

Java & J2EE Made Easy

Java & J2EE Made Easy

PHANI KOSURI

J2EE Consulting Inc.

For online information and ordering of this book, please go to www.j2eeconsultant.net. The author offers discounts on this book when ordered in quantity. For more information, please contact:

Phani Kosuri
J2EE Consulting Inc.,
8400 Gradington Dr,
Westerville, OH 43081

Phone: 614 260 0555
Email: phani.kosuri@gmail.com

© 2007 by J2EE Consulting Inc. The author reserves all rights.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the author.

J2EE Consulting Inc,
8400 Gradington Dr,
Westerville, OH 43081

Cover designer: Chaitanya Thatipamula

ISBN 978-1-4243-3263-2

Printed in the United States of America

Dedicated to

My family and friends.

Contents

*preface xv
what's in this book xvii
acknowledgements xxii*

PART 1

Chapter 1

Introduction 1

What is Java? 1
Java Based Applications 2
How does Java achieve Platform Independence 2
Java Virtual Machine 4
Installing Java Software 4
Environment Setup 4
Compiling and executing a simple program 7
Summary 8

Chapter 2

Java Language Fundamentals 9

Introduction 10
Class 10
Data Types 13
<i>Integer Data Types 14 * Decimal Data Types 14</i>
<i>Character Data Types 14 * Boolean Data Types 15</i>
Casting Data Types 15
Displaying results to User 18
Conditional Statements 19
<i>if, if-else 20</i>
Control Statements 21
<i>while, do-while, for 22</i>
First Java Program 22
Main Method 27
Class Methods 30
<i>Passing arguments to methods 3</i>

Contents

Comments 31
Static Blocks 36
Summary 40
*Time for 50-50 40 * Interview Questions 42*

Chapter 3

Packages 43

Introduction 44
What is a Package? 44
*Creating Packages 44 * Namespace Collisions 46*
*Fully Qualified Class Name 47 * Package Naming Conventions 47*
Access Control using Packages 49
*Public Access 50 * Protected Access 50*
*Default Access 51 * Private Access 50*
Summary 51
*Time for 50-50 52 * Interview Questions 52*

Chapter 4

Object Oriented Programming 54

Introduction 55
What is an Object? 55
Difference between class and object 55
Objects and Methods 58
Understanding object references 68
Encapsulation 73
Method Overloading 78
Object Initialization 81
*Constructor 82 * Constructor Overloading 85*
Inheritance 89
*Method Overwriting 97 * Abstract Methods 109*
*Abstract Classes 109 * Final Methods 113*
*Final Class 113 * Final Variables 113*
Difference between abstract and final keywords 114
Interfaces 114
The *Object* class 120
Summary 122
*Time for 50-50 124 * Interview Questions 126*

Chapter 5

Referential Polymorphism 128

Introduction	129
Loose Coupling	131
Parent Class referencing Child Class	133
Accessing Methods using Inheritance	139
Understanding Interface references	146
Accessing Methods using Interfaces	146
Summary	155

Chapter 6

Exception Handling 156

Introduction	157
Handling Exceptions in Java	157
Exception Types	161
Handling Checked Exceptions	169
<i>Creating Custom Exceptions</i>	169
<i>* Throwing Exceptions</i>	171
Finally Block	177
Summary	178
<i>Time for 50-50</i>	179
<i>* Interview Questions</i>	181

Chapter 7

Core Libraries and Best Practices 182

Introduction	183
Important Core Libraries	183
java.lang package	184
<i>Wrapper Classes</i>	184
<i>* Handling Strings</i>	187
<i>StringBuffer class</i>	191
java.util package	193
<i>ArrayList</i>	194
<i>* Vector</i>	196
<i>* HashSet</i>	197
<i>TreeSet</i>	200
<i>* HashMap</i>	205
<i>* Hashtable</i>	206
<i>Properties</i>	206
<i>* Date class</i>	207
<i>* SimpleDateFormat</i>	208
<i> StringTokenizer</i>	210
Naming Conventions	211
Java Documentation	212
Summary	218

Contents

Time for 50-50 219 * *Interview Questions* 221

Chapter 8

Threads 223

Introduction 224

Multi-Processing 224 * *Multi-Threading* 224

How Multi-Threading is done? 225

Thread Queue 226 * *Scheduler* 226

Thread Lifecycle 227

Multi-Threading in Java 228

Thread Class 228 * *Runnable Interface* 228

Main Thread 229 * *Creating a Thread* 230

Thread Priorities 232 * *Thread Yielding* 234

Synchronization 239

synchronized keyword 239 * *synchronized blocks* 240

Summary 241

Time for 50-50 241 * *Interview Questions* 242

Chapter 9

JAR Files and Using API 243

Introduction 244

Notion of Jar File 244

Creating a Jar file 245 * *Using a Jar file* 246

CLASSPATH 248

Understanding API 251

PART 2

Chapter 10

Introduction to J2EE 255

Introduction 256

Core J2EE Technologies 258

Enterprise Application Architectures 262

2-Tier Architecture 262 * *3-Tier Architecture* 263

N-Tier Architecture 264 * *Enterprise Architecture* 265

J2EE Application Servers 265

Summary 268

Contents

Time for 50-50 269

Chapter 11

JDBC 270

Introduction 272

Database Basics 272

Structured Query Language (SQL) 273

Creating a Table 274 * *Inserting Data* 275

Updating records in Table 275 * *Retrieving records from table* 276

Deleting records 278

Database Drivers 280

JDBC-ODBC bridge 280 * *Partly Java Partly Native Driver* 281

Intermediate Database Access Driver Server 281

Pure Java Drivers 282

JDBC API 283

Creating table 287 * *Inserting Data* 289

Reading Data 292 * *PreparedStatement* 295

Batch Processing 298

Connection Pooling 303

Database Isolation levels 305

Dirty Read 306 * *Repeatable Read* 306

Phantom Read 306

Summary 309

Time for 50-50 310 * *Interview Questions* 311

Chapter 12

XML And Java 312

Introduction 314

What is XML? 314

What is an XML Document 314

Why XML is important? 317

XML Validation 318

Document Type Definition (DTD) 318 * *XML Schema* 321

XML Parsing 325

SAX 325 * *DOM* 325

JAXP 326

SAX Parsing using JAXP 326 * *DOM Parsing using JAXP* 331

Difference between SAX and DOM 336

Summary 337

Contents

Interview Questions 337

Chapter 13

Servlet Programming 339

Introduction 342

HTTP 343

GET Request 343 * *POST Request* 344

Server Side of the Web Application 344

Web Container 344 * *Structure of a Web Application* 346

Servlet Technology 347

Definition of Servlet 347 * *Servlet API* 347

Deployment Descriptor 348

Steps for Writing a Servlet 349

LifeCycle of a Servlet 350

Servlet Initialization 353

Reading HTML Form Data 356

Session Management 361

Creating a Session 362 * *Storing the data in the Session* 363

Reading the data from Session 363 * *Destroying the Session* 363

Request Dispatching 363

The forward() method 364 * *The include() method* 364

Summary 380

Time for 50-50 381 * *Interview Questions* 382

Chapter 14

JSP Programming 383

Introduction 384

JSP Basics 384

JSP Directives 386 * *JSP Declarations* 392

JSP Expressions 393 * *JSP Scriptlets* 395

Implicit Objects 396

Java Beans in JSP 400

jsp:useBean 401 * *jsp:setProperty* 402

jsp:getProperty 402

Custom Tags 420

Summary 427

Time for 50-50 428 * *Interview Questions* 429

Chapter 15

JSTL 431

Introduction	432
Core Tags	432
SQL Tags	437
XML Tags	441
Summary	444
<i>Interview Questions</i>	444

Chapter 16

Struts 445

Introduction	446
Model View Controller Design	447
Why Struts is important to us?	448
How Struts Work?	449
Struts API	450
Struts Requests	451
The Struts Configuration File	452
Action Class	459
JSP Development using Struts Tags	467
<i>Bean Tags</i>	468
<i>* Logic Tags</i>	477
<i>* HTML Tags</i>	481
Struts Validation	483
<i>Validation using FormBeans</i>	484
<i>FormBeans using DynaActionForm class</i>	492
Struts Validator Framework	498
<i>validation-rules.xml</i>	499
<i>* validation.xml</i>	502
Java Script Validation	508
Tiles Framework	510
Internationalization	517
Summary	520
<i>Time for 50-50</i>	521
<i>* Interview Questions</i>	522

Chapter 17

Java Messaging Service 524

Introduction	525
Importance of Asynchronous Messaging	525

Contents

MoM (Message oriented Middleware)	526
Messaging Models	527
<i>Publish/Subscribe Model</i>	527
<i>Point-to-Point Model</i>	528
JMS API	529
<i>JMS Administered Objects</i>	530 * <i>Using JMS API</i> 531
<i>JNDI Lookup for Administered Objects</i>	531
Publish/Subscribe Messaging using JMS	532
<i>Non-Durable Subscriptions</i>	534 * <i>Durable Subscriptions</i> 535
Point-to-Point Messaging using JMS	540
Summary	545
<i>Time for 50-50</i>	546 * <i>Interview Questions</i> 546

Chapter 18

Spring Framework 548

Introduction	549
Spring Framework	550
Goals of Spring Framework	550
Architecture of Spring Framework	551
<i>Core Module</i>	552 * <i>ApplicationContext Module</i> 552
<i>AOP Module</i>	552 * <i>DAO Module</i> 552
<i>Web Module</i>	553 * <i>ORM Module</i> 553
Inversion of Control (IoC)	553
Core Module	557
<i>Why wiring beans</i>	558 * <i>BeanFactory</i> 559
ApplicationContext	561
Aspect Oriented Programming	569
<i>Aspect</i>	570 * <i>Advice</i> 570 * <i>PointCut</i> 571
<i>Before Advice</i>	571 * <i>After Advice</i> 572
<i>Around Advice</i>	573 * <i>Throws Advice</i> 574
<i>Static Pointcuts</i>	575
DAO Module	580
Summary	591
<i>Time for 50-50</i>	591 * <i>Interview Questions</i> 592

Chapter 19

Hibernate 593

Introduction	594
--------------	-----

Contents

Why do we need ORM solution?	596
<i>Features of Hibernate ORM Solution</i>	597
<i>Advantages of Hibernate ORM Solution</i>	597
Mapping Beans to Tables	598
Configuring Hibernate	608
Associations	611
<i>Many-to-One/One-to-Many Association</i>	612
<i>Unidirectional Association</i>	612
<i>Bi-directional Association</i>	616
<i>One-to-One Association</i>	618
<i>Unidirectional Foreign key Association</i>	618
<i>Bi-directional Foreign key Association</i>	620
<i>Unidirectional Primary key Association</i>	621
<i>Bi-directional Primary key Association</i>	623
<i>Many-to-Many Association</i>	623
<i>Unidirectional Association</i>	624
<i>Bi-directional Association</i>	629
Polymorphic Associations	630
<i>Table per Concrete Class</i>	631
<i>Table per Hierarchy</i>	637
<i>Table per Sub Class</i>	639
Hibernate Query Language (HQL)	640
Summary	642
<i>Time for 50-50</i>	643
<i>* Interview Questions</i>	644

Chapter 20

J2EE Design Patterns 645

Introduction	646
What is a Design Pattern?	646
Types of J2EE Patterns	647
Creational Patterns	648
<i>Factory Pattern</i>	648
<i>* Singleton Patterns</i>	650
Structural Patterns	651
<i>Decorator Pattern</i>	651
<i>* Façade Pattern</i>	654
Presentation Tier Patterns	655
<i>Front Controller</i>	655
<i>* Business Delegate</i>	657
Business Tier Patterns	658
<i>Session Façade</i>	660
<i>* Transfer Object</i>	663
Integration Tier Patterns	664
<i>Data Access Object (DAO)</i>	664
Summary	666

Contents

Time for 50-50 667 * *Interview Questions* 668

Non Technical

Inside the Company 669

Introduction 670

Enterprise Application Team Structure 671

Business Analysts 671 * *Technical Project Manager(s)* 672

Non Technical Project Manager(s) 672

Architecture Teams 673 * *Development Teams* 673

Infrastructure Team 673 * *Testing Team* 674

Build/Environment Support team 674

Enterprise Application Lifecycle 674

Framing an Idea 675 * *Business Requirements Gathering* 676

Business Requirement Analysis 676

Application Architecture 677 * *Application Design* 677

Application Development 677

Database Developers 678

Business Component Developers 678

Presentation tier Developers 678

Application Testing 679

Unit Testing 679 * *IST Testing* 680 * *UAT Testing* 680

Performance/Load Testing 681

Production Support 681

Version Control 681

What is a Version Control System? 682

Creating a Project 683

Joining a Project 683

Check in Operation 684

Check out Operation 684

Delivering Work 684

Synchronizing Work 685

Interview Tips 685

Software Installation 686

Installing J2EE 687

Installing MYSQL & DBVisualizer 688

Conclusion

index

preface

In today's world, we all know the role of e-commerce applications and how much impact they had on our lives. With Internet became an integral part of our life, every known business irrespective of size is moving into cyber world from our physical world to capture customers attention. At the same time customers are also searching for ways to make life easy by using internet for most of their day to day activities. Because of this mutual benefit to customers and businesses, e-commerce applications or enterprise applications are taking off from the runway into internet space. These applications have now become the faces of modern businesses on internet and act as a bridge for customers to reach businesses and vice-versa.

One of the key challenges today's e-commerce applications face is sustaining severe competition from other similar businesses. Therefore, to sustain from this immense competition, adoption of sophisticated technologies has become the key factor in exploiting the information assets of a business. Though there are billions of technologies that helped business to some extent to address the challenges, it is the introduction of Java and J2EE that helped businesses to stand firmly and with authority.

For more than a decade, Java and J2EE has relentlessly served and serving the enterprise community in building enterprise e-commerce applications. One of the primary reasons for the success of Java and J2EE is the simplicity that it offers and the flexibility. To make Java and J2EE even strong, several communities have been developed and developing to constantly identify potential challenges and incorporating sophisticated solutions to address challenges.

Java as a programming language is designed to be very simple and yet very powerful. One feature that stands on top of all is the portability factor, the ability to port the code from one operating system to the other. Besides this, Java language includes the most powerful built-in libraries that heavily reduce the development time and also makes the application more reliable. On top of the above, Java programming language has inherent support for concurrent processing, networking and many more.

Though Java programming language like any other programming language allows building standalone applications, its true strength lies on the server side where it is used to build highly sophisticated server applications using networking and multi-threading capabilities. This is what led to the so called client server applications. Java based client server applications over the years have gone through several phases from traditional two-tier applications, where fat clients like GUI's acted as user interfaces for interacting

with server application to the latest n-tier applications that use the standard web browser for interacting with the server via internet.

With enterprises deciding to use internet to do business to increase the customer base, Java applications started getting overloaded due the increase in the number simultaneous requests. This seriously affected the performance and scalability of the application and also resulted in maintenance nightmares. To solve this problem, Java community got into the act and came with a powerful solution. The solution is nothing but the J2EE platform. The entry of J2EE into the world of enterprise application development radically changed the dimensions of enterprise internet computing and has now become almost a de facto standard for building and hosting enterprise internet applications.

J2EE platform is a bundle of several ready made technologies with each being used in different areas within the enterprise application. For instance, some technologies are used to build presentation components, some used for building business components etc. Over the years, the standard J2EE technologies served the industry and are still serving. Though the J2EE technologies provided end to end solutions for building enterprise applications, the quest for building better Java based solutions didn't stop. This quest is what led to the so called open source frameworks.

Open source frameworks don't reinvent the wheel but try to build a better wheel. These frameworks are again built from the core J2EE technologies to further simplify the application development. Few such open source frameworks that have become very popular these days are Struts, Spring and Hibernate. It is very important to understand that these frameworks don't replace the entire J2EE platform, but serves an alternative to some of the standard J2EE technologies within the platform. With open source frameworks also gathering momentum, the blend of Java, J2EE and open source frameworks has become a killer combination for enterprise internet application development.

This book covers all the above three. Starting with Java programming language, it slowly takes you though the journey of J2EE and open source. I hope you'll surely enjoy the journey and have a good time. Good luck.

What's in this Book

Chapter 1: Introduction

This first chapter serves as a brief introduction to Java and its role in today's world. It talks about how Java achieves platform independence and the types of applications that can be built using Java. If you don't have any idea about what Java is all about, this is a great place to start. It also explains how to install Java and set up the environment for the remaining chapters.

Chapter 2: Java Language Fundamentals

This chapter helps you to understand and write simple Java programs and highlights some of the important Java keywords. This chapter ensures that you understand how to write, compile and execute basic Java programs using classes. Understanding this chapter is important, as it forms the basis for the next several chapters.

Chapter 3: Packages

This chapter demonstrates the idea of using packages to logically organize Java classes. It also explains the usage of various keywords associated with packages and how to implement access control for classes stored within packages.

Chapter 4: Object Oriented Programming

This chapter will demonstrate Object Oriented Programming (OOP) concepts and their implementation in Java. This is the most important chapter and understanding it makes the journey through Java more interesting and fun. You'll surely enjoy this chapter since OOP is fun and its associated terminology is heard in our day to day life. This chapter explains all the OOPS concepts like Encapsulation, Inheritance etc.

Chapter 5: Referential Polymorphism

This chapter teaches you on how to use parent class and interface references. By the end of this chapter, you will be able to know how and when to use inheritance and interfaces in real world applications. The ideas and examples in this chapter are totally based on what we learned in the previous chapter. Understanding the concepts in this chapter is utmost important since most of the real world Java applications are based on Object Oriented Design whose primary weapons are inheritance and interfaces.

Chapter 6: Exception Handling

This chapter explains about how to handle abnormal conditions which we call as exceptions in Java programs. Exception handling is one of the most important aspects of any Java based application. Understanding this chapter is very important to get a good feel about what Java is all about.

Chapter 7: Core Libraries and Best Practices

This chapter explains the important built-in Java class libraries. Java is all about writing simple programs using the built-in classes. This chapter will tell you some simple tricks and tips on how to use classes and also demonstrates the coding conventions and best practices that are well and truly followed in all the real world Java applications. Understanding this chapter gives you an edge for a successful career in Java.

Chapter 8: Threads

This chapter will introduce you to concurrent processing in Java. Java language has built-in support for writing programs that run simultaneously. This chapter explains the nuances of multi-threading and the core concepts surrounding it.

Chapter 9: JAR files and Using API

This chapter gives you the details about creating and using Jar files. JAR file is used to archive Java applications and is the universal way for porting Java code. This chapter also explains about how to use various API's to build Java applications.

Chapter 10: Introduction to J2EE

This chapter introduces you to the world of J2EE by highlighting the important J2EE technologies and their application. This will also give an idea about J2EE application servers and their role in building enterprise applications.

Chapter 11: JDBC

This chapter teaches you to write Java applications that talk with databases. This chapter will explain the details of what a database is, how to use JDBC technology to connect with databases, and how to execute the SQL queries against the database. This chapter also gives you a brief idea of SQL and its syntax.

Chapter 12: XML and Java

This chapter explains what XML is and its usage in Java. This chapter will also give you an idea on parsing XML documents using JAXP API. Understanding this chapter is very important from J2EE point of view as XML is widely used in almost all the J2EE applications.

Chapter 13: Servlet Programming

This chapter introduces you to the world of Web Applications using Java. Since Web Applications form the basis for enterprise internet applications, starting with this chapter, the next few chapters give you all the details about Web application development using J2EE and open-source technologies. This first chapter introduces you to basic web application development using Servlet technology and also gives you an idea about Web Containers.

Chapter 14: JSP Programming

This chapter introduces you to the most widely used web technology, the Java Server Pages (JSP). By the end of this chapter, you'll become familiar in building web applications using JSP pages. This chapter will explain you the key concepts like using Java Beans in JSP pages, and building custom tags.

Chapter 15: JSTL

This chapter teaches you how to build JSP pages using JSTL tags. JSTL has become very popular these days and understanding JSTL tags give you an edge while building JSP pages.

Chapter 16: Struts

This chapter introduces to the most popular and widely used web technology called Struts. This is an open source framework developed by Apache Software foundation. By the end of this chapter, you'll understand the important features of Struts and how web applications are built using it. This chapter starts with the discussion about MVC architecture, and then explains all the important Struts components with examples. It covers validator framework, tiles framework and Internationalization.

Chapter 17: Java Messaging Service (JMS)

This chapter introduces you to asynchronous messaging using JMS. Important messaging models such as Publish/Subscribe and Point-to-Point models will be discussed in detail. By the end of this chapter you'll know the basics of messaging systems and how synchronous and asynchronous messaging is implemented.

Chapter 18: Spring Framework

This chapter introduces you to yet another open source framework for building enterprise applications, the Spring framework. This chapter covers spring's Core module, AOP module and DAO module. This chapter also demonstrates the usage of AOP programming.

Chapter 19: Hibernate

This chapter introduces you to one of the most popular open source ORM solution, Hibernate. By the end of this chapter you'll understand how to use hibernate framework for mapping objects to relational database systems and how to retrieve objects from database using hibernate query features.

Chapter 20: J2EE Design Patterns

This chapter explains you the important J2EE design patterns that are normally used while designing enterprise applications. Knowing these design patterns help you to design applications that perform well.

Chapter 21: Inside the Company

This chapter is purely non technical and gives you a glimpse of enterprise application development lifecycle. Understanding this chapter is very important and gives you the details about how applications will be developed, how different teams co-ordinate and all that good stuff.

Who should read this book

Java and J2EE Made Easy is for all Java developers, but people who don't have any background in Java and J2EE and looking for a jump start in their career in Java will find this book extremely useful. This book will take you through the journey starting with core Java and all the way into J2EE and Open Source. All the examples in this book clearly demonstrate the concepts and most of the examples, specially starting from J2EE are accompanied with step by step instructions.

This book is written in such a way to correlate every concept with real world scenarios. Every chapter ends with a small quiz and interview questions that should help every prospective Java developer to land in a good Job. This book also covers the non technical details of enterprise application development such as typical team structures, team collaboration, and application development lifecycle to bring awareness about how things are handled in real world applications.

Code conventions

This book contains several examples. All the examples will appear in `code` font. If there is a particular part of an example that I want you to pay more attention, it will appear in bolded code font. All the examples in this book are complete, and can be executed without making any changes.

Source Code download

The complete source code for all the examples this book can be downloaded from the authors website at <http://j2eeconsultant.net>.

Author Online

This book has free access to web forum where you can make comments about this book and ask technical questions and receive help from the author and from other users. To access the forum, point your web browser to www.j2eeconsultant.net. This page provides information about how to get to the forum, what kind of help is available.

About the author

An Electrical Engineering Graduate from the University of Nebraska, Lincoln, Phani Kosuri is a Sun Certified Architect and a professional IT consultant with a handful of experience in the world of J2EE. Professional activities include designing and developing software tools and training folks in Java and J2EE. Currently lives in Columbus, Ohio.

acknowledgments

Firstly, let me tell you this. This book was not a one man job. Many of my friends played a great role in shaping out this book to what it is.

To my friends - thanks for all their help, motivation that kept me going.

To my family - thanks for giving me everything that I needed.

Part 1

Java

Chapter 1

Introduction

What is Java?

Java™ is a programming language introduced by Sun Microsystems Inc. Like any other programming languages Java is also used for building software applications. If this is the case, then why do we need Java? Good question. Here is what makes Java distinctive from other programming languages. Most programming languages like C, C++ etc, are targeted for a particular *platform* (a.k.a operating system). What this really means is that, if we write a program in a language like C or C++ or any other language, say on an OS like Windows, then the same program cannot be executed on a different OS like, say Linux, without making some changes to the source code. These changes could span anywhere from modifying few lines of code to rewriting the entire program. This is not good, right? Therefore, we say such programming languages as being *non portable* in nature.

When do we say a programming language as being portable?

When the *machine code* generated by *compiler* of a programming language can be used on any OS without any modification, then we say the programming language as being *portable* in nature.

Keeping the above philosophy in mind, creators of Java language made it *portable*. Sweet! We'll see how this portability is achieved later. For now, this portability feature is what makes Java distinctive from other programming languages and this is the reason why we call it as *platform independent* language. Believe me, this one feature is the driving force and motivation for today's modern enterprises that uses Java technologies. And moreover, this is also the reason why we are interested in learning Java.

Java based Applications

Besides Java language being platform independent, it is the nature of applications that we can build using Java makes it even more special. Java can be used to build two types of applications as listed below:

1. Standalone applications
2. Internet applications

Standalone applications are the applications that run on a single computer. All the command line applications and GUI based applications where funky windows are used to interact with the applications fall in this category. This is one side of Java. The other side of Java is that it can be used to build applications that can be run on the internet. This is where Java unleashes its true power. We all agree without any slightest hesitation that internet is part of our life. Here is how I look at internet and Java. Internet connects people and organizations around the globe to exchange information. To achieve this, we need to build high speed applications that run on internet. Prior to Java, there are other languages and technologies with which internet applications are built, but they suffered from serious problems and limitations. The advent of Java and its adaptability on internet has completely changed dimensions of business organizations and the way business is done. This is the true strength of Java, *its adaptability on the Internet*.

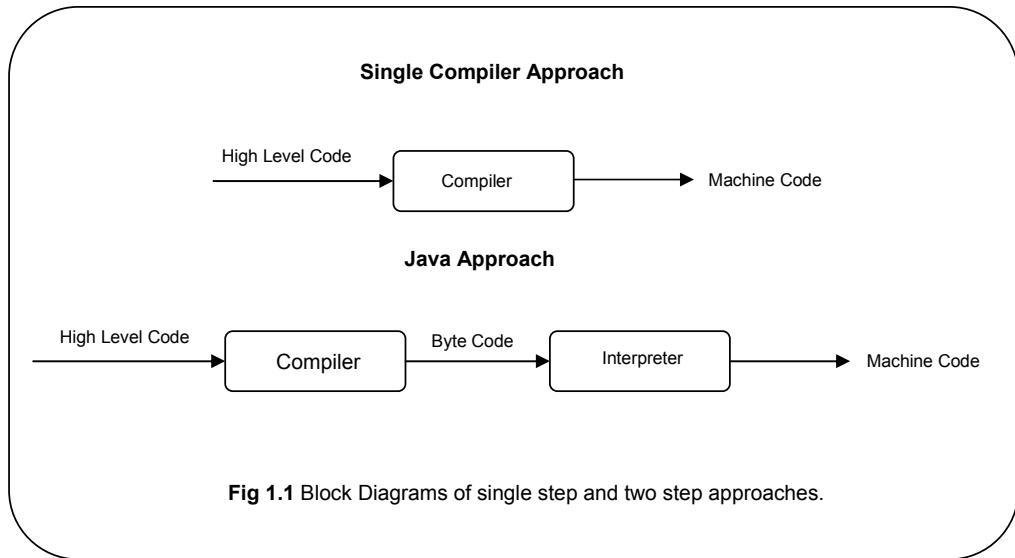
How does Java achieve Platform independency?

First things first. As I said before that Java is platform independent, the question is, how does it achieve this independency? To understand, there is some thing called *bytecode* that we need to know about. Most programming languages use compilers to translate *high level program* to *machine code* of the OS in a *single* step. Once the machine code is generated we cannot port it to a different OS and can only be used on that OS only. Therefore, any language that uses just the compiler to compile as well as execute the program is *platform dependent*. Java recognized this problem upfront and came with a novel idea by breaking the compiler into two different programs as compiler and interpreter.

Java uses *compiler* to translate high level Java code into bytecode, and uses the *interpreter* to translate bytecode to machine code. This is a two step process as opposed to single step process shown in Fig 1.1.

Chapter 1

Introduction



The generated bytecode is like an intermediary code and is 100% platform independent. This bytecode can now be ported across any OS. After porting it, Java uses the second component, the interpreter, to convert the bytecode to machine code of that OS. Smart, isn't it. So in Java, we port the intermediary bytecode but not the OS specific machine code itself. From Java programming standpoint, we treat the bytecode as the executable and this is how we achieve platform independence. In this two step process, it uses *javac* program as a compiler and *java* program as interpreter. Fig 1.2 shows the idea of platform independency.

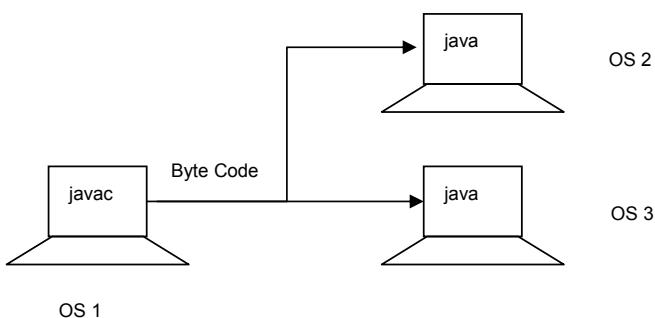


Fig 1.2 Demonstration of Platform Independence in Java.

Java Virtual Machine (JVM)

The *java* program, which is the interpreter is also referred to as *Java Virtual Machine* or simply JVM because of the fact that, it very much behaves like a CPU, except that it is not really a physical CPU. It is rather, a virtual CPU that processes the Java bytecode or in other words a virtual machine that processes Java bytecode, which leads to the name Java Virtual Machine.

Few reasons why we are interested in Java are:

- ✓ Java language is easy to understand.
- ✓ Java programs are portable
- ✓ Java programs are robust and secure.
- ✓ Java can be used to build multithreaded programs.
- ✓ Java can be used to build Internet applications.
- ✓ Programs written in Java are highly scalable and distributable.
- ✓ Above all, Java is based on Object Oriented Programming model.

Now that we know what Java is, we can start learning the intricacies of Java programming language. But before that, we need to install Java software and set up the working environment.

Installing Java Software

Sun Microsystems Inc is the distributor for Java software which is bundled as Java Software Development Kit or simply JDK. The JDK includes both the Java compiler (*javac*) as well as interpreter program (*java*). The latest version of JDK at the time of writing this book is 1.5, which is what we shall be using throughout. The examples in this book also work on Java SDK 1.4. Download the latest version at the following link.

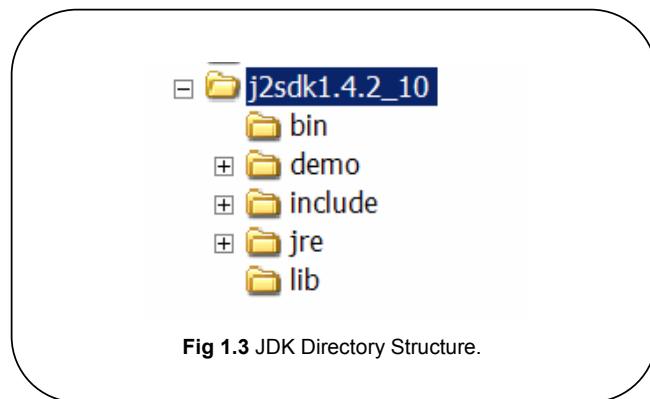
http://java.sun.com/javase/downloads/index_jdk5.jsp

By default, JDK will be installed to “C:\Program Files\j2sdk1.5”. You can however choose a different directory at the time of installation.

Environment Setup

Once you downloaded and installed JDK, we need to set up our working environment to run the examples.

Assuming the JDK is installed to “C:\Program Files\j2sdk1.5” directory, you see several sub directories as shown in Fig 1.3 below.



We call the installation directory “C:\Program Files\j2sdk1.5” as the *home* directory for Java and is referred with the name `JAVA_HOME`. This is one of the global environment variables we need to set up.

Table 1.1 lists various subdirectories within JDK and their purpose

Table 1.1 JDK Directories

Directory	Purpose
bin	This directory includes Java Compiler, Interpreter (JVM) and several other utility tools.
demo	This directory contains several examples.
include	This directory will have header files that be used by the JVM. Not important for us.
jre	This directory includes the Java Runtime Environment (JRE), the core libraries and security settings.

Setting up the global environment variables

In this section, let's set up two environment variables namely `JAVA_HOME` and `PATH`.

Go to *Start -> Settings-> Control Panel -> System -> Advanced -> Environment Variables*

In the system variables section do the following two things.

1. Click New, which opens a window. Type JAVA_HOME for variable name and C:\Program Files\j2sdk1.5 as variable value, and click OK.
2. Select the PATH variable in the same section. Click Edit, and append ;JAVA_HOME\bin to the existing variable value and click OK.

This completes the Java set up process. For running all the examples in this book, let's create the following directory:

C:\JavaTutorial

Finally, to validate the Java setup, let's do the following three things.

1. Open the Command Window and move to the "JavaTraining" directory.
2. Type the command javac, and you should see something as shown below.

```
C:\JavaTraining>javac
Usage: javac <options> <source files>
where possible options include:
  -g                      Generate all debugging info
  -g:none                 Generate no debugging info
  -g:{lines,vars,source}   Generate only some debugging info
  -nowarn                 Generate no warnings
  -verbose                Output messages about what the compiler is doing
  -deprecation            Output source locations where deprecated APIs are us
ed
  -classpath <path>        Specifi where to find user class files
```

3. Type the command java, and you should see something as shown below

```
C:\JavaTraining>java
Usage: java [-options] class [args...]
           (to execute a class)
      or  java [-options] -jar jarfile [args...]
           (to execute a jar file)

where options include:
  -client      to select the "client" VM
  -server      to select the "server" VM
  -hotspot     is a synonym for the "client" VM  [deprecated]
               The default VM is client.
```

If everything worked as above, we successfully completed the Java Setup Process. Good Job.

How to Compile and Execute a Java Program

Let's take a simple Java Program, and learn how to compile and execute it. Don't worry about what the program is, we will see that later. This is just like a match practice before the real game. Copy the following program into your favorite editor:

```
public class Greeting {  
    public static void main (String args[]) {  
        System.out.println("I am ready to taste Java with Passion ");  
    }  
}
```

Save the program as C:\JavaTraining\Greeting.java. To compile the program, use the *javac* command and specify the file name as shown below:

```
C:\JavaTraining>javac Greeting.java
```

The above command will compile the source code in the file. If there are any errors it will display them, otherwise, it generates the bytecode file and displays the prompt. To see the bytecode file, type *dir* command, and you will see the bytecode file *Greeting.class* as shown below:

```
C:\ JavaTraining >dir  
Volume in drive C has no label.  
Volume Serial Number is 685A-CF9E  
  
Directory of C:\JavaTraining  
  
10/26/2006  09:33 PM      <DIR>          .  
10/26/2006  09:33 PM      <DIR>          ..  
10/26/2006  09:33 PM           444 Greeting.class
```

Now, we need to execute the bytecode using *java* command and pass the bytecode file, to see the output as shown below:

```
C:\JavaTraining>java Greeting  
I am ready to taste Java with Passion
```

Note: You must not specify *.class* extension in the *java* command. It will automatically figure it out.

Practice the above two commands 2-3 times. In the next chapters, when I ask you to execute a program, this is what I expect you to do. Now that we know how to compile

and execute a program, we are ready to learn the nuances of Java Programming Language. This completes this chapter, and I am sure you learned something. Let's move forward.

Summary

- ✓ Java Programming language is invented by Sun Microsystems Inc.
- ✓ Java is Platform Independent language.
- ✓ Java uses bytecode to achieve platform independence.
- ✓ The bytecode generated will have ".class" extension.
- ✓ The program that translates the bytecode to machine code is called as Java Virtual Machine (JVM).
- ✓ To compile a Java program we use *javac* command, and to execute a program we use *java* command.

Chapter 2

Java Language Fundamentals

This chapter helps you to understand and write simple Java programs and highlights some of the important Java keywords. Understanding this chapter is important as it forms the basis for the next several chapters. Honestly, from my experience if you know what a *class* is and what a *method* is, and some keywords associated with them, you can call yourself as a *Java Programmer*. With I saying this, it doesn't mean you can skip the rest of the chapters. :-)

Chapter Goals

- ✓ Understand the basic elements of a typical Java program
- ✓ Understand the primitive Java data types
- ✓ Learn to write and execute Java Programs
- ✓ Understand Class concepts
- ✓ Standard Java conventions and best programming practices

Environment Setup

All the programs in this chapter should be stored in the following directory.

C:/JavaTraining/Chapter1

From the command prompt, move to the above directory.

Introduction

This chapter introduces you to basic Java programming. One of the primary goals while designing Java is to keep things *simple*. This is the reason why Java programs are very easy to write and understand unlike other programming languages. The best way to learn anything is to enjoy it, and once you start enjoying it, everything looks unbelievably simple and beautiful. So, let's start rolling the ball.

A typical Java program is usually stored in a file with the extension “.java”. This file represents the Java source code that we write. Upon compiling the Java source code, the Java compiler generates bytecode for the program. This generated bytecode is stored with “.class” extension which we then use to execute the program and get the results.

Class

The building block for any Java program is an entity called *class*. So, what is a class? A class is nothing but an entity that holds *state* and *behavior*. State is represented by *variables* and behavior is represented by *methods* that operate on the state.

There can be *one or more than one class* in a single Java source file. So, we can say that a Java program is nothing but a collection of classes and these classes can interact with each other to process requests and return results.

Now that we know a Java program as a collection of classes, the question that comes to our mind is, which *class* is the starting point of our program? Very good question. Just like a flowchart has a start block from where the process starts, a Java program has a class from where the execution begins. We call this class as the *top-level* class. Like a flow chart cannot have multiple start blocks, there cannot be multiple top-level classes for a single Java program. To conclude, a Java program can only have ONE top-level class which we also call it as **main** class. The Java Virtual Machine (JVM) upon completing all the interactions with other classes finally terminates the main class.

Every class in a Java program is identified by a name called *classname*. If a Java file has multiple classes, then every class must have a unique name. As an example, we can have three classes named TestClass1, TestClass2 and TestClass3 stored in a single Java source file namely Example.java.

Compiling a Java Program with Classes

When we compile a Java source file using *javac* command, it generates “.class” file for *each and every class* in the Java file. For instance, let's say we have three classes namely *TestClass1*, *TestClass2* and *TestClass3* in a single Java source file *Example.java*. When *Example.java* is compiled using *javac* command, it generates three class files namely *TestClass1.class*, *TestClass2.class* and *TestClass3.class*. These class files have the so called bytecode of each class. This bytecode is Java specific code, and can be copied or ported to any operating system. Upon copying the class files, we can then use *java* command on that operating system to execute the class files. Trust me; you will see the same results on any platform. This is how *platform independence* is achieved with Java.

Note: When any Java source is compiled, one bytecode file will be generated for every class in the program, and the name of the bytecode file will be the name of the class with a “.class” extension.

Good Practice: Try to write one class per Java file, and ensure the Java file name is same as class name. For instance, if a program has a class named *Test*, then save it to a file named *Test.java* which upon compilation will generate *Test.class*. This way you don't have to remember multiple filenames for the same program. However, if you save the same program to a file named *SampleTest.java*, upon compilation it will still generate *Test.class* since the class name still remained the same. The problem is you have to remember two file names namely *SimpleTest.java* and *Test.class*. Are we not complicating things here?

Complicating things is what we should always strive to avoid. Keep life simple and easy.

A Typical Java Program

Before we start writing any Java program, it's very important to understand how a typical Java program looks like. So, take a look at a typical Java program.

Listing 2.1 (*Calculator.java*) A typical Java program.

```
// One Package Statement
package chapter1;

// One or more import statements
import java.io.*;
import java.util.*;

// Class Declaration
public class Calculator {
```

```
// State. Variables and Constants  
int i=10;  
double k = 2.5;  
  
// Behavior, one or more methods  
void printSum(){  
    System.out.println("The sum is " + (i+k));  
}  
}
```

As seen from Listing 2.1, a typical Java program will have the following basic elements.

- ✓ One and only one **package** statement. The package statement specifies the directory in which the Java file is stored. In the above program, the package statement denotes that the file is stored in chapter1 directory. The package statement is optional but using packages is a standard programming practice.
- ✓ One or more **import** statements for importing the library classes and other utility classes to use them in the current program.
- ✓ A **class** with a classname that defines the state (as variables and constants) and behavior (as methods). A class can have 'n' number of variables and methods.
- ✓ Comments. These are the text lines explaining the parts of the program.

Now that you know how a typical Java program looks like, it's time to learn how to write simple Java programs, and more importantly following best practices. Wherever applicable, I am going to highlight the best practices and standard coding conventions. It's very important that we follow them right from the beginning to get used to them without having to spend that extra time to learn them. This is one of the goals of this book.

Though we are ready to write our first Java program at this point, I thought it's very useful to learn certain things that are seen frequently in any computer program written in any language. Learning such things upfront will help us not only to write programs faster but also makes understanding a program much easier and fun. So, without wasting any further time, let's see what these common elements are.

The fundamental elements that are seen in any computer program are

1. Data types declarations.
2. Displaying results to the user
3. Conditional statements like **if**, **if-else** etc.

4. Control statements like `while` loops, `for` loops etc.

In my opinion, it doesn't make sense to write a program without knowing how to write the above four mentioned. We will not dedicate a complete chapter for the above, but we learn them as simple essentials for any program. This way we can save more time to learn the important things.

Data Type Declarations

Data types are the most important elements of any program as they allow us to store data. Writing a program without these is almost of no use. Data types basically allow a program to store primitive data like integers, decimal numbers, characters etc. Like any other programming language, Java also defines several primitive data types for storing data as described below.

Integer Data Types

For storing integers, Java uses the following 4 data types based on the size of the integer data.

`byte` - This data type allocates maximum of 1 byte (8 bits) for storing the number.
`short` - This data type allocates maximum of 2 bytes (16 bits) for storing the number.
`int` - This data type allocates maximum of 4 bytes (32 bits) for storing the number.
`long` - This data type allocates maximum of 8 bytes (64 bits) for storing the number.

Table 2.1 shows the range of values that can be safely stored in each of the above integer data types.

Table 2.1 Integer Data Types

Data Type	Size	Range
<code>byte</code>	1	-128 to +127
<code>short</code>	2	-32768 to +32767
<code>int</code>	4	-2147483648 to 2147483647
<code>long</code>	8	-9223372036854775808 to + 9223372036854775807

Most Java programs often use `int` and `long` data types for storing integers. Following are some examples of integer variable declarations.

```
int    i = 10;
long   k = 123456789;
byte   b = 200;
short  s = 12345;
```

Decimal Data Types

For storing decimal numbers Java uses the following two data types based on the precision of the decimal number

`float` - This data type allocates maximum of 4 bytes (32 bits) for storing the decimal number. This data types offers less precision (number of digits following the decimal point). While declaring the `float` data type, we need to use the literal '`f`' at the end of the number as shown below.

```
float rate = 13.5f;
```

Failing use the literal will result in a compilation error.

`double` - This data type allocates maximum of 8 bytes (64 bits) for storing the decimal numbers. This data types offers very high precision.

Unlike `float`, we need not use any literal like '`d`' at the end of the number. This is because Java by default treats any decimal number without a literal as a `double`. Following is how we declare a `double`.

```
double rate = 2.3456789;
```

Most programs use `double` instead of `float`, as it supports more precision.

Character Data Types

For storing single characters, Java uses the following data type.

`char` – This data type allocates a maximum of 2 bytes for storing characters. Characters must be enclosed in single quotes during initialization as shown below:

```
char ch = 'A';
```

In Java, we can also store the ASCII number corresponding to the character. For instance the ASCII code of character 'A' is 65. So, the following declaration is perfectly legal.

```
char ch = 65; // No quotes with this type of representation.
```

The above representation allows us to do character arithmetic, but this form is seldom used in programs. So, don't worry about it. Just remember the previous one.

Boolean Data Types

Java introduces a new data type for storing boolean values `true` and `false` as shown below.

```
boolean isMarried = false;  
boolean isOwner = true;
```

Note that we don't have to use single quotes or double quotes around. This is one of the most important data type and is normally used with decision making statements such as *if, for, while* etc.

Now that we know about several Java data types, there is one concept around data types called *datatype casting* which we need to understand. Let's learn what this is to complete the discussion on data types.

Casting Data types

Casting data types is nothing but converting a data type from one form to another form. For instance, we can convert an `int` data type to a `float` data type and vice-versa. However, there are some simple rules to understand while doing casting. Just read them once and if you think they are confusing, simply ignore them. You will still be fine.

Rule 1: First rule of thumb is, it is always *illegal* to assign a larger integer data type to smaller integer data type. This causes a compilation error. Following is the order of data types from smallest to largest.

`byte < short < int < long`

Based on the above order, following assignments are illegal. The numbers in the parenthesis are the sizes (in bytes) of the data types.

```
int(4) = long(8), byte(1) = int(4), short(2) = int(4)
```

Consider the following integer declarations.

```
int i = 10;
byte b = 100;
short s = 22;
long k = 20;
```

With the above declarations, following assignments are illegal and cause compilation errors.

```
i = k; // Illegal, even if the size of variable i is big enough to store 20.
b = i; // Illegal
s = i; // Illegal
```

To make the above assignments legal, we need to *cast* them as shown below.

```
i = (int) k; // Explicitly convert long k to an int and then assign to i
b = (byte) i; // Same reason
s = (short) i; // Same reason
```

You know what, unknowingly you are paying some price to make the compiler happy and that price is *data loss*. Hmm... Let's see what this is with an example.

Consider the following integer declaration

```
int i = 500; // i can store 500 safely, since it allocates 4 bytes of memory.
```

Now, look at the following casting from *int* to *byte*. We are converting larger data type to a small data type. We call this as *narrowing* the data type.

```
byte b = (byte) i;
```

With this assignment, the value of b will become -12. This is because, a single byte can only store a maximum value of 127 and JVM uses the cyclic process and makes it -12. This is not good, right? So, the moral of the story is, be careful while casting data types. Do casting only when you know for sure that the data in the larger data type is small enough to store in the smaller data type, in which case you don't lose any data.

Rule 2: You can assign any decimal data types (double, float) safely to integer data types (int, long etc.,) even though the decimal data types are *larger than* integer data types. As an example, consider the following assignments.

```
int i = 10; // Allocates 4 bytes for variable i
```

```
double d = 2.5; // Allocates 8 bytes for variable d
```

With the above declarations, look at the following assignment.

```
i=d;
```

The above assignment is perfectly legal in Java and will not result in a compilation error even though a larger data type is assigned to smaller data type. The thumb rule is, `float` and `double` data types can be assigned to any integer data types irrespective of size. But the price we pay is we loose the precision.

Rule 3: The result of any integer arithmetic expression is always

- a) An `int` if all the data types participating in the expression are *equal to or less than int*.
- b) Largest data type in the expression if one of the data types participating in the expression is *greater than int*.

As an example,

```
int i=10;
long k = 20;

i = i+k;
```

The above is a *greater than int* expression and the result is `long` as per *Rule 3(b)*.

We know that `long` cannot be assigned to `int` variable `i`. So the above assignment results in a compilation error. One solution is to convert the result of expression to `int` as shown below:

```
i = (int) (i+k); // Compiler is happy.
```

The other solution is assigning the expression to a `long` variable as shown below.

```
k = i+k;
```

This time we are assigning `long` to `long`. So, the compiler is happy.

Hmmm, lot of rules right. Don't worry, these are seldom seen in programs, but all I say is beware of such things as they may be the root cause in a million dollar loss for a company. Don't believe me? Read at the following paragraph.

Let's say a customer visits a website and purchases a product for \$2.99. When the order is submitted, if the processing system assigns the price it to an "int", the company loses 99 cents on the product, since it trims off the digits after decimal point. If the company is like Amazon.com which sells millions of products a day, think of how many \$\$ it loses. The result is "You are fired".

For the rest of the chapters, just remember the following things and you'll be good.

- ✓ *int* and *long* are used for storing integers,
- ✓ *double* is used for storing decimal numbers,
- ✓ *char* is used for storing single characters, and
- ✓ *boolean* is used for storing true or false.

This completes data types in Java. This is more than enough.

Displaying results to user

If the program doesn't echo results to the user, what is the purpose of writing a program? So, this is my top of the list candidate. In Java, to display any results to the user (in the console), we use the following statement.

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

The above statement echoes anything specified in between the parenthesis to the console. At this point let me tell you one thing. Java is a case-sensitive programming language which means in the above statement, 'S' must be in uppercase. Though it kind of frustrates you initially, you will start to enjoy as you move forward. There is a trick I'll tell you when we start writing programs, and this will no longer be frustrating.

In Java we usually display data in variables, text messages to the console. So, let's see how we display them.

1. To display as text message, use the double quotes around text as shown below,

```
System.out.println( "We learn Java with Passion ");
```

2. To display the value of a variable, we do so as shown below.

```
int i =10;  
i = i*2;  
System.out.println (i);
```

Simply specify the variable name in parenthesis. What if you want to display more than one variable? Separate each variable with an empty text and a + operator as shown below

```
int i = 10;  
int k = 20;  
  
System.out.println (i + "    " + k);
```

Look at even better way of displaying data.

```
int i =10;  
int k = 20;  
int sum = i+j;  
System.out.println ("The Sum of "+ i + " and " + k + " is " + sum);
```

The above statement displays the following

```
The Sum of 10 and 20 is 30
```

We use the above statement a lot in the programs. Just recite the above statement five times and you'll never forget it in your life. Sometimes we need to do this, as it gives you more command on the language, gives you the rhythm and boosts your confidence.

Conditional Statements

Like any programming language, the conditional statements that Java uses are again the same old “if-else” blocks. In Java, we need to specify a *boolean* expression in the “if” statement. If the expression evaluates to *true*, the *if* block gets executed, other wise the control goes to *else* block if one is present. Before we look at the examples, let’s see the conditional operators we have in Java that return a boolean value.

Following are the conditional operators that we can use:

- > Greater than
- < Less than
- >= Greater than Equal to
- <= Less then equal to
- == equals to
- != Not equals to

Any expression that uses the above operators returns a boolean value *true* or *false* as shown below

```
boolean test = (2 >= 5);
```

The variable *test* will have *false*, since the condition is evaluated to false. The above statement is a single condition statement. We can also use multiple conditions in a single statement using logical AND (**&&**) and OR (**||**) operators as shown below:

```
boolean test = ((10 == 10) && (5>=3)) || ( 2<5));
```

As we all know, **&&** returns ‘true’ if both the conditions are true, and **||** returns true if one of the conditions is true.

See the following 4 examples.

Example 1

The following is a simple *if* statement. It executes all the statements within the block when the condition is evaluated to true.

```
int i = 10;  
  
if (i>=10)  
{  
    System.out.println("i is greater than 10");  
}
```

Example 2:

The following demonstrates an *if-else* block. If the condition evaluates to false, the control goes to the else block. This example also demonstrates nested ‘if’. Look at the ‘if’ statement inside the else block.

```
int i = 2;  
  
if ( i>=10) {  
    System.out.println("i is greater than 10");  
}  
else {  
  
    if ( i < 5){  
        System.out.println("i is less than 5");  
    }  
}
```

Example 3:

In the following example we declared a boolean variable 'found'. The first *if* block sets the value to *true*. Therefore the second *if* block will be evaluated to *true*, and displays the statement 'Found Monkey'. It's perfectly legal to pass a boolean variable directly to an if statement.

```
boolean found = false;
int i=10;

if (i == 10) {
    found = true;
}
if (found) {

    System.out.println("Found Monkey");

} else {

    System.out.println("Damn. It escaped");
}
```

Example 4:

The following example demonstrates the usage of multiple conditions in *if* block.

```
int i=10; int j = 20;

if( (i==5) || (j != 10) ){
    System.out.println( "Hello 1");
}
if ( (i ==10) && (j==20) ){
    System.out.println( "Hello 2");
}
if (( i==10 && j==10 )|| ( j==40 ) ){
    System.out.println( "Hello 3");
}
```

Control Statements

Control statements are the most important elements of any program. Control statements allow a set of statements to execute repeatedly until a certain condition is met. There are three types of control statements, namely, *while*, *do-while* and *for* loops.

See the following four examples.

Example 1:

This is an example of *while* loop. The condition is specified in the *while* statement. If the condition is evaluated to *true*, then the statements inside the loop will be executed. At some point of time, it's important that the control comes out of the loop, otherwise the program hangs. So, we usually place a statement inside the loop that makes the condition fail after few iterations. In the following code it is the *i++* statement. This will increment the value in variable *i* by 1 every time the control comes inside the loop. When the value of *i* becomes 11, the *while* loop condition evaluates to *false*, and the control comes out of the loop.

```
int i = 1; // Initialization
while( i<= 10)
{
    System.out.println("Counter value is " + i);

    i++; // Increment or decrement
}
```

Now, a small question for you. What if the condition fails for the first time itself? None of the statements inside the loop will be executed, right? Now, I have a requirement where the statements inside the loop must be executed *at least once* before the condition is checked, what should I do? Is there is simpler solution. Yes, we have one. Look at the next example.

Example 2:

This example uses a *do-while* loop. This works much the same as previous *while* loop, except the statements in the loop will be executed before the condition is checked. So, we are 100% guaranteed that the statements in the loop are executed at least once. Is this what our requirement is? Cool. So, using *do-while* loop is the solution. Notice the semicolon after the *while* statement.

```
int i = 1;

do
{
    System.out.println("Counter value is " + i);
    i++; // Increment or decrement
}
while( i<=10);
```

Example 3:

This example uses a *for* loop. This is the simplest of all in the sense it declares initializer, condition and condition changer all in one line. Note that variable *i* is declared inside

the loop itself. This is perfectly legal. Unlike in C, you need not declare the variables at the beginning of the program. You are free to declare them wherever you want. Declare and use, one of the coolest features in Java.

```
for (int i =1; i<=10;i++)
{
    System.out.println("Count value is " + i);
}
```

Example 4:

We can also define a loop within a loop as shown below. These are called as nested loops. For every value of *i* in the outer loop, the inner one loops 10 times. So, guess the output.

```
for (int i =1; i<=10;i++)
{
    for ( int k = 1; k<= 10; k++)
    {
        System.out.println("Count is " + (i+k));
    }
}
```

That's it guys. We finished learning the basic elements of any Java program and we are now ready to take on the big ones. The reason I've explained all this upfront before even writing a single Java program is that you will feel more comfortable and confident when you start writing real programs. This saves us whole lot of precious time.

To run all the examples in this chapter, please follow the steps outlined in the environment setup at the beginning of the chapter.

First Java Program

As I said before, a Java file simply contains one or more classes. The standard syntax for defining a class is shown below.

```
class <class name>
{
}
```

class is a Java keyword that identifies a Java class. Listing 2.2 shows a single class defined in the Java file.

Listing 2.2 (`SimpleClass.java`) A simple class.

```
class TestClass{  
}
```

As you can see from the above code, the name of the class is `TestClass`. This class can be saved to a file with the name of your choice. In this case, it is saved in the file named `SimpleClass.java`. Compile the program as shown below:

```
C:/>JavaTraining>Chapter1>javac SimpleClass.java
```

The above command generates `TestClass.class` file since the Java source defined a class with the name `TestClass`. In Java, we always compile the '`java`' file and we execute the '`.class`' file to see the results. To execute the program, we need to pass the name of the classfile without the extension as shown below.

```
C:/>JavaTraining>Chapter1>java TestClass
```

The above command produces the following output.

```
Error: Class has no main method.
```

The reason JVM has resulted in the above error message is due to the fact that the class doesn't have anything defined in it. Therefore, the program is of no use. Let's look at next program.

Listing 2.3 (`BasicClass.java`) Java source with more than one class.

```
class BasicClass11  
{  
}  
class BasicClass12  
{  
}  
class BasicClass13  
{  
}
```

The above Java source file has three classes namely `BasicClass11`, `BasicClass12` and `BasicClass13` which are saved to the file named `BasicClass.java`. Once again all the classes are empty. Compile the program as shown below:

```
C:/>JavaTraining>Chapter1>javac BasicClass.java
```

Since we defined three classes, Java compiler generates three class files, one for each. The resulting three class files will be BasicClass11.class, BasicClass12.class and BasicClass13.class. Now the question is, which class should we execute? You can try executing all the three classes as shown below, and you know what, they all give you the same error “Main method not found”. This error makes sense, since all the classes are empty, right? We can’t blame the JVM. Ok.

```
C:/>JavaTraining>Chapter1>java BasicClass11  
Error: Main method not found  
C:/>JavaTraining>Chapter1>java BasicClass12  
Error: Main method not found  
C:/>JavaTraining>Chapter1>java BasicClass13  
Error: Main method not found
```

Now, our goal is to make all the above classes executable. Since JVM complained about the absence of **main** method, let’s add that method in all the classes and see what happens. Modify the above classes as shown in Listing 2.4.

Listing 2.4 (BasicClass.java) A simple class.

```
class BasicClass11  
{  
    public static void main (String args[])  
    {  
    }  
}  
class BasicClass12  
{  
    public static void main (String args[])  
    {  
    }  
}  
class BasicClass13  
{  
    public static void main (String args[])  
    {  
    }  
}
```

This time our program added a main method in each of the classes. If we compile and execute the above program like we did in the previous example, the error will disappear. So, here is the rule. To execute any class, the class **must** have a **main** method, and the main method must be exactly as shown below.

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

At this point of time, don't worry about the words **public**, **static**, **void** and all that. You'll know about these later when time comes. All we need to know is, to execute a Java class, the class must define the **main** method.

Let's do a small experiment with the above code. Add the keyword **public** to BasicClass11 class as shown below.

```
public class BasicClass11
```

Now save the program and recompile it again. To your surprise, you'll see the following compilation error.

```
C:/>JavaTraining>Chapter1>javac BasicClass.java  
Error: The file name must be the public class BasicClass11
```

The moment we added **public** keyword to BasicClass11 class, JVM treats it as **top level** class, and says, "*Hey buddy, if your code has a top level class, the file name must be the class name of the top-level class. Otherwise, I am going to throw an error*". So, the rule is, if a Java file has a class declared as public, then the file name must be same as the public class name. Therefore for the above program to compile, we should obey the JVM order and sincerely rename the file to BasicClass11.java. Save the file, and compile it again, and without any surprises the error disappears.

Let me now ask you a question here. Can more than one class in a single Java file be declared as public? Strictly No. You can't. The reason is simple. You cannot have two filenames for the same file. As simple as that. In essence, a single Java file can only have one public class, and if it is the case then Java source file name must be the name of the public class name.

Let me tell you one thing here. Though we executed the classes successfully without any errors, it's not a good practice to write more than one class in a single java file. Always write one class per Java file. So, from here onwards, I am going to write just **one public class in one Java file with or without the main method**, to keep things simple and clean.

Good Practice: A class name should always start with uppercase character, and every word in class name should also start with uppercase letter. Some example class names are shown below

HelloWorld, InitialContext, HttpServlet, EntityContext,
Connection etc..

The above class naming convention is universal standard in Java, so let's follow it. With the above good practices and naming conventions, our Java program would look like as shown in Listing 2.5.

Listing 2.5 (`Greeting.java`) A simple class.

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

We finally reached a point where, to compile and execute a Java program without any errors,

1. We should write a public class in the Java file, and
2. Write a **main** method in the class. That's it. Very good.

Main method of class

This is the standard method that the JVM looks in a class to start executing the statements. So, the main method is the starting point of execution of any Java program like a start block in a flowchart. So, let's start writing real productive programs.

Listing 2.6 (`Message.java`) A simple class that echoes a greeting.

```
public class Message {  
    public static void main ( String args[] ) {  
        for (int i=1; i<=5; i++) {  
            System.out.println ( "Keep things Simple");  
        }  
    }  
}
```

Listing 2.6 is a simple program that uses `for` loop that repeats five times. So the message gets printed five times in the console. See, learning the basic programming elements upfront makes it easy to write and understand them pretty fast. We don't have to scratch our minds as to what a `for` loop is, what a `System.out` statement is and how they work. Compile and execute the program as shown below.

```
C:/>JavaTraining>Chapter1>javac Message.java  
C:/>JavaTraining>Chapter1>java Message
```

The output of the program is

```
Keep things Simple  
Keep things Simple  
Keep things Simple  
Keep things Simple  
Keep things Simple
```

Listing 2.7 (SumSquare.java) A simple class that computes the sum of squares.

```
public class SumSquare {  
  
    public static void main ( String args[] ) {  
  
        long sum = 0;  
  
        for (int i=1; i<=30; i++) {  
  
            sum = sum + i * i;  
  
        }  
        System.out.println("The sum of squares of first 30 numbers is " + sum );  
    }  
}
```

The above program again uses *for* loop and computes the sum of squares of certain numbers. When the value of variable *i* becomes 31, the loop terminates, and we print the value in the ‘sum’ variable. Copy the above program to the following directory to compile and execute the program.

```
C:/>JavaTraining>Chapter1>javac SumSquare.java  
C:/>JavaTraining>Chapter1>java SumSquare
```

The result of the above program is

```
The sum of squares of first 30 numbers is 9455
```

I know the above two programs are not so interesting. So, let’s write some modular programs using methods. From here onwards, when I ask you to compile and execute a program, I expect you to

- a) Copy the file into the “chapter1” directory and
- b) Execute *javac* and *java* commands like you did until now.

In all the previous examples, our class just had the main method with all the statements. However this is usually not the case. As I said before, a class can have any number of methods besides the main method. So, let's see how to define a method in a class.

Listing 2.8 (`TaxPayer.java`) A simple class that uses methods.

```
public class TaxPayer {  
  
    public static void main(String args[]) {  
  
        System.out.println("I am in main method");  
  
        // Call methods from here  
        payTaxes();  
        sayHi();  
    }  
  
    static void payTaxes() {  
        System.out.println("Thanks for Paying taxes");  
    }  
  
    static void sayHi() {  
        System.out.println("I am in sayHi method");  
    }  
}
```

The example in Listing 2.8 is more interesting and spicy. This example introduces you to methods in a class. A class can have one or more methods which define the behavior of the class. All these methods define statements to compute the results. Moreover, methods can call each other to generate results. A method is like a reusable module with in a class that can be invoked any number of times. Following is how the program works.

1. Program execution begins in the main method
2. Main method prints some message, and then calls the method `payTaxes()` method.
3. `payTaxes()` method prints another message and returns back to main.
4. Main then calls `sayHi()` method, which prints a message and returns back.
5. Since there are no more statements to execute, main method terminates.

Any doubts? No, right? Compile and execute the program only to see the following result.

```
I am in main method  
Thanks for Paying taxes  
I am in sayHi method
```

Before we start writing further examples, let's learn some details about methods and their intricacies.

Class Methods

The syntax for writing a method in a class is,

```
<one or more keywords> <return type> methodName (argument list) {  
}
```

One of the keywords we need to use at least for now is **static**. Why? Good question. In Java, there are only two fundamental entities called *class* and *object*. Right now we are only dealing with the first entity, which is the class. Both class and object can have methods as well as variables. Therefore, to distinguish class level methods, variables from object level methods, variables we use the keyword **static**. If the method or a variable is declared as **static**, then they will be treated as class level properties, otherwise, they will be treated as object level properties. Make sense? Since we are talking about class methods and not object level methods, we use **static** in every method declaration. Don't even think about objects for now. We'll see them later. Ok.

The return type of a method denotes the type of data being returned back to whoever called the method. If the method doesn't return any data back, we need set the return type as **void** as shown below

```
static void sayHello() {  
}
```

If the method does return some data back, it needs to do two things.

1. Need to specify the type of the data being returned as the return type,
2. Return the data using the keyword **return**.

For instance, if a method returns a number of type **int**, following is how the method should look like:

```
static int getAmount(){  
  
    int amount = 1000;  
    return amount;  
}
```

It is very important that the data returned match with the declared return type. Otherwise, it results in a compilation error.

Passing Arguments to Methods

Method arguments are nothing but a set of variables a method uses to take the data for it to process and return results. These variables are declared within the parenthesis in the method declaration. If the method doesn't take any data, the 'argument list' will be empty within the parenthesis as shown in the above two examples. However, if the method does take arguments, then we need to specify the type of data that it takes by declaring the variable types. As an example, if the method takes an integer and a double argument, the method declaration would look like as shown below:

```
public void setData(int a, double b) {  
}
```

When the invoker of this method sends two numbers, the numbers will be stored in variables a and b, which can then be used within the method body for processing. This is all about methods you need to know at this point.

Comments

Using comments in program is very good practice as it improves the readability of the program. In Java, we can write single line comments and multi line comments as shown below:

```
// This is a single line comment  
// This is another comment  
/*  
   The following code convert a given string into a number and inserts  
   into the database.  
*/
```

The examples we are going to write from here inwards are more interesting. If you notice, in all the examples thus far we just used one class to define all the methods and invoked them from the **main** method. We will get away from this approach and start writing code in multiple classes and have them call each other. This is how we need to write programs. Instead of writing all the functionality in one class, try logically distributing the code into multiple classes, so that we can reuse the classes. This is one of the best practices.

Let's implement a search functionality that allows searching for a resource on Internet and directory.

Since we need to search for resources in two different places, let's write two classes namely `InternetSearchEngine.java` and `DirectorySearchEngine.java` and define a `search()` method in each.

Listing 2.8a (`InternetSearchEngine.java`) Class to search on internet.

```
public class InternetSearchEngine {  
    static void search() {  
        System.out.println("Found 20 results on Web");  
    }  
}
```

Listing 2.8b (`DirectorySearchEngine.java`) Class to search on directory.

```
public class DirectorySearchEngine {  
    static void search() {  
        System.out.println("Found 10 results on Novel Directory");  
    }  
}
```

Look at the above two search classes in Listing 2.8 a & b. If you noticed, both the search engine classes do not have a `main` method defined. So you can compile them, but cannot execute them. These are like utility classes that some other class must use them; otherwise their existence doesn't make any sense, right? Therefore, we will write another class named `PremierSearch`, which will use the above classes to search for resources. Listing 2.8c has the implementation for this class.

Listing 2.8c (`PremierSearch.java`) Class to search on internet and directory.

```
public class PremierSearch {  
    public static void main(String args[]) {  
        // Local Search  
        search();  
  
        // Other searches  
  
        InternetSearchEngine.search();  
        DirectorySearchEngine.search();  
  
        System.out.println(" Search Completed ");  
    }  
}
```

```
    }

    static void search() {
        System.out.println("Found 5 results in Local");
    }
}
```

The PremierSearch class has its own `search()` method for local search. It calls this method by just specifying the method name like we did in previous examples. However, to call the `search()` methods in the two search engine classes, it needs to prefix the class name with a dot before the method name. The rule is, for a class to invoke static methods in another class, it must use the following syntax:

`ClassName.methodName();`

If I say, the color of the shirt is red, it probably refers to my shirt. But if I wanted to say the color of other person shirt, I must say "The color of **Tom's** shirt is red". See, I have to use his name. Java is closely related with how we talk every day.

Above is exactly what we did in `PremierSearch` class. We used the class name to invoke the methods in different classes. Writing functionality in multiple classes allows loose coupling and at the same time code reusability. Tomorrow, if you want to add new search functionality, you can write a new class with a search method, and then call it from where ever you want. Isn't this nice? I am sure you are with me.

Note: You need to compile all three classes using `javac` command and then execute the `PremierSearch` class to see the results. Compile and execute the classes as shown below:

```
C:/>JavaTraining>Chapter1>javac InternetSearchEngine.java
C:/>JavaTraining>Chapter1>javac DirectorySearchEngine.java
C:/>JavaTraining>Chapter1>javac PremierSearch.java
C:/>JavaTraining>Chapter1>java PremierSearch
```

The result of the above program will be,

```
Found 5 results in Local
Found 20 results on Web
Found 10 results in Novel Directory
Search Completed
```

Trick: I know it's tedious to compile each and every program using the `javac` command shown above. To get around with this, use the following simplified command to compile all java files in one step (using *).

```
C:/>JavaTraining>Chapter1>javac *.java
```

The above command compiles all the Java files in the directory in one go. You can then execute whichever class you want using the *java* command. Sweet! I like tricks. In fact, who doesn't?

In the next example, we'll design two classes using give and take model. We'll have one class takes the details, and have the other class use the details supplied and return the essentials. The code is shown in listing 2.9.

Listing 2.9a (*LoanCalculator.java*) Class taking the details and returning the essentials.

```
public class LoanCalculator {  
  
    static int creditScore;  
    static double annualIncome;  
    static double approvedAmount;  
  
    static void setCreditScore(int score) {  
        // Store the credit score  
        creditScore = score;  
    }  
  
    static void setAnnualIncome(double income) {  
        // Store the annual Income  
        annualIncome = income;  
    }  
    static double getApprovedAmount() {  
  
        // Apply the conditions and determine the approved amount  
        if (creditScore < 600 && annualIncome > 60000) {  
  
            approvedAmount = 500;  
        } else if (creditScore > 600 && annualIncome < 70000) {  
  
            approvedAmount = 1000;  
        } else {  
  
            approvedAmount = 0.0;  
        }  
        return approvedAmount;  
    }  
}
```

The `LoanCalculator` class has defined three variables and three methods. Whenever we define variables in a class, they can be accessed or shared by all the methods of that class. We call them as global variables of the class. Another important thing is that the class level global variables must be declared **static**. This implies that, static methods can access static global variables only. Note this point.

The `setCreditScore()` method takes the credit score and stores it in `creditScore` variable. Similarly the other set methods. Finally, the `getApprovedAmount()` method reads the data from the populated global variables, and returns the approved amount based on some conditions. Also note that this method has a `return` statement at the end to return the approved amount which we declared as `double`. Therefore, in its method declaration, the return type is set to `double` as shown below:

```
static double getApprovedAmount()
```

Listing 2.9b (`LoanCalculatorTest.java`) Test class that passes the details and take the essentials.

```
public class LoanCalculatorTest {  
    public static void main(String args[]) {  
        // Pass the details  
        LoanCalculator.setCreditScore(650);  
        LoanCalculator.setAnnualIncome(55000.00);  
        // Get the results  
        double amount = LoanCalculator.getApprovedAmount();  
        System.out.println("Approved Amount = $" + amount);  
    }  
}
```

Now look at the `LoanCalculatorTest` class. It simply invokes the appropriate methods on the calculator class by giving the details and taking the essentials, which is the approved amount. Compile the two classes, and execute the test class to see the following results.

```
Approved Amount = $1000.0
```

This is how we need to define variables and methods in Java program. Let's look at next concept.

Static Blocks

So far we used **static** keyword in a class with methods and variables only. Is there another usage of **static** that we missed? Yes there is, and the most important application of static, a *static block*. A static block has the following properties.

- ✓ It has no name
- ✓ It gets invoked before the main method is invoked
- ✓ JVM will only invoke it once
- ✓ There can be more than one static blocks in a class, and the JVM will execute all of them one by one. However, it's not a good practice to use multiple static blocks.

A static block looks as shown below:

```
static {  
    // Initialization Statements  
}
```

The purpose of static block in a class is for class level initialization. More often than not, we require JVM to perform some initialization before it starts executing the main method. For instance, before we use something like a database, it's very important to initialize some of its internal components. Initialization is almost a must for any software application. This is where static blocks come in handy. They allow Java programs to execute certain initialization statements before the actual execution begins. Listing 2.10 demonstrates the usage of static block.

Listing 2.10 (`StaticBlockDemo.java`) Class demonstrating the usage of static block.

```
public class StaticBlockDemo {  
  
    static {  
  
        System.out.println("40% Class is Initialized");  
    }  
    static {  
  
        System.out.println("Remaining 60% Class is also Initialized");  
    }  
  
    public static void main(String args[]) {  
  
        System.out.println("I am in Main");  
    }  
}
```

When the JVM executes the above class, it first invokes all the static blocks one by one before it begins executing the main method. Compile and execute the program, and the above mentioned properties of static block will make sense. The result of the program will be,

```
40% Class is Initialized  
Remaining 60% Class is also Initialized  
I am in Main
```

Use static blocks whenever you want to execute certain statements prior to the execution of main method. The above example is pretty simple and doesn't really demonstrate the beauty of static blocks. So, let's look at a classical database example *with and without* a static block, and you'll notice its importance.

In this example, let's implement a database functionality where the database returns two connections to the requestor.

Challenges

1. We cannot get a connection without initializing the database. At the same time we must not initialize the database every time we get a connection.
2. Only Database should initialize itself. We should not call the initialize method from outside database class.

We will see three cases on this example from bad to good to great. The program is very simple. We will simply write a class named `Database` with two methods namely `initialize()` and `getConnection()` methods. We will then write a test class that only invokes the `getConnection()` method twice to retrieve two connections. The test class is not allowed to invoke the `initialize()` method as it is internal to database. Initialization must be taken care by the database class itself.

For all the cases, the test class remains the same, but we will enhance the `Database` class to come up with a clean and better solution.

Case 1: See listing 2.11a for the version of database class.

Listing 2.11a (`Database.java`) Version 1 implementation of the database class

```
public class Database {  
  
    static void initialize() {  
        // Do the initialization here  
        System.out.println("Database Initialized");  
    }  
  
    public Connection getConnection() {  
        // Implementation of getConnection()  
    }  
}
```

```
    }

    static void getConnection() {
        // Calls the initialization method
        initialize();
        System.out.println("Connection established");
    }
}
```

This class is not smart, but is safe. It bluntly invokes the `initialize()` method every time the `getConnection()` method is invoked, ensuring that the database is always initialized before returning the connection. This implementation is not at all good because it is misusing the resources. Listing 2.11b shows the test class that tries to get the two connections.

Listing 2.11b (`DatabaseTest.java`) Test class for retrieving the connections.

```
public class DatabaseTest {

    public static void main(String args[]) {
        // Get two connections.
        Database.getConnection();
        Database.getConnection();
    }
}
```

If you look at the above test class, it invokes the `getConnection()` method twice to get two connections from the database to meet the requirement. This is exactly what we want to do. Compile the above two classes and execute the test class to see the following result.

```
Database Initialized
Connection established
Database Initialized
Connection established
```

As you can see from the above code, to get two connections successfully, this version of database class initialized itself twice. Usually, database initialization is a costly operation and consumes significant amount of resources. Therefore, it must be done as few times as possible. This is therefore a very bad implementation. Let's see if we can modify the database class to cut down the number of initializations.

Case 2:

In this case, we will tweak the above database class to ensure that it is initialized just once. Look at the following enhanced version of the database class in listing 2.11c.

Listing 2.11c (`Database.java`) Version 2 implementation of the database class

```
public class Database {  
  
    static int val = 0;  
  
    static void initialize() {  
  
        System.out.println("Database Initialized");  
        val = 1;  
    }  
  
    static void getConnection() {  
  
        if (val == 0) {  
  
            initialize();  
        }  
        System.out.println("Connection established");  
  
    }  
}
```

Look at the above enhanced `Database` class. In this case, the database class is good. To ensure that the database is initialized only once, it introduced some additional logic around the call to `initialize()` method in the `getConnection()` method. The way the logic works is, when the test class invokes the connection method for the first time, the variable `val` will be 0, and the control goes inside the `if` block and invokes the `initialize()` method. After successful initialization within the method, it updates the variable `val` to 1. For the second call to the method from the test class, the condition fails and will not get initialized. Smart logic, right? With this updated class, if you execute the same test class, the output will be what we wanted it to be as shown below:

```
Database Initialized  
Connection established  
Connection established.
```

Case 3:

In the previous example, we didn't change anything in the test class. All we did is tweaked the database class to get the right output. At this point, you might be wondering that we were able to solve the problem in second case itself, so why do we need a third case. What I don't like in the second case is the introduction of additional

logic to meet the requirement. Always remember one thing; every additional line code is a potential bug. So try to make the program as simple as possible with fewer statements. And also remember the fact that, when the program is getting complicated, you are always going in the wrong direction. My third case is smart and classical. It permanently removes the `initialize()` method in the database class, and moves the initialization statements into a **static** block as shown in listing 2.11d.

Listing 2.11d (`Database.java`) Version 3 implementation of the database class

```
public class Database {  
    static {  
        System.out.println("Database Initialized");  
    }  
    static void getConnection() {  
        System.out.println("Connection established");  
    }  
}
```

Use the above database class and compile the program. Execute the test class again, and you'll be surprised to see the same result as Case 2. But notice how simple the database class is. No variables, no conditions and no initialization method. Yet, the results are same. This is the best solution we can get, right? Simple and classical. Following is how it works.

As we know that the static block gets invoked only once, we simply put all the initialization statements in this block. By doing this, JVM ensures that it calls this block only once. When the control comes for the first time to this class, i.e., when the `getConnection()` method is called for the first time from the test class, JVM invokes the static block, and then invokes the connection method. From the second call onwards to the connection method, JVM bypasses the static block as it already executed before, and directly invokes the method.

Thumb Rule: Put all the class level initialization statements in the static block.

This is all what we need to know about classes. We have seen several examples and I am sure we learned a lot in no time. If some things are confusing, go through them again, and you'll be fine.

Summary

- ✓ A class is the building block of Java program.
- ✓ A class will have methods and variables.
- ✓ A class can have Class-level variables, methods and Object level variables and methods.
- ✓ Class level methods and variables must be declared as **static**.
- ✓ Static methods can only access static global variables.
- ✓ To execute a class, it must have a **main** method.
- ✓ Several classes can interact with other. For one class to invoke static methods in different class, it must use *ClassName.methodName* syntax.
- ✓ A class uses static blocks for initialization purposes. A class can have any number of static blocks, but it's a good practice to use only one.
- ✓ JVM invokes the static blocks before invoking the main method.

Time to play 50-50

1. Which of the following is a valid main method?
 - a) public void static main(String str[])
 - b) public static void main(String str[])
2. Which of the following is a valid class level method
 - a) double getAmount()
 - b) static double getAmount()
3. Given a class DriverManager with a method getConnection, which of the following is valid way of invoking the method
 - a) DriverManager.getConnection
 - b) DriverManager.getConnection();
4. What is the output of the following program

```
public class InitialContext {  
    static int i;  
    static{
```

```
i = 10;  
System.out.println( " Context Initialized ");  
}  
  
public static void main( String args[]){  
    System.out.println (i);  
}  
}
```

- a) Context Initialized
10
 - b) Context Initialized
0
5. Which of the following are valid assignments
- a) byte b = 5 * 5;
 - b) int k = 5 * 5;

Interview Questions

Question: List the properties of a static block.

Answer:

- ✓ It has no name
- ✓ It gets invoked before the main method is invoked
- ✓ JVM will only invoke it once
- ✓ There can be more than one static blocks in a class, and the JVM will execute all of them one by one. However, it's not a good practice to use multiple static blocks.

Congratulations! You have successfully completed Chapter 1. Let's move forward.

Chapter 3

Packages

By the end of this chapter you will understand the notion of packages in Java. This chapter demonstrates the usage of various keywords associated with packages and how to implement access control for classes within packages.

Chapter Goals

- ✓ Understand the notion of packages
- ✓ Understand the keywords associated with packages

Environment Setup

All the programs in this chapter should be stored in the following directory.

C:/JavaTraining/Chapter3

From the command prompt, move to the above directory.

Introduction

This chapter introduces you to the notion of packages in Java. A typical Java application comprises of several Java classes and organizing the classes in a systematic fashion is very important from maintenance standpoint. Consider for instance we are developing a chatting application. Such application contains several classes that can be categorized as UI related classes, server related classes, FTP classes, utility classes and many more. Storing all these classes in just one directory poses several maintenance problems. It would be much better if we store them in separate directories. This is very much like organizing files on your computer system. This is what Java packages is all about. A package is nothing but a directory in which we store Java classes. The good thing with using packages is that besides organizing classes, we can also provide some level of security to the classes which comes automatically.

Java packages is really not a concept, but we can think of it as a good practice. You really don't have to concentrate seriously to understand this chapter. It hardly takes couple minutes to understand what packages are and how to use them. I am going to keep this chapter really short by giving you the important details about packages.

Without wasting any time let's quickly get though all the important things we need to know about packages.

What is a Package?

A package is a collection of classes that allows organizing the code effectively and efficiently without any namespace collisions.

Creating and Declaring Packages

Creating a package is nothing but creating a directory on hard drive. Once we have this directory (from here onwards I am going to refer this as package) created, we can store all the classes in this package. However, there is one statement we need to add at the beginning of source code which tells the JVM that the class indeed belongs to that package.

Let's create a package named `src` in our `JavaTraining/Chapter3` directory as shown below:

`C:/JavaTraining/Chapter3/src`

If we wish to store a class in the above package, the code will look as shown in listing 3.1.

Listing 3.1 (HelloWorld.java) Class using packages

```
// Package statement
package src;

public class HelloWorld{

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

As you can see from the code, the first line is the package statements that tells the JVM that this class strictly belongs to the package `src`. The keyword we use for package declaration is **package**. The syntax for declaring the package is,

`package <packagename>;`

There are two simple rules to keep in mind though.

1. A class can only have ONE package statement
2. The package statement must be the first statement in the program.

You can store any number of classes in a single package. Moreover, like we create directories within directories we can also have packages within packages. For instance, we can create another package named `examples` within `src` package. The classes within `examples` package will then use the following package statement:

`package src.examples;`

Surprised? May be you thought the package statement should instead be

`package examples;`

There is nothing wrong in your thinking. However, it doesn't work this way since the package names are *relative* names. Before we start using the packages, we first need to fix the *root* directory. All the packages we create must be created within the root directory. So, what is a root directory? A root directory is nothing by the directory of your project. Let's say we are developing a financial application. So, fundamentally, we need to

choose a directory for storing all our work. Let's say we have created the following directory for this application.

C:/RetailBanking

The above directory will now be the root directory for all the components within the application. All the packages we create will be created from this directory onwards for storing the class files. Once created, the package names should be with respect to the root directory. Consider the following package structure for organizing our classes.

```
RetailBanking
  core
    Calculator.java
  logging
    SystemLog.java
    FileLogger.java
  utils
    DateFormatter.java
    StringUtils.java
```

Based the above package structure, the package names for various classes will be as shown in table 3.1.

Table 3.1 Class and Package names

Class Name	Package Statement
Calculator.java	package core;
SystemLog.java FileLogger.java	package core.logging;
DateFormatter.java StringUtils.java	package core.utils;

I am sure the above table made things clear. As you can see, all the package names are now with respect to the root directory "RetailBanking".

Namespace collisions

More often than not, you might want to have one or more Java files with different functionality but share the same name. This would be impossible without packages. The reason is simple. You cannot have two files with the same name in the same directory. For instance, I can have one date class for processing general purpose dates, and other date class for processing database related dates. Rather than using two class names like

RegularDate and DatabaseDate, I can use same name for both the files but should store them in two different packages as shown below:

```
core  
    Date.java  
  
database  
    Date.java
```

The first one uses the package statement with the name core and the second one with the name database.

Fully Qualified Name of a Class

A fully qualified name of the class is the class name along with the package name. For instance in the above example, core.Date and database.Date are the fully qualified names of the two date classes. The fully qualified name will always be unique.

Package Naming Conventions

Although you can have any arbitrary names for package names, we always follow standard naming conventions. A typical Java application uses several third party classes from different companies. For instance, we can use the logging classes from company say, XLOG, some other classes from some other company. All these companies follow a standard naming convention as shown below:

<type of company>. <companyName>. <module name>

The type of company is normally commercial or organization in which cases we use either com or org. Following are some of package names using the above convention.

```
com.xlog.logging  
org.abc.web  
com.zee.xml
```

Using the above conventions, if you own a company named “eTools Inc”, which sells commercial tools, then your application packaging structure can be,

```
com.etoools.calculators  
com.etoools.calculators.general  
and so on....
```

Java Core Libraries

Java comes with a rich set of library classes and all these classes are very cleanly organized in the form of packages. Our Java programs simply need to import these library classes by specifying the package names. As an example, following are the some of the important packages in the Java core library:

```
java.lang  
java.util
```

All the above packages will have several classes. To use the classes stored in above packages, we need to import them using the **import** keyword as shown below:

```
import java.lang.*;
```

The asterisk (*) at the end indicates *all the classes* within the package. However, if you just want to import one specific class in a particular package, you can do so as,

```
import java.io.PrintWriter;
```

Unlike a package statement, a Java program can have any number of import statements as shown below:

```
import java.io.*;  
import java.util.* ;
```

Compiling and executing Programs

There is a standard way of compiling and executing a class that uses packages. To better understand, let's consider the code in listing 3.2.

Listing 3.2 (Calculator.java) Class using packages

```
package examples;  
  
public class Calculator {  
    public static void main(String args[]) {  
  
        int sum = 0;  
        for (int i = 0; i <= 100; i++) {  
            sum = sum + i;  
        }  
        System.out.println(sum);  
    }  
}
```

When ever we use packages, the programs should always be compiled from the root directory only. The above program should be compiled as shown below:

```
C:/JavaTraining/chapter3> javac examples\Calculator.java
```

In the above command, “chapter3” is the root directory. The *javac* command should specify the package name and the class name separated by a “\”. If we want to compile all the classes in the same package, we can use the following syntax

```
C:/JavaTraining/chapter3> javac examples\*.java
```

Once all the programs are compiled, we then have to execute the main class using the following command:

```
C:/JavaTraining/chapter3> java examples.Calculator
```

Note that while executing the program, we need to use dot (.) instead of slash (\) and without any extension. This is how you need to compile and execute the classes using packages.

Now that you know how to create, use and import packages, it’s time to understand the most important aspect of packages which is access control.

Access Control using Packages

So far we’ve seen how to define a class using packages. One of the important features of packages is that when used in programs, JVM will provide some level of security to classes within packages. Though this level of security doesn’t protect the code from hackers, it is very useful from application point of view. We call this as **access control**. Let’s see what this is all about.

Java provides three keywords namely **public**, **protected** and **private** which we call them as access modifiers. Using these three keywords and together with packages, classes and methods can be implemented with restricted access.

A Java application is all about invoking the methods in classes that are distributed in several packages. Based on this, following are the important questions we can ask ourselves.

1. Can a class A in one package access a class B in different package?

2. Even if the answer is “yes” for the above question, can a class A be able to access class B methods?

The answers to both the questions are Yes and No. The answer is “yes” if we use **public** keyword and “no” if we use **private** keyword. Access modifiers are used with classes, methods and variables. Table 3.2 lists which access modifiers are applicable to which entities.

Table 3.1 Class and Package names

	public	protected	private	default
class	Yes	No	No	Yes
method	Yes	Yes	Yes	Yes
variable	Yes	Yes	Yes	Yes

As you can see from the above table, all the access modifier keywords can be used with classes, methods and variables except that we don't use **private** and **protected** with a class. Also, it is important to note that “default” means no keyword. Following are some of the examples:

```
public class Test
class Test
public void sayHello();    ( Public Method)
protected void sayHello();  ( Protected Method)
private void sayHello();   ( Private Method)
void sayHello();          ( Default Method)
private int i=10;
protected int k=20;
public int k = 20;
```

Public Access

This keyword is used with classes, methods and variables and is the most generous one. If used with a class, *will allow classes in the same package or in any other package* access it without any restriction. One good example is the utility classes which should always be declared as public since they are normally shared by all the remaining classes within the application irrespective of package they are present in. The same is applicable to variables and methods if they are declared public. Any class will be able to access them.

Protected Access

This keyword is used with methods and variables. The scope of this keyword is almost close to public keyword with just one small access restriction. If a method or variable is

declared protected, *only classes within the same package and “child” classes within different package* can access them. Don’t worry about what child classes are for now. We’ll see them later.

Default Access

If none of the access modifiers are used with a class, method or variable, then we say that they all have default access. This means *only the classes in the same package can access them.*

Private Access

This keyword is only used with methods and variables. When used, they are *only limited to the class in which they are present* even if the class is public. A class typically has a combination of public and private methods. The private methods as the name suggests are private to that class, and cannot be shared with other classes. This is the lowest level of access we can provide to any entity.

From the above discussion, the order of access from highest to lowest is

public ->protected->default->private

From the next chapters onwards we will start using these access modifier keywords. Don’t worry about them for now. I will explain them in detail when we start using them. This is all you need to know about access modifiers and packages. Using proper access modifiers and packages is only a good practice but not a “must”. Since we are aspiring to become a professional developer, we always follow good practices. With this brief overview about packages, let’s move on with the rest of the chapters.

The next chapter is the most and most important one, as it introduces you to the second type of entity in Java, an Object. You know what, if you noticed carefully, all the programs are based on ‘Structured and Modular Programming models’. Though Java’s strength is ‘Object Oriented Programming’ which we will see in the next chapter, it still supports the old programming models. Sweet!

Summary

- ✓ Packages are used for organizing the classes in different directories.
- ✓ Packages eliminate namespace collisions.
- ✓ A class must use the **package** keyword to specify the package a class belongs to.

- ✓ A class can only have one package statement and it must be the first statement in the code.
- ✓ All the library classes are organized in several packages. For a class to use such library classes, it needs to import them using the **import** statement.
- ✓ Unlike a package statement, a class can have any number of import statements.
- ✓ Using packages provides access control to classes.
- ✓ The keywords that support in access control are **public**, **protected** and **private**
- ✓ The order of access modifiers from highest to lowest is public, protected, default and private.

Time to play 50-50

1. Which is the following keyword is used to specify a package.
 - a) Package
 - b) package
2. A class can have more than one package statement. True/False
3. A class can have more than one import statements. True/False
4. Which of the following statement imports a class named `Hashtable` in the `java.util` package
 - a) `package java.util.Hashtable;`
 - b) `import java.util.Hashtable;`
5. Which of the following access modifier should be used to allow access in the same package
 - a) `public`
 - b) `default`
6. Which of the following access modifier is used with a method to prevent external classes from accessing it?
 - a) `protected`
 - b) `private`

Interview Questions

Question: What are the different access modifiers in Java?

Answer: public, protected and private

Question: Explain the differences between public, protected and private keywords

Answer:

When a class or a method or a variable is declared public, they *can be accessed by any class in any package*.

When a method or a variable is declared protected, they *can be accessed by all the classes in the same package and child classes in different packages*. Usually a class will not be declared as protected.

Private keyword is used only with methods and variables. These *can only be accessed only within the class that contains them*. This is the smallest level of access.

When none of the keywords are used with a class, or a method or a variable, it is given a default access. This means they *can only be accessed by classes in the same package*.

Congratulations! You've successfully completed this chapter and I am sure you are ready to move to the next chapter. Let's move on.

Chapter 4

Object Oriented Programming

By the end of this chapter, you will know how to build Object Oriented Java applications. This chapter will demonstrate Object Oriented Programming (OOP) concepts and their implementation in Java. You'll surely enjoy this chapter since OOP is fun and its associated terminology is heard in our day to day life.

Chapter Goals

- ✓ Understand the difference between Class and Object
- ✓ Learn how to create Objects from Classes
- ✓ Learn to define interaction between objects
- ✓ Understand Object Oriented concepts like Encapsulation, Inheritance etc.
- ✓ Understand the notion of Constructors and some important keywords
- ✓ Understand Abstract classes and Interfaces.

Environment Setup

All the programs in this chapter should be stored in the following directory.

```
C:/JavaTraining/Chapter4/objectconcepts
```

objectconcepts is a package in which we will store the programs. To compile the programs, move to the following root directory and get ready.

```
C:/JavaTraining/Chapter4
```

Introduction

Object Oriented Programming (OOP) as the name suggests, is nothing but *programming using Objects*. OOP model is a very powerful programming model as it allows us to build reusable, modular, loosely coupled and flexible applications unlike applications built using other programming models. The central idea or entity on which OOP model is based is something called *object*. So, without wasting any further time, let's see what it is before doing some programming using objects.

What is an Object?

There are several ways to define an object like the ones below.

- An object is an instance of a Class
- An object is a runtime entity of a Class.
- An object is a memory model of a Class.

We like keeping things simple, so our definition of an object is,

“An object is an instance of Class”.

Difference between a Class and an Object

In simple terms, an *object is a carbon copy of a class*. Let's say you want to build a car. You just can't go ahead and build it right away. You first need to design a car, and then build the car. So, a car is built from a car design, a computer is built using computer design. Likewise, an *object* is built from an *object design* and object design is what we call as *class*. So, to create an object, we need a class. Point to be noted.

Once we have a car design, we can build 'n' number of different cars from it, and every car is an independent entity. It has its own wheels, seats, doors etc. Similarly, if we have a class, we can create 'n' number of different objects belonging to that class and again every object is an independent entity. Another point to be noted.

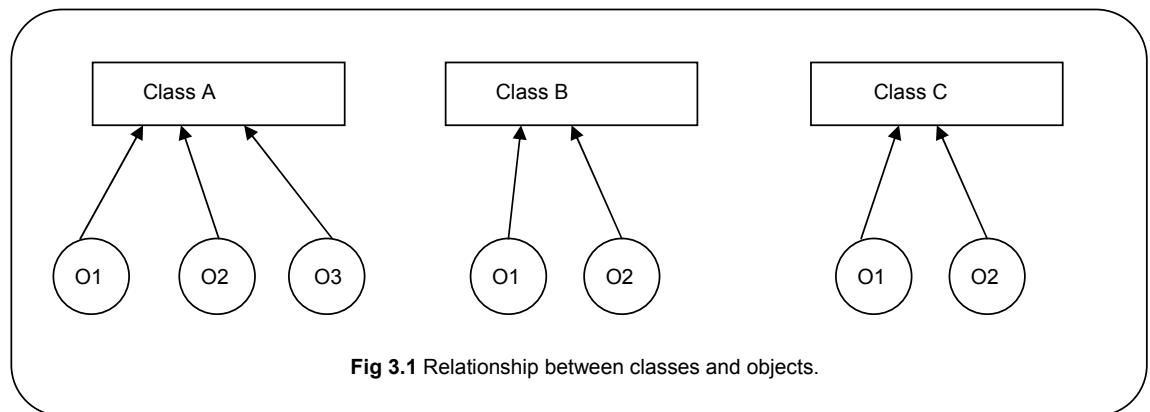
Writing a program that creates objects from a class, and defining the interactions between the objects is what we call as *object oriented program* and such a programming model is referred to as *Object Oriented Programming model* (OOP).

So, in OOP methodology, we write object oriented programs and we write them in Java from here onwards. Cool.

Two important points to remember

1. For every class, we can create 'n' number of objects belonging to the **same class only**.
2. There is no '*object*' without a '*class*'.

Fig 3.1 shows the relationships between classes and objects



Therefore, to write an Object Oriented Java program, we do the following three things.

1. Write a class like we did before,
2. Create objects of that class and
3. Define the interaction between the created objects.

If you recall from chapter 2, we learned something about class level methods and variables and object level methods and variables. Now it's time to clearly understand what this means. Let's say we have a class with two variables and two methods. Now, if we create an object from this class, the question is, does the object also have the same two variables and methods? Hmmm. Tough question, right? The answer is both Yes and No.

Yes, if the variables and methods in the class **are not declared as static**.
No, if the variables and methods in the class **are declared as static**.

Let's stop this theory crap, and see with an example to make our life easy. Like you, I do hate theory, but you know what, sometimes we can't escape from it. But, let me tell you one thing, I'll try my best to cut short the theory and explain things with examples.

Listing 4.1 (`Printer.java`) Class with object level and class level properties.

```
package objectconcepts;

public class Printer {

    static int i = 10;
    int j = 20;

    static void display() {
        System.out.println("The value of i is " + i);
    }

    void print() {
        System.out.println("The value of j is " + j);
    }
}
```

The code shown in listing 4.1 has two variables namely `i`, `j` and two methods namely `display()` and `print()`. Now, when we create objects of this class, the question is, which variables and methods the objects will have?

To understand, here are two very simple rules.

1. Objects will only get **non-static** variables and methods defined in a class So, every object created from this class will only have variable `j` and method `print()`. These are called object level properties.
2. All the static methods and variables namely `i` and `display()` will remain in the class only. These are class level fields and can be shared by all the objects created from the class. This is the reason why we call class level properties (variables and methods) as shared properties.

In Java, all a class will have is static and non-static methods and variables. If someone gives you a class or if you need to use some library class, the first thing you need to do is identify which are class level fields and which are object level fields. If you see the keyword **static** for a method or variable, mark them as class level, and if you don't see **static**, mark them as object level fields. If you could do this, then half the job is done. Trust me. To access class level methods, we use `ClassName.methodname` syntax, and to access variables we use `ClassName.variablename` syntax which we already discussed in chapter 2. I am sure you didn't forget these.

From here onwards in this chapter, we'll see how to access object level properties.

Note: For all the examples in this chapter, we will write at least two classes. One is the class whose objects will be created, the other is a test class that will create the objects and invoke the methods on these objects. For instance, two classes namely `Car` and `CarTest`. The test class is the one that will have the `main` method and it is in this method, we will start creating car objects. To execute the program, we simply have to execute the test class. No confusions, right?

Objects and Methods

Object oriented programming is all about creating objects and invoking methods on objects. So, let's begin our journey into object oriented programming by writing a class that has both class level as well as object level properties. See listing 4.2.

Listing 4.1 (`Printer.java`) Class with object level and class level properties.

```
package objectconcepts;

public class Printer {

    static int paperCount = 1;

    static void print() {

        if (paperCount > 0) {

            paperCount = paperCount - 1;
            System.out.println("Document printed");

        } else {

            System.out.println("Please load the paper in the tray");
        }
    }

    void printDocument() {

        print();
    }
}
```

As you can see from the code, the `Printer` class has one object level method namely `printDocument()` method (non-static). Objects created from this class will only have this method.

Coming to the class level properties, it has two of them namely, `paperCount` variable and `print()` method (static ones). These two properties can be shared by all the objects

created from this class by directly invoking them from the object level methods as shown below:

```
void printDocument () {  
    print(d);  
}
```

Tip: For the class, visualize a rectangle that has a `print()` method and `paperCount` variable.

Now, look at the test class in listing 4.2.

Listing 4.2 (`PrinterTest.java`) Test class to create printer objects.

```
package objectconcepts;  
  
public class PrinterTest {  
  
    public static void main(String args[]) {  
  
        // Create 2 objects  
        Printer p1 = new Printer();  
        Printer p2 = new Printer();  
  
        // Invoke the object level method  
        p1.printDocument();  
        p2.printDocument();  
    }  
}
```

As you can see from the above test class, it first created two printer objects. The syntax for creating an object from a class is shown below:

```
<class name> <objectReference> = new <class name> ();
```

In the above, **new** is the keyword used for creating objects. Based on the above syntax, we created two printer objects as

```
Printer p1 = new Printer();  
Printer p2 = new Printer();
```

The moment objects are created, JVM will load the objects with all the object level properties (`printDocument()` method), and references the objects using the names `p1` and `p2`. Both `p1` and `p2` are two completely different objects independent of each other.

Tip: Visualize two circles (objects) pointed with the names p1 and p2 in your mind each having `printDocument()` method. Now you have one rectangle and two circles in your mind. Good job.

To invoke the object methods, we need to use `referencename.methodName` syntax as shown below:

```
p1.printDocument();
```

The above statement invokes the `printDocument()` method in the object referenced by p1. This method in turn invokes the shared `print()` method in the class. The `print()` method checks the `paperCount` which is 1, decrements to 0 and displays the following message:

```
Document printed with 20
```

Similarly, when `printDocument()` method is called using p2 reference, it invokes the same shared `print()` method in the class, which sees the `paperCount` as 0, and displays the following message

```
Please load the paper in the tray
```

This is how object level methods share the class level methods. Now, let me ask you a question here. Like an object level method invoking the class method, can a class method invoke object method? I must say big No to this. See the following rule.

Rule: Object level methods (non-static) can always access class level methods (static methods), but class level methods cannot access object methods. Violating this rule causes compilation errors.

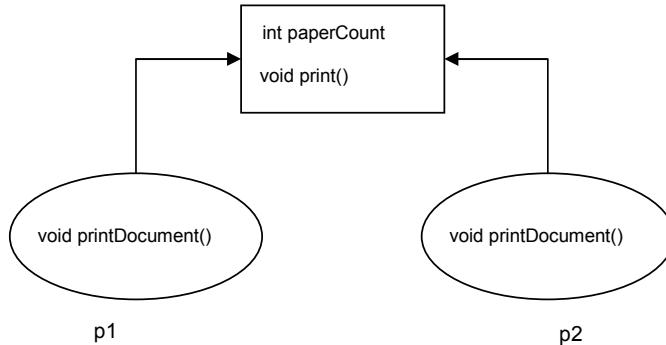
Save the above two files in `objectconcepts` directory and compile them as shown below.

```
C:/>JavaTraining>Chapter4>javac objectConcepts/*.java  
C:/>JavaTraining>Chapter4>java objectConcepts.PrinterTest
```

The result of the above program will be,

```
Document printed with 20  
Please load the paper in the tray
```

Following picture demonstrates the working of this example. This is the picture I asked you to imagine in your mind.



Tip: The best and easiest way to understand objects is by visualizing them. Every time we create object(s) using **new** keyword, right there, imagine a circle(s) with the name of the reference variables or draw it on a piece of paper as shown in the above picture. Do this for the first few examples.

Now that you know how to create objects from class and invoke methods using their references, from here onwards in this chapter, we are going to write classes with only object level methods and variables. We are strictly into object oriented programming business. Ok. See listing 4.3a where the class only has object level properties.

Listing 4.3a (`Calculator.java`) Class with only object level properties.

```
package objectconcepts;

public class Calculator {

    // Instance variable
    double result = 0;

    // Instance methods
    public void sum(int i, int k) {
        result = i + k;
    }

    public void multiply(int i, int k) {
        result = i * k;
    }
}
```

The calculator class in listing 4.3a has two instance methods namely `sum()` and `multiply()` and one instance variable `result`. Both the methods take some data, and update the same `result` variable. This means, all object level methods can share all the object level variables.

Listing 4.3b (`CalculatorTest.java`) Test class creating multiple objects.

```
package objectconcepts;

public class CalculatorTest {

    public static void main(String args[]) {

        // First object
        Calculator call1 = new Calculator();
        call1.sum(4, 16);
        System.out.println(call1.result);

        // Second object
        Calculator call2 = new Calculator();
        call2.multiply(5, 6);
        System.out.println(call2.result);
    }
}
```

As you can see from the above test class, it created two calculator objects and invoked methods using their respective references as shown below:

```
call1.sum(4,16);
```

The above statement updates its `result` variable to 20. The test class then displays the value in the `result` variable using the following statement.

```
System.out.println( call1.result);
```

Similarly, it creates another object `call2` and invokes the `multiply()` method and updates its own `result` variable. Though the object has both the methods `sum()` and `multiply()`, we are invoking only the `sum()` method in one object, and `multiply()` method in the other. You can however, invoke both the methods on the same object as shown below:

```
calc1.sum(10,10);
System.out.println( calc1.result);
calc1.multiply(5,5);
System.out.println( calc1.result);
```

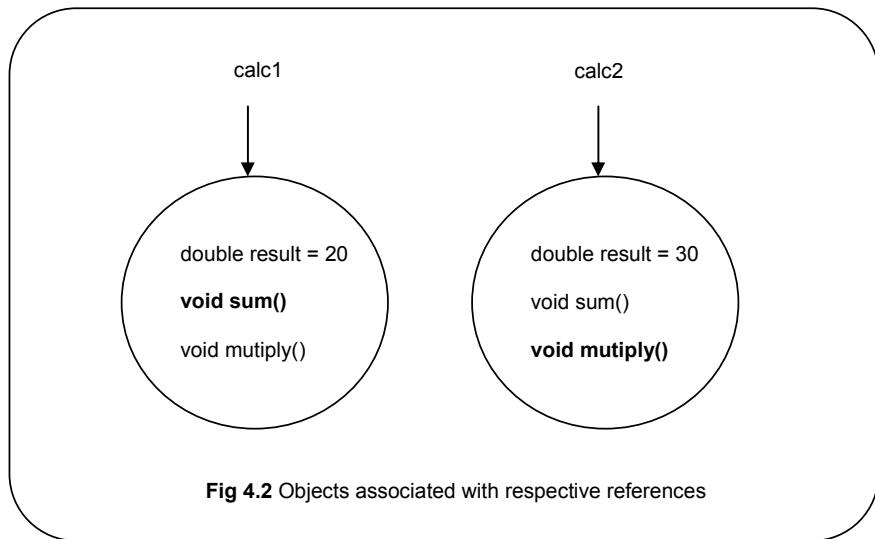
Compile both the classes and execute the test class like as shown below

```
C:/>JavaTraining>Chapter4>javac objectConcepts\*.java  
C:/>JavaTraining>Chapter4>java objectConcepts.CalculatorTest
```

The output of the above program is

```
20  
20
```

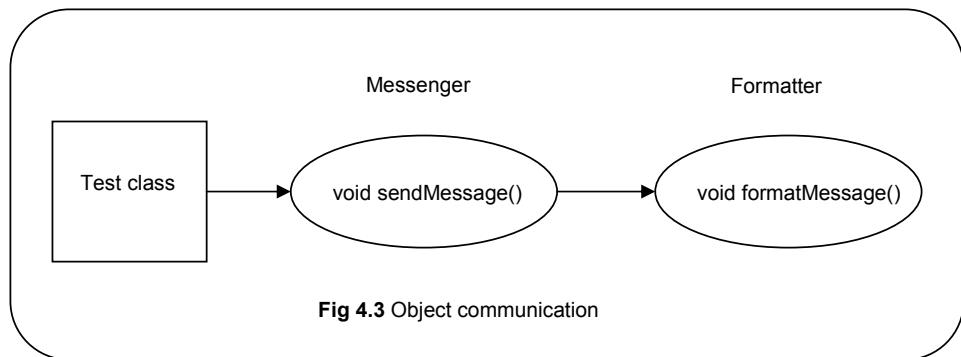
Think of objects as carbon copies made from the original document. You can play with every carbon copy (invoking methods on object) as you like. Once you are done scratching the carbon copy, you can take another copy (create another object) and start working with it in a different way (invoking different methods on the object). Fig 4.2 should give some idea. A picture is worth 100 words.



In the above example, we created two independents as shown in the above figure and just invoked the methods on these independent objects. One thing we should understand is having independent objects simply serves no purpose. As I said right at the beginning of the chapter, an object oriented program creates objects as well as establishes communication between these objects. We now know how to create objects, so let's learn how to tie these objects together and make them more powerful.

Let's say we want to send a message to a legacy system which requires the message to be formatted and then sent. We see two operations here namely 'sending message' and 'formatting message'. If we have two objects with one operation in each, then to send the message successfully to the legacy system, we need to have both objects interact with

each other. In this case, as shown in Fig 4.3, the test object invokes the `sendMessage()` method on the `Messenger` object, which in turn invokes the `formatMessage()` method on the `Formatter` object.



The communication between the above two objects will be established by having the `sendMessage()` method of one object invoking the `formatMessage()` method of the other. In Java, for one object to invoke the method on the other object, the former must first create the object of later, and then invoke the method as shown below:

```
void sendMessage() {  
    Formatter f = new Formatter();  
    f.formatMessage();  
  
    System.out.println("Message sent to legacy system");  
}
```

Listing 4.4 shows the code for the above messenger and formatter classes.

Listing 4.4a (Messenger.java) Class to send message.

```
package objectconcepts;  
public class Messenger {  
  
    public void sendMessage() {  
  
        // Invoke the method in the formatter object  
        Formatter f = new Formatter();  
        f.formatMessage();  
  
        System.out.println("Message sent successfully to legacy system");  
    }  
}
```

Listing 4.4b (`Formatter.java`) Class to format the message.

```
package objectconcepts;

public class Formatter {
    public void formatMessage() {
        System.out.println(" Message formatted");
    }
}
```

If you look at the messenger class in listing 4.4a, its `sendMessage()` method created an object of `Formatter` class and invoked the `formatMessage()` method. This is how messenger object interacts with formatter object, simple, right? The `Formatter` class is pretty straightforward as it simply defines the `formatMessage()` method as shown in listing 4.4b. Finally, the test class simply creates an object of messenger class and invokes the `sendMessage()` method as shown in listing 4.4c.

Listing 4.4c (`MessengerTest.java`) Test class to send a message.

```
package objectconcepts;

public class MessengerTest{
    public static void main(String args[]) {
        // Create a messenger object
        Messenger m = new Messenger();
        // Send the message
        m.sendMessage();
    }
}
```

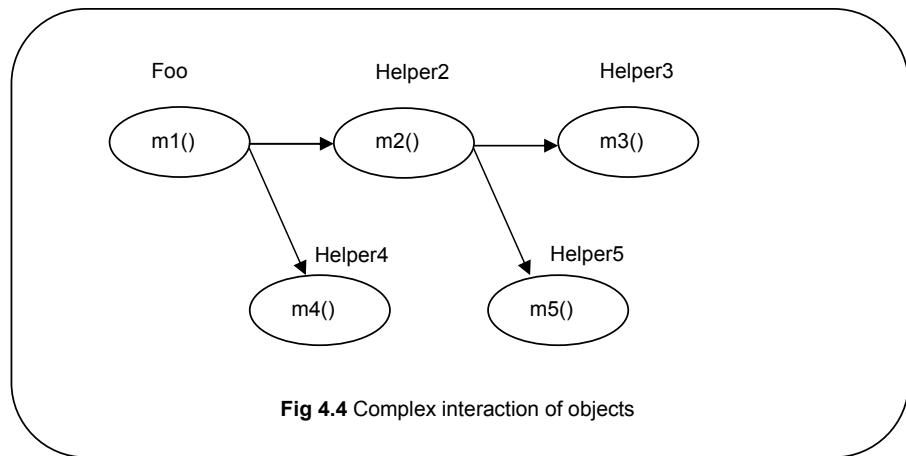
Compile and execute all the three classes as shown below.

```
C:/>JavaTraining>Chapter4>javac objectConcepts\*.java
C:/>JavaTraining>Chapter4>java objectConcepts.MessengerTest
```

The result of the above test class will be,

```
Message formatted
Message sent successfully to legacy system
```

This is how objects interact with each other. The rule is pretty simple. If an object should interact with any another object(s), it should simply create object(s) and invoke the methods. Before we look at the next concept, let's do one complex example where several objects interact with several objects as shown in Fig 4.4.



From the above figure, Foo object interacts with Helper2 and Helper4 objects, and Helper2 object interacts with Helper3 and Helper5 objects. Also, Foo object is the one that initiates the interaction. Each class has methods as shown in the figure. However in real world applications, there could be several objects interacting with each other in much more complex fashion. Look at the code listing 4.5a for each of the above classes, and you'll get the complete picture.

Listing 4.5a Code snippets for the classes shown in Fig 4.4

```

package objectConcepts;

public class Helper3
    public void m3(){
        System.out.println ("I am in Helper3");
    }
}

package objectConcepts;

public class Helper4{
    public void m4(){
        System.out.println ("I am in Helper4");
    }
}

package objectConcepts;

public class Helper5{
    public void m5(){
        System.out.println ("I am in Helper5");
    }
}

```

```
package objectConcepts;

public class Helper2{
    public void m2(){

        System.out.println ("I am in Helper2");

        Helper3 h3 = new Helper3();
        h3.m3();

        Helper5 h5 = new Helper5();
        h5.m5();

    }
}

package objectConcepts;

public class Foo{
    public void m1(){

        System.out.println ("In Foo");

        Helper2 h2 = new Helper2();
        h2.m2();

        Helper4 h4 = new Helper4();
        h4.m4();
    }
}
```

The implementation of all the above classes is pretty straightforward. For instance, `Foo` object to invoke methods in `Helper2` and `Helper4`, created the objects and invoked the methods as shown below:

```
public void m1(){

    System.out.println ("In Foo");

    Helper2 h2 = new Helper2();
    h2.m2();

    Helper4 h4 = new Helper4();
    h4.m4();
}
```

Now, if we write a test class and invoke the method `m1()` in `Foo` object, the interaction initiates. Listing 4.5b shows the test class.

Listing 4.5b (`FooTest.java`) Test class to initiate the object interaction

```
package objectConcepts;
```

```
public class FooTest{
    public static void main (String args[]){
        Foo f = new Foo();

        // Invoke the starting method
        f.m1();

    }
}
```

Compile and execute all the above programs as shown below:

```
C:/>JavaTraining>Chapter4>javac objectConcepts\*.java
C:/>JavaTraining>Chapter4>java objectConcepts.BoxTest
```

The result of the above test class will be is

```
I am in Helper2
I am in Helper3
I am in Helper5
I am in Helper4
```

Understanding Object References

So far we have seen quite a few examples that created objects using **new** keyword, and assigning them to object references which we then used to invoke the object methods. Though this looks pretty simple and straightforward, we need to know how the JVM internally uses the object references to invoke the methods. Let's spend few minutes here and understand the nuances of object references.

Look at the following class.

```
public class Messenger {
    String msg = "";

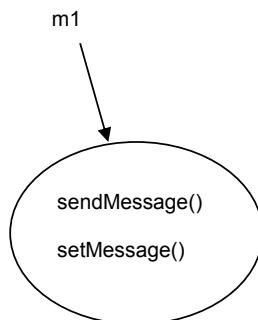
    public void sendMessage() {
        System.out.println("Message" + msg + "Sent");
    }

    public void setMessage(String message) {
        msg = message;
    }
}
```

The above Messenger class has two methods. One method that sets the message and the other method that sends the message. With this class, let's create an object as shown below:

```
Messenger m1 = new Messenger();
```

The above statement creates the following representation in JVM's memory.



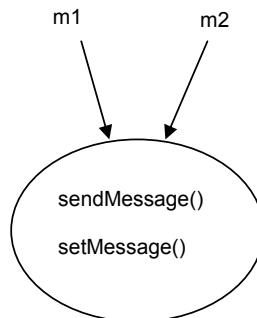
The reference variable 'm1' is like a pointer to the physical object. It is using this pointer, we invoke the methods in the object as shown below:

```
m1.setMessage("Hello ");  
m1.sendMessage();
```

Now, let me ask you a question. Can an object have more than one reference pointing to it? Yes it can. This is achieved by **assigning references**. Assigning object references is very much like assigning primitive variables. Look at the following two statements.

```
Message m1 = new Message();  
Message m2 = m1;
```

As you can see from the above statements, we first created an object with reference 'm1'. We then assigned m1 reference to m2 reference just like we assign primitive variables. The only rule is that the reference variable m2 should also be of the same class type. Now the object in memory will have two references as shown in the following figure.



We can now use any of the above references to invoke the methods as they are pointing to the same object. For instance, we can use 'm1' to invoke one method, and 'm2' to invoke the other method as shown below:

```
m1.setMessage("Hello");
m2.sendMessage();
```

In Java, a single object can have 'n' number of references, and we can use any of the references to update the objects properties. Now that we know how the object references work, the question that comes to our mind is, when and why do we assign references? Good question. The main application of assigning references is when passing 'objects' as parameters to methods. In Java, just like we pass primitive data types (int, float etc) as arguments to methods, we can also pass objects as parameters. Look at the code in listing 4.6 that demonstrates assigning references.

Listing 4.6a (`Book.java`) Simple class with methods and variables.

```
package objectconcepts;

public class Book {

    double bookPrice;

    public void setPrice(double price) {
        bookPrice = price;
    }

    public double getPrice() {
        return bookPrice;
    }
}
```

The Book class shown above is pretty straight forward. It defined two methods, one to set the price, and the other to return the price.

Listing 4.6b (`BookTest.java`) Passing objects as method parameters

```
package objectconcepts;

public class BookTest {

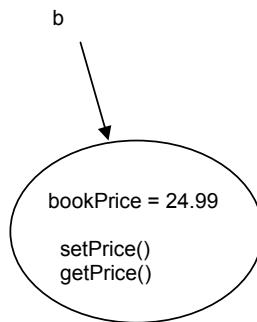
    public static void main(String args[]) {
        //Create a book object
        Book b = new Book();

        // Set the price
        b.setPrice(24.99);

        // Pass this book object as a parameter and display the price
        displayPrice(b);
    }

    static void displayPrice(Book b1) {
        // Read the price
        double price = b1.getPrice();
        System.out.println(price);
    }
}
```

The above test class defined a method `displayPrice()` that echoes the book price. To see how this works, the main method first created the book object 'b' and set the price by invoking the `setPrice()` method. At this point, the book object will be as shown below:



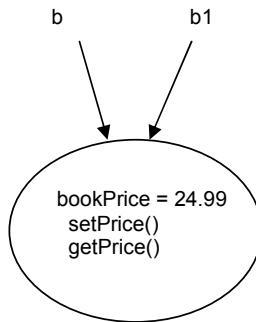
The main method then passed the reference 'b' as a parameter to `displayPrice()` method. Since 'b' is of type `Book`, the method parameter in the `displayPrice()` must also be of type `Book` as shown below:

```
static void displayPrice(Book b1)
```

When 'b' is passed into 'b1', it is indirectly equal to the following assignment:

```
b1 = b;
```

Therefore, 'b1' also points to the same object as 'b' points and the book object will now have two references as shown in the following figure.



Since 'b1' is pointing to the same object, it is like we indirectly passed the object to the method. But, strictly speaking we passed the object reference, but not the physical object, right? So, when the method invokes `getPrice()` method using the reference 'b1' as shown below, it returns the price of the book object.

```
double price = b1.getPrice();
```

Therefore the output of the above program will be,

24.99

This is how we pass objects to methods. Passing objects as parameters to method is another way of making the objects interact with each other. Now that we know how to define interactions among different objects, it's time to know something about two important keywords **public** and **private** when used with object methods and variables. These are also known as "access modifiers".

We all know without any hesitation that an object has methods and variables and we all know that objects interact with other is by calling each others methods. Let's say I am an object for now. I have some personal information that I don't want to share with other objects, but have some other information that I really want to share. So, how can I restrict other objects from accessing my personal stuff and only access what I wanted them to? Is there a way? Yes, there is one. Simply prefix those methods and variables you think are personal with **private** keyword, and for the ones you want to share, prefix with **public**. That's it. If some other class or object tries to invoke private methods, the

compiler spits an error message. Based on these two keywords, let's look at the first OOPS concept, *Encapsulation*.

Encapsulation

Encapsulation allows an object to hide the details and present only the essentials. Details and essentials are nothing but the variables and methods of object. We use **private** keyword for hiding the details and **public** keyword for presenting the essentials. That's it. Let's look at a classical tax calcualtor example. See listing 4.7a.

Listing 4.7a (`TaxCalculator.java`) Class with private and public methods.

```
package objectconcepts;

public class TaxCalculator {

    // Details
    private double federalReturn;
    private double stateReturn;
    private double miscReturn;

    private double computeFederal(double amount) {
        return 0.1 * amount;
    }

    private double computeState(double amount) {
        return 0.05 * amount;
    }

    private double computeMisc(double amount,double misc) {
        return 0.1 * amount + 0.05*misc;
    }

    // Essentials
    public double getMyReturns(double federalWithHeld, double stateWithHeld,
                               double expenses, double misc) {

        federalReturn = computeFederal(federalWithHeld);
        stateReturn = computeState(stateWithHeld);
        miscReturn = computeMisc(expenses,misc);

        return federalReturn + stateReturn + miscReturn;
    }
}
```

Look at the TaxCalculator class.

- It has three private variables and three private methods and no class can directly access these properties. If they try, you know what happens. Compilation errors.
- It has one public method `getMyReturns()`. This is the only method any one can ever access.
- The `getMyReturns()` method internally uses the private methods as helper methods to computes the taxes. The method finally returns the results back. This method is hiding the details and giving only the essentials. Isn't this what encapsulation is all about? Cool.

Now look at the test class shown in listing 4.7b.

Listing 4.7b (`TaxCalculatorTest.java`) Test class for encapsulation.

```
package objectconcepts;

public class TaxCalculatorTest {

    public static void main(String args[]) {

        TaxCalculator calc = new TaxCalculator();
        // Pass the details, and take the essentials
        double returns = calc.getMyReturns(10300, 1600, 700,200);

        System.out.println("My total tax returns = " + returns);
    }
}
```

Now, look at the `TaxCalculatorTest` class. This class creates the tax calculator object first, and sincerely calls the public method `getMyReturns()` by passing all the required data (essentials). It never even dreams to call the private methods as it knows what happens. You can try calling the private method if you want.

If you noticed `TaxCalculator` class carefully, the use of private and public keywords makes complete sense, right? As a user of the tax calculator, I really don't care how it computes my taxes (details). All I care is how much tax returns it gives me back (essentials), right? This is what we call as give and take system. Compile and execute the programs to see the following result.

```
My total tax returns = 1190.0
```

Thumb Rule: Keep **all variables private**, and **all methods** either **private or public**. Never expose the object state (variables) to outside world. If some one needs the data in the variable, write a method that returns it. Ok. At any cost, make sure that an external

object can only interact through methods. No direct access to variables should be allowed. This is a good practice.

You know what; I am tired of using *int's* and *double's* in my programs for declaring the data. Storing text data is the common thing that we would see in any application since most of the data is text based. For instance, things like name, address, city, country and what not. So, I need a new primitive data type to store just text data. Do we have one? The answer is very big NO. Shoot! What should we do now? You know what, don't worry. There is a **special entity** called as **String** which is not really a primitive data type, but behaves in a similar way. We will start using this from now onwards to store texts, but don't even worry about its details for now. Just use it like any other data type. Ok. We'll see its details later. There is a big section dedicated to it. Trust me.

The usage of **String** is shown below. The text to be stored must be enclosed in double quotes.

```
String s = "Thank God, finally I found one for storing text data ";
```

Let's get back to encapsulation for a moment. In real world applications how and where do we use it? Let's say we want to store the details of a customer like `firstName`, `lastName`, `address`, `ssn` etc,. A good solution is to create a class called `Customer`, and define variables like `firstName`, `lastName` etc,. To, get a clear picture, see the code in listing 4.8a.

Listing 4.8a (`Customer.java`) Customer class using encapsulation.

```
package objectconcepts;

public class Customer {

    private String firstName;
    private String lastName;
    private String address;

    public void setFirstName(String fname) {
        firstName = fname;
    }

    public void setLastName(String lname) {
        lastName = lname;
    }

    public void setAddress(String add) {
        address = add;
    }
}
```

```
public String getFirstName() {  
    return firstName;  
}  
  
public String getLastName() {  
    return lastName;  
}  
  
public String getAddress() {  
    return address;  
}  
}
```

Damn, the programs are getting longer. You know what; the programs are instead getting simpler and classier as we move forward. Length doesn't matter for us. Concept matters most. Look at the `Customer` class and see how simple it is. All it did is,

1. Declared three private string variables for storing text data.
2. Since another class cannot directly access these private variables, to facilitate storing of data, it defined three *set* methods to accept the data which will then simply store in the private instance variables.
3. To again return the data back, it defined three *get* methods to return the data in the private instance variables.

This class simply encapsulated the variables and provided getter methods and setter methods for storing and retrieving the data. We call such a class as **JavaBean** or a **POJO** (*Plain Old Java Object*). In real world applications, to store complex data, there will be several of these classes. They simply declare private variables and define getters and setters. Don't be surprised even if you see thousands of above like classes.

Note: Some people call the above class as JavaBean and some others call it as POJO. Technically, both are same,ok.

Now, look at the test class shown in listing 4.8b.

Listing 4.8b (`CustomerTest.java`) Test class using the customer bean.

```
package objectconcepts;  
  
public class CustomerTest {  
    public static void main(String args[]) {  
  
        // Create the Customer Object  
        Customer c1 = new Customer();
```

```
// Store the data  
c1.setFirstName("John");  
c1.setLastName("Smith");  
c1.setAddress(" Gove City, CA ");  
  
// Retrieve the data  
String s1 = c1.getFirstName();  
String s2 = c1.getLastName();  
String s3 = c1.getAddress();  
  
// Display the results  
System.out.println(s1);  
System.out.println(s2);  
System.out.println(s3);  
}  
}
```

As you can see from the above test class, it created Customer object c1 and invoked the set methods to store the data, and then invoked the get methods to retrieve the data back. Compile and execute the test class to see the following result.

```
John  
Smith  
Grove City, CA
```

In the above customer class, I see one small bad practice, the names of method parameters in the set methods. For instance, look at the following set method:

```
public void setFirstName(String fname) {  
  
    firstName = fname;  
}
```

In the above method, the argument name is fname, and the instance variable name is firstName. We are using two different names to represent the same data, right? In such situations, it's a good practice to use the same names for both. If this is the case, then which one refers to instance variable and which refers to argument variable? To solve this dilemma, we need to use the keyword **this** as shown below.

```
public void setFirstName(String firstName) {  
  
    this.firstName = firstName;  
}
```

In the above method, **this.firstName** refers to instance variable, and **firstName** refers to argument variable name. Likewise, to be consistent, in the get methods we should return the instance variables as

```
public String getFirstName() {  
    return this.firstName;  
}
```

The above technique of using **this** keyword is called as *instance variable hiding*. Modify all the methods in the Customer class using the above convention, and you'll see the same results.

The above case of encapsulation is very simple, yet very powerful. It offers **restricted** access to methods and variables by using **private** and **public** keywords.

Trick: Always try to visualize the objects as circles and method names within these circles for the first few examples. This makes understanding objects easier.

Let me once again reiterate that to work with objects, all we need to do is

1. Create an object using **new** keyword and
2. Start invoking the methods using the object reference.

Now, let's learn another concept called method *Overloading*, another classical OOPS concept.

Method Overloading

Method Overloading is a mechanism where more than one method in a single class shares the same name. Following are the two rules for overloading methods.

1. One or more methods in a class must have the same name
2. The **number** of arguments or the **type** of arguments must be different.

If the above two rules are satisfied, then we say that the method is overloaded. Method overloading is very useful in some situations where you have one generic operation that can be implemented in number of ways.

For instance, let's say we have a messenger whose sole purpose is to send text messages to a mainframe system. Now, what if this messenger should have three sending operations to send one text message, two text messages and three text messages based on the situation? This is like same generic operation but different behavior. Such scenarios are best implemented using overloading. Listing 4.9a demonstrates method overloading.

Listing 4.9a (`MultipleMessenger.java`) Class with overloaded methods

```
package objectconcepts;

public class MultipleMessenger {

    // One message
    public void sendMessage(String msg1) {
        System.out.println(msg1 + " sent to Mainframe");
    }

    // Two messages
    public void sendMessage(String msg1, String msg2) {
        System.out.println(msg1 + msg2 + " sent to Mainframe");
    }

    // Three messages
    public void sendMessage(String msg1, String msg2, String msg3) {
        System.out.println(msg1 + msg2 + msg3 + " sent to Mainframe");
    }
}
```

If you look at the `MultipleMessenger` class, we overloaded the `sendMessage()` method. This method obeys both the rules of overloading (same name but different number of arguments). This feature allows us to use the same names for methods whose high-level behavior is the same (send messages in our case). Now look at the test class shown in listing 4.9b.

Listing 4.9b (`MultipleMessengerTest.java`) Test class invoking overloaded methods.

```
package objectconcepts;

public class MultipleMessengerTest {

    public static void main(String args[]) {
        MultipleMessenger messenger = new MultipleMessenger();

        // Invoke the overloaded methods
        messenger.sendMessage("Hello ");
        messenger.sendMessage("Hello", "World");
        messenger.sendMessage("Hello", "Beautiful", "World");
    }
}
```

As you can see from the above code, we invoked all the three methods passing different data. Based on the data, JVM will figure it out as to which overloaded method in the `messenger` class it should invoke. Isn't this cool? I am sure you are with me.

Another point to be noted is that you can also have another set of overloaded methods in the same messenger class. For instance, you can write three more `formatMessage()` overloaded methods next to the `sendMessage()` methods in the same messenger class. There is absolutely no limitation on the number of methods you can overload, as long they adhere to the two rules of overloading.

You'll see a lot of method overloading in real world applications. As an example, we can get a customers profile based on last name, or based on first name and last name, or based on SSN etc. In all the cases, the number of parameters or the type of parameters is changing, but the operation is still the same, right? In such cases, it's a good practice to use method overloading.

Compile the above two classes and execute the test class to see the following result.

```
Hello sent to mainframe system  
Hello World sent to mainframe system  
Hello Beautiful World sent to mainframe system
```

Now, here is a certification question you can expect. Look at the following two methods and tell me whether they are overloaded or not?

```
public int sum( int n1, long n2);  
public int sum( long n1, int n2);
```

You are correct. They are overloaded. If you look at rule 2 of method overloading, it says *Number of arguments or Type of arguments must be different*. In this case, the type of arguments is different. So, they are legally overloaded. Good. Since they are legally overloaded, let's say my test class passes two numbers to the overloaded method as shown below:

```
c.sum(10, 10);
```

Now you tell me, which of the above two methods will be invoked? Most probably you are stumped. You know what, JVM is also stumped, and gives a compilation error saying "method sum is ambiguous".

Here is how the compiler interprets the code. It treats both 10 and 10 as int and looks for a method that takes int and int. When it cannot find such method, it looks for a method whose parameter combinations are *equal to or greater than int*. To its surprise, it finds two methods and gets confused as to which method to invoke. Any time the compiler gets confused, its only weapon to show frustration is by echoing an error. So, the solution is, either remove one of the above method, or pass the arguments correctly as shown below:

```
c.sum( 10, (long)10) // to invoke the first method  
c.sum( (long)10, 10) // to invoke the second method.
```

The above statements use explicit data casting to invoke the right method. This is just to demonstrate the highly improbable scenario, but it's always good to know such things. Things like this are frequently asked in certification exams. If they confuse you, simply ignore them. You'll not see them in real world applications. But understanding these things will surely boost your confidence by many times.

Just as a recap, at this juncture, we learned the following things

1. How to create an object using new keyword
2. How to invoke methods in objects using reference variables
3. How to implement encapsulation using private and public keywords and
4. How to overload methods in a class based on two simple rules.

Now, we will look at one of the most important feature in object oriented programming, "Object Initialization".

If you noticed carefully, in all the examples thus far in this chapter, we simply created objects and invoked its methods. We never initialized objects. Did we? It's time now to start initializing objects. Let's rock.

Object Initialization

If you recall from chapter 2, we learned something about class level initialization using the so called **static** blocks. Let me ask you a question. Since objects are created from a class, if we initialize a class, does this mean we also initialized all the objects of that class? I have to say 'no' again. An object though created from a class, can still have its own initialization.

Let's consider a classical printer example that prints documents in both color as well black and white. Printer can be viewed as a class, and documents as objects of printer, right? We need to feed the printer with papers. This is printer (class) initialization. Let's say the printer must print documents in both black and white as well as color. Here color is specific to the document (object), and not to the printer. Choosing the color of the document (object) is object level initialization. Make sense?

So, the question is, how do we initialize objects? We initialize objects using something called **constructor**. Let's see what this is.

Constructor

A constructor is a *special* method defined in the class used for object initialization. Special? In what way is it special? Look at the following.

A constructor is a method in a class whose,

1. Name must be same as class name and
2. Must not have a return type.

Now we should agree that it's indeed special. Next thing to know is, who invokes the constructor and when is it invoked? Great question. JVM invokes the constructor, and invokes it immediately after creating an object using **new** keyword. Sweet! So, this is all automatic, right? We don't have to worry about invoking constructors at all. All we need to do is define a constructor in the class with whatever initialization we want to do for the object and leave the rest to JVM. Let's look at the details as to how JVM invokes the constructor.

Consider a class named **Printer**. So, based on the above 2 rules, the constructor of this class would be as shown below:

```
Printer()  
{  
}
```

As you can see from the above constructor definition, its name is same as class name and doesn't have any return type like **void** or **int** preceding the method name adhering to the constructor rules. When we create an object of this class as shown below,

```
Printer p = new Printer();
```

It is at this point, JVM identifies the constructor based on the characters after **new** keyword highlighted in bold. Simple, right? Now that it identified the constructor, it simply executes the statements inside the constructor method right away.

Let me ask you a question here. Can a constructor take arguments or parameters like a method? Absolutely, why not? We didn't specify anything in the constructor rules, not to take any arguments. Did we? So, we can also have constructors that take arguments. It is using these arguments that we do different initializations for different objects. Note this point.

As an example, let's say we want to pass an integer value to a constructor of some class say, Test. The constructor of this class would then look as,

```
Test (int val){  
    // Some initialization statements here  
}
```

To create an object that uses the above constructor, we need to pass the integer number right at the point of creating the object as shown below:

```
Test t1 = new Test(10);
```

Since 10 by default is of type int, JVM will invoke the constructor that takes int argument and assigns the number to the constructor argument val. Inside the constructor, the object uses the val variable for its initialization. To create another object and initialize with say 20, we simply do it as shown below:

```
Test t2 = new Test(20);
```

Note: If you define a constructor and don't pass the arguments correctly, you get compilation errors. So, be careful.

I think it's time to write an example using constructors. Let's implement the printer and documents idea we discussed in the beginning. Listing 4.10a shows the printer class.

Listing 4.10a (`AdvancedPrinter.java`) Class using constructor.

```
package objectconcepts;  
  
public class AdvancedPrinter {  
  
    String printType;  
  
    // Document initialization  
    AdvancedPrinter(String type) {  
  
        printType = type;  
    }  
  
    public void printDocument(String text) {  
  
        System.out.println(text + "printed in " + printType);  
    }  
}
```

As you can see from the printer class in listing 4.10a, it defined a constructor that takes type as an argument and initializes the instance variable printType. When the printDocument() method is invoked, it uses the printType to print the text. Now look at the test class shown in listing 4.10b.

Listing 4.10b (AdvancedPrinterTest.java) Test class invoking the constructors.

```
package objectconcepts;

public class AdvancedPrinterTest {

    public static void main(String args[]) {

        AdvancedPrinter bw = new AdvancedPrinter("Black And White");
        bw.printDocument(" Test Document 1");

        AdvancedPrinter colorPrinter = new AdvancedPrinter("Color");
        colorPrinter.printDocument(" Test Document 2");

    }
}
```

The above test class simply created two printer objects 'bw' and 'colorPrinter'. For the first object, we passed 'Black And White' as the print type. So, when we invoke the printDocument() method passing the text, we get the following displayed.

```
Test Document 1 printed in Black And White
```

The colorPrinter object passes 'Color' as the print type, and displays the following when the printDocument() method is invoked.

```
Test Document 2 printed in Color
```

Simple and cool, right? We were able to create objects with different initializations. Compile and execute the test class to see the results.

Now that you know how to define a constructor, and how to create objects to invoke the constructors, here is the tricky situation. In all the several examples we wrote thus far, we didn't define any constructor and we created objects as shown below:

```
Display d = new Display();
```

Yet the examples compiled and executed fine. Didn't they? However, during the discussion on constructors, we learned that JVM *invokes the constructor* at the time of creating an object. So, how is this possible when none is defined? Stumped again? Don't

worry. Here is the rule. *If no constructor is defined, JVM is gracious enough of provide a class with a "default constructor".*

A default constructor is nothing but an empty constructor with no parameters and no body as shown below:

```
Display()  
{  
}
```

Though we didn't define any constructor in our examples, JVM provided us with an *invisible to us and visible to JVM* default constructor. When a class doesn't define any constructor, visualize the presence of a default constructor. This is the reason why all the examples compiled and executed fine without any problems. See, JVM is so generous. :-)

Constructor Overloading

As I said before, constructor is a special method. If this the case, let me ask you a simple question. Like we overloaded methods, can we also overload constructors? You're right. We can well and truly overload constructors. This means an object can have different initializations or more specifically *different levels of initialization*. Moreover, the same rules apply for constructor overloading. Take a look at listing 4.11a.

Listing 4.11a (`Computer.java`) Class with multiple constructors

```
package objectconcepts;  
  
public class Computer {  
  
    int ramSize;  
    int hardDiskSize;  
    String os;  
  
    public Computer() {  
  
        // Default Initialization  
        ramSize = 512;  
        hardDiskSize = 40;  
        os = "Windows XP Home";  
    }  
  
    public Computer(int rs) {  
  
        // Initialization with differnet RAM size  
        ramSize = rs;  
  
        hardDiskSize = 40;  
        os = "Windows XP Home";  
    }  
}
```

```
public Computer(int rs, int hd) {  
    // Initialization for both RAM and HD size  
    ramSize = rs;  
    hardDiskSize = hd;  
  
    os = "Windows XP Home";  
}  
  
public Computer(int rs, int hd, String osl) {  
    // Initialization for all the three  
    ramSize = rs;  
    hardDiskSize = hd;  
    os = osl;  
  
}  
  
public void compute() {  
    System.out.println("Computing with " + os + " with " + ramSize  
        + "MB RAM and " + hardDiskSize + "GB hard disk");  
}
```

The above code defines a `Computer` class with four overloaded constructors. One is default constructor that initializes all the instance variables to default values. A computer object created using this constructor is always a Windows XP Home with 256 MB RAM and 40 GB hard disk.

The second constructor allows the objects to specify the RAM capacity with other two variables to default values. A computer object created using this constructor is always a Windows XP Home with 40 GB hard disk and a specified RAM capacity.

The third constructor allows to create objects with different RAM and hard disk size but with default OS. A computer object created using this constructor is always a Windows XP Home with specified hard disk and RAM capacity.

Similarly, the last constructor allows us to specify all three parameters. This is the most *specific* constructor. The `compute()` method simply uses the values initialized by the constructors for computing. Now look at the test class shown in listing 4.11b.

Listing 4.11b (`ComputerTest.java`) Test class using different constructors for creating objects.

```
package objectconcepts;  
  
public class ComputerTest {
```

```
public static void main(String args[]) {  
    Computer c1 = new Computer();  
    Computer c2 = new Computer(256);  
    Computer c3 = new Computer(512, 40);  
    Computer c4 = new Computer(1024, 80, "Max OS");  
  
    c1.compute();  
    c2.compute();  
    c3.compute();  
    c4.compute();  
}
```

The above test class creates four different types of computer objects using different constructors. Based on the parameters passed, the corresponding constructor will be invoked. `c1` uses default constructor, `c2` invokes second constructor, `c3` the third and `c4` the fourth constructor. No complexities and confusions, right?

Compile and execute the test class to see the following result.

```
Computing with Windows XP Home with 512MB RAM and 40GB hard disk  
Computing with Windows XP Home with 256MB RAM and 40GB hard disk  
Computing with Windows XP Home with 512MB RAM and 40GB hard disk  
Computing with Max OS with 1024MB RAM and 80GB hard disk
```

If you noticed carefully at the `Computer` class, you'll see some duplicate initialization statements such as,

```
os = "Windows XP Home";
```

The above statement appears thrice and some other statements too. So, is there a way by which we can eliminate these duplicate initialization statements? Yes. We can solve this problem by having constructors *invoke each other*. For one constructor to invoke another constructor, it needs to use `this` keyword with the constructor arguments. For instance, to call the default constructor we need use the following statement:

```
this();
```

Similarly, to call the constructor that takes an integer argument, we use the following

```
this(10);
```

The above representation is like replacing the constructor name with the `this` keyword, right? There is one important rule with invoking the constructors. The above statements must appear as the **first statement** inside the constructor. Otherwise, a compilation error

will be thrown at us. Look at the following updated Computer class in listing 4.11c and see how the duplicate code is classically eliminated.

Listing 4.11c (Computer.java) Updated computer class invoking constructors using **this** keyword.

```
package objectconcepts;

public class Computer2 {

    int ramSize;
    int hardDiskSize;
    String os;

    public Computer2() {
        ramSize = 512;
        hardDiskSize = 40;
        os = "Windows XP Home";
    }

    public Computer2(int rs) {
        this();
        ramSize = rs;
    }

    public Computer2(int rs, int hd) {
        this(rs);
        hardDiskSize = hd;
    }

    public Computer2(int rs, int hd, String os1) {
        this(rs, hd);
        os = os1;
    }

    public void compute() {
        System.out.println("Computing with " + os + " with " + ramSize
            + "MBRAM and " + hardDiskSize + "GB hard disk");
    }
}
```

Look at how the different constructors used **this** keyword to invoke other constructors. The second constructor invokes the default constructor which initializes all three variables to default values. Then it sets the ram size with the new value. Similarly, the other constructors invoke the appropriate ones.

This is how we overload constructors and have them invoke each other. As we write many more examples throughout this book, we will come across constructor overloading for sure. As always, compile and execute the test class to see the same results.

We now arrived at a point where we are ready to learn the most important OOPS concept, the *Inheritance*.

Inheritance

First, let's define what inheritance is. *Inheritance* is a mechanism where one class acquires or inherits the properties of another class.

Properties are nothing but variables and methods. For applying inheritance, we need at least two classes. One is the *parent* class whose properties will be inherited, and the other is the class that inherits the properties, called as the *child* class.

The keyword **extends** is used to create a child class from a parent class. Moreover, during inheritance, a child class *cannot* inherit the **private** members of the parent class. Before we write programs, it's important to understand as to when we use inheritance.

Let's say we wrote a class today with four methods for client 1. Tomorrow, we got a new client and needs the same four methods plus one additional method. If we don't have inheritance, what we need to do is simply create a new class, copy all the four methods from the old class and add the one new method, right? At this point, you might wonder why not just add the one new method in the old class? You can do, but why do you want to give additional functionality to client 1 when he doesn't need it? Moreover, if we get another client needing for 6 methods, we again have to update the class, and therefore we are impacting client 1 as well as client 2. Aren't we? This is a very bad practice.

The best solution is, have the new class automatically inherit the functionality that it needs from the old class so that it doesn't have to duplicate the code and just define the new functionality. This is exactly what inheritance allows us to do. To better understand inheritance, look at the code in listing 4.12a.

Listing 4.12a (`BasicCalculator.java`) Calculator class for client 1

```
package objectconcepts;  
public class BasicCalculator {
```

```
private double result;

public double sum(double d1, double d2) {

    result = d1 + d2;
    return result;
}
```

Assume that the above `BasicCalculator` class is sold to client 1. Now we got a new client 2 who needs the `sum()` method as well as `multiply()` method. See listing 4.12b.

Listing 4.12b (`ScientificCalculator.java`) Calculator class for client 2.

```
package objectconcepts;

public class ScientificCalculator extends BasicCalculator {

    private double result;

    public double multiply(double n, double m) {

        result = n * m;
        return result;
    }
}
```

As you can see from the above code, we created a `ScientificCalculator` class for client 2 that **extends** from the `BasicCalculator` class as shown below:

```
public class ScientificCalculator extends BasicCalculator
```

The above statement will tell the JVM to inherit (copy) all the non-private methods from the `BasicCalculator` to the `ScientificCalculator`. Though you don't physically see them, they are implicitly there in the class. You have to visualize their presence. This class then defined the new `multiply()` method which leads to this class having both `sum()` that is inherited as well as the new `multiply()` methods. Notice that the private variable in the `BasicCalculator` class will never be inherited which is why the child class declared its own `result` variable.

Here, `BasicCalulator` is the parent class and `ScientificCalculator` is the child class. Now look at the test class shown in listing 4.12c that verifies the inheritance.

Listing 4.12c (`CalculatorInheritanceTest.java`) Test class for client 1 and client 2

```
package objectconcepts;
```

```
public class CalculatorInheritanceTest {  
    public static void main(String args[]) {  
        // Client 1 code  
        BasicCalculator bc = new BasicCalculator();  
        double result = bc.sum(10, 20);  
  
        // Client 2 code  
        ScientificCalculator sc = new ScientificCalculator();  
        result = sc.sum(20, 30);  
        result = sc.multiply(20, 30);  
    }  
}
```

As you can see from the above test class, it first created an object of `BasicCalculator` class and invoked the `sum()` method. This is client 1 code. Next, it instantiated the `ScientificCalculator` class and invoked both the `sum()` as well as `multiply()` methods. This is possible only because the child class inherited the `sum()` method. Isn't this cool? Now a small question. What if I got a new client, who needs the methods `sum()`, `multiply()` and `divide()`?

Do you want to,

- Write a class `AdvancedCalculator` that inherits from `BasicCalculator` and define `multiply()` and `divide()` methods (or)
- Write a class `AdvancedCalculator` that inherits from `ScientificCalculator` and just define the `divide()` method

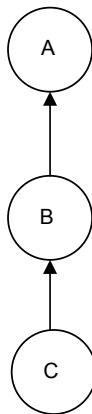
I am sure you would go with choice b. This is exactly what we should and can do. `AdvancedCalculator` will now be the child of `ScientificCalculator` and following is how its class declaration will be.

```
public class AdvancedCalculator extends ScientificCalculator
```

The above class inherits the `sum()` as well as `multiply()` methods and defines the `divide()` method. We can then create an object of this class and invoke all the three methods as shown below. Isn't this cool and classical?

```
AdvancedCalculator ac = new AdvancedCalculator();  
ac.sum(10,10);  
ac.multiply( 10, 10);  
ac.divide(10,10);
```

The above example follows the pattern in which A is the parent of B, B is the parent of C as shown in the following figure.



In the previous example, we learned how to create a child class from a parent class. Let me ask you another question here. Can a parent class have just one child class? No. A parent class can have any number of child classes. For instance, in the above example `BasicCalculator` has just one child called `ScientificCalculator`. However, we can also create another child from it like `MathCalculator` as shown below:

```
public class MathCalculator extends BasicCalculator
{
}
```

This child class will now have `sum()` method plus its own methods. There is absolutely no limit on the number of child classes a parent class can have. To demonstrate the real power of inheritance, let's look at a complex inheritance hierarchy diagram and understand what methods are inherited by various classes in the diagram. Since, I cannot write the entire method in the figure, let's use some conventions.

Look at the following block that represents a class in our diagram.

Class: A
+ j : int
- val : String
+ m1 : void
+ m2 : void
- m3 : void

In the above class block, + represents public members, and - represents private members.

Based on this convention, the above block represents a class named A having one public variable `j`, one private variable `val`, two public methods `m1` and `m2` and one private method `m3`. Now, look at the Fig 4.5 that represents a complex inheritance hierarchy.

The inheritance diagram shows six classes namely A, B, C, D, E, F each having their own methods and variables. From the class diagram, we observe the following things.

A,B,D is one branch

A,B,E is another branch

A,C,F is another branch

A is the ultimate parent class

B and C inherit from A

D inherit from both B and A as they belong to the same branch

E inherit from both B and A as they belong to the same branch

F inherit from C and A as they belong to the same branch

Note: Private variables and methods are never inherited.

Based on the above rule, let's see that methods and variables various classes will have.

Class B:

From A it inherits

Variables: j

Methods: m1 and m2

Combining the above with its own variables and methods, B will have

`j` and `val` as variables, and methods `m1, m2, m4, m5, m6`.

(See the figure in the next page)

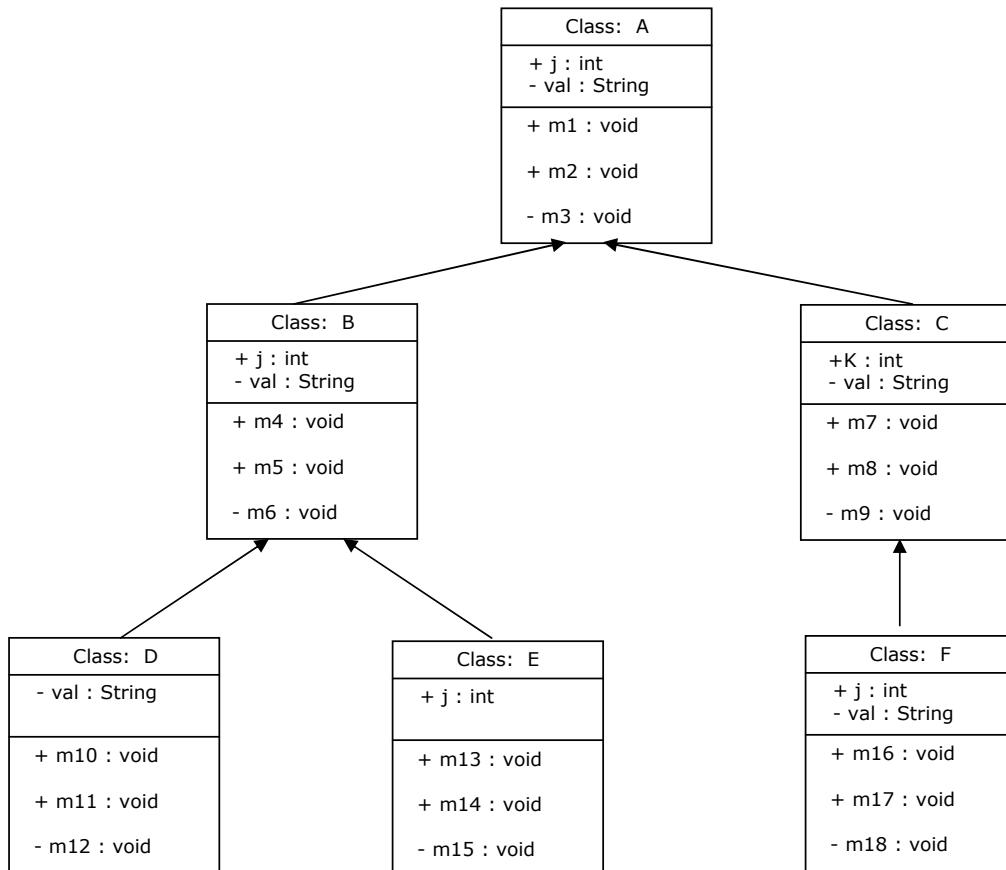


Fig 4.5 Complex inheritance hierarchy

Class C:

From A it inherits:

Variables: j

Methods: m1 and m2

Combining the above with its own variables and methods, C will have

j, **k** and **val** as variables, and methods **m1, m2, m7, m8, m9**.

Class D: Inherits from both A and B

From A, B it inherits

Variables: j

Methods: m1, m2, m4, m5

Combining the above with its own variables and methods, D will have

j, val as variables, and methods m1, m2, m4, m5, m10, m11, m12.

Class E: Inherits from both A and B

From A, B it inherits:

Variables: j

Methods: m1, m2, m4, m5

Combining the above with its own variables and methods, E will have

j, val as variables, and methods m1, m2, m4, m5, m13, m14, m15.

Class F: Inherits from both A and C

From A, C it inherits

Variables: j, k

Methods: m1, m2, m7, m8

Combining the above with its own variables and methods, D will have

j, k, val as variables, and methods m1, m2, m7, m8, m16, m17, m18.

I hope the above picture and the derivations are pretty straightforward. This is how we can build the inheritance hierarchy and determine which methods and variables are inherited. Now, based on the above diagram, let me ask you couple questions here.

Question 1: I need to build a class with the methods m1, m2, m4, m5 from existing classes and a new method p1. Which of the following classes my new class must extend:

- a) Class A
- b) Class B

Question 2: I need to build a class with the methods m1, m2, m7, m8, m16, m17 from existing classes and a new method p2. Which of the following classes my new class must extend:

- a) Class F
- b) Class C

The correct answers are ‘choice b’ for Q1 and ‘choice a’ for Q2. I am sure you got them right. Now you are an expert in inheritance and its intricacies.

If you carefully observe the above inheritance diagram, we don’t see a class having more than one parent. So the question is, can a class have more then one parent, which is also known as multiple inheritance? The answer is big no. In Java, multiple inheritance is strictly not allowed which means we cannot create a child class from two different parents. Therefore the following class declaration causes compilation error.

```
public class G extends A,B // Not allowed in Java
```

All this is to say that, in Java, only single inheritance is allowed. A class can only have one parent. The reason why Java doesn’t allow multiple inheritance is that it introduces whole lot of confusions and complications which is fundamentally against the primary goal of Java, “Keeping things Simple”. Fig 4.6 shows both legal and illegal inheritance hierarchies.

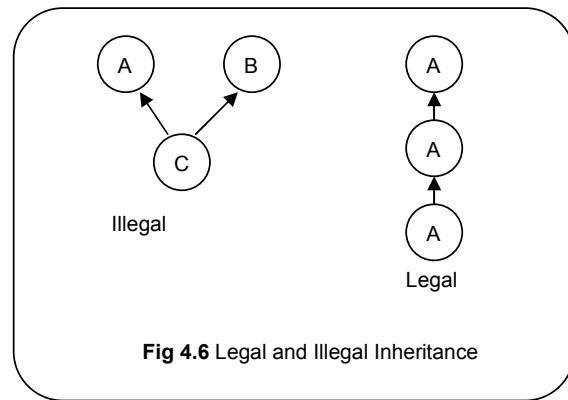


Fig 4.6 Legal and Illegal Inheritance

This completes the basics of inheritance. More and deeper details on inheritance are investigated in the next chapter. For now, let's move on to the next concept.

Method Overwriting

We all now know that we use inheritance when we need to create a class with most of existing functionality plus some new functionality. Good. Now, let's say we have a class named `MainframeMessenger` with three methods being used by client 1. Now let's say we got a new client who needs a new messenger class with same functionality as the above class, but with slight modification to one of its methods (no new method this time, ok). What shall we do? Shall we update the method in existing class? No, we can't, because the updates made to method for client 2 will surely break client 1 functionality, right? So, this is clearly not the solution.

The only solution for the above scenario is to create a completely new class. Again, how do we want to create this new class? Shall we copy all the methods from the old class and update the method that needs to be updated or use inheritance to eliminate code duplication? So, at this point the best solution is using inheritance. Good. I therefore created a new messenger class from the mainframe messenger class as shown below:

```
public class SAPMessenger extends MainframeMessenger
{ }
```

When we created our new class as above, what happened is, by the rule of inheritance the new class inherits the exact same methods that are in the parent class. Hmm. Looks like we are back to square one, since our new `SAPMessenger` class is exactly same as `MainframeMessenger` class at this point of time. The challenge we have now is to update the inherited method, right? This is where method overwriting comes into picture.

Method overwriting requires us to *overwrite* the inherited old method with the new updated method. To clearly understand, take a look at the code in listing 4.13.

Listing 4.13a (`MainframeMessenger.java`) Class for client 1

```
package objectconcepts;

public class MainframeMessenger {

    public void formatMessage(String str) {
        System.out.println("Message Formatted");
    }
}
```

```
public void sendMessage(String message) {  
    formatMessage(message);  
    System.out.println("Message" + message + " sent to Mainframe");  
}
```

Looking at the mainframe messenger class, it has two methods namely `formatMessage()` and `sendMessage()`. This class is used by client 1. Now look at the code in listing 4.13b

Listing 4.13b (`SAPMessenger.java`) Class for client 2

```
package objectconcepts;  
  
public class SAPMessenger extends MainframeMessenger {  
  
    public void sendMessage(String message) {  
        formatMessage(message);  
        System.out.println("Message" + message + " sent to SAP System");  
    }  
}
```

As you can see from the above code, we created a `SAPMessenger` class for client 2 and just implemented the `sendMessage()` method with updated functionality. Is this going to work? Yes it will. This is how it works.

1. `SAPMessenger` inherits the methods `formatMessage()` and `sendMessage()` from `MainframeMessenger`.
2. The updated `sendMessage()` method *overwrites* the inherited `sendMessage()` method. This class will now have the old `formatMessage()` method and the updated `sendMessage()` method.

This process of defining the exact same method in the child class that is in the parent class, and then changing the behavior is what we call as **method overwriting**. The child class is now said to have overwritten the method in the parent class. In the above code, if you noticed, we eliminated duplicating the `formatMessage()` method, right? In real world applications there will be tons of methods in the parent class, and overwriting just one method eliminates lot of code duplication.

Now, look at the test class shown in listing 4.13c.

Listing 4.13c (OverwritingTest.java) Test class for client 1 and client 2

```
package objectconcepts;

public class OverwritingTest {

    public static void main(String args[]) {

        MainframeMessenger m1 = new MainframeMessenger();
        m1.sendMessage("Hello");

        SAPMessenger s1 = new SAPMessenger();
        s1.sendMessage("Hello");
    }
}
```

The above test class created both the messengers and invoked the `sendMessage()` methods. By using `m1` reference it invokes the method from the mainframe class which produces the following output.

```
Message Formatted
Message Hello sent to Mainframe
```

Using `s1` reference, it invokes the updated method in the SAP messenger class, and produces the following output

```
Message Formatted
Message Hello sent to SAP System
```

Compile all the above 3 classes, and execute the test class to see the above results.

In the above example, we clearly know that we eliminated writing *duplicate methods in the class* using inheritance and method overwriting. Let's look at another classical example that also eliminates *duplicate statements in the methods*.

Consider a class called `Processor` with a method `process()` that processes four steps used by client 1 as shown below:

```
void process(){

    System.out.println("Processing Step 1");
    System.out.println("Processing Step 2");
    System.out.println("Processing Step 3");
    System.out.println("Processing Step 4");
}
```

Now my new client requires a `process()` method to process six steps as shown below

```
void process() {  
    System.out.println("Processing Step 1");  
    System.out.println("Processing Step 2");  
    System.out.println("Processing Step 3");  
    System.out.println("Processing Step 4");  
    System.out.println("Processing Step 5");  
    System.out.println("Processing Step 6");  
}
```

So, what we need to do is, create a child class of Processor class and overwrite the above method to include six steps, right? Though this is what we should do, the child class `process()` method will now have the *duplicate* statements from parent `process()` method plus additional statements, right?. The situation is even worse if we have several hundred statements like in real world applications.

In such situations, the best solution is have the child class `process()` method invoke the parent class `process()` method, and just define the additional statements. This is when we use the keyword **super**. The **super** keyword is used by child classes methods to invoke the parent class methods. The syntax for invoking methods using super keyword is shown below:

```
super.methodName();
```

Now look at the code in listing 4.14 that demonstrates the usage of **super** keyword and smartly eliminating the duplicate statements.

Listing 4.14a (`Processor.java`) Client 1 code

```
package objectconcepts;  
  
public class Processor {  
  
    public void begin() {  
        System.out.println("Processing Initiated");  
    }  
  
    public void process() {  
        System.out.println("Processing Step 1");  
        System.out.println("Processing Step 2");  
        System.out.println("Processing Step 3");  
        System.out.println("Processing Step 4");  
    }  
  
    public void stop() {  
        System.out.println("Processing Completed");  
    }  
}
```

```
    }

    public void execute() {
        begin();
        process();
        stop();
    }
}
```

The above code is pretty straightforward and let's assume that it is used by client 1. Now take a look at code in listing 4.14b that uses the **super** keyword.

Listing 4.14b (`AdvancedProcessor.java`) Client 2 code

```
package objectconcepts;

public class AdvancedProcessor extends Processor {

    public void process() {
        // Invoke the parent class process method
        super.process();

        System.out.println("Processing Step 5");
        System.out.println("Processing Step 6");
    }
}
```

If you look at the `process()` method in the `AdvancedProcessor`, the overwritten method first calls the parent class `process()` method and then adds whatever additional statements it should. This is really cool way of doing things, isn't it? Look how we completely eliminated the duplicate statements. The child class still inherits all the other methods. Listing 4.14c shows the test code for the above classes.

Listing 4.14c (`ProcessorTest.java`) Test class for client 1 and client 2 code

```
package objectconcepts;

public class ProcessorTest {

    public static void main(String args[]) {
        System.out.println("Using Processor");

        Processor p1 = new Processor();
        p1.execute();

        System.out.println("Using AdvancedProcessor");

        AdvancedProcessor p2 = new AdvancedProcessor();
```

```
        p2.execute();  
    }  
}
```

In the above test class, when the `execute()` method is invoked using `p1` reference, it internally invokes the `begin()`, `process()` and `stop()` methods from `Processor` class. So, the result will be,

```
Processing Initiated  
Processing Step 1  
Processing Step 2  
Processing Step 3  
Processing Step 4  
Processing Completed
```

When the `execute()` method is invoked using `p2` reference, it invokes the `begin()`, `process()` and `stop()` methods in the `AdvancedProcessor` class. Don't ask me where the `begin()` and `stop()` methods came from. They are inherited from the parent class. The `process()` method first invokes the parent class method, executes the four statements, comes back and executes the remaining two statements. So the result will be,

```
Processing Initiated  
Processing Step 1  
Processing Step 2  
Processing Step 3  
Processing Step 4  
Processing Step 5  
Processing Step 6  
Processing Completed
```

This is all about method overwriting. The rule is pretty simple. Overwrite a method in a class when some one needs a completely updated behavior in the existing class. When you overwrite a method, the rule is that the **method name, argument list, keywords and return type all must be the same**. No changes at all, ok. This is opposite to overloading, right?

Interview question: What is the difference between Overloading and Overwriting?

Answer: Overloading is related with one class where more than one method in the same class shares the same name. Overwriting is related with 2 classes namely parent class and child class, where the child class defines the exact same method as the one in parent class. You need to have this on your finger tip. :-)

Hopefully I didn't confuse you until now. I am sure the examples thus far are pretty straightforward. One of the primary goals of this book is not to create any confusion at

any point of time. If something is still confusing, read the same again. If you still don't understand, skip it. You'll be fine. Trust me. The important thing is we should always keep moving forward, not just with this book, but anywhere :-)

Constructors Revisited

It's time to revisit constructors again. You know, we already learned how to define a constructor and how the JVM invokes the constructors. We also know how to overload constructors. Is there anything that we missed? Yes, a small concept around constructors during inheritance. Consider the two classes shown in listing 4.15a and 4.15b.

Listing 4.15a (`Parent.java`) A typical parent class with 2 constructors.

```
package objectconcepts;

public class Parent {

    public Parent() {
        System.out.println("I am in Parent default constructor");
    }

    public Parent(int val) {
        System.out.println("I am in Parent int constructor");
    }
}
```

Listing 4.15b (`Child.java`) A typical child class with 2 constructors.

```
package objectconcepts;

public class Child extends Parent {

    public Child() {
        System.out.println("I am in Child default constructor");
    }

    public Child(int val) {
        System.out.println("I am in Child int constructor");
    }
}
```

The code shown in listing 4.15 shows two classes namely `Parent` and `Child` each having a default constructor and a constructor that takes `int` as an argument. Let's see four cases here.

Case 1: What will be the result of the following statement?

```
Parent p1 = new Parent();
```

The above statement is creating a parent object with default constructor. So, the JVM invokes the default constructor, and the result will be,

```
I am in Parent default constructor
```

Good. This is pretty simple. Let's look at next case

Case 2: What will be the result of the following statement?

```
Parent p1 = new Parent(10);
```

The above statement is creating a parent object with constructor taking an integer argument. So, the JVM invokes the constructor that takes int. The result will be,

```
I am in Parent int constructor
```

No surprises until now. Which ever constructor we used while creating the Parent class object, the same constructor is executed. Let's look at next case.

Case 3: What will be the result of the following statement?

```
Child c1 = new Child();
```

The above statement is creating a child object with default constructor. So, like before we expect the result to be,

```
I am in Child default constructor
```

But the actual result will be

```
I am in Parent default constructor  
I am in Child default constructor
```

Stumped! How can this be possible? Take a look at the next case.

Case 4: What will be the result of the following statement?

```
Child c1 = new Child(10);
```

The above statement is creating a child object with integer constructor. So, we expect the result to be,

```
I am in Child int constructor
```

But the actual result will be

```
I am in Parent default constructor  
I am in Child int constructor
```

Once again stumped. Shoot, what the hell is going on here. No need to get frustrated guys. Here is the JVM's rule of executing the constructors under inheritance.

Rule: Before executing any child class constructor, JVM *always* executes parent class **default** constructor.

With the above rule, look at Case 3 and Case 4 again. You'll no longer be stumped.

Thumb Rule: If a class is participating in inheritance, when creating an object of that class, JVM always executes **all** of its predecessors (all the parents in the inheritance chain) default constructors. It's very important to remember this point.

Listing 4.15c is the test class for the above code.

Listing 4.15c (`ConstructorTest.java`) Test class for constructors during inheritance.

```
package objectconcepts;  
  
public class ConstructorTest {  
  
    public static void main(String args[]) {  
  
        Parent p1 = new Parent();  
        Parent p2 = new Parent(10);  
  
        Child c1 = new Child();  
        Child c2 = new Child(10);  
    }  
}
```

Compile all the above three classes, and execute the test class as shown below

```
C:/>JavaTraining>Chapter4>javac objectConcepts\*.java  
C:/>JavaTraining>Chapter4>java objectConcepts.ConstructorTest
```

The output of the above program will be

```
I am in Parent default constructor
I am in Parent int constructor
I am in Parent default constructor
I am in Child default constructor
I am in Parent default constructor
I am in Child int constructor
```

If you look at the previous example, all the child class constructors are only invoking the parent class **default** constructor. But in real world applications, it would be more beneficial if the child class constructors have the ability to invoke which ever parent constructor it want to rather than JVM always invoking the default constructor as shown above.

If you recall one of the previous examples, we saw how constructors invoked other constructors of the same class using **this** keyword. Similarly, for a child class to explicitly invoke any of the parent class constructors, it should use **super** keyword. For instance, the statement *super()* invokes the parent class default constructor, *super(20)* invokes the parent class integer constructor and so on.

Now that we know how to use super to invoke the parent class constructors, the question that comes to my mind is when and why a child class should invoke the parent class constructor? Again the reasoning is simple, to eliminate duplicate initialization statements. Just like in one of the example we used **super** to invoke a parent class method to eliminate the duplicate statements, we use **super()** constructor calls to avoid duplicate initialization code in child classes.

Let's look at more practical application. Extreme Solutions Inc is a company that develops a database called "ExtremeDB". Today it released version 5 of the database which has certain initialization. As part of continuous enhancements, it decided to release version 6 after 1 year and fix some of the bugs in the previous release. Let's say the version 5 database is suffering from serious performance issues. After serious research the company attributes the problem due to bad initialization of the database, and decides to add more initialization steps on top of existing initialization process hoping to improve the performance. Let's now write a program to implement this scenario.

Look at the code in listing 4.16.

Listing 4.16a (`ExtremeV5.java`) Version 5 code of the database.

```
package objectconcepts;

public class ExtremeV5 {
```

```
public ExtremeV5(int counter) {  
    for (int i = 1; i <= counter; i++) {  
        System.out.println(" Initializing Partition " + i);  
    }  
}  
  
public void execute() {  
    System.out.println("Executed Statements");  
}
```

If you look at the `ExtremeV5` class, it defines a constructor that takes an integer argument. This constructor has some initialization, ok. Now, take a look at the code in listing 4.16b.

Listing 4.16b (`ExtremeV6.java`) Version 6 code of the database.

```
package objectconcepts;  
  
public class ExtremeV6 extends ExtremeV5 {  
  
    public ExtremeV6(int counter) {  
        super(counter);  
  
        // Additional Initialization  
        for (int i = 1; i <= counter; i++) {  
            System.out.println(" Initializing Channel " + i);  
        }  
    }  
}
```

As you can see from the above code, we created `ExtremeV6` class whose constructor explicitly invokes the parent class constructor as shown below:

```
super(count);
```

The above statement executes all the initialization statements in `ExtremeV5` parent class constructor. Once completed, the control comes back to the `ExtremeV6` constructor and executes the rest of the initialization statements. Listing 4.16c shows the test class for the above two classes.

Listing 4.16c (`ExtremeTest.java`) Test class for version 5 and version 6 databases.

```
package objectconcepts;
```

```
public class ExtremeTest {  
    public static void main(String args[]) {  
        System.out.println("Using Oracle Database V5");  
        ExtremeV5 v5 = new ExtremeV5(4);  
        v5.execute();  
        System.out.println("Using Oracle Database V6");  
        ExtremeV6 v6 = new ExtremeV6(4);  
        v6.execute();  
    }  
}
```

The test class creates objects of both the classes by passing the argument value 4 to their constructors and invokes the `execute()` method. The output of the program will be,

```
Using Oracle Database V5  
Initializing Partition 1  
Initializing Partition 2  
Initializing Partition 3  
Initializing Partition 4  
Executed Statements  
Using Oracle Database V6  
Initializing Partition 1  
Initializing Partition 2  
Initializing Partition 3  
Initializing Partition 4  
Initializing Channel 1  
Initializing Channel 2  
Initializing Channel 3  
Initializing Channel 4  
Executed Statements
```

As you can see from the above result, the version 6 of the database used the initialization from version 5 followed by its own initialization statements. This is how we use **super** keyword to invoke the parent class constructors. There is one rule here shown below.

Rule: The **super** statement must be the first statement in the constructor, and a single child constructor cannot invoke more than one parent constructor.

With this, we now know everything about constructors. Good.

Now, it's time to learn something about abstract methods, classes and their significance. Abstract classes play a very important role in object oriented design. Let's see what these are.

Abstract Methods

In object oriented design, one of the common things we try to do is define a generalized class with method definitions, and let other classes implement the methods in their own way. The generalized class is like a definition class that only knows what methods a class must have, but doesn't know how to implement the methods (define the body). Such methods are said to be **abstract** as they don't have any implementation.

To define an abstract method, the keyword we need to use is **abstract**. Following are the two simple rules to define an abstract method.

1. Define the method and put a semicolon after the method signature. No curly braces {}.
2. Declare the method as abstract using **abstract** keyword.

The syntax for declaring abstract methods is shown below

abstract <return type> methodName(argument list);

As an example, following is an abstract `process()` method.

```
public abstract void process();
```

All we did is included the **abstract** keyword in the method declaration, and terminated with a semicolon without writing any statements.

Abstract classes

Now, what is an abstract class? If a class has at least one abstract method, the class is said to be abstract and must be declared abstract as shown below:

```
public abstract class Test
```

A class can have any number of abstract methods. However, a class besides having abstract methods can also have non-abstract methods (ones that are implemented). This is why we also call abstract class as a partially implemented class.

Note that we again have to use the **abstract** keyword in the class declaration. Look at following 2 simple rules with abstract classes:

1. A class **must** be abstract if it has at least **one** abstract method.
2. If a class is abstract, we cannot create objects of that class using **new** keyword. This makes sense because, the class is partial, and we don't want to create partial objects. So the JVM itself forbids creating objects.

Rule 2 is interesting. If the class is declared abstract, it says we are not allowed to create objects of that class. If this is the case, then this class is fit for nothing even though it has few implemented methods, right? To make the class useful, what we therefore need to do is, create a child class and have it implement all the abstract methods. Now the child class becomes fully implemented class and we should use this class for creating objects. Make sense? Yes it does.

Here is the de facto rule. If we have an abstract class, there **should** be one or more child classes of the abstract class to implement all the abstract methods. Every child class can implement the abstract methods in its own way. But the rule is, it must implement all the methods.

If you have a situation where you know what methods a class must have, but cannot implement some of the methods, then use an abstract class. Then somebody who knows how to implement abstract methods should write a child class of the abstract class, and provide the implementation for those some unimplemented methods. Let's look at an example and see how abstract methods are used.

I am a car designer. I create the design and sell it to various car manufacturers. Car manufacturers will then use my design and build the car with their own technology and features. As a designer, I know there will be some things that are common to every car, like the way the car should be driven and the manufacturers should never implement them. Instead, they should only implement other specific components. Listing 4.17 shows the code for this scenario using abstract classes and child classes.

Listing 4.17a (Car.java) Abstract car class.

```
package objectconcepts;

public abstract class Car {

    public abstract void assembleTyres();
    public abstract void addCover();
    public abstract void addEngine();
    public abstract void addSeats();
    public abstract void startCar();
    public abstract void shiftGears();
    public abstract void drive();
```

```
public void testDrive() {  
    assembleTyres();  
    addCover();  
    addEngine();  
    addSeats();  
    startCar();  
    shiftGears();  
    drive();  
}  
}
```

Look at the `Car` class in the above listing. This class defined several abstract methods. It also defined one method `testDrive()` which is not abstract, as it knows how to implement the method. This is the common method that the car class expects the child classes to always use. This is a partially implemented class and the class is therefore declared abstract as shown below:

```
public abstract class Car {
```

Now, take a look at the child class shown in listing 4.17b.

Listing 4.17b (`Honda.java`) Child class implementing the abstract methods.

```
package objectconcepts;  
  
public class Honda extends Car {  
  
    public void assembleTyres() {  
  
        System.out.println("Assembled GoodYear tyres");  
    }  
  
    public void addCover() {  
  
        System.out.println("Added Metallic Cover");  
    }  
  
    public void addEngine() {  
  
        System.out.println("Added 500 HP engine");  
    }  
  
    public void addSeats() {  
  
        System.out.println("Added 2 Leather seats");  
    }  
  
    public void startCar() {  
  
        System.out.println("Car started with AutoStart");  
    }  
}
```

```
public void shiftGears() {  
    System.out.println("Has Automatic Transmission");  
}  
  
public void drive() {  
    System.out.println("Driving at 80 mi/hr");  
}
```

As you can see from the above code, the Honda class is created from the Car class and implemented all the abstract methods as it knows how to implement them. This class inherits the testDrive() method. Therefore, it has all the methods implemented and we can create objects of this class. Likewise, you can create another class like Toyota and implement the same methods but in a different way. Now look at test class shown in listing 4.17c.

Listing 4.17c (CarTest.java) Test class for invoking methods.

```
package objectconcepts;  
  
public class CarTest {  
  
    public static void main(String args[]) {  
  
        Honda h = new Honda();  
        h.testDrive();  
    }  
}
```

The test class simply created Honda object and invoked the testDrive() method. This method internally invokes all the methods and generates the following output.

```
Assembled GoodYear tyres  
Added Metallic Cover  
Added 500 HP engine  
Added 2 Leather seats  
Car started with AutoStart  
Has Automatic Transmission  
Driving at 80 mi/hr
```

In Java, only methods and classes can be abstract. Variables cannot be abstract. This is how we use abstract classes and methods. Using abstract classes is very common in real world applications as it allows multiple implementations of certain behavior. The next topic is final methods and final classes.

Final methods

Like abstract methods and abstract classes, methods and classes can also be final. This is done using the **final** keyword. The usage of final with methods and classes is very simple. If you want a certain method in a class *not* to be overwritten in the child class, then simply declare the method as final as shown below

```
public final void testDrive();
```

When the method is declared as above, JVM will not allow any of the child classes to overwrite the method. In the previous example, the `testDrive()` method in the `Car` class is not final. So, `Honda` class can overwrite it. But it didn't. So the `Car` class is at the mercy of `Honda` class not to overwrite it. But now that we know what a final can do, without any hesitation attach the **final** keyword to the `testDrive()` method, and the `Honda` class can't overwrite it. The `Car` class need not live at the mercy of `Honda` class anymore. This is like enforcing the rule. Now I am 100% sure that no car manufacturer will ever dare to overwrite the method. They must adhere to the design. This is how we use final with methods.

Final class

Let me ask you a question here. If one of the methods in a class is final, should the class be declared final? Nope. Unlike abstract, you need not declare the class as final if it has final methods. However, you can make the class final at any time to gain one advantage. The advantage is *preventing inheritance* of that class. So, if a class is declared as final, it cannot have any child classes. Following is how a final class looks like:

```
public final class Test  
{  
}
```

Trying to create a child class of the above class using **extends** keyword will cause a compilation error.

Final variables

Can a variable be final? Yes. A variable can be **final**. If a variable is final, its value cannot be changed once assigned as shown below:

```
final int MAX_QTY = 100;
```

Trying to alter the value of MAX_QTY will cause a compilation error. This is why we call final variables as CONSTANTS.

Let's summarize the usage of final and abstract, and see the differences between the two.

Difference between abstract and final

If you noticed carefully, final keyword and abstract keyword work totally opposite. Here are the differences between **final** and **abstract** keywords. This is 100% guaranteed interview question.

final

- Used with Class, method and variable
- If a class is declared final, it cannot be inherited
- If a method is declared final, it cannot be overwritten in the Child class.
- If a variable is declared final, its value cannot be changed. It is a CONSTANT

abstract

- Used with Class and method only. Not with variable.
- If a class is abstract, then it must be inherited (sub classed) before using it.
- If a method is abstract it must be overwritten in the Child class.

Now the last concept in this chapter, Interfaces.

Interfaces

Let's recall the notion of abstract classes here for a second. As I said before, an abstract class is a partially implemented class. This means, some of its methods are well and truly implemented and some of them are abstract. Let me ask you a question. What if all methods in the class are abstract? Good guess. The class is said to be 100% abstract, right? Keeping this in mind, Java has provided us with a special entity to use whenever all the methods are known to be abstract. This special entity is what we call as **interface**.

An interface is nothing but an entity whose methods are all abstract by default. The good thing with interfaces is that, you need not declare the methods using abstract keyword anymore. The moment we use an interface, JVM knows the methods are implicitly abstract. Following is the syntax for declaring an interface with methods.

```
public interface <interfacename>
{
    Method 1 ();
    Method 2 ();

    .
    .
    Method N();
}
```

The keyword used to declare an interface is **interface**. Once we defined an interface with all the methods, are we done? No. We need to write a class that *implements* all the methods in the interface. Make sense? Yes, it does. Otherwise, the existence of an interface doesn't make any sense unless someone implements its methods. So the rule is, if we have an interface, we need to write a class that **implements** the interface. Good. Let's look at a classical Database example using interfaces.

Problem: As a designer for databases, I know how to design a database, such as what elements a database must have and all that good stuff. However, I don't know how to create a working database based on this design. But there are good number of people out there who knows how to implement database based on the design. So, I give my design to those people, who will then use their expertise to implement the database. Problem solved. Cool. Let's implement this scenario.

Designer

This guy knows what elements a database must have and cannot implement any of these elements (methods). So he creates a `Database` interface with all the methods a database must have.

Database Creators: (Oracle, MySQL etc)

These guys use the above design (`Database` interface) and provide implementation of all the elements (methods)

Users: (Test class)

These are the users of the database who use the above created databases like Oracle and MySQL. We use them by creating objects of databases and invoking the methods.

Take a look at the code in listing 4.18.

Listing 4.18a (`Database.java`) Interface defining the methods.

```
package objectconcepts;

public interface Database {

    public void getConnection();
    public void createStatement();
    public void executeQuery();
    public void shutDown();
}
```

The above Database interface defined four methods without any implementation. Notice that we didn't use any **abstract** keyword here as they are implicitly abstract by default. Since the above interface is of no use alone, we need to write a class to implement the above interface. Now look at the implementation class shown in listing 4.18b.

Listing 4.18b (`Oracle.java`) Class implementing the database interface.

```
package objectconcepts;

public class Oracle implements Database {
    static {
        System.out.println("Oracle DB Initialized");
    }

    public void getConnection() {
        clearCache();
        System.out.println("Got the connection from Oracle DB");
    }

    private void clearCache() {
        System.out.println("Cache Cleared");
    }

    public void createStatement() {
        System.out.println("Created Oracle Statement");
    }

    public void executeQuery() {
        System.out.println("SQL Executed");
    }

    public void shutDown() {
        System.out.println("Oracle Shutdown Completed");
    }
}
```

As you can see the above code, we defined a class named `Oracle` that implements all the methods in the interface. For a class to implement an interface, it must use the

keyword **implements** just like we used **extends** keyword when creating a child class. See the following class declaration.

```
public class Oracle implements Database
```

This class besides implementing the methods in the interface can also have its own methods. No restriction, ok. At the bare minimum it must implement all the methods of the interface. In our case, the Oracle class also implemented its own private method `clearCache()`, which it uses as a helper method while implementing the interface methods. That's it. We now have a class with all the functionality implemented, and we can now create the objects of this class to invoke the methods. Listing 4.18c shows the test class.

Listing 4.18c (`OracleUser.java`) Test class for using Oracle database.

```
package objectconcepts;

public class OracleUser {

    public static void main(String args[]) {

        // Create the implementaton object
        Oracle db = new Oracle();
        // Invoke the methods.
        db.getConnection();
        db.createStatement();
        db.executeQuery();
        db.shutdown();
    }
}
```

The test class is pretty straight forward. It simply created an instance of `Oracle` class, and invoked the methods. Just like we created `oracle` class, we can create another class from the same interface and implement the methods in its own way as shown below:

```
public class MySQL implements Database{

    // implement all the methods
}
```

The moral of the story is, given an interface, we can write multiple implementation classes for the same interface.

When do we use abstract classes and interfaces?

Good question. If you have a situation where you know what methods a class must have, but cannot implement any of the methods, then use an interface. You must then write a class that *implements* the interface, and provide the implementation for all the methods. Finally, use this class to create objects.

If you have a situation where you know what methods a class must have, but cannot implement some of the methods, then use an abstract class. You must then write a child class that *extends* the abstract class, and provide the implementation for those some methods that are not implemented. Then use this class to create objects.

The only difference between an abstract class and interface is, in the former "some" methods are unimplemented, and in later "all" methods are unimplemented. This is a very important interview question. So learn it twice. Let me ask you a question here. Is the following statement true or false?

"Like a child class **cannot** inherit from more than one parent class, a class **cannot** implement more than one interface." True/False

The answer is false. Classes can implements 'n' number of different interfaces. The syntax is as shown below:

```
public class Test implements A,B,C
```

where A, B, C are three different interfaces.

The only rule is that, if a class implements more than one interface, it must implement all the methods from all the interfaces. Make sense? Yes it does. So, let's look at the next example that uses multiple interfaces. See listing 4.19.

Listing 4.19a (`Database.java`) Database interface.

```
package objectconcepts;

public interface XYZDatabase {
    public void getConnection();
    public void executeQuery();
}
```

Listing 4.19b (Driver.java) Driver interface.

```
package objectconcepts;

public interface Driver {

    public void loadDriver();

}
```

As you can see the above code, the two interfaces defined three methods. These are pretty straight forward. Now look at the code in listing 4.19c that implements both the above interfaces.

Listing 4.19c (XYZDatabase.java) Class implementing both the interfaces.

```
package objectconcepts;

public class XYZDatabase1 implements Driver, XYZDatabase {

    public void loadDriver() {
        System.out.println("XYZ Driver loaded");
    }

    public void getConnection() {
        System.out.println("Got Connection");
    }

    public void executeQuery() {
        System.out.println("Executed Query");
    }
}
```

The above class implements both the interfaces. Notice that we implemented all the three methods from the two interfaces. Once we have this class, we can write a test class as shown below and invoke its methods.

Listing 4.19d (XYZDatabaseUser.java) Test class invoking all the methods in the interface.

```
package objectconcepts;

public class XYZDatabaseUser {

    public static void main(String args[]) {
        XYZDatabase db = new XYZDatabase();

        db.loadDriver();
        db.getConnection();
    }
}
```

```
        db.executeQuery();
        db.executeQuery();
    }
}
```

All we did in the above test class is invoked the methods in the implementation class. Compile and execute the test class to see the following result.

```
XYZ Driver loaded
Got Connection
Executed Query
Executed Query
```

This is how you need to use interfaces, a very powerful concept in Object Oriented design. There will hardly be any real world Java applications that ever use interfaces and abstract classes. So, read them twice.

The Object Class

When we learned about inheritance, we have seen how to create a child class from parent classes using **extends** keyword. This logically means, if a class uses extends keyword in its declaration, then we can be 100% sure that there is a parent class, right? Let me ask you a question. If a class doesn't use extends keyword, does this mean it has no parent? Is it an orphan class? The answer is No. Surprised? We all know we have parents because we physically see them. But, there is one ultimate parent whom we cannot see in our life who is none other than 'God'. Likewise in Java, there is an ultimate parent called **Object** class. If a class doesn't extend from any class, it's an **implicit** child of Object class. This is the ultimate parent of any inheritance hierarchy. This is why I say Java is close to real life.

By saying that any class is a child of **Object** class if it doesn't use the **extends** keyword, based on inheritance rule it must inherit the methods from the Object class, right?, Before we verify the this fact, let's first look at what methods the Object class has.

Table 4.1 lists the methods in the Object class

Table 4.1 Object class API

Method	Description
Object clone()	Creates and returns a copy of this object.
boolean equals(Object o1)	Indicates whether some other object is "equal to" this one.
void finalize()	Called by the garbage collector on an object when garbage

	collection determines that there are no more references to the object.
<code>Class getClass()</code>	Returns the runtime class of an object.
<code>int hashCode()</code>	Returns a hash code value for the object.
<code>void notify()</code>	Wakes up a single thread that is waiting on this object's monitor.
<code>void notifyAll()</code>	Wakes up all threads that are waiting on this object's monitor.
<code>String toString()</code>	Returns a string representation of the object.
<code>void wait()</code>	Causes current thread to wait until another thread invokes it.

Out of all the above methods, the only method that we are interested and most widely used is the `toString()` method even though all the above methods are inherited. So, to verify the fact that these methods are indeed inherited into any class that doesn't extend from any class, let's execute the following programs.

Listing 4.20a (`PhoneNumber.java`) Simple class.

```
package objectconcepts;

public class PhoneNumber {

    public String getNumber() {
        return "12345678";
    }
}
```

Listing 4.20b (`PhoneTest.java`) Test class.

```
package objectconcepts;

public class PhoneTest {

    public static void main(String args[]) {
        PhoneNumber ph = new PhoneNumber();

        String number = ph.getNumber();
        // Invoke the inherited method
        String str = ph.toString();

        System.out.println(str);
        System.out.println(number);
    }
}
```

If you look at the `PhoneNumber` class, it just defined the `getNumber()` method. Since this class didn't extend any class, it inherits from the `Object` class, and therefore the class will have all the methods from `Object` class plus its own method. One such inherited method from the `Object` class is the `toString()` method which returns a `String` value of the object. To verify this fact, the test class created the object referenced by `ph` and invoked both `getNumber()` as well as `toString()` invisible method. If you compile this class, you'll not see any compilation errors which proves that the method `toString()` is inherited from the `Object` class. The output will be some funky alphanumeric for the first one, and the phone number for the second like as shown below:

```
abcd1234h@PhoneNumber  
12345678
```

Just remember that `Object` class is the ultimate parent of any class that doesn't extend from any other class. This completes all the concepts we need to know to work with objects. So, let's summarize all we learned until now in this chapter.

Summary

- ✓ Object Oriented Programming is “Programming using Objects”.
- ✓ An Object is an instance of class. A single class can have ‘n’ number of objects of the same class.
- ✓ The keyword **static** differentiates between Class level methods and Object level methods in the sense Object level methods are non-static and class level methods are static.
- ✓ All object methods can access static class methods as shared methods, but the reverse is not true.
- ✓ Objects interact with each other by invoking each others methods.
- ✓ Encapsulation hides the details and presents the essentials of an object. This is achieved by using the keywords **private** and **public**.
- ✓ Objects can be passed as arguments to methods. In this case, we pass the reference (pointer) of the object, but not the Object itself.
- ✓ Method overloading allows more than one method to share the same name in a single class.
- ✓ A Constructor is used for object initialization, just a static block is used for class level initialization.
- ✓ A Constructor is like a special method whose name is same as class name and doesn't have a return type.
- ✓ JVM automatically invokes the constructor at the time of creating object using **new** keyword.
- ✓ Constructors can also be overloaded like methods. The rules are same.

- ✓ For one constructor to invoke another constructor, we use **this** keyword and pass arguments within the parenthesis. Ex: `this(10)`, `this(10, 'a')` etc.
- ✓ Inheritance allows one class to inherit the properties of another class. Using this, a child class can inherit all the **non-private** members of the parent class.
- ✓ Inheritance eliminates the code duplication.
- ✓ Method Overwriting allows a child class to define the exact same method as the one in the parent class.
- ✓ **super** keyword is used by a child class to explicitly invoke the parent class methods.
- ✓ A default constructor is an empty constructor with no arguments and no body.
- ✓ If a constructor is not defined in a class, JVM provides with a default constructor.
- ✓ When a child class object is created with any child class constructor, JVM always invokes the parent class default constructor first before invoking the child constructor.
- ✓ For the child class constructor to explicitly invoke an appropriate parent class constructor, it can use **super** keyword. Ex: `super()`, `super(10)` etc.
- ✓ An **abstract** method is a method that doesn't have any body. In such cases, the method must be declared using **abstract** keyword.
- ✓ If a class has any abstract methods, the class must also be declared **abstract**.
- ✓ An abstract class however can also have some methods implemented. This is why, abstract class is also called as partially implemented class.
- ✓ If a class is abstract or partial, we cannot create objects of its class using **new** keyword.
- ✓ We need to create a child class from abstract class and implement all the abstract methods. We then have to use this class to create objects.
- ✓ If a **final** keyword is used with a method, then it cannot be overwritten in the child class.
- ✓ If a class is **final**, we cannot create child classes from it.
- ✓ If a variable is **final**, then its value cannot be changed. It's a Constant.
- ✓ An **interface** is an entity whose all methods are implicitly **abstract**.
- ✓ A class must **implement** the interface and define the body for all the methods in the interface.
- ✓ A class can implement any number of interfaces. But a class can never have more than one parent class. Multiple inheritance is strictly not allowed in Java.
- ✓ The ultimate parent of any class is the **Object class**. If a class doesn't use extend keyword in its declaration, it is implicitly a child of Object class, and implicitly has all the methods in the Object class.
- ✓ The most important method in the Object class is **toString()** method.

Time to play 50-50

1. Which of the following statement is true
 - a) Class is an instance of Object
 - b) Object is an instance of Class
2. Which of the following keyword is used for distinguishing class methods and object methods
 - a) void
 - b) static
3. Which of the following is an object level method?
 - a) void display();
 - b) static void display();
4. Which of the following statement creates an Object?
 - a) Test t = new Test();
 - b) Test t = Test();
5. Which of the following is a legal way of invoking an Object level method?
 - a) Test t = new Test();
t.display();
 - b) Test.display();
6. Which of the following are valid overload methods?
 - a) void test() and int test()
 - b) void test() and int test(int k)
7. Given a class Test, which of the following is a valid constructor?
 - b) int Test()
 - c) Test()
8. Which of the following keyword is used for invoking constructors of same

class?

- a) super
- b) this

9. Which of the following keywords is used for hiding the details of Object?

- a) public
- b) private

10. A simple class with variables and getter methods and setter methods is called as

- a) Java Bean or POJO
- b) Enterprise Java Bean

11. Which of the following is valid for overwriting a method?

- a) Same name, same return type and same arguments
- b) Same name, same return type, different arguments

12. Which of the following is a valid abstract method?

- a) abstract void display(){ }
- b) abstract void display();

13. Which of the following is a valid class?

- a) class Test{
 abstract void display();
 }
b) abstract class Test{
 abstract void display();
 }

14. Which of the following is a valid interface?

- a) interface Test{
 void display() {
 System.out.println("Test");
 }
 }

```
b) interface Test{  
    void display();  
}
```

15. Which of the following is a valid class declaration

- a) public class Test extends Test1,Test2
- b) public class Test implements Test1,Test2

I am sure you cracked all of the above in a minute. The only reason I want to give you just two options is that you don't have to think out of the blue. More the options, more the thoughts, more the questions, and more the confusion which is what I intend to avoid. Keeping life simple and easy, way to live.

Interview Questions

Question: Can you tell me what are the important OOPS concepts?

Answer: The most important OOPS concepts are Encapsulation, Inheritance, and Polymorphism.

Encapsulation hides the details and presents the essentials of an object using private and public keywords.

Inheritance is a technique which allows one class to inherit the variables and methods of another class. One is called the Parent class and the other is called the Child class. A child class can only inherit the non-private members of the parent class only.

Polymorphism allows methods in a class to be represented in multiple forms. This is done through Overloading and Overwriting.

Question: What is Overloading and Overwriting ?

Answer: Please refer to the Summary section.

Question: What is the difference between abstract and final keyword?

Answer: Please refer to page 114.

Question: What is the difference between **this** and **super**

Answer: **this** keyword is used for invoking the methods, variables and constructors of the same class. **super** keyword is used by the child class to invoke the variables, methods and constructors of parent class.

That's all we need to know in this chapter. This completes all the concepts in Object Oriented Programming. Wow. Let me tell you one thing. Though there are plenty of concepts, all of them are very simple and easy to understand. Moreover, the terminology is almost what we speak in our day to day life, which is why Object Oriented Programming is so fun and interesting. I am sure you thoroughly enjoyed this chapter and more importantly with passion. If possible read this chapter twice before going to next chapter. Java programming is all about creating objects and invoking methods.

Let's move one more step forward.

Chapter 5

Referential Polymorphism

This chapter teaches you on how to use parent class and interface references. By the end of this small chapter, you will be able to know how and when to use inheritance and interfaces in real world applications. The ideas and examples in this chapter are totally based on what we learned in the previous chapter. We are not trying to reinvent the wheel here, ok. Understanding the concepts in this chapter is utmost important since most of the real world Java applications are based on Object Oriented Design whose primary weapons are inheritance and interfaces.

Chapter Goals

- ✓ Understanding parent class references
- ✓ Understanding Interface references
- ✓ How to restrict access to methods using Inheritance
- ✓ How to restrict access to methods using Interfaces

Environment Setup

All the programs in this chapter should be stored in the following directory.

C:/JavaTraining/Chapter5/objectRefs

objectRefs is a package in which we will store the programs. To compile the programs, move to the following root directory and get ready.

C:/JavaTraining/Chapter5

Introduction

In Java, it's all about different objects invoking methods in different objects. Methods are like sophisticated operations and sometimes it's important that we restrict access to some of the operations to certain objects and allow the same to some other group of objects. In the previous chapter, we learned something called encapsulation where access to methods is restricted using **private** keyword. However, the **private** keyword restricts access to **all** the outside objects which is not what we wanted. We wanted to restrict access to **some** objects and allow access to **some** others. So, using **private** keyword is definitely not the solution.

The solution to this problem is using inheritance and interfaces. In the previous chapter we learned how inheritance and interfaces allows us to have multiple implementations of the same methods, and at the same time, how to eliminate duplicate code. This is one side of inheritance and interfaces. The other side of it is what the most powerful one is, **controlling access to methods**, one of the key application in Object Oriented Design. Let's see how this is achieved.

We already know how to create objects from a parent class and a child class, and what methods we can invoke on the same. Just as a recap, we'll see one example and invoke the methods in both the classes. Take a look at the code in listing 5.1.

Listing 5.1a (`Parent.java`) A simple parent class.

```
package objectrefs;

public class Parent {

    public void formatMessage() {
        System.out.println("Message Formatted");
    }

    public void sendMessage() {
        System.out.println("Message Sent");
    }
}
```

As you can see from the above code, the `Parent` class defined two methods namely `formatMessage()` and `sendMessage()`. Its child class is shown in listing 5.1b.

Listing 5.1b (Child.java) A simple child class.

```
package objectrefs;

public class Child extends Parent {

    public void formatMessage() {
        System.out.println("New Message Formatted");
    }

    public void recieveMessage() {
        System.out.println("New Message Formatted");
    }
}
```

As you can see from the above code, the child class overwrote the `formatMessage()` and defined new method namely `recieveMessage()`. Now take a look at the following two simple cases.

Case 1: A parent object is created as shown below.

```
Parent p = new Parent();
```

Using reference variable 'p', we can **only** invoke the methods in parent class as shown below:

```
p.formatMessage();
p.sendMessage();
```

Case 2: A child object is created as shown below.

```
Child c = new Child();
```

Using reference variable 'c', we can **only** invoke the methods in child class as shown below:

```
c.formatMessage(); // Overwritten method
c.sendMessage(); // Inherited from parent
c.recieveMessage(); // Its own method
```

No confusions, right? We simply invoked the respective methods on each object. This is what we learned in the previous chapter. Good.

Loose Coupling

One of the primary goals in Object Oriented Design is to design the system in such a way that various objects are loosely coupled with each other to increase the flexibility of the overall system. To understand loose coupling, consider the following example.

Let's say my company want to build a *search engine that uses web tool* to search for resources over the web. To develop this system, I need two classes namely SearchEngine and WebTool. Look at the following code in listing 5.2 for both the classes.

Listing 5.2a (WebTool.java) A simple web tool class.

```
package objectrefs;

public class WebTool {

    // Search method
    public void search() {

        System.out.println("Found 10 results in Web");
    }
}
```

I am sure you know how the above code works. The WebTool class defined a search() method that displays the search results. This is the method our search engine class invokes to display the search results. Now look at the search engine class in listing 5.2b.

Listing 5.2b (SearchEngine.java) A simple search engine class.

```
package objectrefs;

public class SearchEngine {

    // This method takes the tool as parameter and performs
    // the search using it.
    public void performSearch(WebTool tool) {

        tool.search();
    }
}
```

As you can see from the above code listing, the SearchEngine class defines a method performSearch() that takes the WebTool object as a parameter, and invokes the

`search()` method using the `tool` reference to display the results. We know all this from the previous chapter. Look at the test class in listing 5.2c.

Listing 5.2c (`SearchEngineTest.java`) A simple search engine test class.

```
package objectrefs;

public class SearchEngineTest {

    public static void main(String args[]) {
        // Create the tool
        WebTool tool = new WebTool();

        // Create the search engine
        SearchEngine engine = new SearchEngine();

        // Pass the tool as a parameter to engine.
        engine.performSearch(tool);
    }
}
```

The test class created a `WebTool` object and passed it as a parameter to `performSearch()` method of the search engine object as shown below:

```
engine.performSearch(tool);
```

The above method internally invokes the `search()` method in the `WebTool` class. Executing the test class will give you the following result.

```
Found 10 results in Web
```

So far so good. Let's say my client who is using the above code for searching comes to me and say, "Hey! Right now you are searching only using `WebTool` which is good. Sometimes we also want to search using another tool called `ImageTool` to search for images". To fulfill this requirement, we need to implement the following two changes:

1. Create a class called `ImageTool` just like `WebTool` as shown below:

```
package objectRefs;

public class ImageTool{
    public void search(){
        System.out.println("Found 10 Image results");
    }
}
```

2. Add another overloaded `performSearch()` method in the `SearchEngine` class that takes `ImageTool` as an argument, and again invoke its `search()` method as shown below:

```
package objectrefs;

public class SearchEngine{

    public void performSearch(WebTool tool) {
        tool.search();
    }
    public void performSearch (ImageTool tool) {
        tool.search();
    }
}
```

Using the above `SearchEngine` class, my client will then be able to search using either tool. So, every time my client comes with a new search tool, I need to add a new `search()` method in the search engine class, right? We can say this class as a tightly-coupled class, since we need to update it every time our client comes with a new requirement, ok. This is not a good design. So the question is, what is the best design for the above class? To get to the best design, our goal is two fold:

- a) Eliminate the tight coupling and
- b) Use just one generic `performSearch()` method for all the searches.

Trust me, we will definitely reach our design goal, but before that we need to know something about parent class and child class references.

Parent class referencing the Child Object

In Java, a child class object can be referenced by a parent class variable as shown below:

```
SearchTool p = new WebTool();
```

The above statement is *legal only if* `WebTool` is a child class of `SearchTool`. Let me ask you a question here. Using the reference variable 'p', which methods can we access?

- a) All methods in `SearchTool` class
- b) All methods in `WebTool` class

I am sorry to say that both are wrong. Here is the rule. Using reference 'p', we can invoke,

- 1) Only the methods in parent class where,
 - 1.1) Non-overwritten methods are invoked from the parent class, and
 - 1.2) Overwritten methods are invoked from the child class.

Confusing? Let's see an example and the confusion will fly away. See the code in listing 5.3

Listing 5.3a (`SearchTool.java`) A simple search tool class.

```
package objectrefs;

class SearchTool {

    public void format() {
        System.out.println("In Parent Format");
    }

    public void search() {
        System.out.println("In Parent Search");
    }
}
```

Listing 5.3b (`EnhancedWebTool.java`) A simple web tool class.

```
package objectrefs;

class WebTool {

    // Overwriting the search method
    public void search() {
        System.out.println("In Child Search");
    }

    // Its own method
    public void display() {
        System.out.println(" In display method");
    }
}
```

With the above two classes, when we create a child object as shown below,

```
SearchTool tool = new WebTool();
```

Rule 1 says "Only the methods in parent class". So, the only methods we can invoke using 'tool' reference are `format()` and `search()` as shown below.

```
tool.format();
tool.search();
```

Since, `format()` method is **not overwritten** in the child class, it invokes this method from the parent (Rule 1.1) . The result is

In Parent Format

Since `search()` method is **overwritten** in the child class, it invokes this method from the child class (Rule 1.2) . So the result of this invocation will be

In Child Search

Here is a simple technique JVM uses to invoke the methods when a child object is referenced with a parent class variable. It first goes to parent class method and checks if the method is overwritten in the child class. If so, it invokes the child class method. Otherwise, it invokes the parent class method. This should take away your confusion.

From the above statement, if we **don't override** any methods in the child class, then all the methods will be invoked from the parent class, right? In such cases, following two statements are identical:

```
Parent p = new Parent(); and  
Parent p = new Child();
```

Tip: Use the parent = child relationship, only when the parent class is **abstract**.

Using the above concept, let's improve our previous search engine example. Before we understand the program, let's write the code and then look at the details. See listing 5.4.

Listing 5.4a (`SearchTool.java`) An abstract search tool class.

```
package objectrefs;  
  
public abstract class SearchTool {  
  
    // Abstract search method  
    public abstract void search();  
}
```

As you can see from the above code, we created an abstract class called `SearchTool` with an abstract `search()` method. Any search tool who knows how to search must extend the abstract class and implement the `search()` method. See listing 5.4b in which we create a child class of this class.

Listing 5.4b (`WebTool.java`) Class extending the abstract class.

```
package objectrefs;

// Tool class extending the abstract class
public class WebTool extends SearchTool {

    public void search() {

        System.out.println("Found 10 results in Web");
    }
}
```

The above `WebTool` class is inherited from the abstract `SearchTool` class and implemented the `search()` method. This is a typical parent child implementation. Now look at the `SearchEngine` class in listing 5.4c.

Listing 5.4c (`SearchEngine.java`) Search engine class.

```
package objectrefs;

public class SearchEngine {

    public void performSearch(SearchTool tool) {

        tool.search();
    }
}
```

The above `SearchEngine` class instead of taking `WebTool` as an argument, used its parent class `SearchTool` as a parameter and invoked the `search` method. Now look at the test class shown in listing 5.4d.

Listing 5.4d (`SearchEngineTest.java`) Test class for search engine.

```
package objectrefs;

public class SearchEngineTest {

    public static void main(String args[]) {

        SearchEngine engine = new SearchEngine();

        engine.performSearch(new WebTool());
    }
}
```

To understand the above code, look at the two snippets of code shown below:

```
engine.performSearch(new WebTool());
```

```
public void performSearch(SearchTool tool)
{
    tool.search();
}
```



As you can see from the above, the `WebTool` object is passed into the `SearchTool` leading us to the following assignment.

```
SearchTool tool = new WebTool();
```

So, when the `performSearch()` method invokes the `search()` method using its argument `tool` reference, it actually invokes the method in the `WebTool` object passed, right? This is because the `search()` method is overwritten in the `WebTool` class and according to our rule, overwritten methods are invoked from the child class.

Compile and execute the above classes, and the output will be,

```
Found 10 results in Web
```

With the above code, did we eliminate tight coupling? Yes we did. To prove this, let's fulfill our client's requirement who also want to use `ImageTool` to search for images. With the above code, all we need to do is create this class just like the `WebTool` class as shown below:

Listing 5.4e (`ImageTool.java`) New search tool class.

```
package objectrefs;

public class ImageTool extends EnhancedSearchTool {
    public void search() {
        System.out.println("Found 10 Image results");
    }
}
```

Trust me; you don't have to change a single piece of line in the `SearchEngine` class. You don't have to add any new method like we did in example 5.2 (Page 133). If my client needs to use this tool, all he has to do is simply pass the object of `ImageTool` class as shown in listing 5.4f.

Listing 5.4e (`SearchEngineTest.java`) Updated test class for search engine.

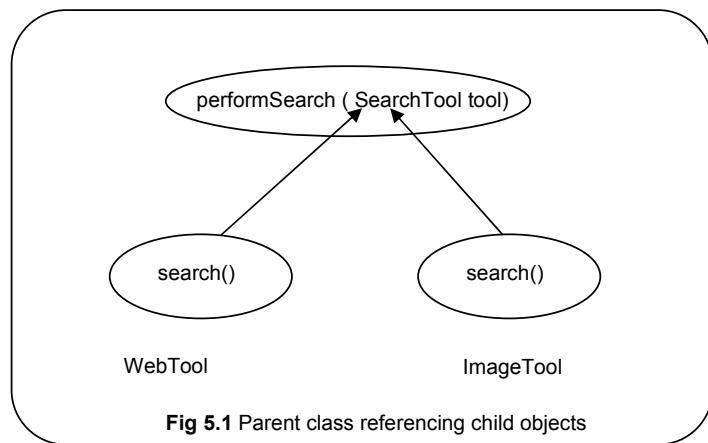
```
package objectrefs;

public class SearchEngineTest {
    public static void main(String args[]) {
        SearchEngine engine = new SearchEngine();
        engine.performSearch(new WebTool());
        // Updated functionality
        engine.performSearch(new ImageTool());
    }
}
```

Our `performSearch()` method in the `SearchEngine` class is now 100% generic. *By changing the argument to parent class 'SearchTool', it can invoke the `search()` methods on any child search tools (WebTool, ImageTool etc).* Now we should agree that our engine class is loosely coupled with the `WebTool` and `ImageTool` classes. Even if we update the `WebTool` code or completely stop using it, we still don't have to modify our search engine class. You agree with me?

The funda here is using the parent class references as method arguments instead of child class references. This is the trick. In most of the real world applications, the arguments of a method will always be a parent class references, and who ever invokes the method, should always pass the child objects.

Fig 5.1 shows the design of our search engine class.



Accessing methods using Inheritance

As I said before that we can restrict access to methods using inheritance and interfaces, let's first look at this using inheritance. To clearly understand the concept, let's now look a more practical example and see different cases for accessing methods in a class.

Case 1: Client A needs a search tool with the following 6 search methods

```
searchWeb();searchImages();searchNovel();  
searchLdap();searchOracleDB(); searchMySqlDB();
```

The above requirement is very simple to implement. All we need to do is write a class that implements all the above methods, right? So, our class will look as shown in 5.5a.

Listing 5.5a (`PremierSearchTool.java`) Case 1 search tool

```
package objectrefs;  
  
public class PremierSearchTool {  
  
    public void searchWeb() {  
        System.out.println("Found 10 Web results");  
    }  
  
    public void searchImages() {  
        System.out.println("Found 10 Image results");  
    }  
  
    public void searchNovel() {  
        System.out.println("Found 5 results in Novel");  
    }  
  
    public void searchLdap() {  
        System.out.println("Found 5 results in LDAP");  
    }  
  
    public void searchOracleDB() {  
        System.out.println("Found 2 results in Oracle DB");  
    }  
  
    public void searchMySqlDB() {  
        System.out.println("Found 3 results in MySQL DB");  
    }  
}
```

Now, let's write a utility class that will return the above search tool to Client A as shown in listing 5.5b.

Listing 5.5b (`SearchTools.java`) Utility class to return the search tools for clients.

```
package objectrefs;

public class SearchTools {

    public static PremierSearchTool getClientASearchTool() {

        // Create and return the search tool for client A
        PremierSearchTool p = new PremierSearchTool();

        return p;
    }
}
```

Client A can use the above class and invoke the method `getClientASearchTool()` to get his search tool. Note that the above method is declared static (class level) so that client code can call the method directly using class name. His code will therefore look something like the one shown in listing 5.5c.

Listing 5.5c (`ClientASearchToolTest.java`) Client A search code.

```
package objectrefs;

public class ClientASearchToolTest {
    public static void main(String args[]) {

        PremierSearchTool tool = SearchTools.getClientASearchTool();

        // Invoke the search methods

        tool.searchWeb();
        tool.searchImages();
        tool.searchNovel();
        tool.searchLdap();
        tool.searchOracleDB();
        tool.searchMySqlDB();

    }
}
```

Compile and execute the above test class to see the following result.

```
Found 10 Web results
Found 10 Image results
Found 5 results in Novel
Found 5 results in LDAP
Found 2 results in Oracle DB
Found 3 results in MySQL DB
```

Now let's look at another requirement.

Case 2: Client B wants a superior search tool that has *enhanced* searching in **Novel** and **Ldap**, and also a *new* search for searching **libraries**. All other searches must remain the same.

For this requirement, what we can do is, create a child class from the above `PremierSearchTool` class, *overwrite* `searchNovel()` and `searchLdap()` methods, and *create* a new method `searchLibraries()`, right? This is nothing but reusing the code using inheritance. So, my search tool class for this requirement will be as shown in listing 5.5d.

Listing 5.5d (`SuperiorSearchTool.java`) Client B search tool.

```
package objectrefs;

public class SuperiorSearchTool extends PremierSearchTool {

    // Overwritten methods
    public void searchNovel() {

        System.out.println("Performing enhanced Novel Search");
        System.out.println("Found 20 Novel results");
    }

    public void searchLdap() {

        System.out.println("Performing enhanced LDAP Search");
        System.out.println("Found 15 results in LDAP");
    }

    // New search Method
    public void searchLibraries() {

        System.out.println("Found 9 results in Libraries");
    }
}
```

As you can see from the above code, all we did is created `SuperiorSearchTool` class from `PremierSearchTool` class and,

- ✓ Overwrote the two methods for novel search and ldap search
- ✓ Added one new search method for library search.

Non-overwritten searches (four of them) in the `PremierSearchTool` class will simply be inherited into `SuperiorSearchTool` by the rule of inheritance. So, the `SuperiorSearchTool` now has 7 methods with 4 old search methods from parent and 2 updated methods and 1 new method. Hence this is really a superior search tool. My utility class will return this search tool to Client B as shown below in bold.

```
package objectrefs;

public class SearchTools{

    public static PremierSearchTool getClientASearchTool(){
        PremierSearchTool p = new PremierSearchTool();
        return p;
    }
    public static SuperiorSearchTool getClientBSearchTool(){
        SuperiorSearchTool s = new SuperiorSearchTool ();
        return s;
    }

}
```

Client B can use the above class and invoke the method `getClientBSearchTool()` to get his search tool. His code will therefore look something like the one shown in listing 5.5e.

Listing 5.5e (`ClientBSearchToolTest.java`) Client B search code.

```
package objectrefs;

public class ClientBSearchToolTest {

    public static void main(String args[]) {
        // Get the search tool.
        SuperiorSearchTool tool = SearchTools.getClientBSearchTool();

        // Old Searches
        tool.searchWeb();
        tool.searchImages();
        tool.searchOracleDB();
        tool.searchMySqlDB();

        // Updated searches
        tool.searchNovel();
        tool.searchLdap();

        // New Search
        tool.searchLibraries();
    }
}
```

Compile all the classes and execute the above test class to see the following result.

```
Found 10 Web results
Found 10 Image results
Found 2 results in Oracle DB
Found 3 results in MySQL DB
```

```
Performing enhanced Novel Search
Found 20 Novel results
Performing enhanced LDAP Search
Found 15 results in LDAP
Found 9 results in Libraries
```

Time for a small quiz. Bob is the creator of search tools and Harry is the user of the search tools.

Based on the above statement, answer the following questions.

1. If Harry should only be allowed to use the following 6 search methods, which of the following statement should he be using to create the tool object?

```
searchWeb,searchImages,searchNovel(old),searchLdap(old),searchOracleDB,searchMySqlDB
```

- a) PremierSearchTool ptool = new PremierSearchTool();
- b) SuperiorSearchTool stool = new SuperiorSearchTool();

2. If Harry should only be allowed to use the following 7 search methods, which of the following statement should he be using to create the tool object?

```
searchWeb,searchImages,searchNovel(updated),searchLdap(updated),searchOracleDB,searchMySqlDB,searchLibraries (new)
```

- a) PremierSearchTool ptool = new PremierSearchTool();
- b) SuperiorSearchTool stool = new SuperiorSearchTool();

I am sure you knocked out the questions in a sec. Very good. Keep it going. The answers are 'a' and 'b'.

Case 3: Client C requires the following 6 search methods only.

```
searchWeb,searchImages,searchNovel(updated),searchLdap(updated),searchOracleDB,searchMySqlDB.
```

This case is same as Case 2, except that he doesn't need the library search. Let's say we used the following statement to create the search tool in our utility (SearchTools) class.

```
PremierSearchTool ptool = new PremierSearchTool()
```

With this he can use all 6 searches; however the *novelsearch* and *ldapsearch* will be the old ones. But the requirement says we need the updated ones, right? So this is incorrect.

Say we use the following statement to create the search tool in our utility (SearchTools) class

```
SuperiorSearchTool stool = new SuperiorSearchTool();
```

With this he can use all the required 6 searches with updated novel and ldap searches as well. But you know what, he can also use the new *searchLibraries* and he should not be allowed to search libraries, right? So, this too is incorrect

If we have statement that allows access to all the methods in the PremierSearchTool with only the overwritten methods from the SuperiorSearchTool, then it is the right answer. So, do we have one? I am sure you guessed the answer. This is parent = child relationship we read in the previous section. Following is the statement and correct answer.

```
PremierSearchTool tool = new SuperiorSearchTool();
```

My utility class will return the search tool to Client C as shown below in bold:

```
package objectRefs;

public class SearchTools{

    public static PremierSearchTool getClientASearchTool() {
        PremierSearchTool p = new PremierSearchTool();
        return p;
    }
    public static SuperiorSearchTool getClientBSearchTool() {
        SuperiorSearchTool s = new SuperiorSearchTool ();
        return s;
    }
    public static PremierSearchTool getClientCSearchTool(){
        PremierSearchTool p = new SuperiorSearchTool ();
        return p;
    }
}
```

Client C can now use the above class and invoke the method `getClientCSearchTool()` to get his search tool. His code will therefore look something like the one shown in listing 5.5f.

Listing 5.5f (`ClientCSearchToolTest.java`) Client C search code.

```
package objectrefs;
public class ClientCSearchTool {
```

```
public static void main(String args[]) {  
    PremierSearchTool tool = SearchTools.getClientCSearchTool();  
  
    tool.searchWeb();  
    tool.searchImages();  
  
    // These two methods will be called from child class  
    tool.searchNovel();  
    tool.searchLdap();  
  
    tool.searchOracleDB();  
    tool.searchMySqlDB();  
}  
}
```

Compile and execute the test class to see the following result

```
Found 10 Web results  
Found 10 Image results  
Performing enhanced Novel Search  
Found 20 Novel results  
Performing enhanced LDAP Search  
Found 15 results in LDAP  
Found 2 results in Oracle DB  
Found 3 results in MySQL DB
```

When you have parent and a child class you can only get 3 different levels of access to methods in an object.

1. All methods in parent class ONLY (Case 1)
2. All methods in child class ONLY (Case 2) and
3. All methods in parent class with overwritten methods from child class.
(Case 3)

A good design must provide as many as possible levels of access to methods. So what we will do is, implement the same search tool using a better design by using interfaces and see if it provides more levels of access control.

Understanding Interface References

Before we look other cases, let's understand one simple rule with objects and interfaces.

Rule: In Java, you can also assign an object of a class to an interface, provided the class implements that interface.

For instance, if we have a class declaration as shown below,

```
public class X implements A,B,C
```

then the following statements are perfectly legal since class X implements interfaces A,B,C.

```
A a = new X();  
B b = new X();  
C c = new X();
```

With the above statements,

Reference **a** can be used to access only the methods defined in interface A that are implemented in X

Reference **b** can be used to access only the methods defined in interface B that are implemented in X

Reference **c** can be used to access only the methods defined in interface C that are implemented in X

Reference variables a,b,c are called as *interface references* to object of class X.

Accessing methods using Interfaces

Using the above concept of interfaces references, let's see if we can improve our PremierSearchTool and SuperiorSearchTool classes.

As we know that the total search methods we have are seven, let's distribute these 7 methods into four interfaces namely InternetSearch, DirectorySearch, DatabaseSearch and LibrarySearch as shown in listing 5.6a-d.

Listing 5.6a (InternetSearch.java)

```
package objectrefs;  
  
public interface InternetSearch {  
  
    public void searchWeb();  
    public void searchImages();  
}
```

Listing 5.6b (DirectorySearch.java)

```
package objectrefs;
```

```
public interface DirectorySearch {  
    public void searchNovel();  
    public void searchLdap();  
}
```

Listing 5.6c (`DatabaseSearch.java`)

```
package objectrefs;  
  
public interface DatabaseSearch {  
    public void searchOracleDB();  
    public void searchMySqlDB();  
}
```

Listing 5.6d (`LibrarySearch.java`)

```
package objectrefs;  
  
public interface LibrarySearch {  
    public void searchLibraries();  
}
```

Since our `PremierSearchTool` should have six methods (case 1) in the `InternetSearch`, `DirectorySearch` and `DatabaseSearch` interfaces, it will implement all these three interfaces. Make sense?

Our `SuperiorSearchTool` will again be a child class of `PremierSearchTool`, as it needs the four old search methods. Since it must also have the two updated directory methods (novel and ldap) which happen to be the methods in `DirectorySearch` interface, and a library search method which happen to be the method in `LibrarySearch` interface, we will have this class implement these two interfaces. Make sense? This is case 2 requirement.

Now my updated `PremierSearchTool` and `SuperiorSearchTool` classes will be as shown in listing 5.6e-f.

Listing 5.6e (`PremierSearchTool.java`) Updated search class.

```
package objectrefs;  
  
public class PremierSearchTool implements InternetSearch, DirectorySearch,  
    DatabaseSearch {  
  
    // Methods from InternetSearch
```

```
public void searchWeb() {
    System.out.println("Found 10 Web results");
}

public void searchImages() {
    System.out.println("Found 10 Image results");
}

// Methods from DirectorySearch
public void searchNovel() {
    System.out.println("Found 5 results in Novel");
}

public void searchLdap() {
    System.out.println("Found 5 results in LDAP");
}

// Methods from DatabaseSearch
public void searchOracleDB() {
    System.out.println("Found 2 results in Oracle DB");
}

public void searchMySqlDB() {
    System.out.println("Found 3 results in MySQL DB");
}
}
```

Listing 5.6f (SuperiorSearchTool.java) Updated search class.

```
package objectrefs;

public class SuperiorSearchTool extends PremierSearchTool implements
    DirectorySearch, LibrarySearch {

    // Methods from Directory Search
    public void searchImages() {
        System.out.println("Performing enhanced Image Search");
        System.out.println("Found 20 Image results");
    }

    public void searchLdap() {
        System.out.println("Performing enhanced LDAP Search");
        System.out.println("Found 15 results in LDAP");
    }

    // Method from LibrarySearch
    public void searchLibraries() {
        System.out.println("Found 9 results in Libraries");
    }
}
```

If you look at the updated `PremierSearchTool` class in listing 5.6d, the only difference with the previous version of the code is that implements three interfaces in its declaration.

The SuperiorSearchTool class not only **extends** from PremierSearchTool class, but also **implements** two interfaces. In Java, a class can extend from one class, and at the same time can implement any number of interfaces. In this case, it inherits to get the four old search methods, ok.

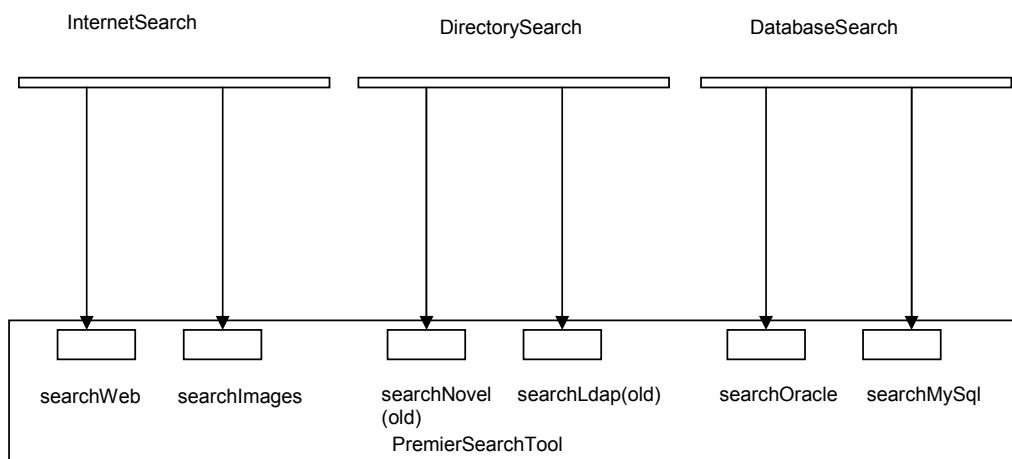
Using the above two updated classes, the previous three cases will also work. You can copy the above two files, and again run all the three test classes for the clients A,B,C. Trust me. You'll get the same results. This is to say that our new design still supports the old clients without breaking them. Now let's see if this design offers some additional cases.

Note: In real world applications, if there is a piece of code that is used by multiple clients, for some reason when you update the code or redesign, make sure that none of the clients functionality is broken. You need to do lot of regression testing when ever you make changes. Ok. Always keep this in mind.

An interface reference is basically a restricted view of the underlying object. Look at the following class declaration:

```
public class PremierSearchTool implements InternetSearch, DirectorySearch,
DatabaseSearch
```

The above class declaration should be viewed as shown in the following figure.



Let's see what meaning the above figure conveys us. `PremierSearchTool` object has six methods shown in the rectangle. This object has three interfaces (views) as shown. Since `InternetSearch` interface defines `searchWeb()` and `searchImages()`

methods, using its reference one *can only access these two methods from the underlying object*. Same is the case with other two interfaces. Now, let's look at three cases shown below:

Case 4: Client D must be allowed to access only the following methods:

```
searchWeb(), searchImages()
```

If you noticed carefully, the above two methods are the `InternetSearch` interface methods. So, I can give the search tool for this client by just creating an object of `PremierSearchTool` and referencing it with `InternetSearch` interface type as shown below:

```
InternetSearch tool = new PremierSearchTool();
```

So, we add the following method to the `SearchTools` utility class to return the search tool as shown below:

```
public static InternetSearch getClientDSearchTool() {  
    InternetSearch p = new PremierSearchTool();  
    return p;  
}
```

Using the above method, Client D search code will be as shown in listing 5.6g.

Listing 5.6g (`ClientDSearchToolTest.java`) Client D test class.

```
package objectrefs;  
  
public class ClientDSearchToolTest {  
  
    public static void main(String args[]) {  
        // Get the tool.  
        InternetSearch tool = SearchTools.getClientDSearchTool();  
  
        tool.searchWeb();  
        tool.searchImages();  
    }  
}
```

Executing the above client test code will produce the following results.

```
Found 10 Web results  
Found 10 Image results
```

The beauty is, though we are giving him the object that has all the six methods, he can only invoke two methods. If he tries to invoke other search methods, he gets a compilation error. Aren't we restricting access to some methods? The `InternetSearch` interface reference acts as a restricted point of entry into the search tool.

Tomorrow if Client D requests access for another search method, all we need to do is declaring the new method in the `InternetSearch` interface. That's it. Isn't this simple?

Case 5: Client E must be allowed to access only the following methods:

```
searchNovel(old), searchLdap(old)
```

'old' means methods in `PremierSearchTool`.

Follow the same steps as we did for Case 4. Take this as home work.

Case 6: Client F must be allowed to access only the following methods:

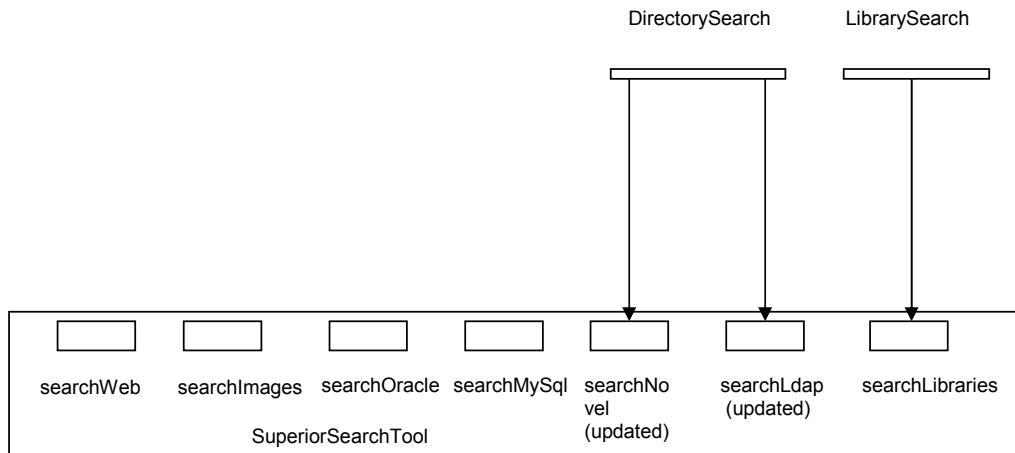
```
searchOracle(old), searchLySql(old)
```

Follow the same steps as we did for Case 4. Take this is home work.

Now, look at the following declaration for `SuperiorSearchTool` class.

```
public class SuperiorSearchTool extends PremierSearchTool implements  
    DirectorySearch, LibrarySearch
```

The above class declaration can be viewed as shown in the following figure.
(See the figure in the next page).



In the above figure, the first four methods are inherited from `PremierSearchTool` class. The other three methods are the methods from `DirectorySearch` and `LibrarySearch` interfaces.

Let's look at 2 more cases.

Case 7: Client G must be allowed to access only the following methods:

```
searchNovel(updated), searchLdap(updated)
```

'updated' means methods in `SuperiorSearchTool`.

For this case, we need to return the `DirectoryInterface` reference of `SuperiorSearchTool` object as shown below:

```
DirectoryInterface tool = new SuperiorSearchTool();
```

You can then write the new method in the `SearchTools` class and Client G test code following the steps in Case 4.

Case 8: Client H must be allowed to access only the following method:

```
searchLibrary()
```

This is like Case 7. Take it as home work.

Here is a simple thumb rule. If a class implements 'n' interfaces, then the number of ways you can access methods using interfaces is also 'n'. Since PremierSearchTool implements three interfaces, it gave us 3 additional cases. Likewise, Superior SearchTool implements two interfaces, so it gave us 2 more cases. The moral of the story is, more the interfaces a class implements, more the access levels you can get.

I know you might have cursed me for not giving the answers for Case 4, 5, and 8. So, here are the answers.

Case 4:

```
DirectorySearch tool = new PremierSearchTool();
```

Case 5:

```
DatabaseSearch tool = new PremierSearchTool();
```

Case 8:

```
LibrarySearch tool = new SuperiorSearchTool();
```

Using the above, you can add methods in the SearchTools utility class, one for each case and then write the test code for each of the classes.

In Java 50% of the method parameters are usually of type interfaces, and 50% are of type abstract parent classes. Just remember the following three important points, and you can be a Java champ.

1. If any method takes Object as a parameter, you can pass *an object of any class* to that method. As an example, consider a class having a method as shown below:

```
public class ArrayList{  
    public void add( Object obj)  
    {  
        // Some Statements  
    }  
}
```

Since we know that Object is the ultimate parent class, the above method can take any object as shown below:

```
ArrayList list = new ArrayList();  
  
list.add (new Box());  
list.add (new Demo());
```

In the above code, we can pass any class object to the add() method without any

compilation errors. This is because both `Box` and `Demo` are the child classes of `Object` class by default.

2. If a method argument is an interface, then *we must pass an object of class that implements the interface*. As an example, consider a class having a method as shown below

```
public class Database{  
    public void setDataSource( DataSource ds) {  
        // Some Statement  
    }  
}
```

Assuming `DataSource` parameter as an interface, to invoke the method, we need to create a class that implements the `DataSource` interface as shown below:

```
public class TestDataSource implements DataSource{  
    public void getConnection() {  
        // Some Statements  
    }  
}
```

We then pass the object of `TestDataSource` as the method parameter .

```
Database d = new Database();  
d.setDataSource ( new TestDataSource() );
```

3. If a method parameter is *any class other than Object class*, then

- ✓ If the class is **not abstract**, then we **pass an object of same class**.
- ✓ If the class is **abstract**, then we **pass an object of its child class**.

Just remember these three rules. Read them 100 times. You can do it.

This concludes the chapter. We learned a lot in this one. I suggest you to read this chapter at least 2 times. Trust me, if you understood the concepts until now, you can take the liberty to even skip the rest of the chapters if you don't have time. You'll still be fine. But I recommend you to spend some time with the later chapters to become even more comfortable with Java.

Summary

- ✓ When a parent class object is assigned to a parent class reference, we can only invoke the methods in the parent class.

Referential Polymorphism

- ✓ When a child class object is assigned to a child class reference, we can only invoke the methods in the child class. This includes the methods inherited from parent class.
- ✓ When a child class object is assigned to a parent class reference, we can only invoke the non-overwritten methods in the parent class and overwritten methods from the child class.
- ✓ An object of a class can be legally assigned to interface reference only if the class implements the interface.
- ✓ If an object is assigned to an interface reference, we can only invoke the methods defined in the interface, even though the object has more methods.

Interview Questions

No one will ask you any questions from this chapter during interview, since this chapter gives you better design ideas (not concepts) using the concepts from the previous chapter.

All right then, catch you in the next chapter.

Chapter 6

Exception Handling

By the end of this chapter, you will know for sure as to how to handle abnormal conditions which we call as exceptions in Java programs. Exception handling is one of the most important aspects of any Java based application.

Chapter Goals

- ✓ Understand what exceptions are.
- ✓ Handling Exceptions.
- ✓ How to throw exceptions
- ✓ How to create custom exceptions.

Environment Setup

All the programs in this chapter should be stored in the following directory.

C:/JavaTraining/Chapter6/exceptions

exceptions is a package in which we will store the programs. To compile the programs, move to the following root directory and get ready.

C:/JavaTraining/Chapter6

Introduction

In the last few chapters, we have written several programs and they all worked fine without exhibiting any *abnormal* behavior, right? They are therefore good programs for behaving well without giving us any trouble. However, in real world applications we come across several abnormal conditions in programs due to which JVM abruptly terminates the program. There is absolutely nothing wrong terminating the program, but it would be really nice if the JVM tells us what the abnormal condition is, due to which it terminated the program. We are kind of asking JVM to be somewhat liberal by being co-operative. We will find out later if the JVM is really co-operative or not.

Sometimes, there will be abnormal conditions which are really **abnormal** that the program cannot recover from. For instance, if the JVM run out of memory space is there anything the program can do to fix the problem? No, right? In such cases, JVM terminating the program is the best solution. However, there will be several situations where the *abnormal conditions from JVM point of view* are *normal from the program point of view*, and we don't want JVM to terminate the program.

In order that JVM not to terminate the program, there must be some mutual understanding between JVM and the Java program. The understanding should be like the Java program requesting the JVM, “*Hey JVM, I know you always try to terminate me in the event of an abnormal condition. But you know what, why don't you tell me what the problem is, so that I'll take the decision whether or not to terminate myself*”.

This is exactly what the JVM does. It wraps all the information about the abnormal condition in an **exception** object, and rubs off its hands by **throwing** the exception object to the Java program. Now the exception object is in Java program's court. The program should sincerely **catch** the exception object, take the appropriate decision and continue with the rest of the program. This is called **Exception handling**. However, if the program **fails to catch** the exception object, then the program terminates.

Now tell me this. Is the JVM co-operative or not? Good. JVM is really with us and gave us all the co-operation that it can possibly give. So, let's use its support and move on.

As a programmer, at any cost we don't want the program to terminate abruptly. For instance, I have a client who is using my program to calculate his taxes. All of sudden the program terminates abruptly and the data is lost. You tell me, what would be his state in that situation. If my program conveyed a meaningful message to him explaining what the problem is, then he will be more than happy, right? This is exactly what we can do if we handle exceptions.

From here onwards, let's write programs that can also handle abnormal conditions (exceptions).

Handling Exceptions in Java

To handle exception objects in our Java program, all we need to do is,

1. Identify the harmful statements that cause the abnormal conditions.
2. Enclose the statements in *exception handling block*.

That's it. Pretty simple, right? Let's say we completed Step 1. To complete the second step, we first need to know is what an exception handling block is.

An exception handling block is formed using the keywords **try** and **catch** as shown below:

```
try {  
    // Harmful Statements  
}  
catch ( exception object ) {  
    // Exception handling statements  
}
```

The exception handling block comprises of **try** block, immediately followed by a **catch** block. If one of the harmful statements in the **try** block causes an abnormal condition, JVM will create an **exception object** with all the information and throws it. The **catch** block will catch the exception object and executes the statements in its block. Once all the statements are executed in the catch block, the program continues with the rest of the statements following the catch block, resuming the program execution.

Let me tell you one thing. You don't have to scratch your mind trying to understand exceptions. There are simple tricks that I am going to tell you how to handle exceptions. Just follow them and you'll be good. Also, you really don't have to spend too much time on this chapter.

I'll keep this chapter as simple and as short as possible, so that we can spend more time learning the important ones.

Let's first write a program without any exception handling. See listing 6.1.

Listing 6.1 (ExceptionTest.java) Simple class creating an abnormal condition.

```
1 package exceptions;
2
3 public class ExceptionTest {
4
5     public static void main(String args[]) {
6
7         int i = 10;
8         int j = 0;
9
10        // Harmful statement, division by 0
11        int k = i / j;
12
13        System.out.println(k);
14        System.out.println("Hello World");
15    }
16}
```

If you compile and execute the above program, the result will be as shown below:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
at exceptions.ExceptionTest.main(ExceptionTest.java:11)
```

Notice the output. There are three important pieces of information.

1. The exception object belongs to a class called `java.lang.ArithmetricException`.
2. Exception message "`/by zero`"
3. Point where the exception occurred

```
"exceptions.ExceptionTest.main(ExceptionTest.java:9)"
```

In the above program we did not handle the exception, so the program abruptly terminated at line 11, and therefore we didn't see the message "Hello World" at line 14. In this case, JVM sincerely did its job of throwing the exception object. But the above program failed to catch the exception object and this is why the program terminated. The exception is mainly because we are dividing with 0 as shown below:

```
int j = 0;
int k = i/j;
```

This is the culprit statement. So the trick is, simply enclose this statement in a try-catch block as shown in listing 6.2.

Listing 6.2 (ExceptionTest.java) Updated class handling abnormal condition.

```
package exceptions;

public class ExceptionTest {

    public static void main(String args[]) {

        int i = 10;
        int j = 0;

        try {
            int k = i / j;
            System.out.println(k);

        } catch (java.lang.ArithmeticeException ae) {
            System.out.println("Arithmetice Exception handled");
        }

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

In the above program, the try block is pretty obvious. The catch block is the one that deserves some attention. If someone throws a ball at us we need to catch the ball and not something else, right? Based on this analogy if the JVM throws an exception object of `java.lang.ArithmeticeException` class, the catch block must catch the same. This is why the catch statement is like as shown below:

```
catch(java.lang.ArithmeticeException ae)
```

In the above statement, `ae` is some name that we gave to the exception object. Don't worry about why we need the name at this point of time. We'll see that later. For now, just remember that the catch statement must always give a name to the exception object. In the catch block, we can write any statements to handle the exception, and convey a message that exception is handled. This is what we did in this example.

If you compile and execute the above updated program, the output will be:

```
Arithmetice Exception handled
Hello World
```

Once the catch block is executed, JVM will continue with the rest of the program. This is why you see "Hello World" getting displayed. No more program termination.

The Real Challenge

If you look at the previous example, JVM has thrown `ArithmaticException`. We came to know about this exception only when we ran the program without the try-catch block, right? Then we smartly put the right name in the catch statement and fixed the problem.

Without doing the above, how the hell are we supposed to know what type of exception object JVM throws? There can be infinite different types of exceptions JVM can throw. How do I know which exact exception to handle? We'll surely address this challenge but only after learning the next section. Keep this challenge in mind for now.

Exception Types

In Java, there are tons of built-in exception classes. JVM can only throw exception objects of these classes only. The good thing is that we will neither have learn about all these different exceptions nor store them in our mind. However, we need to know very few important ones that are most commonly thrown by the JVM. Thank God.

In Java, all the built-in exception classes are categorized into two types as,

1. Runtime Exceptions
2. Checked Exceptions

Runtime Exceptions as the name suggests, will be **created and thrown by the JVM**, only when the program is executed. With these exceptions, the compiler doesn't know for sure whether an exception will be really thrown or not. Therefore it compiles the program without any errors even if you **don't put a try-catch block**. When you run the program, that's when an exception may or may not pop up based on the data. If an exception pops up, then the program terminates. This is the price you need to pay with runtime exceptions. 50-50 chances, right?

Checked Exceptions on the other hand are **created and thrown by Java programs themselves**. Because they are thrown by the Java program itself, the compiler knows exactly what possible exceptions particular statements in a program can throw, and checks whether the program handled these exceptions using try-catch block. Failing to handle will not let the program to compile at all. So, **a try-catch block is a must for checked exceptions**.

Now that we know the difference between the two, let's understand five runtime and two checked exceptions. Fair enough, right?

Five important Runtime Exceptions

`java.lang.ArrayIndexOutOfBoundsException`

This is thrown when we try to access an element from an array outside the bounds of its length.

`java.lang.ClassCastException`

This is thrown when we improperly cast objects. We'll see an example later.

`java.lang.NullPointerException`

This is thrown when we try to call a method on a null object. Most frequent one.

`java.lang.ArithmaticException`

Thrown when illegal arithmetic operations are performed.

`java.lang.ClassNotFoundException`

Thrown when JVM is unable to load a class.

Two important Checked Exceptions

`java.io.IOException`

Thrown when working with files.

`java.io.FileNotFoundException`

Thrown when a file is not found.

In Java, all the built-in exceptions are nothing but classes. We really don't have to worry about how the above classes are written and all that. All we need to know is how they are related with each other. To understand this, look at the following inheritance diagram.

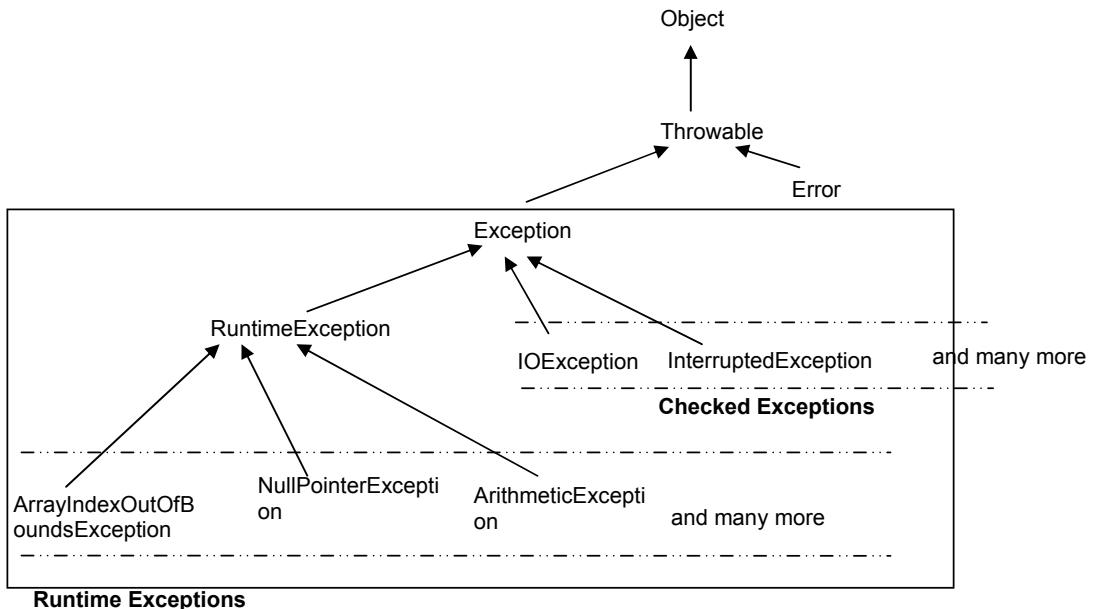


Fig 6.1 Exception hierarchy

Look at the above hierarchy. As I said before, `Object` is the ultimate parent of all. The actual exception class hierarchy begins from the `Throwable` class. This is the ultimate exception parent class. This class has two child classes namely, `Exception` and `Error`. In Java, we must never handle **errors**. Good. This leaves us with just one option, which is handling `Exception` objects (ones in the rectangle). Therefore, we are only interested in the hierarchy enclosed in the rectangle.

If you notice the hierarchy in the rectangle,

1. All the Checked Exceptions (within upper dotted line) are child classes of `Exception` class.
2. All the runtime exceptions (within lower dotted line) are child classes of `RuntimeException` class which in turn is the child class of `Exception` class.

It is the `Exception` class at the top that will help us to address the challenge we faced before. With this knowledge let's get back to writing programs.

Take a look at the code in listing 6.3.

Listing 6.3 (ExceptionTest3.java) Multiple exceptions

```
package exceptions;

public class ExceptionTest3 {

    public static void main(String args[]) {

        try {

            int i = 10;
            int k = i / 0; // Culprit 1
            System.out.println("Hello 1");

            int a[] = new int[2];
            a[0] = 1;
            a[1] = 2;
            a[2] = 3; // Culprit 2

            System.out.println("Hello 2");

        } catch (ArithmaticException ae) {
            System.out.println("Arithmatic exception handled");
        }
        System.out.println("DONE");
    }
}
```

When you compile and execute the above test program, it will produce an Arithmatic Exception due the following statement:

```
int k = i/0;
```

So, all the following statements will be skipped, and the control comes to catch block that handles ArithmaticException. So, the output of the program will be,

```
Arithmatic exception handled
DONE
```

Knowing the problem, let's fix it by updating the division statement to

```
int k = i/2;
```

Good. Compile and execute the program only to see another exception as shown below:

```
Hello 1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
        at exceptions.ExceptionTest1.main(ExceptionTest.java:16)
```

Now we started to hate exceptions. We fixed one, and we got another. Shoot! This time my new exception is `ArrayIndexOutOfBoundsException`. This is the first of the five runtime exceptions we listed before. Since the exception message gave us the line number as 13, the statement shown below is the culprit.

```
a[2] = 3;
```

If you look at the code, we created an array of size 2, but trying to store the element in the 3rd position that doesn't even exist. Aren't we going out of bounds? So is the exception. The question is, how should we make the program to also handle `ArrayIndexOutOfBoundsException`? Very simple. Put another catch block for this exception as shown in listing 6.4.

Listing 6.4 (`ExceptionTest4.java`) Multiple catch blocks.

```
package exceptions;

public class ExceptionTest4 {

    public static void main(String args[]) {

        try {

            int i = 10;
            int k = i / 2; // We fixed this
            System.out.println("Hello 1");

            int a[] = new int[2];
            a[0] = 1;
            a[1] = 2;
            a[2] = 3; // This is the culprit
            System.out.println("Hello 2");

        } catch (ArithmaticException ae) {
            System.out.println("Arithmatic exception handled");
        } catch (ArrayIndexOutOfBoundsException ae) {
            System.out.println("Out of Bounds handled");
        }
        System.out.println("DONE");
    }
}
```

This time when the out of bounds exception is thrown, the control goes to the catch block that handles it (second one). Upon executing this new catch block, the program continues with the statements after the **last** catch block. The output of the above program will therefore be:

```
Hello 1
Out of Bounds handled
DONE
```

So the moral of the story is, *a single try block can be associated with 'n' number of catch blocks* each catching its own exception. What if my try block throws 100 different exceptions? Should we put 100 different catch blocks? Even if we are patient enough to add 100 catch blocks, we should still know the exact names of all 100 exceptions. Holy cow! Now, I am stuck. Some how we found the names of 100 exceptions, and placed 100 catch blocks. Now we are smart. You know what, all our code will be cluttered with catch blocks than the real code. Is there a cleaner solution that addresses all the above problems? Yes, we have. Look at the code in listing 6.5.

Listing 6.5 (ExceptionTest5.java) Single generic catch block.

```
package exceptions;

public class ExceptionTest5 {

    public static void main(String args[]) {

        try {

            int i = 10;
            int k = i / 0; // First Culprit
            System.out.println("Hello 1");

            int a[] = new int[2];
            a[0] = 1;
            a[1] = 2;
            a[2] = 3; // Second culprit
            System.out.println("Hello 2");

        } catch (Exception e) {
            System.out.println("Exception handled");
        }

        System.out.println("DONE");
    }
}
```

In the above code, we activated both the culprits. When you run the program, it first throws `ArithmaticException`, and the output will be:

```
Exception handled
DONE
```

Now, let's just fix the first culprit by making it `i/2`. This time after the `Hello 1` statement, the out of bounds exception will be thrown. The time the result will be,

```
Hello 1
Exception handled
DONE
```

Wow. The single magical catch block shown below caught both the exceptions unlike before. If the try block throws some other exception, the catch block will still catch it. What a catch block. It's like a super cop catching all the culprits, right?

```
catch (Exception e)
```

All this happened because `Exception` is the parent class of all the exceptions. Look at the figure couple pages back to verify this. So, strictly speaking if we use this type of catch statement, we don't have to know about any of the infinite exceptions we have. Life is cool :-) I don't have to learn anymore in exceptions. Hurray! Just remember this one catch block and you can handle any exceptions. Now you can relax and read the next topics without any worries. This generic catch block is indeed the solution for the challenge we had couple pages ago.

Though the above trick is useful, it also suffers from a serious drawback. Damn. The reason is simple. This single catch block handles all the exceptions in the *same* way. This is like a **consistent catch**. *Some times being consistent is bad*. You will see the same 'Exception Handled' message for both `ArithmaticException` as well as `ArrrayIndexOutOfBoundsException`. Can someone please answer the following questions?

- ✓ How do you know which exception really popped up?
- ✓ What if I want to handle different exceptions differently?

The answer to both the questions is the same, which is using **multiple catch blocks**. Haha. Sorry guys. The trick doesn't work all the times. It all boils down to how you want to handle exceptions. If it is **ok** for you to handle all the exceptions in the same way, then use the single catch solution trick. Otherwise, use the multiple catch solution.

For instance, consider this scenario. Some piece of code throws five different exceptions namely A,B,C,D,E. The requirement is that, exceptions A, B must be handled differently, while the remaining exceptions C, D, E can all be handled in the same way. How do you implement this scenario? Look at the following two solutions:

```
public class ExceptionTest
{
    public static void main(String
args[]){
        try{
            // Statement throws exception A
            // Statement throws exception B
            // Statement throws exception C
            // Statement throws exception D
        }
        catch(ArithmaticException e){
            // Statement throws exception A
            // Statement throws exception B
            // Statement throws exception C
            // Statement throws exception D
        }
    }
}
```

```
// Statement throws exception E
}
catch(A){
    System.out.println("Handled A");
}
catch(B b){
    System.out.println("Handled B");
}
catch(C c){
    System.out.println("Handled");
}
catch(D d){
    System.out.println("Handled");
}
catch(E e){
    System.out.println("Handled");
}
}

// Statement throws exception E
}
catch(A a){
    System.out.println("Handled A");
}
catch(B b){
    System.out.println("Handled B");
}
catch(Exception e ) {
    System.out.println("Handled");
}
}
```

Which one of the above two implementations do you like? I like the second one. The rule is simple. Put a separate catch block for every exception you need to handle differently. Lastly, you put the generic (trick) catch statement for all the remaining exceptions that you want to handle in the same way. So, if you have multiple catch blocks, the **generic catch block must be the last catch block**. Very important rule. If we try to put it as the first catch block, then it will also swallow exceptions A and B, right? So, the A and B catch blocks will never be reached by the JVM to execute. Compiler recognizes this fact upfront and gives you an error saying “Unreachable catch blocks”. So, the rule is always to put the parent class exception after the child class exceptions.

Now, it's time to understand the most common exception that any Java program faces. It's the very popular `NullPointerException`. This exception is thrown when we try to invoke a method on a `null` object. Null object means the object doesn't exist at all.

Can you tell me when an object doesn't exist? The answer is very simple. When we don't create an object using **new** keyword, right? Look at the following statements.

```
Date d;  
d.getTime();
```

In the above statements, the Date object is actually not created, and we are trying to invoke a method on it. In Java, when an object is not created, the default value of any object reference variable is null. Therefore, in the above code, the object pointer 'd' is null, and so is the NullPointerException. To avoid this exception, we simply have to check that the object is **not null** before invoking the method as shown below:

```
Date d;  
if ( d != null){  
    d.getTime();  
}
```

The above code will never throw a `NullPointerException` because of the check we are doing. Always check for the object reference variable as not null before invoking the methods using it. This is a good practice.

Handling Checked Exceptions

Handling checked exceptions is no different from handling runtime exceptions. We still use the same try-catch blocks. With runtime exceptions as I said before, JVM creates and throws them. We just have to handle them using try-catch block. However with checked exceptions, the Java program itself will **create, throw and handle** exceptions. I know you are curious to know, why the hell the program should create and throw exceptions, and why should it handle them again? Doesn't make sense, right?

I told you at the very beginning of the chapter, that exceptions are both good and bad. We saw the bad side of exceptions until now. Let's now look at the good side of exceptions. In real world applications, it's extremely important to convey meaningful error messages to the users. This is exactly what we can do with exceptions, convey proper error messages.

To convey an error message we need do the following things:

1. Create a checked exception object with a meaningful message
2. Throw the exception object and
3. Handle the exception object using try-catch block and display the message.

Creating Exceptions

All the exceptions in Java are nothing but **classes**. Creating an exception means, creating object of one such class. These exception classes can either be built-in classes or user defined custom exception classes. Let's look at both the cases.

Case 1: Creating an exception from built-in class.

Let's take the following built-in checked exception class.

```
java.io.IOException
```

(Note that `java.io` is the package in which `IOException` class is stored in the library)

To create an exception of the above class, we simply create an object like we did using `new` keyword as shown below:

```
String message = "We are unable to locate the file. Please try again later";
java.io.IOException ex = new java.io.IOException( message );
```

(or)

```
java.io.IOException ex = new java.io.IOException("We are unable to locate the
file. Please try again later");
```

You can define a message and pass the message to the exception object or can directly pass in the message as shown above. Most folks use the second approach. This is how we create exception object **ex** from built-in exception class. Easy, right?

Case 2: Creating an exception from a user-defined or custom exception class.

In this case, we need to first write our custom exception class as shown below:

```
public class MyCustomException extends Exception{

    public MyCustomException ( String message ){

        // Pass the message to the parent class constructor
        super ( message );
    }
}
```

If you look back at our exception class hierarchy diagram, all the checked exceptions are child classes of `Exception` class. Therefore, to create a custom checked exception,

- a) We **must** create a class that extends from `Exception` class as shown above.
- b) We then have to define a constructor that takes `String` as an argument for message and then pass this message to the parent class `Exception` constructor using `super` statement as shown above.

That's it. We now created our own exception class. Finally we create the exception object as shown below:

```
MyCustomException ex = new MyCustomException ("Please call 1-800-888-
8244 for Help");
```

If someone asks you to create a custom exception, this is what you need to do. Keep this in mind since we often create whole lot of custom exception classes in real world applications.

Throwing Exceptions

Now that we know how to create checked exceptions, we need to see how to *throw* these exceptions. To throw an exception we simply use the **throw** keyword as shown below:

```
MyCustomException ex = new MyCustomException (" Please call 1-800-888-  
8244 for Help ");

throw ex;  
  
(or)  
  
throw new MyCustomException (" Please call 1-800-888-8244 for Help ");
```

You can use one of the above conventions. The second one is simplest as it does all in one line. This is the commonly used convention. Keep it in mind.

Handling Custom/built-in Checked Exceptions

This is the simplest of all. Like we did with runtime exceptions, we also use the same try-catch block for handling the custom/built-in checked exceptions as shown below:

```
try {  
    // Statements to create and throw exceptions  
}  
catch ( MyCustomException ex) {  
    // Do what ever you want here  
}
```

One important point to be noted here is that, all the above 3 steps namely creating, throwing and handling should be done in the *same method* in the Java program (at least for now).

Knowing all the above three things like creating, throwing and handling checked exceptions, let's write a program to get a complete picture. See listing 6.6.

Listing 6.6a (`MyCustomException.java`) A custom exception class

```
package exceptions;

public class MyCustomException extends Exception {

    public MyCustomException(String message) {

        // Pass the message to the parent class constructor
        super(message);
```

```
    }  
}
```

Listing 6.6b (`Calculator.java`) Method throwing an exception

```
package exceptions;  
  
public class Calculator {  
  
    void divide(int i, int j) {  
  
        if (j == 0) {  
  
            throw new MyCustomException("Call 1-800-888-8244 for Help");  
  
        } else {  
  
            System.out.println(i + "/" + j + "=" + i / j);  
        }  
    }  
}
```

If you look at the above two classes in listing 6.6a-b, we first created our custom checked exception class which we already know. We then wrote a class named `Calculator` which defines the `divide()` method. This method creates and throws the exception object if the value of `j` is 0. Otherwise, it computes and displays the result. Try compiling the above classes as shown below:

```
C:>JavaTraining>Chapter6>javac exceptions\*.java  
Error: Unhandled exception type MyCustomException.
```

The above compilation error is pretty obvious because the method missed the 3rd step, which is handling the exception, right? So, let's fix the code as shown below:

```
if (j == 0) {  
  
    try{  
  
        throw new MyCustomException("Call 1-800-888-8244 for Help");  
    }  
    catch (MyCustomException ex){  
  
        String message = ex.getMessage();  
  
        System.out.println (message);  
    }  
}
```

Compile the code again with the above fix, and the error disappears. I know what you are wondering about. The statements in the catch block, right? Where the hell did the

`getMessage()` method come from? If you look at our custom exception class, we inherited it from `Exception` class. So, by the rule of inheritance, all of the methods in the built-in `Exception` class will be inherited into our `MyCustomException` child class. One such method is the `getMessage()` method shown below:

`getMessage()` – This method returns back the `String` message that we supplied while creating our exception as shown below:

```
MyCustomException("Call 1-800-888-8244 for Help");
```

Since it returns the message, like good folks we took it into the `message` variable. Java is all about give and take. Once we got the message, we simply echoed it. Now, look at the test class shown in listing 6.6c.

Listing 6.6c (`ExceptionTest6.java`) Test class for exceptions,

```
package exceptions;

public class ExceptionTest6 {

    public static void main(String args[]) {
        Calculator calc = new Calculator();

        calc.divide(10, 2); // Good one
        calc.divide(10, 0); // Bad one
    }
}
```

In the above test class, the first call to `divide` method produces the correct result without any exception. It is the second one that results in an exception which the `divide()` method handles and displays the message. Compile and execute the above program to see the following result.

```
10/2 = 5
Call 1-800-888-8244 for Help
```

The above error message is more meaningful and my client will be happier then ever. This is possible only by throwing the exception and then handling it, right? Let's introduce a tricky situation here.

Say we have a requirement where the above `divide()` method can create and throw an exception, **but should not handle the exception at all**. Is this possible? Don't we get a compilation error when we do this? You know what, there is a solution. If the method doesn't know how to handle a checked exception, it must throw the exception to the

outside world using the **throws** keyword. Then the one who actually invoked the method must handle the exception. Again, if the called method doesn't handle the exception, it should throw it to some other method. This is like throwing the ball until some one finally catches it. The syntax for throwing an exception to the outside world is shown below:

void somemethod() throws <list of exceptions>

So let's update our `divide()` method by removing the try-catch block and using the **throws** keyword to throw the exception to the invoker of the method. See listing 6.6d.

Listing 6.6d (`Calculator.java`) Updated calculator class using throws keyword.

```
package exceptions;

public class Calculator {

    void divide(int i, int j) throws MyCustomException {
        if (j == 0) {
            throw new MyCustomException("Call 1-800-888-8244 for Help");
        } else {
            System.out.println(i + "/" + j + "=" + i / j);
        }
    }
}
```

As you can see from the above code, the `divide()` removed the try-catch block, and declared a **throws** clause specifying the name of the exception to throw to the caller of that method. This is like the method saying, *"Hey guys, my code might throw an exception named **MyCustomException** which I honestly don't know how to handle it. You guys take care of it. Bye"*. This is how it rubs off her hands. Smart method, right? Now the one who invokes the method must handle this exception by putting the try-catch block around it. This poor guy will be cursing the `divide()` method and painfully adds a try-catch block. See listing 6.6e.

Listing 6.6e (`ExceptionTest7.java`) Caller handling the exception.

```
package exceptions;

public class ExceptionTest7 {

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

```
Calculator calc = new Calculator();

try {
    calc.divide(10, 2); // Good one
    calc.divide(10, 0); // Bad one

} catch (MyCustomException ex) {
    String message = ex.getMessage();
    System.out.println(message);
}
}
```

The try-catch block is now moved into the test class and **all** the calls to the divide() method must be enclosed in this block. With this updated code for both Calculator and ExceptionTest7, compile and execute the test class. The result will still be the same.

Let me ask you a question. What if the test class too doesn't want to handle the exception? Can it also throw the exception to some other world? Yes, it can, but you know what, that world is the final world which is none other than JVM (because the main method is called by JVM). JVM will then display a weird *exception trace* on the console.

Modify the test class by removing the try-catch block and declare a **throws** clause in the **main** method as shown below:

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

    Calculator calc = new Calculator();
    calc.divide(10,2); // Good one
    calc.divide(10,0); // Bad one
}
```

If you execute the test class with the above update, the output will be

```
10/2=5
MyCustomException: Call 1-800-888-8244 for Help
    at Calculator.divide(Calculator.java:9)
    at ExceptionTest.main(ExceptionTest.java:8)
Exception in thread "main"
```

If no one handles the exception, this is what JVM will give us. We call this as **exception stack trace**. Though this looks weird at the first place, if we observe it carefully, we can't wait to appreciate what it has to offer. It tells you the following things:

1. What exact exception it is, in this case it is MyCustomException.
2. What is the exception message like Call 1-880 etc.
3. The way the exception propagated. In this case, if you start reading, it says it first originated in the divide() method at line 9, and then moved to line 8 in the main() method. I am sure this information is more than enough to help us fixing the problem. You agree with me?

Note: Whenever you see an exception, don't panic. Take it positively and be thankful for JVM for giving all the information about the error.

Let me ask you a question here. Can a single method throw more than one exception? The answer is yes. A method can throw more than one exception. All we need to do is, list all the exceptions in the throws clause with a comma delimiter as shown below:

```
public void compute() throws IOException, TestException1, TestException2....
```

Then whoever calls this method, must handle all the exceptions by putting multiple catch blocks, one for each (or) one catch block with the generic Exception as shown below:

```
try{  
    calc.compute();  
}  
catch(IOException e1){  
}  
catch(TestException1 t1){  
}  
catch(TestException2 t2){  
}  
  
(or)  
  
try{  
    calc.compute();  
}  
catch( Exception e){  
}
```

We finally arrived at the last concept of this chapter, using **finally** block. Let's see what this is all about.

Finally block

Let's say we have some piece of code that *may or may not throw* an exception. What ever be the case, I *always* intend to execute some functionality. What should I do? The answer

is use **finally** block and put all the functionality there. To get a clear picture, take a look at the code in listing 6.7

Listing 6.7 (FinallyBlockDemo.java) Class using finally block.

```
package exceptions;

public class FinallyDemoTest {

    public static void main(String args[]) {

        Calculator calc = new Calculator();
        try {

            calc.divide(10, 2);
            calc.divide(10, 5);

        } catch (MyCustomException ex) {

            String message = ex.getMessage();
            System.out.println(message);

        } finally {

            System.out.println("This is always executed");
        }
    }
}
```

If you execute the above code, both the divide() methods are passing good data, and neither an exception will be thrown nor will be handled. The result will be,

```
10/2 = 5;
10/5 = 2;
This is always executed.
```

Change the second compute method to divide(10, 0), and the result will be

```
10/2 = 5;
Exception handled
This is always executed.
```

There are couple of rules here with finally block.

1. It must be placed **after the last catch block** of the try block.
2. There can be **only one** finally block per try-catch block.

The moral of the story is, the statements in the **finally** block will always be executed whether or not an exception is thrown. We usually place the clean up statements here.

However, there is one way we can stop the JVM from executing the finally block. This is by manually terminating the program in the **try** block or the **catch block** as shown below:

```
try{
    System.exit(0);
}
catch( Exception e){
    System.exit(0);
}
finally{
    System.out.println("This is never executed");
}
```

Interview Question: How can you make the program not execute the finally block?

Answer: By placing a `System.exit(0)` in both try and catch blocks.

That's it guys. This is all about exceptions you need to know. Here is a simple trick with exceptions.

Trick: If any method in a class has a **throws** clause in its declaration, simply enclose that method call in a **try-catch** block. This will save your life for sure.

In real world applications, exception handling is one of the important aspects. So, it's always nice to know something about exceptions and their details. I am sure you thoroughly enjoyed this chapter like others. Trust me; life get's real easy and easy from next chapter onwards. Before we get there, let's summarize this chapter.

Summary

- ✓ Exception is an abnormal condition that occurs in a program due to which JVM terminates the program abruptly.
- ✓ To prevent the termination of the program, exceptions must be handled.
- ✓ In Java, exceptions are handled using **try-catch** block.
- ✓ The **try** block encloses all the harmful statements that result in throwing exceptions.
- ✓ The **catch** block catches the exceptions thrown by the **try** block and takes the appropriate action.
- ✓ Exceptions are classified as Runtime Exceptions and Checked Exceptions.

- ✓ Runtime Exceptions are created and thrown by the JVM when the program is executed. With these exceptions, the program will compile without any errors even without a **try-catch** block.
- ✓ Checked Exceptions on the other hand are created and thrown by the program itself. The program must always handle these exceptions using **try-catch** block. Failing to handle will result in compilation errors.
- ✓ The parent class of all the checked exceptions is **Exception** class.
- ✓ To create a custom checked exception, we need to create a child class of **Exception** class and define a constructor that takes **String** argument.
- ✓ A method can throw an exception using **throw** keyword.
- ✓ If a method throws an exception using **throw** keyword, it must also handle the exception using **try-catch** block.
- ✓ If the method decides not to handle the exception, it must throw the exception to the invoker of that method using **throws** keyword. The invoker of the method must then handle the exception using the **try-catch** block.
- ✓ A method can also throw multiple exceptions. In this case, the method must declare all the exceptions in the **throws** clause.
- ✓ Use **finally** blocks to always execute some statements irrespective of whether an exception is thrown or not.

Time to play 50-50

1. Which of the following keywords are used for handling exception?
 - a) try-finally
 - b) try-catch
2. Which of the following exceptions need not be handled, and yet the program compiles without any errors
 - a) Checked Exceptions
 - b) Runtime Exceptions
3. All the checked exceptions are child classes of which of the following exception class.
 - a) RuntimeException
 - b) Exception
4. Which of the following is a Runtime exception

- a) NullPointerException
b) IOException
5. A NullPointerException is caused by which of the following:
 - a) Invoking a method after creating an Object
 - b) Invoking a method without creating an Object
6. To create a custom checked exception class, which of the following classes it must extend from:
 - a) RuntimeException
 - b) Exception
7. Which of the following keyword is used by a method to throw an exception?
 - a) throws
 - b) throw
8. Which of the following keyword is used in the method declaration to throw the exceptions to the outside world?
 - a) throw
 - b) throws
9. Which of the following must be used to always execute the statements whether or not an exception is thrown?
 - a) catch blocks
 - b) finally blocks
10. Which of the following is used prevent the execution of finally block?
 - a) System.out.println("Program terminated");
 - b) System.exit (0);

Interview Questions

Question: What is an exception in Java?

Answer: An exception in Java is an abnormal condition due to which JVM terminated the program abruptly.

Question: What should you do to handle exceptions?

Answer: Use try-catch blocks.

Question: What are the two categories of exceptions, and what is the difference?

Answer: Runtime exceptions and checked exceptions are the two categories. The difference between the two is that, with runtime exceptions, the program compiles without any errors even without try-catch block. With checked exceptions, the program will not compile without a try-catch block.

Question: How do you create a custom exception?

Answer: To create a custom checked exception we need to create a class that extends **Exception** class and define a constructor that takes a **String** argument.

Question: What are the keywords by which a method can throw an exception?

Answer: throw and throws.

Question: What is the only way to bypass a finally block?

Answer: Use System.exit(0) statement in **try** and **catch** block.

That's it guys. This concludes the chapter. Let's move ahead.

Chapter 7

Core Libraries and Best Practices

By the end of this chapter, you'll be an expert in using Java built-in core classes. Java is all about writing simple programs using the built-in classes. This chapter will also tell you some simple tricks and tips on how to use classes and also demonstrates the coding conventions and best practices that are well and truly followed in all the real world Java applications. Understanding this chapter gives you an edge for a successful career in Java.

Chapter Goals

- ✓ Understand the important core libraries
- ✓ Understanding the most important core library classes
- ✓ Coding conventions and best practices
- ✓ Generating Java Documentation using javadoc tool.

Environment Setup

All the programs in this chapter should be stored in the following directory.

```
C:/JavaTraining/Chapter7/utilities
```

`utilities` is a package in which we will store the programs. To compile the programs, move to the following root directory and get ready.

```
C:/JavaTraining/Chapter7
```

Introduction

In all the previous chapters we have written several programs understanding the object oriented concepts, handling abnormal situations in a program using exceptions and may more. All the examples we wrote thus far didn't use any of the built in classes that Java comes with. In real world applications there will be several complex requirements and it's time consuming to implement each and every requirement from scratch. Since time is the primary constraint, it would be more beneficial if there are some ready made components that can do the bulk of the job, and leaving the rest to us.

By making our Java program use the ready made components, not only the development gets faster, but also the code becomes more reliable since the ready made components are built by reliable people. In Java, these components are nothing but the classes. In this chapter we'll first understand the important built-in classes and then start writing programs using them, ok.

Let me tell you one thing here. Using the built-in classes is a fine art and mastering this art is the key to become a successful Java programmer. The best part of it is, mastering this art just takes few minutes. Trust me. I am going to tell you few tricks and tips. We'll follow them and make our life easy.

Important Core Libraries

In Java, all the built-in classes are very cleanly organized in the form of packages. A package as we already know is nothing but a collection of classes. Though there are infinite packages in the Java core library, we are only interested in just 2 packages. Fair enough, right? 99% of the any real world applications just use these two packages. The two packages are listed below:

`java.lang` and
`java.util`

The above two packages have tons of interfaces and classes. So, should we learn all of them? Don't worry. Again, we just have to understand few classes in each package. Awesome. This is why I like Java. Let's don't waste any time and understand these few classes to become a master Java programmer.

java.lang package

This package though contains several classes and interfaces; we are interested in just the following classes:

1. Wrapper Classes
2. String class
3. StringBuffer class

Before using the above classes, let me tell you this one more time. 99.9% of the time, a Java class will simply have no more than variables and methods. Don't worry about the other 0.1% special classes for now. Using a class means, using the methods in the class.

Wrapper Classes

In Java there are some special built-in classes that can only work with objects and not primitive variables like *int*, *long*, *double* etc. So, if we have primitives in our hand, there is no way we can use the so called special classes, right? If I cannot move forward without using the special classes, then I definitely need someone to convert the primitives to objects. That someone is none other than wrapper classes. Wrapper classes simply convert the primitives into equivalent objects, which can then be used with the special classes. There is one wrapper class for every primitive data type as shown in Table 7.1.

Table 7.1 Wrapper Classes

Primitive	Equivalent Wrapper
<i>int</i>	<i>Integer</i>
<i>short</i>	<i>Short</i>
<i>byte</i>	<i>Byte</i>
<i>long</i>	<i>Long</i>
<i>float</i>	<i>Float</i>
<i>double</i>	<i>Double</i>
<i>char</i>	<i>Character</i>

The wrapper class names are pretty obvious. Note the class names start with uppercase letter. All we need to know is how to wrap a primitive variable and convert it to object, and vice-versa. So without wasting any time, let's see how to do this.

Integer

This class is an object representation of `int` primitive variable.

Usage:

To convert an `int` to an object we do it as shown below:

```
int i=10;  
Integer ii = new Integer( i );
```

`ii` is now an object that contains 10.

To get the primitive back from the object, we use

```
int val = ii.intValue();
```

Given a number as `String`, to convert to primitive `int` we use the following:

```
String s = "10 ";  
int val = Integer.parseInt( s );
```

Double

This class is an object representation of `double` primitive variable.

Usage:

To convert a `double` to an object we do it as shown below:

```
double d=12.2345;  
Double dd = new Double( d );
```

`dd` is now an object that contains 12.2345.

To get the primitive double back from the object, we use

```
double val = dd.doubleValue();
```

Given a number as `String`, to convert to primitive `double` we use the following

```
String s = "12.2345 ";  
double val = Double.parseDouble( s );
```

Likewise, the usage is same for the rest of primitives. This is all you need to learn about the wrapper classes. Let's do one quick example. See the code in listing 7.1.

Listing 7.1 (WrapperClassDemo.java) Using wrapper classes.

```
package utilities;

import java.lang.Integer;

public class WrapperDemo {

    public static void main(String args[]) {

        int i = 10;

        // Wrap i in the object
        Integer ii = new Integer(i);

        // Pass the object ii as a parameter
        add(ii);
    }

    static void add(Integer object) {

        int val = object.intValue() + 5;

        System.out.println(val);
    }
}
```

In the above code, we have a method named `add()` whose parameter is of type `Integer`. So, if we have a primitive `int`, we need to convert it to `Integer` object and then pass it as a parameter. This is exactly what we did in the `main` method. The `add()` method takes the `Integer` argument, computes the sum by converting back to the primitive. Finally, look at the import statement that we added at the beginning of the program. If at anytime we use a built-in class, we need to *import* the class as shown below. Since `Integer` class is in `java.lang` package, the import statement will be,

```
import java.lang.Integer;
```

Compile and execute the above program to see the result 15 displayed.

Now, it's time to understand the most important and widely used class, the `String` class.

Handling Strings

String class is the most widely used library class. There will hardly be any Java program without using this class. This is because most of the real world data is represented in the form of text. For instance things like customer name, address, city, state, country and what not. Everything can be represented as text. You agree with me?

I am sure you might be wondering that we used strings to store text data in previous chapters. So why are we discussing this again? You are right. But you know what, I said that string is like a special data type and now is the time to understand why it is indeed special. Ok.

To store text data, we need to create an object of String class, and pass the text as shown below:

```
String str = new String(" This is a simple text message ");
```

All the text must be enclosed in the double quotes. Let me say one thing here. In any real world application, there will be million places to store text. So, it's really tedious to always use the **new** keyword and then pass the text, right? Is there a simpler solution? Fortunately, the inventors of Java language recognized this fact upfront, and gave us a shortcut representation for strings as shown below:

```
String str = "This is a simple text message ";
```

Now the above declaration is simple. By looking at it, doesn't this shortcut representation look like a primitive data type declaration? Yes, it does look like that. This is why I called it as special data type for storing text data.

So, no more using **new** keyword when it comes to declaring strings. This shortcut is **only applicable** to String class, ok. Once the text is declared using String class, it offers several methods to process the text. Let's see the most important methods.

Consider the following string. Every character in it will have a position or index. The indexing of characters will start with 0 as shown below:

0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2								
T	h	i	s		i	s		r	e	a		l		y		r	e	a		l		y		b		i		t		e		x		t

Let's look at the methods in String class for working with text data. For all the following examples, please use the above table.

1. `int length()` : This method returns the length of the text.

Example:

```
String str = "This is really really big text ";
int length = str.length();
System.out.println("The length of the text is " + length );
```

The above statement displays the following:

```
The length of the text is 30
```

2. `char charAt(int index)` : This method returns the character at the specified index.

Example:

```
String str = "This is really really big text ";
char ch = str.charAt( 6);
```

ch will now have the character 's'.

3. `int indexOf(char ch)`
`int indexOf(String s)`

There are two overloaded `indexOf()` methods. One takes a single character and returns the position of its first occurrence, and the second takes a string, and returns the first occurrence of that string.

Example:

```
String str = "This is really really big text ";
int pos1 = str.indexOf( 's');
int pos2 = str.indexOf("realy");
```

pos1 will be 3, and pos2 will be 8.

4. `String substring(int start)`
`String substring(int start, int end)`

The above two overloaded methods return part of the text from the original text. The first one returns the text from the specified index to the end of the text. The second one returns the text between two specified indices.

Example:

```
String str = "This is really really big text";
String s1 = str.substring(22);
String s2 = str.substring(8,20);
```

The content in s1 will be big text, and the content in s2 will be really really.

5. boolean equals(Object anotherString)

This method compares one text with another text, and returns either true or false. The comparison is **case-sensitive**.

Example:

```
String str = "Hello";
if ( str.equals ("hello") ){
    System.out.println(" Strings are equal");
} else{
    System.out.println("Strings are unequal");
}
```

The above code displays **Strings are unequal**. This is because 'h' is in lower-case.

Tip: If any method returns a boolean, you can call the method from conditional statements like if, while etc.,

6. boolean equalsIgnoreCase(String anotherString)

This method compares the text with another text, and returns either true or false. The comparison is **case-insensitive**.

Example:

```
String str = "Hello";
if ( str.equalsIgnoreCase ("hello") ){
    System.out.println(" Strings are equal");
} else{
    System.out.println("Strings are unequal");
}
```

The above code displays **Strings are equal**.

7. String uppercase() : This method returns the text in uppercase.

Example:

```
String oldText = "abcdefghijklm";
String newText = oldText.toUpperCase();
```

newText will now have ABCDEFGHIJKLMNOP.

8. To concatenate multiple texts, we use + operator as shown below:

```
String s1 = "Hello";
String s2 = s1 + "World";
String s3 = "This" + " is " + "easy";
```

These are all the methods you need to know about strings. The most important one is the equals() method which compares two strings. To conclude this section, let's do one quick example. See the code in listing 7.2.

Listing 7.2 (StringClassDemo.java) Using String class.

```
package utilities;

import java.lang.*;

public class StringDemo {

    public static void main(String args[]) {

        String str = "This is really really big text";

        int length = str.length();
        System.out.println("The length of the text is " + length);

        char ch = str.charAt(6);
        System.out.println(ch);

        int pos1 = str.indexOf('s');
        int pos2 = str.indexOf("really");

        System.out.println(pos1);
        System.out.println(pos2);

        String s1 = str.substring(22);
        String s2 = str.substring(8, 20);

        System.out.println(s1);
        System.out.println(s2);

        if (str.equals("this is really really big text")) {
            System.out.println(" Not Equal. Check the Case");
        }

        if (str.equalsIgnoreCase("This is really really big text")) {
            System.out.println("Equal");
        }
    }
}
```

```
    }  
}  
}
```

Compile and execute the above class to see the following result.

```
The length of the text is 30  
s  
6  
8  
big text  
really reall  
Not Equal. Check the case.  
Equal
```

Note: The substring method will throw `StringIndexOutOfBoundsException` when negative numbers are passed as shown below:

```
str.substring(3, -1);
```

StringBuffer

The `StringBuffer` class is another class used for processing the texts. When we already have the `String` class that does this, why do we need another class? Great question. The answer is simple. `String` class consumes more memory when processing texts than `StringBuffer` class. This is because string objects are *immutable*. This means, whenever we try to change the text in the string object, JVM will create a new object. More processing means more string objects in memory.

It is important to note that this way creating of `String` objects is done purposefully to improve the performance. But many a times this solution becomes an anti-performance while complex text processing is involved. In such situations, `StringBuffer` class is an ideal candidate.

With `StringBuffer` class, there will just be one object in the memory and all the text processing is done within the same object. For complex text processing always try to use `StringBuffer` class instead of `String` class. Following is how we use this class.

Create the `StringBuffer` object as shown below:

```
StringBuffer buffer = new StringBuffer();
```

To add text, we use the `append()` method as shown below:

```
buffer.append("A silly young cricket ");
```

```
buffer.append(" accustomed to sing ");
buffer.append(" through the warm sunny days ");
```

You can insert a text in the middle using the `insert()` method as shown below:

```
buffer.insert(5, "Hello ");
```

The above will insert Hello after the 5th character.

Finally, to get the text out of the buffer, we use the following statement:

```
String finalText = buffer.toString();
```

See how easy it is to process text using this class. Let's write an example. See listing 7.3.

Listing 7.3 (StringBufferDemo.java) Using string buffer class.

```
package utilities;

public class StringBufferDemo {

    public static void main(String args[]) {

        StringBuffer buffer = new StringBuffer();

        buffer.append("A silly young cricket ");
        buffer.append(" accustomed to sing");
        buffer.append(" through the warm sunny days");

        buffer.insert(5, "Hello");

        String finalText = buffer.toString();

        System.out.println(finalText);
    }
}
```

Compile and execute the above program to see the following result:

```
A sillHello young cricket accustomed to sing through the warn sunny days
```

This is all about processing text using `String` and `StringBuffer` classes. These are very simple to use, and you'll learn automatically when you start writing programs. No big deal.

These are the only two classes (`String`, `StringBuffer`) we need to know in the **java.lang** package. Most of the real world applications use these classes extensively and understanding them will make our life easy. There are several other methods in above

three classes, and you can try them to see if they help you in any other way. The next package we are going to see is the java.util package.

java.util Package

This package as the name suggests, is a package that has several utility classes that we can use in our programs. The classes in this package are called as **collection** classes. These classes represent various data structures that can be used for storing complex data. Though there are several of these classes, we will learn only those that are frequently used in real world applications.

The collection classes in this package **can only store objects**. They cannot store primitive data types like int, double etc. Point to be noted. Before we learn the usage of the classes in this package, let's find out why we need to use these classes.

In any real world applications, most of the data is highly complex in nature, and it's extremely tedious to store the data using primitive variables like **int**, **double** etc. Instead we store the data in the form of **object**. Therefore we need some thing that provide flexible way of storing and retrieving data objects. This is where the data structures in this package come in handy. Not only the classes in this package allow storing and retrieving data objects, but also provide several convenient methods to process the data like sorting, deleting etc.

All the data structure classes in this package forms the collection hierarchy. There are two broad classifications for all the data structures in this package as shown below:

1. One dimensional data structures
2. Two dimensional data structures

One dimensional data structures

The class and interface hierarchy for one dimensional data structures is shown in the following figure.

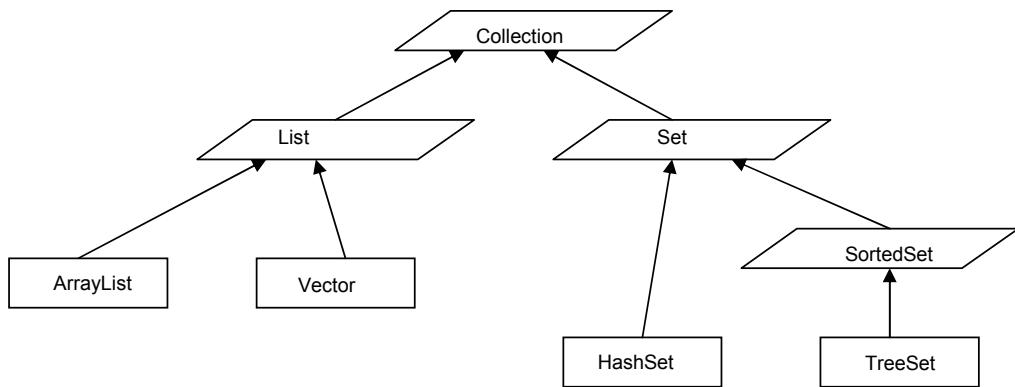


Fig 7.1 Collection Hierachry

If you look at the above hierarchy, we observe the following things:

`Collection` is the top level interface.

`Set` and `List` are the child interfaces of `Collection` interface.

`ArrayList` and `Vector` implement `List` interface

`HashSet` implements `Set` interface

`TreeSet` implements `SortedSet` interface.

First, let's look at the differences between `List`, `Set` and `SortedSet` interfaces.

`List` represents **ordered**, **duplicate** and **unsorted** elements. The classes that implement this interface are `ArrayList` and `Vector`.

`Set` represents **unordered**, **unique** and **unsorted** elements. The important class that implements this interface is the `HashSet`

`SortedSet` represents **unordered**, **unique** and **sorted** elements. The important class that implements this interface is the `TreeSet` class.

Though each class and interface has several methods, like smart folks we'll only learn the most important ones.

Let's first look at the `ArrayList` and `Vector` classes of `List` interface.

ArrayList

Following figure shows the structure of `ArrayList` data structure.

0	1	2	3	4
object1	object2	object2	object3	object4

This class should be used for **storing objects in an order** like an array. Following are the properties of this data structure.

1. All the elements are indexed starting from 0.
2. There is no limit on the number of elements you can store.
3. Allows storing duplicate objects.

Following are the basic operations we can do with this class.

1. Create an ArrayList
2. Add objects to the list
3. Display objects in the list

The three important methods in this class are:

`void add(Object obj)` : This method adds an object to the ArrayList.

`Object get(int pos)` : This method returns the object from the specified index.

`int size()` : This method returns the size of the list.

Let me ask you a question here. If you notice the above class hierarchy, `ArrayList` class implements `List` interface. Based on this, is the following statement legal or illegal?

```
List list = new ArrayList();
```

You are correct. The above statement is perfectly legal and 'list' is an interface reference to the `ArrayList` object. This is the convention we normally use and is also one of the best practices. Let's do a quick example to get a complete picture of `ArrayList` class. See listing 7.4.

Listing 7.4 (ArrayListDemo.java) Using array list class.

```
package utilities;  
import java.util.*;  
  
public class ArrayListDemo {  
    public static void main(String args[]) {  
        // Create an ArrayList
```

```
List list = new ArrayList();

// Add objects.
list.add("Hello");
list.add("World");
list.add("World");
list.add("ABC");

// Get the objects
for (int i = 0; i < list.size(); i++) {

    String s1 = (String) list.get(0);
    System.out.println(s1);

}

}
```

The above example code created an `ArrayList` and added four `String` objects using the `add()` method. It then used a `for` loop using the `size()` method to retrieve the strings back. Notice how we used the object casting within the loop. Since the `get()` method returns `Object`, we need to narrow it down to `String` object since we stored strings. Compile and execute the above program to see the following output:

```
Hello
World
World
ABC
```

If you noticed the above result, we can observe the following things:

1. The string objects are displayed in the same order that we added.
2. The list allowed duplicate objects (World)
3. The strings in the `ArrayList` are not automatically sorted.

This is one of the widely used data structure in real world applications since we need to preserve the order of the data. So, keep this in mind.

Vector

If you look back at the class hierarchy diagram, both `ArrayList` and `Vector` implement the same `List` interface. This means, both of them will have same methods, right?. If this the case, why do we need `Vector` when we already have `ArrayList`? This is an interview question. The answer is, using `Vector` is safer in multi-threaded environments than using `ArrayList`. So, depending on what the context is, you can use

either `Vector` or `ArrayList` classes. The usage of `Vector` is exactly same as using `ArrayList` as shown in listing 7.5.

Listing 7.5 (`VectorDemo.java`) Using vector class.

```
package utilities;

import java.util.*;

public class VectorDemo {

    public static void main(String args[]) {

        // Create a Vector
        List list = new Vector();

        // Add objects.
        list.add("Hello");
        list.add("World");
        list.add("World");
        list.add("ABC");

        // Get the objects
        for (int i = 0; i < list.size(); i++) {

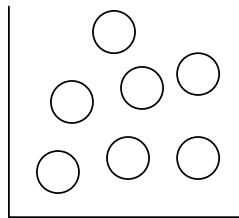
            String s1 = (String) list.get(0);
            System.out.println(s1);

        }
    }
}
```

If you noticed the above code, the only difference is that we created `Vector` instead of `ArrayList`. One important thing with `ArrayList` and `Vector` is that the objects will *never* be sorted. The next couple of examples will demonstrate how we can automatically sort objects in a data structure.

HashSet

This class implements the `Set` interface and should be used when the order of the elements is not important. Look at the following figure to see how the objects are stored within `HashSet`.



Following are the properties of this data structure

1. It doesn't store the elements in an order
2. It doesn't allow duplicate elements
3. It doesn't sort the elements

With this class, we can add the elements using the `add()` method just like we did with the `ArrayList` and `Vector` classes. However, to display the objects, we need to follow two simple steps listed below:

1. Get the iterator object.
2. Use the iterator with a `while` loop to display the elements.

The three important methods in this class are:

`void add(Object obj)` : This method adds an object to the `HashSet`.

`Iterator iterator()` : This method returns the `Iterator` object for the elements. The `Iterator` class in turn has the following two important methods for retrieving the elements.

`boolean hasNext()` : Returns true if there are more objects
`Object next()` : Returns the object.

I know this is some what complex than array list and vector classes. Take a look at the code in listing 7.5 and you'll get the picture.

Listing 7.6 (`HashsetDemoDemo.java`) Using hash set class.

```
package utilities;  
import java.util.*;  
public class HashsetDemo {  
    public static void main(String args[]) {
```

```
// Create a HashSet  
Set set = new HashSet();  
  
// Add objects.  
set.add("Hello");  
set.add("World");  
set.add("World");  
set.add("ABC");  
  
// Get the iterator  
Iterator itr = set.iterator();  
  
while (itr.hasNext()) {  
  
    String s1 = (String) itr.next();  
    System.out.println(s1);  
}  
}
```

If you look at the above example, after adding the strings to the `HashSet`, we got hold of the iterator object using the following statement:

```
Iterator itr = set.iterator();
```

We then used the two methods in the `Iterator` class with a while loop, and displayed the elements. The `hasNext()` method returns `true` until all the elements are returned. The working of the while loop is like, “*If there are more elements, then give me the next element*”. Compile and execute the above class to see the following result.

```
World  
ABC  
Hello
```

If you noticed the above result, we can observe the following things:

1. Strings are displayed in random order
2. Didn't allow duplicates when we added **World** twice.
3. The strings are not sorted.

The only advantage with this data structure is that we can eliminate the duplicate elements. For instance, if you want to count the number of unique objects in a given set of objects, a simple solution is that you add all the objects to `HashSet` and the size of the `hashset` gives you the count of unique elements. Without this, you have to compare the current object with the previous object and all that crap. Are you with me? I am sure you are.

TreeSet

This class works just the same as the `HashSet` class, except that it sorts the elements in an ascending order. In the above program, change the following statement

```
Set set = new HashSet() to  
Set set = new TreeSet();
```

Compile and execute the program with the above change, and you'll notice the strings sorted alphabetically as shown below:

```
ABC  
Hello  
Word
```

Let me ask you a question here. Does `TreeSet` sort any type of objects or just `String` objects? The answer is not that simple. Look at the following rule.

Rule: With `String` objects and `Wrapper` objects, `TreeSet` by default will sort the objects in ascending order. With all other objects, `TreeSet` expects the user to define how the sorting should be done.

Let's say we have a complex object (Java Bean) like `Customer` which encapsulates data like `firstName`, `lastName` and `SSN`. Now let's add the customer objects to the `TreeSet` as shown below:

```
TreeSet set = new TreeSet();  
  
set.add(newCustomer("John", "Smith", 12345));  
set.add(newCustomer("Laura", "Jackson", 456546));  
set.add(newCustomer("Thomas", "Edison", 12345));
```

Based on the above rule, the `TreeSet` will *not* sort the objects. This makes sense because the `TreeSet` doesn't know which particular field in the object to use for sorting, right? In such situations, `TreeSet` expects us to pass an object of `Comparator` class that defines how the sorting should be done. To better understand, look at the following comparator class in listing 7.7a.

Listing 7.7a (`LastnameComparator.java`) Comparator class.

```
package utilities;  
  
import java.util.*;  
  
public class LastnameComparator implements Comparator {
```

```
public int compare(Object o1, Object o2) {  
  
    Customer c1 = (Customer) o1;  
    Customer c2 = (Customer) o2;  
  
    int index = c1.getLastName().compareTo(c2.getLastName());  
  
    return index;  
}
```

Here is what we did in the above code.

1. Created a custom comparator class by implementing the Comparator interface.

```
public class LastNameComparator implements Comparator
```

2. Implemented the compare() method. This method should return 1, 0 or -1 based on comparison implemented inside the method. This method takes two objects as parameters, which should first be **casted** to the type of objects that we will store in the TreeSet and then return an int. In our case, we compared the last names of the two customer objects as shown below:

```
int index = c1.getLastName().compareTo(c2.getLastName());
```

3. We finally returned the index back.

Once we have the above comparator, we are now ready to use the TreeSet to sort the customer objects. Listing 7.7b & c shows the customer class and the test class.

Listing 7.7b (Customer.java) Customer class.

```
package utilities;  
  
public class Customer {  
  
    String firstName;  
    String lastName;  
    int ssn;  
  
    public Customer(String firstName, String lastName, int ssn) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.ssn = ssn;  
    }  
  
    public String getFirstName() {  
        return this.firstName;  
    }  
  
    public String getLastName() {
```

```
        return this.lastName;
    }

    public int getSSN() {
        return this.ssn;
    }
}
```

Listing 7.7c (CustomerSortTest.java) Customer class.

```
package utilities;

import java.util.*;

public class CustomerSortTest {

    public static void main(String args[]) {

        TreeSet set = new TreeSet(new LastNameComparator());

        set.add(new Customer("John", "Smith", 12345));
        set.add(new Customer("Laura", "Jackson", 456546));
        set.add(new Customer("Thomas", "Edison", 78989));

        Iterator itr = set.iterator();
        while (itr.hasNext()) {

            Customer customer = (Customer) itr.next();

            System.out.println(customer.getLastName() + "    "
                + customer.getFirstName() + "    " + customer.getSSN());
        }
    }
}
```

Look at the CustomerSortTest class. Before it added the customer objects to the treeset object, it created a TreeSet as shown below:

```
TreeSet set = new TreeSet(new LastNameComparator());
```

The rule is that we need to pass the object of the comparator class to the TreeSet constructor. Once we passed this object, the treeset knows how to sort the elements. We can start using the TreeSet add methods as usual to add the different customer objects. Compile and execute the test program and you'll see the following result.

Edison	Thomas	78989
Jackson	Laura	456546
Smith	John	12345

You can see from the above result, that the customer objects are sorted by last name in ascending order. If we want the lastnames sorted in descending order, all we need to do is put a negative sign in front of index variable in the `compare()` method as shown below:

```
return -index;
```

Homework: Modify the comparator class to sort by SSN.

Hint: The method should return +1 or 0 or -1 for greater than, equal to and less than comparisons.

This completes all the one dimensional data structures. Following is the summary of 1-D data structures.

- ✓ Use `ArrayList` to store objects when you **want to preserve the order** of the objects
- ✓ Use `add()` method to store the data
- ✓ Use `get()` method in a `for` loop to read the data

- ✓ Use `HashSet` to store objects when you **don't want duplicate objects**
- ✓ Use `add()` method to store the data
- ✓ Get the `Iterator` and use it with a while loop to read the objects.

- ✓ Use `TreeSet` to store objects when you **want to sort the objects**
- ✓ Use `add()` method to store the data
- ✓ Get the `Iterator` and use it with a while loop to read the objects.

You are good if you can just remember the above points.

Disadvantage with 1D data structures

The main disadvantage with 1D data structures is **searching** for a particular object. We need to iterate through the entire list and then get the object that matches the specified. This is where 2D data structures come in handy. They offer flexibility and faster way of retrieving the objects. So, let's see what they are.

Two Dimensional Data Structures

In two dimensional data structures, objects are stored in the form of a table using **key-object** pair as shown below:

key	Data Object
Susan	object 1
Tim	object 2
Mac	object3

If you notice the above representation, every data object (object 1, object 2, object3) is associated with a unique key. When a **key** is passed, the data structure will return the data object corresponding to this key. So, given a **key**, the data object can be retrieved in just ONE step, right? This is why searching is fast with 2D data structures.

Following is the hierarchy diagram for 2D data structures.

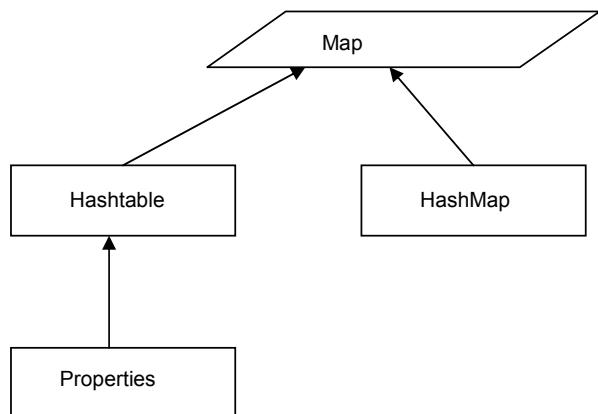


Fig 7.2 Map interface hierarchy

The top-level interface is the `Map` interface. `Hashtable` and `HashMap` are the two classes that implement this interface. `Properties` class is a child class of `Hashtable` class.

In this category of classes, we just have to learn how to use `HashMap`, `Hashtable` and `Properties` classes. They are very simple to use. Trust me. One more important thing is that we can only use these data structures only when every object we store has a unique key that identifies the object. There is no way you can store an object that doesn't have a key. Makes sense?

In all the above three classes, there are just 2 important methods we need to know, one for storing the object and the other for retrieving the object. With these things in mind, let's look at the 2D data structures one by one.

HashMap

A Hashmap object stores the data in the form of key-value pairs. Both the key and the value can be object. Normally, a String object is used as the key to store any other objects. Let's say we wish to store data about several customers who has firstname, lastname and ssn. To store this data, we need to first identify the unique key associated with each customer. In our case, we can use SSN as the key, since it is unique.

Let's use the same customer class we wrote in the previous example and use a HashMap to store different customer objects. Take a look at the code in listing 7.8.

Listing 7.8 (HashMapDemo.java) Class using Hashmap

```
package utilities;

import java.util.*;

public class HashMapDemo {

    public static void main(String args[]) {
        // Create 3 Customers

        Customer c1 = new Customer("John", "Smith", 12345);
        Customer c2 = new Customer("Laura", "Jackson", 456546);
        Customer c3 = new Customer("Thomas", "Edison", 78989);

        // Create a hashmap

        HashMap map = new HashMap();

        // Store the data using ssn as the key
        map.put("12345", c1);
        map.put("345678", c2);
        map.put("98765", c3);

        // Get Smiths info
        Customer customer = (Customer) map.get("345678");

        System.out.println(customer.getFirstName() + "      "
            + customer.getLastName() + "      " + customer.getSsn());
    }
}
```

Look at the above class. We first created three customer objects namely c1, c2, c3 by supplying all the data. We then created a HashMap object and used the `put()` method to store the data. This method takes the **key** as the first argument, and the data object as the second argument. Since we decided to use **ssn** as the key, the `put` method will be as shown below:

```
map.put("12345 ", c1);
```

Once we stored all the three customers data, we used the `get()` method to retrieve the data. This method takes the **key**, which is the **ssn**, and returns the customer object associated with that key. Note that, we need to do the object casting here to get the required customer object as shown below:

```
Customer customer = (Customer) map.get("345678");
```

Once we got the customer object, we simply displayed its properties. Compile and execute the class to see the following result.

```
Laura Jackson 456546
```

Hashtable

Hashtable works much the same as HashMap. If this is the case, then why do we need Hashtable? This is yet another interview question. The answer is again the same as with ArrayList and Vector. Hashtable can be safely used in multi-threaded environment, while HashMap cannot.

In the above program, just change the following statement

```
HashMap map = new HashMap() to  
Hashtable map = new Hashtable();
```

If you recompile and execute the above class, the output will still be the same. In most of the real world applications we use HashMap instead of Hashtable.

Note: If you look at both HashMap and Hashtable classes, the data objects that we can store can be any object. It can be Customer object, or Box object or anything else. But it's a good practice to always use String object as the key to all the data objects.

Properties

This data structure works much the same as HashMap, but the only difference is that we use it in situations where both the **key** and **value** is a String object.

This is one class that is widely used in real world applications. With this class, instead of using `get()` and `put()` methods like in HashMap, we use `getProperty()` and `setProperty()` methods. Take a look at the code in listing 7.9.

Listing 7.9 (`PropertiesDemo.java`) Class using properties.

```
package utilities;

import java.util.*;

public class PropertiesDemo {

    public static void main(String args[]) {
        Properties p = new Properties();
        p.setProperty("driver", "com.test.Driver");
        p.setProperty("url", "localhost:8080");

        String str = p.getProperty("url");
        System.out.println(str);
    }
}
```

As you can see in the above example, we used `setProperty()` method to which we passed string as both key and data. We then used `getProperty()` method by passing the key and reading the string value back. The `getProperty()` method instead of returning `Object` like the `get()` method, returns the `String` object and we don't have to do the casting. Compile and execute the above class to see the following result:

```
localhost:8080
```

This completes all the important data structures that we need to know. Most of the real world applications don't go beyond the above mentioned data structures unless otherwise the requirements are highly complex. Let's now look at two more important utility classes that are widely used while representing and processing dates.

Date class

This class is used to get the current system date. The usage is pretty simple. Look at the code in listing 7.10.

Listing 7.10 (`DateDemo.java`) Class using Date class.

```
package utilities;

import java.util.*;

public class DateDemo {

    public static void main(String args[]) {
        Date d = new Date();
    }
}
```

```
        System.out.println(d);
    }
}
```

The above program simply created a Date object and displayed it. If you compile and execute the program, you'll see the date displayed as shown below:

```
Tue Jan 13 12:34:789 2006 EST
```

If you notice the above result, the Date object displays both date and time. Though this is not harmful, sometimes in real world applications, we require the date to be displayed in different formats. For instance like **MMddyyyy** or **yyyy/MM/dd** etc. So, the question is, how do we format dates? The answer is using `SimpleDateFormat` class which is the next class we are going to discuss.

SimpleDateFormat

This class belongs `java.text` package, not the `java.util` package. We use this class, to format and parse the dates. To better understand how this class works, let's see the following two cases.

Case 1: Convert the given Date object into String in the form of **MM/dd/yyyy** format.

Following is how we achieve this.

```
SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy ");
String formattedDate = sdf.format( new Date() );
```

As you can see from the above statements, we first have to create `SimpleDateFormat` object by passing the format that we need. JVM will interpret **MM** as month, **dd** as day and **yyyy** as year. We then have to invoke the `format()` method by passing the `Date` object that is to be formatted. This method will then return the date as `String` object in the desired format.

Case 2: Given a date as String, convert it to Date object. This is the reverse of the above and we call it as *parsing* the date. Following is how we parse the dates.

```
SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy ");
Date d = sdf.parse("03/12/2005 ");
```

Following is what we did.

- ✓ Created a `SimpleDateFormat` object with the format of the date string

- ✓ Passed the date to the `parse()` method.

This `parse()` method returns the constructed `Date` object from the string. Just in case if it fails to parse the date, it will throw a `ParseException` that should be caught. This is a checked exception and we need to enclose the method call in a **try-catch** block. See the code in listing 7.11.

Listing 7.11 (`SimpleDateFormatDemo.java`) Class using date formatter class.

```
package utilities;

import java.util.*;
import java.text.*;

public class SimpleDateFormatDemo {

    public static void main(String args[]) {

        // Case 1
        SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy");
        String formattedDate = sdf.format(new Date());

        System.out.println(formattedDate);

        // Case 2
        sdf = new SimpleDateFormat("MM/dd/yyyy");
        try {

            Date d = sdf.parse("03/12/2005");
            System.out.println(d);

        } catch (ParseException pe) {
            System.out.println(" Unable to parse Date");
        }
    }
}
```

Compile and execute the above program to see the following result:

```
03/12/2006
Tue Mar 03 12:34:789 2005 EST
```

Try changing the following statement

```
Date d = sdf.parse("03/12/2005"); to
Date d = sdf.parse("2005/03/12");
```

You'll see the message "Unable to parse Date", as it throws exception due to improper format.

Let's see one last class called `StringTokenizer`, which is also used very frequently in real world applications.

StringTokenizer

This class is a cool class that breaks the text using a delimiter. For instance, consider the following text data.

```
String str = "12345,abcd,Y,98765,ABCD ";
```

Let's say that someone gives us a comma delimited text in the above form, and we are required to break it into pieces at the delimiter. This is when we use the `StringTokenizer` class. It *tokenizes* the original text and gives us individual *tokens*. This class has two methods namely `hasMoreTokens()` and `nextToken()`. Let's see how to use this class.

Firstly, we need to create an object of the class by passing the original text and the delimiter where we want to break it as shown below:

```
String str = "12345,abcd,Y,98765,ABCD ";
StringTokenizer st = new StringTokenizer (str , ",");
```

We then have to use the `hasMoreTokens()` and `nextToken()` methods to get the individual strings (tokens). The description for these two methods is given below:

`boolean hasMoreTokens ()` : This method returns true if there are more tokens left.
`String nextToken ()` : This method returns the actual token.

To better understand, look at the code in listing 7.12.

Listing 7.12 (StringTokenizerDemo.java) Using string tokenizer class.

```
package utilities;

import java.util.StringTokenizer;

public class StringTokenizerDemo {

    public static void main(String args[]) {

        String str = "12345,abcd,Y,98765,ABCD";
        StringTokenizer st = new StringTokenizer(str, ",");

        while (st.hasMoreTokens()) {
            String token = st.nextToken();
            System.out.println(token);
        }
    }
}
```

```
    }  
}  
}
```

The while loop works like, *"If you have more tokens, then give me the next token"*. Compile and execute the program to see the following result.

```
12345  
abcd  
Y  
98765  
ABCD
```

This are all the utility classes that we need to know to write any complex Java programs. Most of the real world applications use these classes to simplify the development. Just learn them once. Even if you couldn't remember the method names, that's OK as long as you know how to use them. So, be cool.

The next few pages give you an insight of some of the best coding practices, conventions that we usually follow in al the real world Java applications.

Naming Conventions

Using meaningful names in programs improves the readability of the program. We don't want to write a program in such a way that people who use the program have a tough time understanding what the program is all about.

Following are the naming conventions we should follow while writing programs:

1. Package names must be in lowercase. As an example.
`src, com.utilities etc.`
2. Class names must always start with a uppercase letter, and every word in the class name must also start with an uppercase letter as shown below:

`InitialContext, BusinessDelegate, BeanUtilityFactory etc.`

3. Method names must always start with a lowercase letter, and every other word in the method name must start with an uppercase letter as shown below:
`getName, setName, parseNameLine etc.`
4. Any method that returns a boolean should start with "has" or "is" as shown

below:

hasMoreElements() , isNameValid(), hasPermissions() etc

5. All the boolean variable names should start with the words “is” or “has”, as shown below:

```
boolean isMarried;  
boolean hasStatements;
```

Avoid using names like ‘flag’, ‘status’ etc.

6. Constant names must always be declared as **static** and **final**, and should always be in uppercase with words separated by an underscore.

```
static final int MAX_QTY = 100;  
static final String VAL_CHECKING = "Checkings";  
static final String PROVIDER_URL = "localhost:8080";
```

Always follow the above rules while writing the Java programs. This will really help you in making your life easy. Moreover, these conventions are followed by every one in the Java community. So, let’s also be on the same page with them.

Note: Writing a Java program is not the key here. Wring a good Java program following the best practices and coding conventions is the key. In every single Java project, there will be dedicated personal who strictly enforces the usage of the above standards. So, it’s always good that we learn to use them right from the beginning.

Java Documentation

Documentation of the program is the most important thing that is very strictly enforced in all the real world applications. A program without any documentation is simply of no use at all. Trust me. We write programs so that someone uses them. If there is no documentation, how will the users of the program know how to use it?

Documenting a Java program is different from using comments in a program. Using comments is a low-level documentation that aids the programmers, but not the public users. For public users, we need high-level documentation, which is what we’ll see in this section.

We use comments inside the program to make the program self-explanatory. When somebody looks at our program, reading the comments must tell that what the program

is all about. The comments must be written in such a way, that even non-Java programmers must be able to understand the program. We already wrote several programs using comments as shown below:

```
// This is a single line comment  
  
/*  
 Following statements uses the SSN  
 to check whether or not the candidate is  
 eligible for a Credit Card.  
 */
```

Now, it's time to learn how to generate high-level documentation for a Java class. Trust me, this is very simple. We use a tool called **javadoc** to generate this documentation. All the documentation will be generated in HTML format, so that you can also publish it online. The documentation that the javadoc tool generates is so cool, that you'll be surprised after seeing it.

Documentation using javadoc tool

javadoc is a documentation generating utility tool that comes along with the java installation. Good. This tool generates the following documentation for a given class.

1. Class details like who the parent class is, what interfaces it implements, what child classes it has and so on.
2. Lists all the method names and a description on how to use the methods, what exceptions does the method throws, what parameters it takes, what values it return back and all that good stuff.
3. Lists all the global variables and constants.

In order to generate the above documentation for a class, we need to first write the descriptions in the class using **javadoc** tags. These are very simple to use, and you can master them in few minutes. Following table lists the six most important **javadoc** tags.

Table 7.2 Javadoc tags

Tag Name	Description
@author	Used to specify the name of the person who wrote the class
@since	Used to specify the creation date of the class
@version	Used to specify the version of the class

<code>@param</code>	Used to specify the method parameters
<code>@return</code>	Used to specify what the method returns
<code>@throws</code>	Used to list all the exceptions a method throws.
<code>@author</code>	Used to specify the name of the person who wrote the class

The best way to learn the usage of the above tags is by writing a program. Look at the following example that uses the javadoc tags.

```
package utilities;

import java.util.*;
import java.io.*;

/**
 * Gateway class that has several methods to retrieve customers
 * information.
 *
 * @author Steve
 * @since October, 2006
 * @version 1.0
 */
public class CustomerGateway {

    // Global Variables
    private static final String MAX_ACCOUNTS = 3;

    /**
     * Retrieves the Customer profile based on SSN.
     *
     * @param ssn - The SSN of the Customer
     * @return String - The profile of the Customer
     * @throws IOException - Thrown when Customer is not found
     */
    public String getCustomerProfile (String ssn) throws IOException
    {
        return "John, 1234 Broad St, NY";
    }
}
```

Copy the above program into the “utilities” directory and execute the **javadoc** command as shown below:

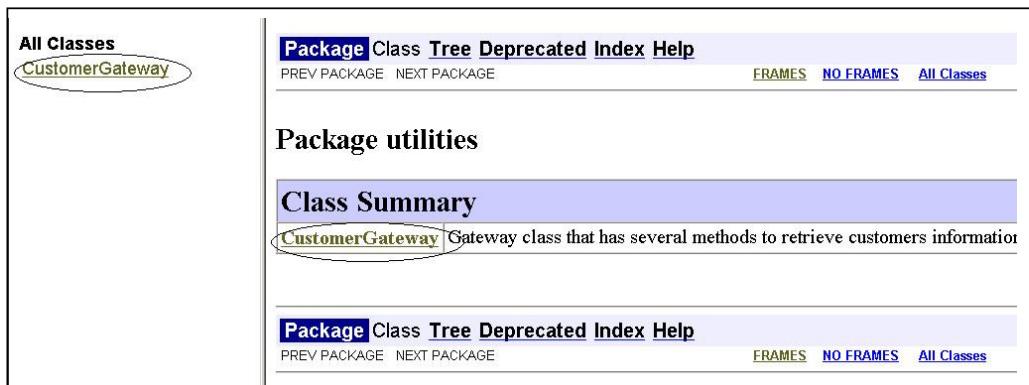
```
C:\JavaTraining\Chapter6>javadoc utilities\CustomerGateway.java -d docs
```

In the above command,

- ✓ **javadoc** is the tool name
- ✓ **utilities\CustomerGateway.java** is the program we wanted to generate the documentation and

- ✓ **-d docs** specifies the directory in which we want to store the documentation

The above command generates several HTML files and stores them in the **docs** directory. The starting point of the documentation is `index.html`. If you open this file, you'll see the following:



The `index.html` lists the class as shown above. Click on the class name, and it displays the content shown in Fig 7.3 in the next page.

The page displays all the details of the class. It has four sections as listed below:

Section 1 lists the name of the class, and its inheritance hierarchy. Since the class doesn't extend from any class, it is an automatic child of `Object` class as shown.

Section 2 displays the class description, the author name, creation date and version. This information is specified in our class right before the class name as shown below:

```
/**  
 * Gateway class that has several methods to retrieve customers  
 * information.  
 *  
 * @author Steve  
 * @since October, 2006  
 * @version 1.0  
 */  
public class CustomerGateway {
```

The screenshot shows a Java documentation page for the `CustomerGateway` class. The page is structured as follows:

- All Classes**: A link to a list of all classes.
- CustomerGateway**: The current class being viewed.
- utilities**: The package name.
- Class CustomerGateway**: The title of the class.
- java.lang.Object**: The superclass of the class.
- public class CustomerGateway**: The class definition line.
- extends java.lang.Object**: The inheritance line.
- Gateway class that has several methods to retrieve customers information.**: The class description.
- Since:** October, 2006
- Version:** 1.0
- Author:** Steve
- Constructor Summary**: A section header.
- `CustomerGateway()`**: A constructor entry.
- Method Summary**: A section header.
- `java.lang.String getCustomerProfile(java.lang.String ssn)`**: A method entry with its return type and name.
- Retrieves the Customer profile based on SSN.**: A brief description of the method's purpose.

Fig 7.3 Class Details Page

Any Javadoc comment will have a short description followed by the tags. All the javadoc comments **must** be enclosed within `/**` and `*/` and every line must begin with a `*` as shown above.

Section 3 displays the constructors of the class. Since we didn't define any constructor, as we learned before, JVM will supply with a default constructor. This is what is shown here. However, if you defined your own constructors, they would be displayed too.

Section 4 gives the method summary of the class. It will display all the methods in the class as hyperlinks along with a description, the method parameters and the return type. If you click on the link, it will show you the complete details of the method as shown below:

Method Detail

getCustomerProfile

```
public java.lang.String getCustomerProfile(java.lang.String ssn)
                                         throws java.io.IOException
```

Retrieves the Customer profile based on SSN.

Parameters:

ssn -- The SSN of the Customer

Returns:

String -The profile of the Customer

Throws:

java.io.IOException -- Thrown when Customer is not found

If you notice the above figure, it displayed the entire method signature with the following items:

1. The description of the method.
2. The parameters to it
3. The data it returns back
4. All the possible exceptions it can throw.

All the above 4 items came from the following chunk in our code:

```
/**
 * Retrieves the Customer profile based on SSN.
 *
 * @param ssn - The SSN of the Customer
 * @return String - The profile of the Customer
 * @throws IOException - Thrown when Customer is not found
 */
public String getCustomerProfile (String ssn) throws IOException
```

Isn't the documentation beautiful? Are you surprised? I am sure you are. The good thing is that we just have to specify the information in the program, and the **javadoc** tool will take care of the rest. In real world applications, there will be several methods, and you need to follow the same procedure for all the methods.

In the above example, we just generated the documentation for one class. However, in real world applications there will be tons of classes in a single package, and you can use the following command to generate documentation for all in one step as shown below:

```
C:\JavaTraining\Chapter7>javadoc utilities\*.java -d docs
```

The above command generates the documentation for all the java files in the **utilities** package, and stores it in the **docs** directory.

This is all you need to know about the **javadoc** tool and the **tags** to use in the program to specify the descriptions.

This completes all the important core classes that I'd like you to know to be more comfortable with Java programming. All we did is, learned some library classes that are frequently used in real world applications and more importantly the best practices and coding conventions. I am sure you enjoyed this chapter as much as you did with the previous chapters, so let's summarize.

Summary

- ✓ Java comes with rich set of built in library classes organized in packages.
- ✓ The two most important packages are **java.lang** and **java.util**.
- ✓ To use a library class in a package, we first need to import the package using the **import** statements. A program can import any number of classes and packages.
- ✓ The important classes in **java.lang** package are **wrapper classes**, **String**, **StringBuffer** and **StringTokenizer**.
- ✓ Wrapper classes like **Integer**, **Float**, **Double**, **Long** etc represent the corresponding primitive variables as objects.
- ✓ **String** class is the widely used class as it represents **text** data, and also has several methods for **text** processing. The only disadvantage with **String** class is that JVM creates a completely new **String** object when the text is modified.
- ✓ **StringBuffer** class must be used in situation where complex text processing is required.
- ✓ **StringTokenizer** class is used to break a large string into smaller strings using a delimiter.
- ✓ The classes in **java.util** package forms the Collection Framework. These classes are used to store data objects.
- ✓ Data Structures are broadly classified as **one-dimensional** and **two-dimensional**.
- ✓ **List** and **Set** interfaces represent 1D data structures.
- ✓ **List** interface represents **ordered**, **duplicate** and **unsorted** objects.
- ✓ **ArrayList** and **Vector** are the two important classes that implement **List** interface.

- ✓ **Set** interface represents **unordered**, **unique** and **unsorted** objects.
- ✓ **HashSet** is one important class that implements the **Set** interface.
- ✓ **TreeSet** class is same as **HashSet** class, but it sorts the objects in ascending order.
- ✓ **Map** interface represent 2D data structures.
- ✓ Data is stored in the form of table using **key-object** pairs.
- ✓ **HashMap** and **Hashtable** classes implements **Map** interface.
- ✓ Use **HashMap** instead of **Hashtable** in thread-safe environments.
- ✓ **Properties** object is used when **key** and **object** are **Strings**.
- ✓ Use **Date** class to display dates.
- ✓ Use **SimpleDateFormat** to **format** and **parse** dates.
- ✓ **javadoc** tool is used for generating documentation for classes. To use the tool, we first need to write the **javadoc tags** in the program.
- ✓ Using **javadoc tags** is a best practice and is always enforced in all the real world applications.
- ✓ A good Java program must always follow the best practices and coding conventions.

Time to play 50-50

1. Which of the following is a wrapper class?
 - a) integer
 - b) Integer
2. Which of the following is a method in String class?
 - a) substring
 - b) substrng
3. Which of the following should be used for breaking the Strings?
 - a) StringBuffer
 - b) StringTokenizer
4. Which of the following are the methods in StringTokenizer class?
 - a) hasNext() and next()
 - b) hasMoreTokens() and nextToken()

5. Which of the following is the top-level interface in the Collections framework
 - a) Collection
 - b) Object
6. Which of the following interface represents **ORDERED** elements?
 - a) Set
 - b) List
7. Which of the following class implements **List** interface?
 - a) HashMap
 - b) ArrayList
8. Which of the following is used to store **UNIQUE** elements?
 - a) ArrayList
 - b) HashSet
9. Which of the following sorts the elements in ascending order?
 - a) HashSet
 - b) TreeSet
10. Which of the following class stores the data as key-value pairs?
 - a) HashSet
 - b) HashMap
11. Which of the following are methods in Hashtable class?
 - a) get() and put()
 - b) getAttribute() and setAttribute()
12. Which of the following are methods in Properties class?
 - a) getAttribute() and setAttribute()
 - b) getProperty() and setProperty()
13. Which of the following class is used to display dates?

- a) Date
- b) SimpleDateFormat

14. Which of the following class should be used to format and parse dates?

- a) Date
- b) SimpleDateFormat

15. Which of the following are the methods in SimpleDateFormat class?

- a) format() and parse()
- b) formatDate() and parseDate()

16. Which of the following checked exception is thrown by the parse() method in the SimpleDateFormat class?

- a) ParseException
- b) ParsingException

Interview Questions

Question: What is the difference between String and StringBuffer classes?

Answer: String is an immutable object which means that JVM will create a new string object when ever the contents are modified. StringBuffer is a mutable object and JVM will use the same object while the contents are modified. This is also the reason why StringBuffer offers better performance than String class when complex text processing is involved.

Question: List some of the collection data structures in java.util package.

Answer: ArrayList, Vector, HashSet, TreeSet, HashMap, Hashtable, Properties.

Question: What is the difference between ArrayList and Vector class?

Answer: Vector is thread safe than ArrayList since the methods in Vector are synchronized.

Question: What is the difference between HashMap and Hashtable?

Answer: Hashtable is thread safe than HashMap. This is because all the methods in Hashtable are synchronized by default. The second difference is that, HashMap allows null keys and values, while Hashtable doesn't.

Question: Which class is used to format dates?

Answer: SimpleDateFormat

Question: What is the difference between Hashtable and Properties?

Answer: Properties is a child class of Hashtable which is used when both the key and value are Strings. Using Properties, we don't have to cast the data, while casting data is required with Hashtable.

This is the end of this chapter. You know what, this chapter is an unofficial end to the Java fundamentals section. However, I suggest you to take few minutes to read the next two very small chapters. So, let's rock and roll.

Chapter 8

Threads

This chapter will introduce you to concurrent processing in Java. Java language has built-in support for writing programs that run simultaneously. By the end of this chapter, you'll understand the basics of how multi threading is done.

Chapter Goals

- ✓ Understand the importance of concurrent processing
- ✓ Understand the API for multi-threading.
- ✓ Understand the life cycle of threads
- ✓ Understand synchronization

Environment Setup

All the programs in this chapter should be stored in the following directory.

```
C:/JavaTraining/Chapter8/threads
```

`threads` is a package in which we will store the programs. To compile the programs, move to the following root directory and get ready.

```
C:/JavaTraining/Chapter8
```

Introduction

Ever since the invention of first computer named Abacus to the latest and super fast computers, our primary goal is to improve the “speed” with which computing is done. Over several years, based on the need for high speed computing, computers have gone through radical improvements using sophisticated semiconductor technologies and today we are in the world of Supercomputers that compute trillions of instructions per second. So, how is this achieved? Good question. Let’s start with the term “processor”. A processor is a hardware device in a computer that processes the instructions. So, for faster processing we need super fast processors. This is where the silicon technology comes into picture. This technology helped us to build super fast processors for computing. Once we have super fast processors it is very important to use them to their full potential, right? There is no point in having a supercomputer and making it sit idle for most of the time waiting for something to happen. All I mean to say is that, using a super fast processor effectively and efficiently is more important than just having the processor.

Creating the processor is the job of semi conductor industry. Using the processor effectively and efficiently is the job of the operating system of the computer. Over the years operating systems too have undergone significant changes incorporating better algorithms to better utilize the CPU and one thing that contributed significantly in this space is the “multi-tasking” or “multi-processing”. There is also something called “multi-threading” which is often confused with “multi-processing”. To tell you, both these are completely different but they are related with other. Let’s see what these are.

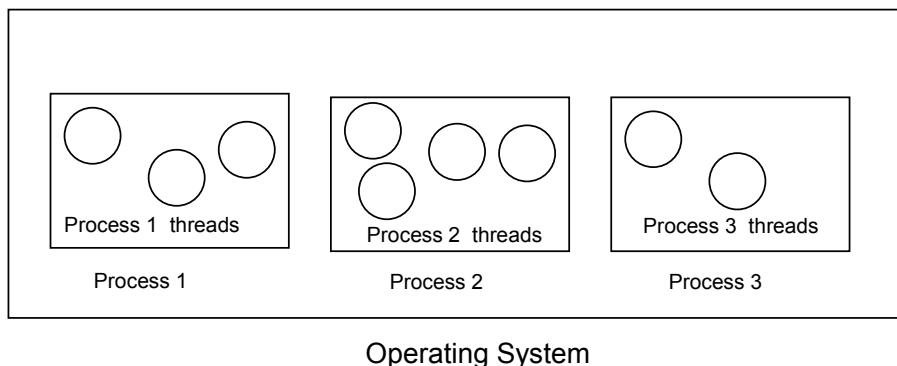
Multi-Processing

Multi-Processing is the ability of CPU to concurrently execute multiple processes at the same time. So, what is a process? A process is nothing but any currently running application. For instance word processing application, Email application, Web application etc. We see multi-processing when ever we use a computer. We open a word document, a web page and many more applications and use them simultaneously, right? Some operating systems also allow us to monitor the processes that are running on the computer.

Multi-Threading

Before we understand what multi-threading is, we need to know something called “Thread”. A thread is nothing but a sub process that runs within a process. For instance a grammar check thread in a word processing document. Based on the type of the

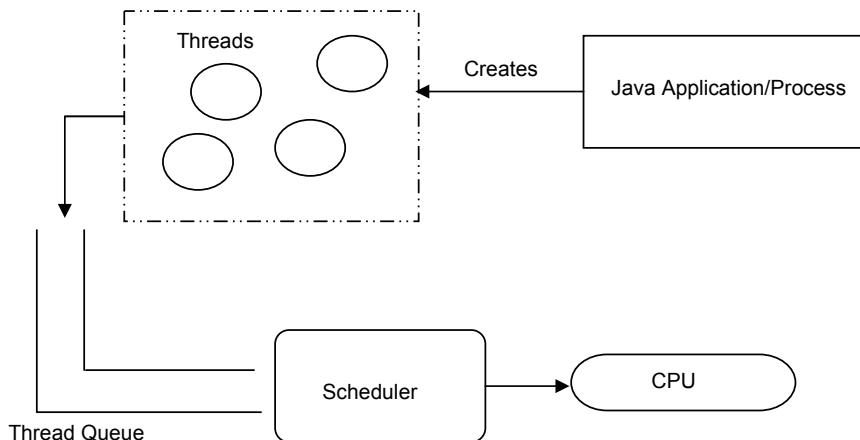
process, it can have any number of sub-processes (threads) and it may require that these threads be running concurrently within the process. This is what we call as multi-threading. So, multi-threading is the ability of the CPU to execute multiple threads within a “single process” simultaneously. Following picture should help you understand the difference between multi-processing and multi-threading.



As you can see from the above figure, every process will have its own threads and will never interfere with another process threads. In this chapter, we will see how we can use Java to work with threads and how to build multi-threaded applications.

How Multi-Threading is done?

This is a big question. To understand this clearly, look at the following figure.



As you can see from the above figure, there are several components that participate in multi-threaded programming. Following explains the roles and responsibilities of each of the above components.

Java Application/Process

This is the application that we as Java developers write using the multi-threading features of Java language. Its main responsibility is to *create* threads, *set the priorities* to threads and *start* the threads. Once it does all these actions, its job is done. There is nothing it can do or control how the threads should run or when they should run and all that stuff.

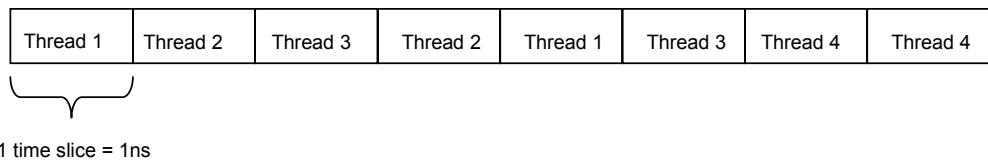
Thread Queue

This is a component that will contain all the threads that are created by the above Java application and also the threads from other processes. There will not be a separate queue for every process. All the threads will be lined up in this Queue waiting to get into the CPU.

Scheduler

This is the component that manages the threads. The scheduler acts like a manager and decides which thread should be sent to the CPU based on things like priority and some thing else. If there is a thread in the queue with high priority sitting behind than the lesser priority one, it **may** pick the highest priority thread and send it to the CPU. However, it is important to understand that this behavior is never guaranteed. The scheduler can send a low priority thread to the CPU ahead of a high priority thread.

The scheduler component comes with the operating system. Most schedulers follow “time-slicing” algorithm for managing threads. With this algorithm, the scheduler will send a thread to the CPU for a small duration of time (called as time slice). Once the time slice is completed, it pulls the thread out of the CPU even if it is partially executed and puts it back in the thread queue. It then sends the next thread in the queue to the CPU and again pulls it back after the time slice. When a previously pulled out thread gets a chance again to go to the CPU, it will resume executing from where it stopped before. This switching between the threads by the scheduler will be so fast that from the application point of view it looks like all the threads are running simultaneously. Following figure should give you some idea on how the scheduler manages the threads.

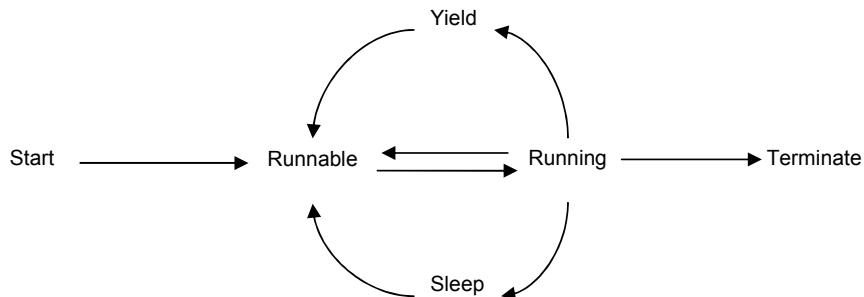


As you can see from the above figure, every thread will get a 1 slice of CPU time at a time. By the end of the cycle, the scheduler makes sure that all the threads are executed completely by the CPU. The time slice will normally be in nano and pico seconds based on the CPU frequency which cannot be recognized by the naked eye and this why we feel the simulation of multi-threading or concurrency behavior.

Now that we know how multi-threading is implemented internally, let's understand the life cycle of a typical thread.

Thread Lifecycle

Following are the states a thread can go from the creation to its death.



As you can see from the above state diagram, once the thread is started, it goes right away to the runnable state. This is the state of the thread in the *queue* and is ready to be executed whenever the scheduler decides. From the runnable state, it goes to the running state. This is the state when the thread is inside the CPU. If the thread is completely executed, then from the running state it directly goes to the terminated state which is a dead state. Otherwise, it will or can go to the runnable state again in the following 3 ways.

1. A running thread can voluntarily yield or relinquish the CPU and go back to the runnable state. This is normally done by the thread itself when ever it thinks that

its job is not a high priority one. These are good threads because they care about other threads. If however there are no other threads in the queue, the scheduler will send it back to the CPU again.

2. A running thread can sleep for a definite amount of time and then go to the runnable state. In this case, you are always guaranteed that the next thread comes to the CPU only after t_1 sec where t_1 is the amount of time the thread slept. Usually threads will sleep because their operation is dependent on the execution of other threads.
3. A running thread can be normally brought to the runnable state by the scheduler.

Now that we know the typical lifecycle of a thread, let's see how we can do multi-threading in Java.

Multi-Threading in Java

Java language supports multi-threading using the standard built-in classes. The good thing is that there is just one class and one interface we need to know to start writing multi-threaded applications. These are

1. `java.lang.Runnable` interface
2. `java.lang.Thread` class

Following table lists the important methods in each of the above.

Thread class

Table 8.1 Thread API

Method Name	Description
<code>String getName()</code>	Returns the name of the thread
<code>int getPriority()</code>	Returns the priority of the thread
<code>boolean isAlive()</code>	Determines if the thread is running or not
<code>void run()</code>	Entry point of a thread
<code>void start()</code>	Method to start the thread
<code>void sleep()</code>	Makes the current thread sleep of certain duration of time
<code>void setPriority()</code>	Sets the priority

static Thread currentThread()	Returns the current thread
----------------------------------	----------------------------

Runnable Interface

The only method in this interface is the “run” method as shown below:

```
public void run();
```

Main Thread

As I said before, threads are nothing but sub processes within a parent process. It is this parent process that creates the threads. Let me ask you a question. What if there is no parent process? The answer is simple. There will not be any threads. Therefore, before we start creating the threads, we need to have a parent process. So, the question is, who is this parent process? You know what; the class that has the **main** method is itself the parent process and we call it as the **main thread**. It is within this **main** method, we will start spawning threads. So, let’s prove this fact by writing a simple program.

Take a look at the code in listing 8.1.

Listing 8.1 (TestThread.java) Main thread class.

```
package threads;  
  
public class TestThread {  
  
    public static void main(String args[]) {  
  
        Thread t = Thread.currentThread();  
  
        System.out.println(t.getName());  
        System.out.println(t.getPriority());  
    }  
}
```

Compile and execute the above program to see the following result.

```
main  
5
```

As you can see from the output, the name of the thread is “main” which is the parent thread and its priority is 5. This is the default priority of the main thread. The `currentThread()` method returns the `Thread` object that is actually running the

code. In our case it returned as the **main** thread. This proves that main method itself is the parent thread. Let's look at mode details on threads from here onwards.

Creating a Thread

In Java, there are two ways of creating a thread as shown below:

1. Implementing the `Runnable` interface
2. Extending the `Thread` class.

You might wonder why we need two different ways of creating a thread. We'll get to the answer later. Let's first understand how to create the threads using both ways.

Runnable Interface

Creating a thread using a runnable interface involves 3 simple steps.

1. Write a class that implement the `Runnable` interface and implement the `run()` method.
2. Create a `Thread` object by passing the `Runnable` object
3. Start the thread.

See the following code in listing 8.2 that demonstrates the above three simple steps.

Listing 8.2a (MyThread.java) Using Runnable Interface

```
package threads;

public class MyThread implements Runnable {

    /*
     * Any statements that should be executed concurrently
     * must go in this method.
     */
    public void run() {
        System.out.println("Executed the thread");
    }
}
```

As you can see from the above code, we first wrote a thread class that implements the `Runnbale` interface with the `run()` method. All the statements that should be executed concurrently must be placed in this method. This is the rule. Ok. Now, take a look at the test class in listing 8.2b that creates and starts this thread.

Listing 8.2b (`RunnableTest.java`) Test class creating a runnable thread.

```
package threads;

public class RunnableTest {

    public static void main(String args[]) {

        // Step 1
        MyThread t1 = new MyThread();

        // Step 2
        Thread th = new Thread(t1);

        // Step 3
        th.start();

        System.out.println("Main Thread about to terminate");

        System.out.println("Main Thread terminated");
    }
}
```

In the above test class, the **main** method (parent thread) created the object of `MyThread` class, and passed it to the `Thread` object constructor. Finally it started the thread using the `start()` method. When the `start()` method is invoked, the thread will go to runnable state (queue) and is now at the mercy of scheduler. When the CPU executes the thread (which is `MyThread`), it executes the statements in its `run()` method. The rule is, place all the statements that are to be executed inside the thread in the `run()` method.

Compile and execute the programs as shown below:

```
C:/JavaTraining>chapter8>javac threads\*.java
C:/JavaTraining>chapter8>java threads.RunnableTest
```

The result of the above test class *may* be the following:

```
Main Thread about to terminate
Main Thread terminated
Executed the thread
```

From the output, you can see that it executed the main thread first, followed by `MyThread`. On some other computer, the output statements may be reversed. One thing you need to always keep in mind is that with multi-threading *the output is never guaranteed to be same* on all computers or even by running the same program again and again. This is because every OS will have its own scheduler program that functions differently from others.

Thread Priorities

In Java, we can also set the priority of a given thread. Thread priority gives the programmer a way to *only suggest* the scheduler to run the thread ahead of other low priority threads. But again, the scheduler decides whether or not to obey this order. You as a developer can only suggest but not command the scheduler. There are three priorities we can set on a given thread such as low, medium and high as shown below:

```
t.setPriority(Thread.MIN_PRIORITY);  
t.setPriority(Thread.NORMAL_PRIORITY);  
t.setPriority(Thread.MAX_PRIORITY);
```

where

```
MIN_PRIORITY = 0  
NORMAL_PRIORITY = 5  
MAX_PRIORITY = 10
```

In the previous program, let's set a high priority as shown below to see if our thread gets executed before the main thread.

```
MyThread t1 = new MyThread();  
Thread th = new Thread(t1);  
th.setPriority(Thread.MAX_PRIORITY);  
th.start();
```

Compile and execute the program again with the above statement and see if the output is reversed or not. The result can be something like shown below:

```
Main Thread about to terminate  
Executed the thread  
Main Thread terminated
```

Creating a Thread using Thread class

In this case, we need to write a thread class that extends the Thread class and overwrite the run() method. This is just a different style, ok. See the code in listing 8.3.

Listing 8.3a (AnotherThread.java) Thread class.

```
package threads;  
  
public class AnotherThread extends Thread {  
  
    public void run() {  
        System.out.println("Executed the thread");  
    }  
}
```

```
}
```

As you can see, we created a thread class by inheriting from Thread class as shown below:

```
public class AnotherThread extends Thread
```

We then overwrote the `run()` method. Easy, right? Let's now write the main class that creates the above thread.

Listing 8.3b (ThreadTest.java) Test class for creating a thread.

```
package threads;

public class ThreadTest {

    public static void main(String args[]) {
        AnotherThread t1 = new AnotherThread();
        t1.start();
        System.out.println("Main Thread completed");
    }
}
```

As you can see from the above test class, to create a thread all we need to do is,

1. Create the object of thread class and
2. Start the thread.

This is the simplest way of creating a thread. Most applications uses this style of creating threads. Now, let's find out the answer to the question we left couple pages ago. When do we use Thread class and when do we use Runnable interface?

Answer: Use **implements Runnable** style when the thread class should also be a child class of some other non-thread parent class as shown below:

```
public class MyThread extends SomeNonThreadParent implements Runnable
```

If however the class doesn't have to inherit from any other non-thread parent class, then use the **extends Thread** style.

Thread Yielding

A thread can voluntarily give up the CPU to give other threads a chance to get into the CPU. So why should the thread so graciously yield the CPU? Usually when we have a threads that need to run for long duration of time say for several hours, in such cases it would be beneficial to make the thread yield at regular intervals of time so that we can take decision whether or not we want to proceed further or terminate the thread.

As an example, let's say your notebook started running a background process to backup your hard disk and takes like 4 hours to complete the job. If it doesn't yield, you have to patiently wait for 4 hours before you leave home which I know is not good at all. On the other hand, if the thread gives you some alert at regular intervals of time whether or not to continue further, then based on the importance of the job you can decide to either kill it or run it.

Another typical application is the threads that do less important work. In such cases graciously giving up the CPU for other threads that are deemed to be of utmost critical is a good thing. Let's write a small example where we have 4 threads namely A,B,C,D. What we will do is, set a priority of 5 for thread A, and priority 10 for B,C,D making them more critical than A. Ok. Listing 8.4 shows the code for low priority thread and high priority threads.

Listing 8.4a (`LessImportantThread.java`) Low Priority Thread

```
package threads;
public class LessImportantThread extends Thread {

    public LessImportantThread(String name) {
        super(name);
    }
    public void run() {

        System.out.println(getName() + " Running");
        System.out.println(getName() + " yielded");

        yield();

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
        }

        System.out.println(getName() + " Running Again");
    }
}
```

Let's look at the `run()` method in the above `LessImportantThread` class.

```
public void run() {
    System.out.println( getName() + " Running");
    System.out.println( getName() + " yielded");
    yield();
    System.out.println( getName() + " Running Again");
}
```

As you can see from the above code, the thread yielded by calling the `yield()` method. This method inherited from the `Thread` class. When it yielded, the scheduler may give the other threads a chance to execute because they are set high priority.

Listing 8.4b (`CriticalThread`) High priority Thread

```
package threads;
public class CriticalThread extends Thread {
    public CriticalThread(String name) {
        super(name);
    }
    public void run() {
        System.out.println(getName() + " Running");
    }
}
```

The `run()` method in the `CriticalThread` class is pretty simple. Its constructor takes a name argument and passes it to parent class constructor. The `Thread` class has a method called `getName()` which when used, will return back the thread name. With the above two thread classes, let's now look at the `ThreadStarter` parent thread class.

Listing 8.4c (`ThreadStarter`) Main thread class spawning multiple threads.

```
package threads;
public class ThreadStarter {
    public static void main(String args[]) {
        LessImportantThread t = new LessImportantThread("A");
        t.setPriority(5);
        t.start();
    }
}
```

```
CriticalThread t1 = new CriticalThread("B");
t1.setPriority(10);
t1.start();

CriticalThread t2 = new CriticalThread("C");
t2.setPriority(10);
t2.start();

CriticalThread t3 = new CriticalThread("D");
t3.setPriority(10);
t3.start();

System.out.println("Main Thread completed");
}
}
```

The above class simply created one less useful thread with the name A and priority 5, and three critical threads with names B,C,D with priorities 10. Usually the thread name should be passed to the Thread class which is why we used the following statement in the thread constructors:

```
super(name);
```

To display the thread name for debugging purposes, we can call the `getName()` method which prints the thread name. This is exactly what we did in the `run()` methods. Compile and execute the above classes as shown below:

```
C:/JavaTraining>chapter8>javac threads\*.java
C:/JavaTraining>chapter8>java threads.ThreadStarter
```

The result of the above program *may* be as shown below:

```
B Running
A Running
A yielded
C Running
D Running
Main Thread completed
A Running Again
```

Looking at the above result, when A yielded, our scheduler sent C and D threads to the CPU followed by main thread. Finally our less useful thread A came back and finished its work.

One of the widely used application of threads is the resource sharing where multiple threads compete for the same resource at the same time. This is when the resource will get messed up. This scenario can be viewed as four kids fighting for the same toy and you all know what will happen to the toy ultimately. So the solution to keep the toy

from breaking is to make the kids (threads) get hold of the toy(resource) one after the other. This is what we call as synchronizing the threads to share the resource.

In our example we will use a printer object as the resource that multiple threads use to print the documents. We will look at two cases, one without synchronizing and other with synchronizing the threads and compare the results. Fig 8.1 demonstrates the idea for this example.

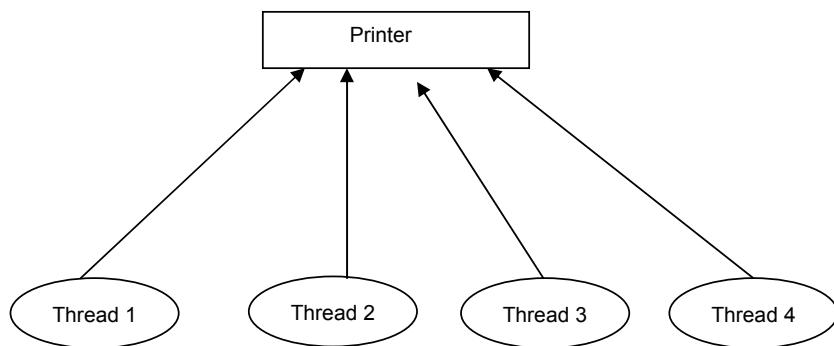


Fig 8.1 Multiple threads accessing the same resource

Listing 8.5a (Printer) Resource Class

```
package threads;

public class Printer {

    public void print(String data) {
        System.out.println("Printing " + data + " started");

        try {
            // Sleep for 1 second (Simulating the printing process)
            Thread.sleep(1000);
        } catch (Exception e) {
        }

        System.out.println("Printing " + data + " completed");
    }
}
```

Look at the above `Printer` class. This class has a `print()` method that takes the data and prints it. Just to make the printer take at least 1sec (1000ms) to print the document, we add the `sleep()` method to sleep for 1 sec. Now look at the thread class shown in listing 8.5b.

Listing 8.5b (PrinterThread) Thread class using Printer resource

```
package threads;

public class PrinterThread extends Thread {

    Printer p;
    String data;

    public PrinterThread(Printer p, String data) {
        this.p = p;
        this.data = data;
    }

    public void run() {
        p.print(data);
    }
}
```

The above thread class takes the `Printer` object reference along with the data through the constructor. When the CPU executes the thread, the `run()` method simply calls the `print()` method on the `Printer` object by passing the data as shown below:

```
public void run() {
    p.print(data);
}
```

Listing 8.5c (PrinterTest) Test class creating threads to access the same printer.

```
package threads;

public class PrinterTest {

    public static void main(String[] args) {
        // Create the resource
        Printer p = new Printer();

        // Create Threads

        PrinterThread p1 = new PrinterThread(p, "Document1");
        PrinterThread p2 = new PrinterThread(p, "Document2");
        PrinterThread p3 = new PrinterThread(p, "Document3");
        PrinterThread p4 = new PrinterThread(p, "Document4");

        p1.start();
        p2.start();
        p3.start();
        p4.start();
    }
}
```

```
    }  
}
```

Now, look at the test class. It first created the `Printer` resource object and then created four threads passing the *same* resource (`p`) with different data as shown below:

```
PrinterThread p1 = new PrinterThread(p, "Document1");  
PrinterThread p2 = new PrinterThread(p, "Document2");  
PrinterThread p3 = new PrinterThread(p, "Document3");  
PrinterThread p4 = new PrinterThread(p, "Document4");
```

Finally it started all the threads. If all the threads behaved well, then we expect the output to be in the pattern shown below:

Started, completed, started, completed, started, completed and so on....

Let's compile and execute the above test code with the following commands and cerify if the result indeed follows the above pattern or not.

```
C:/JavaTraining>javac threads/*.java  
C:/JavaTraining>java threads.PrinterTest
```

The result of the above program will be something like,

```
Printing Document1 started  
Printing Document4 started  
Printing Document2 started  
Printing Document3 started  
Printing Document2 completed  
Printing Document4 completed  
Printing Document1 completed  
Printing Document3 completed
```

As you can see from the above, printing the documents is messed up. It didn't follow our expected pattern. This is what happens when all the threads simultaneously act on a single resource. To prevent this from happening we need to *synchronize* the printer resource. Let's see how we do this.

Synchronization

Thread synchronization is the art of making the threads access the resource one after the other without messing up the resource. The question is, who does this synchronization? Will the resource does this or will the threads do it. You know what, both can do it. In Java, there is a keyword named **synchronized** that we can use to implement synchronization.

If the resource itself want to implement the synchronization, all it need to do is declare its shared methods (print method) using the **synchronized** keyword. In our example, the resource is the `print()` method in the `Printer` class, so we declare the method as,

```
public synchronized void print(String data){  
    -----  
    -----  
}
```

Modify our `print()` method adding the above keyword and run the same test program again. This time to your surprise, the output will be

```
Printing Document2 started  
Printing Document2 completed  
Printing Document1 started  
Printing Document1 completed  
Printing Document4 started  
Printing Document4 completed  
Printing Document3 started  
Printing Document3 completed
```

Now the output looks great, right? See how simple it is for the resource to get the threads in control. Let's say the `Printer` class is obtained from a third party company who didn't synchronize the method and at the same time didn't give us the permission to access to the source code. What would you do in such situation? You cannot add **synchronized** keyword to the `print()` method, right? Does this mean our poor printer would be messed up by the four threads? Not really if the threads behave properly like good threads. In such cases, the threads themselves need to take the responsibility to synchronize the `Printer` object in their own `run()` method. This is where **synchronized block** comes in handy. Let's see what this is.

Synchronized blocks

A synchronized block is normally used by a thread to lock the resource from other threads before using it. The syntax for this block is,

```
synchronized (object to lock){  
}
```

In our example, let's again remove the synchronized keyword from the `print()` method. Now update the `run()` method in the `PrintThread` class as shown below:

```
public void run() {  
    synchronized ( p ) {  
        p.print();  
    }  
}
```

As you can see from the above code, we first synchronized the printer object “p” and then invoked the method. This is like a thread locking the printer object and putting the key in its pocket until it is done with the object.

Run the same program again with the above updated code, and the result will still be the same good result. This is called as external synchronizing.

This completes all the important things you need to know to work with threads. There are however, some concepts like joining the threads etc., which we seldom use in real world applications. Read the interview questions in the later section, and you'll be good to go. For now, let's summarize what we learned this far in threads.

Summary

- ✓ Multi-threading is a technique in which multiple threads in a single process are executed simultaneously.
- ✓ Java supports multi-threading using the threads API.
- ✓ A thread goes through several states before it finally dies.
- ✓ A thread can voluntarily relinquish control of CPU by yielding. It can later come back to CPU and resume its execution cycle.
- ✓ Java uses Thread class and Runnable interface to create threads.
- ✓ All the statements that should be executed simultaneously must be placed inside the run() method of the thread class.
- ✓ Synchronization is the process by which multiple threads can access the resource one after the other without corrupting the resource.
- ✓ To synchronize the threads with the resource, we use **synchronized** keyword.

Time to play 50-50

1. Which of the following interface is used to create a thread?

- a) Thread
- b) Runnable

2. Which of the following class is used to create a thread?
 - a) Thread
 - b) Runnable
3. When a thread is started, which of the following state does it go immediately?
 - a) Running
 - b) Runnable
4. Which of the following method should be implemented to create a thread?
 - a) public void run()
 - b) void run();
5. Which of the following keyword is used to sync the threads with the resource?
 - a) synchronize
 - b) synchronized

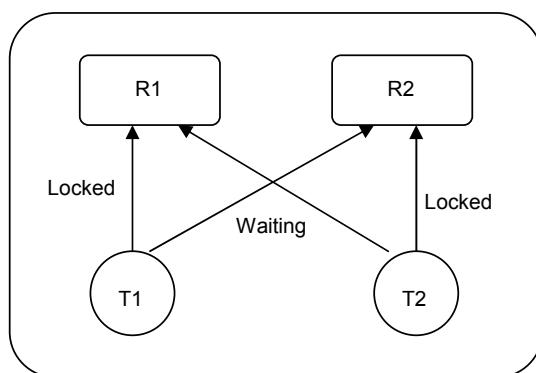
Interview Questions

Question: Explain the lifecycle of a thread.

Answer: Refer to page

Question: What is deadlocking?

Answer: When one thread T1 locked resource R1 and another thread T2 locked resource R2, and if T1 is waiting for R2 and T2 is waiting for R1, then the condition is a deadlock condition. See the following figure.



Chapter 9

JAR Files and Using API

By the end of this last chapter, you will become familiar about creating and using JAR files, understand how to use API's and how to work with CLASSPATH.

Chapter Goals

- ✓ Learn how to create JAR files
- ✓ Learn how to use JAR files
- ✓ Understand CLASSPATH variable
- ✓ Learn to use any Java API

Environment Setup

All the programs in this chapter should be stored in the following directory.

C:/JavaTraining/Chapter9/misc

misc is a package in which we will store the programs. To compile the programs, move to the following root directory and get ready.

C:/JavaTraining/Chapter9

Introduction

As I said at the very beginning of the chapter, a Java application is nothing but a collection of several classes. We also know that Java is platform independent because we can port the class files to any operating system. Though porting a Java application is simply copying the class files from one system to another, it's not as easy as it looks. We need to use some utility tools that aid in porting the code conveniently and this is what we will learn in this chapter.

Notion of JAR file

JAR stands for Java ARchive. This is a standard compression format invented by the creators of Java to archive Java related files. Therefore, a Jar file is nothing but a compressed file that compresses all the given Java related files into a single file. The JAR file uses "jar" extension. This is just like we compress the files using ZIP tool that has ".zip" extension.

Usually Java applications comprises several class files, and it's tedious to distribute each and every file to the clients. Instead, we pack all the files into one or more JAR files and then distribute the JAR files to who ever need it.

The primary reason for the success of Java is the **usability** factor. A single Java application can use any third party Java applications with ease. Let me clarify this. Say, you want to build chatting application in Java. To build this application, there are several components you need to build, like FTP application to transfer files during chatting, voice transfer application for voice chatting, image transfer application for video chatting and so forth. Developing all these components is time consuming task if we build all from scratch. What if there are some companies that distribute free and reliable Java code for the above components? This way we don't have to write all the complex code using complex algorithms. All we need to do is use the third-party code and integrate it into our application. Is'nt this cool? This is the best part of Java. These companies usually distribute the Java code in the form of JAR files, and once we get these JAR files, we can use the classes in the JAR files as if they are our own classes. Sweet!

There are two things we need to know here. As a sender, we need to know how to create JAR files, and as a receiver we need to know how to use the JAR files in our application. So, let's learn both.

Creating a JAR file

A Jar file is created using the **jar** tool that comes along with the Java software installation. Let's first write couple of classes and then create a JAR file with the two classes shown in listing 9.1a&b.

Listing 9.1a (VideoTransferTool) A simple Java class

```
package misc;  
  
public class VideoTransferTool {  
  
    public void startWebCam() {  
  
        System.out.println(" Streaming Images for Visual Chatting");  
    }  
  
    public void stopWebCam() {  
  
        System.out.println("Visual Streaming Stopped");  
    }  
}
```

Listing 9.1b (AudioTransferTool) A simple Java class

```
package misc;  
  
public class AudioTransferTool {  
  
    public void streamAudio() {  
  
        System.out.println(" Streaming Audio for Voice Chating");  
    }  
  
    public void stopAudio() {  
  
        System.out.println("Audio Streaming Stopped");  
    }  
}
```

The above two classes are pretty straightforward. Let's don the role of sender and create a Jar file with the above two classes in it. Following are the steps to create a Jar file.

1. Copy the `AudioTransferTool.java` and `VideoTransferTool.java` into the **misc** directory.
2. Compile the classes using the following command. All the “java” and “.class” will now be in this directory.

```
C:/>JavaTraining/Chapter9> javac misc\*.java
```

3. Finally, create the Jar file named `audiovideo.jar` to include the source files and class files that are in `misc` package using the following command:

```
C:\JavaTraining>jar cvf audiovideo.jar misc

added manifest
adding: misc/(in = 0) (out= 0)(stored 0%)
adding: misc/AudioTransferTool.class(in = 540) (out=
333) (deflated 38%)
adding: misc/AudioTransferTool.java(in = 263) (out=
144) (deflated 45%)
adding: misc/VideoTransferTool.class(in = 545) (out=
342) (deflated 37%)
adding: misc/VideoTransferTool.java(in = 269) (out=
153) (deflated 43%)
```

The Jar command displays all the verbose of what it did on the console. You'll now see `audiovideo.jar` created in the same directory. If you noticed the above verbosity, it added both ".java" files ".class" files. Most people would only archive the class files only. They will never distribute the source code. This is how they maintain the ownership of the code. If you only want to give the class files, then we tweak the above command as shown below:

```
C:\JavaTraining>jar cvf audiovideo.jar misc\*.class

added manifest
adding: misc/AudioTransferTool.class(in = 540) (out= 333) (deflated 38%)
adding: misc/VideoTransferTool.class(in = 545) (out= 342) (deflated 37%)
```

The above command just added the ".class" files. Many a times, classes will be distributed in more then one package. In such cases, to create a Jar file with classes in different packages, all we need to do is simply list all the directories we want to archive with a space delimiter as shown below:

```
C:\JavaTraining>jar cvf audiovideo.jar misc utilities com src
```

This is all you need to know about creating JAR files. Copy the above Jar file to the following directory:

```
C:\Jars
```

Using a JAR file

Using the Jar files is the best part of Java. Assuming that we received the above `audiovideo.jar` file via email or some other means, let's write a chatting program that uses the two classes in the jar file.

Look at the code in listing 9.2.

Listing 9.2a (`ChatApplication.java`) A simple chat application class

```
package chatapplication;  
  
import misc.*;  
  
public class ChatApplication {  
  
    public static void main(String args[]) {  
  
        System.out.println(" Initializing Chat Application ");  
        System.out.println(" Using the Audio Tool");  
  
        AudioTransferTool atool = new AudioTransferTool();  
        atool.streamAudio();  
  
        System.out.println("Using the Video Tool");  
  
        VideoTransferTool vtool = new VideoTransferTool();  
        vtool.startWebCam();  
  
        System.out.println("Application Ready for chatting");  
        System.out.println("Application finished chatting");  
  
        atool.stopAudio();  
        vtool.stopWebCam();  
  
        System.out.println(" Chat application terminated");  
    }  
}
```

For convenience purposes, we created the above class in the `chatapplication` package. Therefore, this program should be saved in the `chatapplication` folder as,

C:\JavaTraining\chapter9\chatapplication\ChatApplication.java

Look at the following import statement:

```
import misc.*;
```

Since the classes in the jar file are in the `misc` package, we need to import them using the `import` statement. The above statement imports all the class files in the `misc` package, which is what we want, right?

Once we finished importing, we are ready to use the classes in the Jar file like our own classes. Therefore, we created objects of the classes, and invoked the methods as shown below:

```
AudioTransferTool atool = new AudioTransferTool();  
atool.streamAudio();
```

Let's compile the above class with the following command and see what happens.

```
C:\JavaTraining>chapter9>javac chatapplication\*.java  
  
chatapplication\ChatApplication.java:3: package misc does not exist  
import misc.*;  
^  
chatapplication\ChatApplication.java:12: cannot resolve symbol  
symbol : class AudioTransferTool  
location: class chatapplication.ChatApplication  
    AudioTransferTool atool = new AudioTransferTool();  
    ^  
chatapplication\ChatApplication.java:12: cannot resolve symbol  
symbol : class AudioTransferTool  
location: class chatapplication.ChatApplication  
    AudioTransferTool atool = new AudioTransferTool();  
    ^  
chatapplication\ChatApplication.java:17: cannot resolve symbol  
symbol : class VideoTransferTool  
location: class chatapplication.ChatApplication  
    VideoTransferTool vtool = new VideoTransferTool();  
    ^  
chatapplication\ChatApplication.java:17: cannot resolve symbol  
symbol : class VideoTransferTool  
location: class chatapplication.ChatApplication  
    VideoTransferTool vtool = new VideoTransferTool();  
    ^  
5 errors
```

Shoot. The compiler is not happy. It says it cannot resolve the imports for Audio/Video classes. Did we miss anything here? Yes, we did. We wrote the program that uses the classes in the Jar file. From program point of view, we did everything right. This is just not sufficient. How does the Java compiler know where these classes are? This is like asking the compiler to compile a program that uses third party classes without telling it where the third party classes are. No wonder why the compiler bombed at us.

We know that the above classes are in the `audiovideo.jar` file, but how do we tell this to the compiler? Good question. This is where something called CLASSPATH variable comes into picture. Let's first understand what the CLASSPATH variable is before we fix the above errors.

CLASSPATH

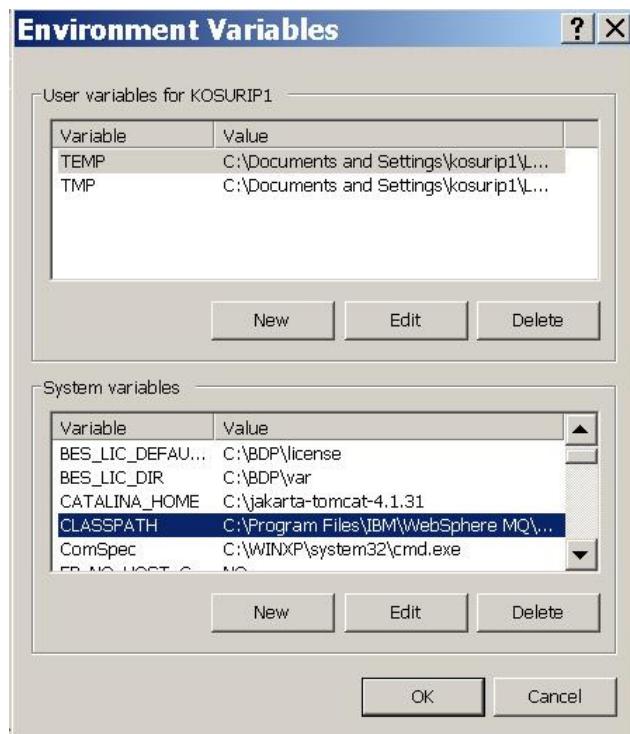
CLASSPATH is a standard variable that Java compiler uses to determine the location of classes that are to be loaded before compiling the program. During the installation of Java software, a default value for this CLASSPATH variable will be set. The default

value is nothing but the location of the built-in library classes. To specify the location of third-party Jar files, we need to append the location of the new Jar file to the existing classpath as shown below:

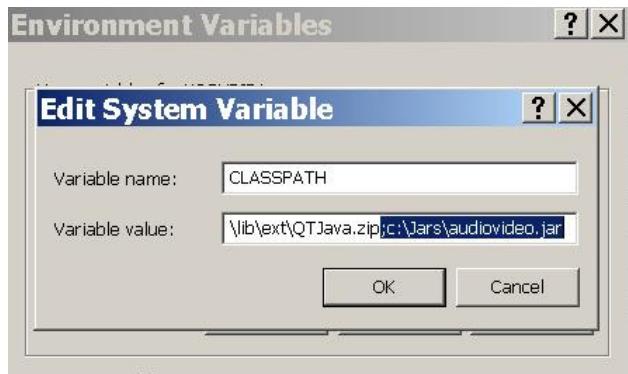
```
C:\JavaTraining>chaper9>set CLASSPATH = %CLASSPATH%;c:\jars\audiovideo.jar
```

The problem with the above command is that, if you close the command window, then you again have to type the above command before executing the program. There is a way to get rid of this problem using the following process:

1. Goto Start->Settings->Control Panel-> System->Advanced->Environment Variables. It will display the following window:



2. Select the CLASSPATH and click Edit button. This will display a small window as shown below:



3. Append the following at the end as shown and click OK. Keep Clicking OK until all the windows disappears.

```
;c:\Jars\audiovideo.jar
```

When the Java compiler compiles the program, it first checks the above CLASSPATH variable and then compiles the program. Since the location of the audiovideo.jar is also specified in the class path, it will then compile the program without any errors.

Now, compile the chat application program again, and all the errors will disappear. You can then execute the program like we did before to see the following output.

```
C:\JavaTraining>chapter9>java chatapplication.ChatApplication
```

```
Initializing Chat Application
Using the Audio Tool
Streaming Audio for Voice Chating
Using the Video Tool
Streaming Images for Visual Chatting
Application Ready for chatting
Application finished chatting
Audio Streaming Stopped
Visual Streaming Stopped
Chat application terminated
```

If you have more than one Jar file, you have to append all the JAR files to the CLASSPATH variable. This is all you need to know about creating Jars and using Jars. I am sure the above commands are simple and pretty straight forward. In real world applications, there will be fancy editors where you simply click the easy button to do all the above tasks. Cool.

You just have to remember one thing.

*******Put the Jar files in the CLASSPATH before using them.*******

Understanding API

This is the buzzword that you always hear with any Java application. API stands for “Application Programming Interface”. This is nothing but the Java documentation that we saw in the previous chapter. This is a funky name given to the Java documentation. As I said before, a Java application is a collection of several classes and interfaces. Since it’s virtually impossible to remember the names of each and every class, there must be documentation about all the classes that we can use to learn the functionality the classes offer.

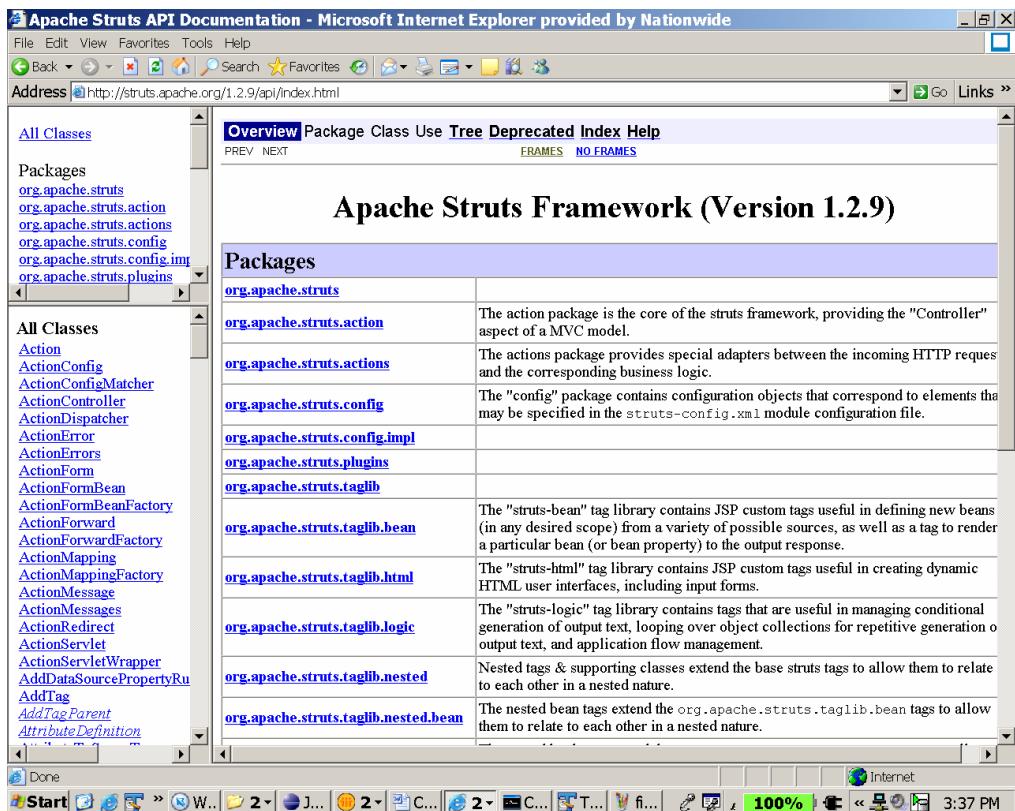
Usually any company that develops a technology using Java, provide us with the so called API which is nothing but Java documentation in HTML format that we saw in the previous chapter. We then use the API to understand the functionality of the classes, and then use them in our programs.

If someone like your manager comes to you and say, “Use Spring API to build database applications”, what he means is,

1. Use the Jar files that Spring has provided to build the application and
2. Use the API (HTML documentation) to learn about the classes in the Jar files.

Like wise, if I say, “Let’s learn Servlet API”, what I mean is, use the documentation and understand the details about the servlet classes. The good thing about Java is that, all this documentation is available online. You can also download the documentation if you want. When you download a Java technology from a company, they also give you the Java documentation for all the classes used by the technology. The reason is simple. If they don’t give you the documentation, then we cannot use their technology and they are the ones who lose their business. So, trust me, there will always be documentation for any Java technology. So, don’t panic to learn all the classes.

If you need the API for any Java technology, for instance Struts technology, just type “Struts API” it in Google, and it will pull it up for you. The API will look as shown in the following figure.



If you look at the above figure, it will display all the class names (left bottom), and all the packages it has (right side). You can click on the links to know about each class.

Finally, simple rules to explore any Java documentation

1. Open the Class by clicking on the hyperlink.
2. To use a particular method, check if the method has **static** keyword or not.
3. If it has **static**, use *ClassName.methodName* syntax to invoke the method. Pass whatever parameters it needs, and take whatever it returns.
4. If it is **non-static** (object method), then create the object using one its constructors listed on the same page, and then invoke the method using the object reference. Again, pass all the parameters and take whatever it returns.

Now is the moment to call yourself as an expert Java Programmer. So, hearty congratulations and a job well done. Take my high five.

Now that you have become a very good Java Programmer, take a small break and enjoy yourself. When you come back, I am going to teach you something called J2EE. Trust me, J2EE is lot easier than Java fundamentals.

Suggestion

Most people are scared of Java because of the number of technologies it has and remembering the class names in these technologies. But you know what, you don't have to memorize all the classes as long as you know what the program is doing. If some technology or even a program frustrates you, skip it temporarily and revisit it later. When you revisited and if it still frustrates you, then forget about the technology. You'll be fine. Take my word. Always make sure that you move forward and make sure you don't loose your heart. Very important.

Congratulations on successfully completing the Core Java fundamentals. Keep the ball rolling.

Part 2

J2EE

Chapter 10

Introduction to J2EE

This chapter introduces you to the world of J2EE by highlighting the important J2EE technologies and their application. This will also give an idea about J2EE application servers and their role in building enterprise applications.

Chapter Goals

- ✓ Understand the notion of J2EE
- ✓ Understand the key challenges during enterprise application development
- ✓ Understand the key challenges when an application is run on internet
- ✓ Understand what a J2EE application server is.
- ✓ Understand the core J2EE technologies supported by the application servers
- ✓ Know the list of free and commercial servers available in the market
- ✓ Understand the various J2EE architectures.

Environment Setup

Not Needed.

Introduction

Having learned all the good things in Java, we finally landed in the world of J2EE. Let me tell you one thing. J2EE is unbelievably easy and simple. You can take my word on this. As I told you at the beginning itself, Java is used to build two types of applications; standalone and internet based applications. All the Java programs we learned and developed until now are standalone applications. These applications are either console based applications that spit the output to the console, or GUI based applications like word processors, paint tools etc where funky windows show up to display information. These applications are like independent applications that are used by several users and every user will have a copy of the application. Such Java applications are no different from applications built using other languages like C, C++ etc, except being platform independent. This is one side of Java that we all know about.

The other side of Java is that it can be used build Internet based applications that can be used by millions of users at the same time. This is the true power of Java. Such internet applications are popularly known as *enterprise level applications*, as they form the faces of today's modern enterprises on the internet. So, from now onwards, we will use Java to build enterprise level applications that run on internet.

An enterprise or e-commerce application is nothing but an application that an enterprise or an organization uses to do business using Internet.

The advent of internet has completely changed the world of communication. With the widespread usage of internet, businesses quickly realized a whole new market had opened up to exploit and started taking advantage through e-commerce. E-commerce is nothing but doing business using internet and has become an instant success. E-commerce applications have now become the global faces of business companies and completely changed the dimensions of businesses. With almost all the businesses going online, the information assets of a company became more and more valuable. In order to sustain the competition, adoption of sophisticated technologies has become the key factor in exploiting the information assets of a business. J2EE is one such technology that helps businesses to build better and flexible e-commerce applications by securing the critical business data.

Now that we know what an enterprise application is, let's look at the key challenges that every enterprise application faces.

Performance: Since the application is now exposed to millions of users, it's very important that it responds faster enough to user requests. This is one of the most important aspects that play a key role in the success of a business.

Reliability: The application must be reliable in terms of processing the business transactions successfully and accurately.

Availability: It's very important that the application be up and running all the times with almost zero downtime. Few seconds of downtime can result in losing thousands of \$\$.

Security: The application must be able to provide a secure environment during the exchange of information between customers and businesses.

All the above are the challenges that an enterprise application faces after it is completely developed. However, there are few challenges while developing the application itself. Building enterprise application is not that simple as it involves interactions with several software systems like databases, mainframe systems, messaging systems and what not. We see two important challenges here.

Challenge 1: Making our Java program interact with such complex systems. It is very important for us to understand the details of the system that we want to interact with like say, what format the request should be sent, what protocol to use to send the request, what kind of response we get back and all that good stuff. Understanding such systems and writing code to interact with such systems is a time consuming task and delays building enterprise applications. The result is "*Your company is losing serious business*". You agree with me? Good.

If you own a business, you want to spend more time writing code that serves your customers rather than writing code to connect with databases, legacy systems etc, right? But again, you can't serve your customers without connecting to such systems. Therefore, we need some *ready made* specialized *helper* technologies that already know how to interact with such complicated systems, and all we need to do is *use* such technologies to talk with them.

Challenge 2: We are trying to build enterprise applications in Java. What if the above mentioned ready made helper technologies are written in some other language say C, C++ etc.? Again it gets complicated (but not impossible) for our Java program to use the technologies. So, to make our life easy, we need someone to develop these technologies in Java itself. Then a Java program using Java technologies will be pretty straight forward, right?

Challenge 3: Assuming that we have all the ready made helper technologies written in Java, managing such technologies is yet another challenge. For instance, configuring the technologies, managing the lifecycle of the technologies and above all, running our enterprise application that uses the technologies. So, we need some sort of sophisticated environment which we also call as *runtime* that helps in addressing this challenge.

We now came to a point, where we need a **sophisticated infrastructure** that

1. Addresses the challenges during application development through *ready made* technologies, thus promoting rapid application development
2. Provide a *runtime* to run or host the applications on the internet
3. Address the challenges (Performance, Availability etc) when the application runs on the internet

J2EE provides one such sophisticated environment which we call it as *J2EE Application Server*. Learning J2EE is nothing but,

1. Learning the helper technologies that it comes with and
2. Learn to build enterprise applications using the above technologies.

This is exactly what we will do from here onwards.

Core J2EE Technologies

As I said before, J2EE technologies are standard helper technologies that we use to build enterprise applications. Table 10.1 shows the list of important technologies that every J2EE application server supports:

Table 10.1 Core J2EE Technologies

Technology Name	Description
JDBC	Used for easy interaction with various Databases
Servlets/JSP	Used for building dynamic web applications in Java
JNDI	Used for interacting with Naming and Directory Services
JMS	Used for interacting with Messaging Systems
EJB	Component technology for building distributed components
Javamail	Used for sending Emails

The above listed technologies are standard J2EE technologies that are supported by every application server. However, there are several open-source technologies (free of cost) in the market though not adapted as standard technology in J2EE platform, are very popular and demanding in the world of J2EE. These technologies have become very popular and almost every real world J2EE application since the last couple of years is using them. Following is the list of popular open-source Java technologies.

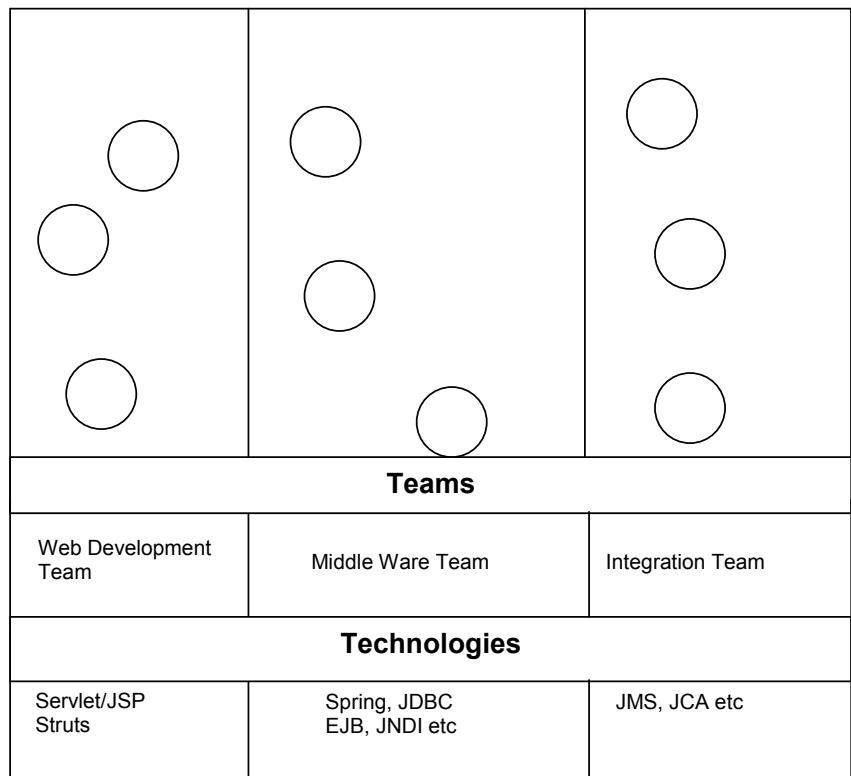
Table 10.2 Open Source J2EE Technologies

Technology Name	Description
Struts	Used for building dynamic web applications
Spring	Container framework for building J2EE applications
Hibernate	Framework for building Persistent Java applications

Question: How is an open-source Java technology different from standard Java technology? Good question.

In simple terms, Open Source technology is free to use and a standard J2EE technology is proprietary. With open-source technology, companies will also distribute the *source code* (Java source files) along with the class files, so that you can modify and use if needed. This is why we call them as *open source* technologies. But this is not the case with standard J2EE technologies listed in Table 10.1. Companies will never give the source code, ok.

I am sure you are disappointed to know that you have to learn all the above technologies in the two tables. You know what; you really don't have to know all the technologies. An enterprise level application is a very complex and huge application. It is impossible for a single person to build it from scratch. Instead, there will several teams to build the entire application, and every team is specialized in a particular area. To understand this, take a look at Fig 10.1.

J2EE Enterprise Application**Fig 10.1** Enterprise Application development Teams and Technologies

If you look at the above figure, we see like three different teams involved in the entire application development. The technologies used by the teams are listed under the team names. The web development team uses Servlet/JSP and Struts technologies. So, if you happen to get a Job as Java Web Developer, then you are good if you know just these two technologies. All the teams together are called as Server Side developers which means if you are get a job as server side developer, you might work in any of the above teams based on your specialization and expertise. However you can also work in all the teams at the same time.

All this is to say that you don't have to worry about the number of technologies you have to learn. But it's always good to know the purpose of each and every technology so that you are not lost during discussions with other teams.

This book covers in detail the following latest and greatest technologies that are used in today's real world J2EE applications. Trust me; knowing these is more than enough to land in a good job in J2EE world. Fair enough, right. The whole purpose of this book is teaching you Java and J2EE that is needed to land in a good job.

Table 10.2 Important J2EE Technologies

Technology Name	Description
JDBC	Used for easy interaction with various Databases
JMS	Used to talk with messaging systems.
Servlets/JSP	Used for building dynamic web applications in Java
Struts	
Spring	Container framework for building J2EE applications
Hibernate	Framework for building Persistent Java applications

EJB technology is an ocean and the complete details of it are beyond the scope of the book. Most of the real world applications are slowly trying to get away from EJB due its complexity and moving to Spring/Hibernate which will be covered in detail. We will only learn those technologies that are widely used in real world applications.

At this point let me tell you one thing. Writing J2EE programs is very simple. It hardly takes few minutes to write the program, but the real challenge is getting it running. This is because J2EE is process oriented. You need to strictly follow a systematic process to make the application running and working.

This challenge can be addressed by keeping things simple and following a systematic approach which is exactly what we'll do in this book. I will outline all the steps that you need to follow to run the applications. If you still don't get something to work, don't loose your heart. Learn J2EE with **passion** and more than that with **patience**. I am sure you'll enjoy it. The way to learn J2EE is, learn to build applications using technologies. As long as you know what you are trying to accomplish, you are good. If you can get the application working, it's a BONUS. That's how I learned J2EE. I never worried even if the program failed to work as long as I know that I followed a systematic process and what I did makes sense.

At the bare minimum, you need to know the following:

1. What is the name of the technology
2. What is it used for.

Let me ask you a question here. Do all the technologies require the use of an application server? Good question. The answer is No. You only need a server primarily when you build Web applications. The only technologies used to build web applications are:

- ✓ Servlets/JSP and
- ✓ Struts

We can call the above two technologies as **mainstream** technologies. All others are helper technologies that the mainstream technologies use to build the applications. Therefore, when we learn helper technologies like JDBC, Spring, Hibernate we will run them as standalone applications using *javac* and *java* commands. We will only use the server when we write applications using Servlet/JSP and Struts technologies. Keep this in mind.

Enterprise Application Architectures

Since we are almost ready to start building enterprise applications in Java, we need to know the architecture of a typical enterprise application. Before we look into it, let's learn something about the various enterprise architectures that we currently have.

2-Tier Architecture

Look at Fig 10.2 that shows typical 2-tier architecture. In this architecture, all the *business logic* and *presentation logic* of the application is embedded in the clients computer itself. Therefore if my application has 100 clients, I need to install the application on all the 100 client computers. However the database will still reside on a separate computer that will be shared by all the clients.

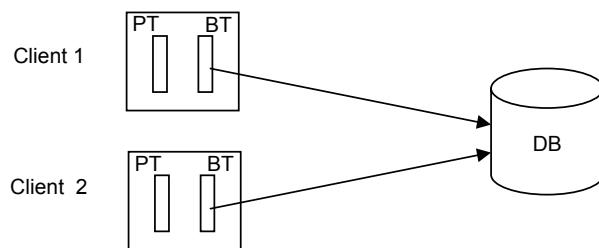


Fig 10.2 Typical 2-tier architecture

Advantages

- ✓ Simple and easy to build
- ✓ Low cost

Disadvantages

- ✓ Even a small change to the application requires a reinstall of the application on all the client computers.
- ✓ If some clients are not ready to take the changes, then multiple versions of the application will prevail, thereby causing maintenance nightmares.
- ✓ The performance of the application is dependent on the performance of client computer.
- ✓ Heavy network traffic due to multiple requests to the database from all the client computers.

The disadvantages of this architecture outweigh the few advantages making it less useful.

3-Tier Architecture

Fig 10.3 shows typical 3-tier architecture. In this architecture, the presentation logic, business logic and data are logically distributed in three tiers. The main difference is that the business logic is permanently isolated from all the client systems and moved to centralized middle tier. The presentation logic on the client systems will query the business logic in the middle tier which in turn accesses the database. This architecture overcomes all the cons of 2-tier system.

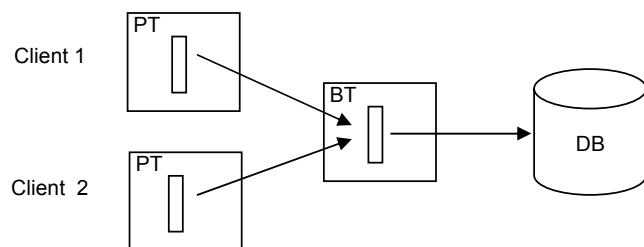


Fig 10.3 Typical 3-tier architecture

Advantages

- ✓ Centralized business logic will offer more flexibility. Business logic is only required to be changed at one place thereby eliminating the installation process of the application on client systems.
- ✓ Less network traffic, thereby improving the performance of the application.
- ✓ Application performance is no longer dependent on client computer due to the business logic isolation.
- ✓ No more maintenance nightmares.

Disadvantages

- ✓ Any update to the business logic must be accepted by all the clients even if they are not ready for updates.

N-Tier Architecture

In this type of architecture, the application logic is divided based on the functionality rather than physically like in 2-tier and 3-tier architectures. A typical n-tier architecture contains the following elements:

User Interface: This is something like a web browser that handles the client interactions.

Presentation Logic: This defines format using which the information will be displayed.

Business Logic: Encapsulates all the business rules by interacting with data sources.

Infrastructure Services: These are utility services that the presentation and business logic makes use of.

Data tier: This contains all the enterprise data.

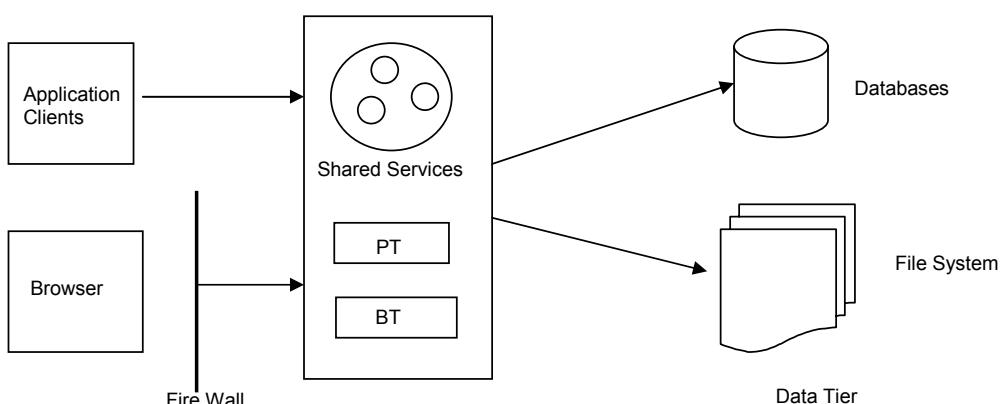


Fig 10.4 Typical n-tier architecture

Breaking the application logic based on functionality offers several benefits like flexibility, better maintenance, improved performance, reusability of code and may more. This architecture is also referred to as *Model-View-Controller* (MVC) architecture.

Enterprise Architecture

An N-tier architecture is applicable to a *single application* in an enterprise. However, an enterprise application is a collection of several applications within the enterprise with all the applications working in tandem and interacting with each other through well defined interfaces as shown in the following figure.

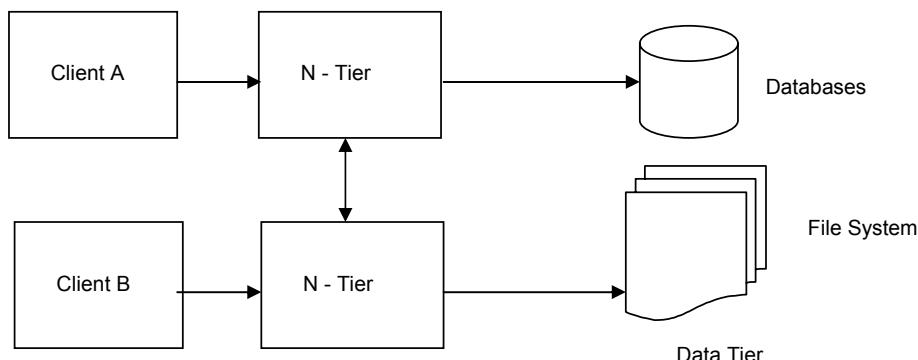


Fig 10.5 Enterprise Architecture

Simply stating, enterprise architecture is a combination of several n-tier architectures. With this basic knowledge of various architectures, let's now try to understand somethings about application servers.

J2EE Application Servers

There is some terminology you need to understand when it comes to J2EE. Unlike simple Java programs, we don't use *java* command to execute the J2EE applications. We use a special program called *application server* to run the applications. However, we still use the same old *javac* command to compile the applications.

An application server is the heart of any J2EE application. This is the one that runs the enterprise application on the internet. So, we can't even move a step forward without knowing what this is.

A J2EE application server is a ready made sophisticated application that will host and run all the J2EE applications. There are several free and commercial application servers available in the market today that are developed by noted companies like Sun Microsystems Inc, IBM, BEA and many more. What is the purpose of all these companies developing the same application server? Good question. Here is the deal. Sun Microsystems first came up with the notion of J2EE platform that can be used to build large scale enterprise applications in Java. To simplify the development of such applications, they felt the need for the following two things:

1. A *Server* that runs the applications
2. Helper *technologies* to build the applications.

They did the hard work and came up with the specifications for the above. These are popularly known as *J2EE specifications*. Having just the specifications is not good enough. We need someone to convert the specifications into workable realities. Therefore, Sun Microsystems decided to distribute the specifications to companies who can implement them. This is how IBM, BEA, Apache and several other companies came into picture. These companies used the J2EE specifications from Sun Microsystems, and built the two workable components listed above using their expertise. Once they built the application server along with the helper technologies, they started selling it.

A company to build and host enterprise level applications in Java needs a J2EE application server. It will therefore hunt for those who sell the J2EE servers and come across several companies like SUN, IBM, BEA, JBoss and many more. After serious negotiations with the companies, it finally chooses the company who sells it for less \$\$ and more value.

Once you have a J2EE application server, you can start building and running enterprise applications. All the companies that built the J2EE Application Server gave funky names to their product shown in the following table:

Table 10.3 J2EE Application Servers

Company	Server Name
SUN	SunONE Studio Application Server
IBM	IBM WebSphere Application Server
BEA	BEA WebLogic Application Server
JBoss	JBoss Application Server

To summarize, “A J2EE application server runs enterprise internet applications that are built using standard J2EE technologies”. Remember this one point, and you are good.

Look at the following figure that shows the architecture of a J2EE application server.

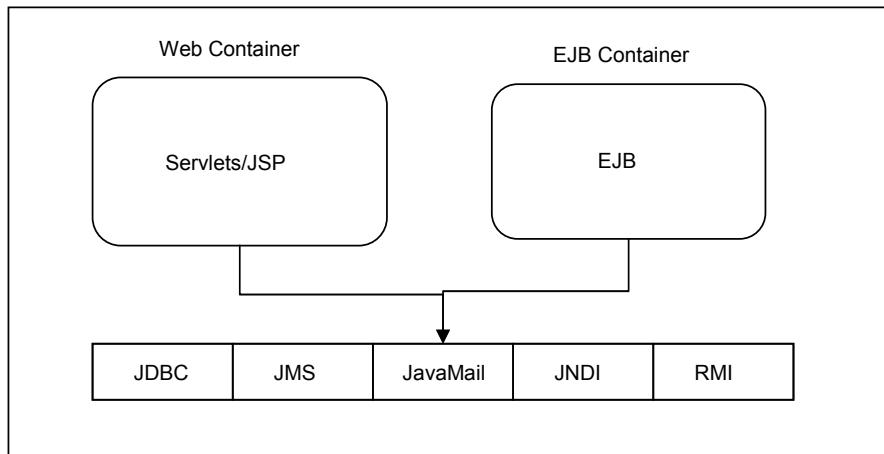


Fig 10.6 J2ee Application Server Components

A J2EE application server comprises of a *Web Container* and an *EJB Container*. A container is nothing but a runtime environment to manage the application components. In J2EE, there are two mainstream technologies like Servlets/JSP/Struts and EJB. Application components built using Servlet/JSP/Struts technologies run within Web container, and application components built using EJB technology are run within EJB container. You cannot run EJB components in Web container and Servlets/JSP components in an EJB container, ok. All the remaining technologies (JDBC, JMS, JNDI, Spring, Hibernate etc) are like helper technologies that are used by the above two mainstream technologies in both the containers.

Let me ask you one question here. An application server has both Web container and EJB container. If our application doesn't use EJB technology, do we need an EJB container? You are right. We don't need an EJB container. In such cases, having just a web container is good enough. Therefore, purchasing an application server is not a good idea. We therefore need a vendor who just gives us a Web container. This is where a company named Apache Software Inc, comes into picture who developed a Web container named *Tomcat*. This web container is FREE to download. Wow. This is really cool, isn't it? Not only that we got a web container, but also for free of cost. Cool deal.

In this book, we will use this FREE Tomcat web container to run the applications. When we get to a point where we need to use the Tomcat container, we'll see how to install it. Until then don't worry about it.

With all this basic understanding of what J2EE is all about and J2EE application servers, let's begin the journey into J2EE with passion. Trust me, developing J2EE application is fun and I am sure you'll enjoy it. So, without wasting any further time, let's start learning the J2EE technologies. But for now, lets summarize this chapter.

Summary

- ✓ J2EE stands for Java 2 Enterprise Edition.
- ✓ J2EE is a platform and not a language.
- ✓ We use J2EE platform to build large scale enterprise level applications in Java that run on Internet.
- ✓ A J2EE platform is also known as J2EE Application Server.
- ✓ A J2EE application server supports all the standard J2EE specifications like JDBC, Servlet, JSP, EJB, JMS etc.
- ✓ Application server also comes with runtime to run the applications that use standard J2EE technologies.
- ✓ An application server is bundled with both Web Container and EJB Container.
- ✓ There are several free and commercial J2EE application servers available in the market.
- ✓ The most popular application servers are IBM WebSphere and BEA Weblogic servers.
- ✓ We don't need an application server if we don't use EJB's for application development. In such cases, having just the web container will solve the purpose.
- ✓ Tomcat is a FREE web container available in the market today. This is developed by Apache Software Foundation.
- ✓ Besides the standard J2EE technologies, there are also several popular open-source technologies. These include Struts, Spring and Hibernate.
- ✓ Most of the J2EE applications using n-tier architecture in which the application logic is logically separated based in the functionality. One example of n-tier architecture is a Service Oriented Architecture (SOA).

Time to play 50-50

1. Which of the following statement is true?

- a) J2EE is a language like Java

- b) J2EE is a platform used to build enterprise applications using Java
2. Which of the following is a standard J2EE technology?
- a) Servlet/JSP
 - b) Struts
3. Which of the following is an open-source technology to build Web applications?
- a) Struts
 - b) Hibernate
4. Which of the following is the name of the application server from IBM?
- a) Websphere
 - b) Weblogic
5. Which of the following statements is true?
- a) An application server comes only with Web Container
 - b) An application server includes both Web container and EJB container
6. Which of the following is a Web Container?
- a) WebLogic
 - b) Tomcat
7. Which of the following architecture is widely used for building J2EE application?
- a) 2-tier
 - b) n-tier
8. Which of the following are the main challenges with a J2EE application?
- a) Performance, Scalability, Availability, Security and Maintenance
 - b) Usability and Presentation

There will not be any interview questions in this chapter, as it just an introductory chapter. Let's move on to the next chapters to begin our journey into the world called J2EE.

Chapter 11

JDBC

By the end of this chapter, you will be an expert in writing Java applications that talk with databases. This chapter will explain the details of what a database is, how to use JDBC technology to connect with databases, and how to execute the SQL queries against the database. This chapter also gives you a brief over of SQL and it syntax.

Chapter Goals

- ✓ Understand the basics of Databases and Drivers
- ✓ Understand the basics of SQL language
- ✓ Understand the important components in JDBC API
- ✓ Learn to write Java programs to connect with databases
- ✓ Learn to write programs to store data in the database tables
- ✓ Understand various database operations.
- ✓ Understand the notion of Connection Pooling
- ✓ Understand the common problems in typical databases and solutions to address them.

Environment Setup

Compiling and executing JDBC program requires,

1. Placing the required Jar files in the CLASSPATH
2. Compiling and executing the program

Since it's always tedious to set the classpath, we will define the classpath setting in batch file named 'env.bat' and execute the batch file before we start executing the programs. This makes life easy. Follow the instructions below to complete the JDBC setup process.

1. Install J2EE SDK, MySql and DbVisualizer software following the instructions in "Software Installation" chapter.
2. Create a directory named **j2eelib** as shown below:

C:/JavaTraining/j2eelib

3. Copy **j2ee.jar** from C:\sun\AppServer\lib and **mysql-connector-java-5.0.3-bin.jar** from C:\MYSQL\drivers\mysql-connector-java-5.0.3 into the C:/JavaTraining/j2eelib directory.
4. Create a batch file named "env.bat" as shown below

C:/JavaTraining/chapter11/env.bat

and copy the following contents into it

```
set CLASSPATH=.;  
C:\JavaTraining\j2eelib\j2ee.jar;C:\JavaTraining\  
j2eelib\mysql-connector-java-5.0.3-bin.jar
```

5. Create a directory named **jdbc** as shown below to store the JDBC examples.

C:/JavaTraining/chapter11/jdbc

Note: Whenever you start working with JDBC, the first thing you need to do is execute the batch file to setup the classpath as shown below:

C:/JavaTraining>chapter11>env.bat

Once you execute the above command you are ready to compile and execute any JDBC program in this chapter.

Introduction

The success of any business depends on the information and data assets it possesses. Therefore it is very important for a business to have a system for storing and retrieving data. Though files are used for storing data, they are highly inefficient in many respects like the amount of data that can be stored, retrieving the data and many more. In order for the businesses to be competent, they need sophisticated systems for storing complex and heavy volume of data and at the same time retrieve the data faster enough to process the data. One such systems are the so called *Databases*.

Databases typically form the back bone for any e-business company. Any enterprise application usually queries the databases to store, manipulate and retrieve the data to fulfill user's requests. Therefore understanding the basic details of databases is very important for building enterprise applications.

In this chapter, we will first learn the basics of databases and then learn how to use JDBC helper technology to access the databases from Java programs. So, let's rock.

Database Basics

A database as the name suggests is a sophisticated system for storing the data. Databases store the data in the form of *tables* with each table containing several records of data. Databases are broadly categorized into two types as listed below:

1. Single table databases and
2. Multi table or relational databases.

A single table databases doesn't mean that they have just one table. It will have several tables of data and all the tables are *independent* of each other. The main drawback with these databases is the lack of flexibility while retrieving the data. On the other hand in multi table databases, two or more tables are related with other. This feature offers more flexibility and power to retrieve the data. These databases are therefore called as *relational* databases. Most of the modern enterprises use relational databases.

Following figure shows how the data is stored in databases:

Table: **Customers**

Name	SSN	Age	Country
John	1234567	47	USA
Smith	3435345	21	Canada

The above table shows customers table that has two records of data. A typical database will have 'n' number tables with each having 'n' number of records.

A relational database can store the data in several tables that are related with each other as shown below:

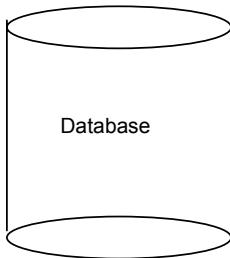
Table: Customers

Name	SSN	Age	Country
John	1234567	47	USA
Smith	3435345	21	Canada

Table: Addresses

SSN	AddressLine 1	AddressLine 2	City	State
1234567	Apt # 20	1111 S St	LA	CA
3435345	Apt #2345	2222 N St	Detroit	MI

If you notice the above tables, they are related with each other by the common column SSN. The two tables together represent the customer information. To pull the customer information, we need to pull the data from both the tables based on the matching SSN. For here onwards, let's use the following figure to represent a database.



Now that we know what relational databases are capable of, we need to know something about how to store the data and retrieve the data to and from databases. This is where SQL comes into picture. So, let's see what this is all about.

Structured Query Language (SQL)

Structured Query Language which is abbreviated as SQL is a standard query language for working with relational databases. SQL is used for creating the tables, populating the tables with data, manipulating the data in the tables, deleting the data in the tables and the tables itself. Let's quickly learn the usage of SQL to do various database operations.

Creating a table

Let's create a table named **Customers** to store customer data like firstname, lastname, ssn, age, city, state, and country as shown below:

Table : Customers

FirstName	LastName	SSN	Age	City	State	Country

The SQL syntax for creating a table is shown below:

```
CREATE TABLE <TABLE NAME>
(
    COLUMN 1 NAME      COLUMN 1 TYPE,
    COLUMN 2 NAME      COLUMN 2 TYPE,
    .
    .
    .
    COLUMN N NAME      COLUMN N TYPE
);
```

Using the above syntax, to create the customers table, the SQL query will be as shown bellow:

```
CREATE TABLE Customers
(
    FirstName      VARCHAR(50),
    LastName       VARCHAR(50),
    SSN            NUMERIC(10),
    Age            NUMERIC(10),
    City           VARCHAR(32),
    State          VARCHAR(2),
    Country        VARCHAR(32)
);
```

While creating table, it's very important to mention the type of data that will be stored in various columns. VARCHAR is a standard datatype for storing text data. NUMERIC is used for storing numbers. The numbers in parenthesis represent the maximum length of data that can be stored. Likewise, there are several other data types. Table 11.1 lists some of the most important SQL data types:

Table 11.1 Important SQL data types

SQL Data Type	Description
VARCHAR	Used for storing text data
NUMERIC	Used for storing numbers
DATE	Used for storing dates

Inserting the data into the table

To insert the data into the tables, we use the INSERT query. A single insert query inserts one record or row of data. Following is the sql syntax.

```
INSERT INTO <TABLE NAME> VALUES (COL1 DATA, COL2 DATA ... COL (N) DATA);
```

Using the above syntax, we insert the record into customers table as

```
INSERT INTO Customers VALUES ('Smith', 'John', 123456, 20, 'Vegas', 'CA', 'USA');
```

The above SQL inserts the data in the **Customers** table as shown below:

Table : **Customers**

FirstName	LastName	SSN	Age	City	State	Country
Smith	John	123456	20	Vegas	CA	USA

While using the INSERT sql,

1. Every data must be separated by a comma,
2. All the text data must be enclosed in single quotes,
3. Numeric data must not be enclosed in single quotes.

If you don't know what value to insert in a particular column, you must specify a NULL as shown below:

```
INSERT INTO Customers VALUES ('Smith', null, 123456, 20, null, 'CA', 'USA');
```

Updating records in table

To update the data in an existing record, we use the UPDATE query statement. The syntax for this query is shown below:

```
UPDATE <TABLE NAME> SET
COLUMN 1 NAME = NEW COLUMN 1 VALUE,
COLUMN 2 NAME = NEW COLUMN 2 VALUE
...
WHERE
COLUMN 1 NAME = VALUE 1 AND
COLUMN 2 NAME = VALUE 2 .....
```

Let's say we want to change the last name to 'Roberts' and first name to 'Joe' whose SSN is 123456. The update query will be,

```
UPDATE CUSTOMERS SET
FirstName = 'Joe',
LastName = 'Roberts'
WHERE
SSN = 123456
```

The above query will update all the customers firstname and lastname whose SSN equals to 123456. The contents of the table would be as shown below:

Table : **Customers**

FirstName	LastName	SSN	Age	City	State	Country
Joe	Roberts	123456	20	Vegas	CA	USA

Retrieving records from table

Retrieving the records is the most important database operation. We use the SELECT query to retrieve the records. The syntax for this query is as shown below:

```
SELECT COL 1 NAME, COL2 NAME ..... FROM <TABLE NAME> WHERE <CONDITION>
```

There are several ways you can use the SELECT query. To better understand, consider the following table:

Table : **Customers**

FirstName	LastName	SSN	Age	City	State	Country
Joe	Roberts	123456	20	Vegas	CA	USA
Sam	Perry	34534543	22	LA	CA	USA
Serra	Bates	567657	67	SFO	CA	USA
Serry	Sanders	123213	33	DEN	CO	USA
Smith	Murray	89789	45	NEV	NV	USA
Slater	Mike	676576	32	COL	OH	USA

If the query has no WHERE clause, then it retrieves the selected columns from all the records in the table. Based on this, let's see few cases here:

Case 1: Retrieve all the customers data

```
SELECT * from Customers;
```

In the above sql asterisk(*) means all columns of data. Since there is no WHERE clause in the query, it will pull up all the records. The result of the above query is the entire table.

Case 2: Select all the customers who belong to the state of CA.

```
SELECT * FROM Customers WHERE State = 'CA';
```

The above query will retrieve those records whose state is CA as shown below

Table: **Customers**

FirstName	LastName	SSN	Age	City	State	Country
Joe	Roberts	123456	20	Vegas	CA	USA
Sam	Perry	34534543	22	LA	CA	USA
Serra	Bates	567657	67	SFO	CA	USA

Case 3: Retrieve the first names and last names whose country is USA and state is CA (multiple conditions)

```
SELECT firstName, lastName from Customers WHERE State='CA' AND Country='USA';
```

The above sql displays the following results:

Table: **Customers**

FirstName	LastName
Joe	Roberts
Sam	Perry
Serra	Bates

This is how we need to use select query. The SQL language uses very general terminology. Reading the SQL itself will tell us what it is doing.

All the above select queries retrieve the data from just ONE table only (Customers). However, in real world applications data is required to be pulled from multiple tables. Let's take a look at a simple example.

Case 4: Let's say we have the following two tables.

Table 1: **Books**

Publisher	ISBN	Title
Wrox	AB3456	Perl Scripting
McGraw	YZ6789	Core Java

Table 2: **Authors**

Author	ISBN	Country
Smith	AB3456	Germany
John	YZ6789	Australia

If you noticed the above two (Books and Authors) tables, they are joined together using the ISBN column. If we need to pull the publisher, author, title and country (data in both the tables) for a particular ISBN, we do so as shown below:

```
SELECT t1.Publisher, t1.Title, t2.Author, t2.Country from Books t1, Authors t2 WHERE t1.ISBN = t2.ISBN;
```

Whenever we need to pull data from multiple tables, we need to use *alias* names for tables. In the above query, t1 is the alias name for **Books** table, and t2 is the alias name for **Authors** table. Therefore, all the columns in books table must be prefixed with "t1." and columns in authors table must be prefixed with "t2." as shown below:

```
SELECT t1.Publisher, t1.Title, t2.Author, t2.Country
```

Now, look at the *where* clause. Since we need to pull the data if the ISBN in both the tables match, the where clause will be as shown below:

```
WHERE t1.ISBN = t2.ISBN;
```

Deleting the records

To delete the records in the table we use DELETE query. The syntax for this query is shown below:

```
DELETE FROM <TABLE NAME> WHERE <CONDITION>
```

For instance, to delete all the customers whose state is CA, the query will be,

```
DELETE FROM Customers where State='CA';
```

The above SQL deletes 3 records in the customers table. These are all the basic SQL operations you need to know to work with any database. I am sure things are pretty simple. With this basic knowledge of databases and SQL, we are now ready to learn JDBC technology to access databases. Let's rock and roll.

JDBC

JDBC stands for Java Database Connectivity which is a technology that allows Java applications to,

1. Interact with relational databases
2. Execute SQL queries against the databases

To better understand JDBC technology, look at the following figure:

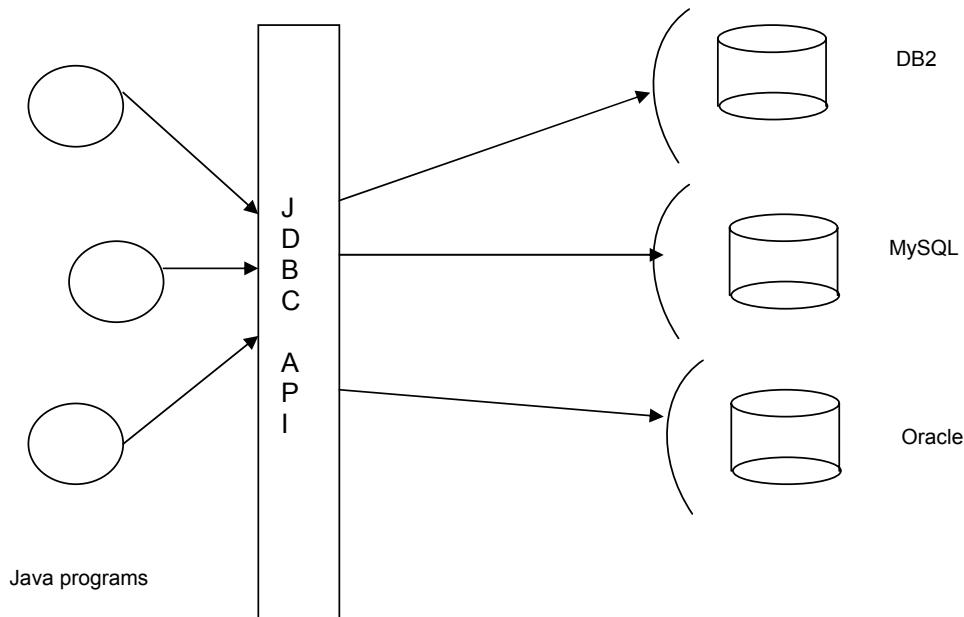
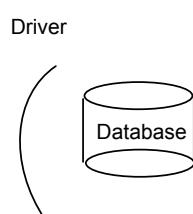


Fig 11.1 JDBC and Databases

The above figure should tell you the complete story. Java applications simply use JDBC API to interact with any database.

How JDBC talks with Databases

Databases are proprietary applications that are created by noted companies like Oracle, Microsoft and many more. These could be developed in any language like C, C++ etc by these vendors. Every database will expose something called *Driver* through which one can interact with database for SQL operations. A driver is like a gateway to the database. Therefore, for any Java application to work with databases, it must use the driver of that database. Note this point. Following figure shows the relationship between driver and database.



Database Drivers

A driver is not a hardware device. It's a software program that could be written in any language. Every database will have its own driver program. Given a driver program of a particular database, the challenge is how to use it to talk with database. To understand this, we need to know the different types of drivers.

There are basically 4 different types of database drivers as described below:

JDBC-ODBC Bridge Driver

This is also called as *Type-1* driver. Fig 11.2 shows the configuration of this driver.

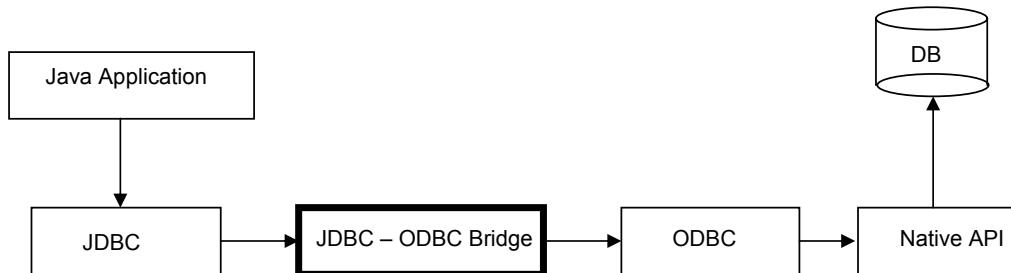


Fig 11.2 JDBC-ODBC Bridge driver

As you can notice from the above figure, this driver translates the JDBC calls to ODBC calls in the ODBC layer. The ODBC calls are then translated to the native database API calls. Because of the multiple layers of indirection, the performance of the application using this driver suffers seriously. This type of driver is normally used for training purposes and never used in commercial applications. The only good thing with this driver is that you can access any database with it.

In simple words, with this driver for JDBC to talk with vendor specific native API, following translations will be made.

JDBC -> ODBC -> Native

It's a 2 step process.

Partly Java and Partly Native Driver

This is also called as *Type-2* driver. As the name suggests, this driver is partly built using Java and partly with vendor specific API. Fig 11.3 shows the configuration of this driver.

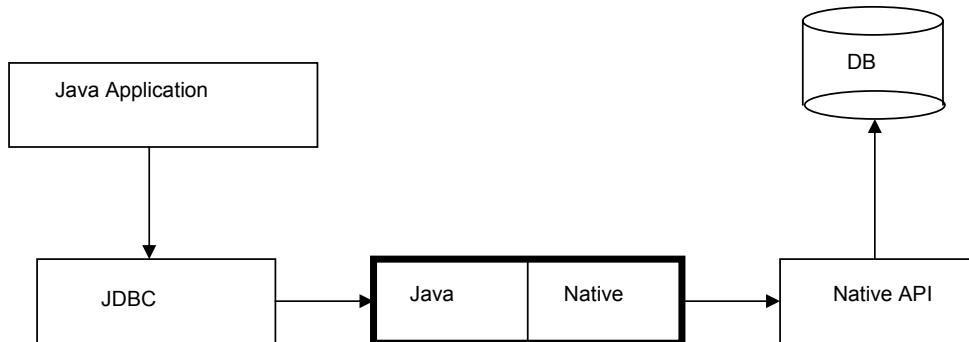


Fig 11.3 Partly Java - Partly Native driver

With this driver, the JDBC calls are translated to vendor specific native calls in just one step. It completely eliminates the ODBC layer. Because of this ODBC layer elimination, the applications using this driver perform better.

Intermediate Database access Driver Server

This is also known as *Type-3* driver. If you look at the Type-2 driver configuration, we only access one database at a time. However there will be situations where multiple Java programs need to access to multiple databases. This is where Type-3 driver is used. It acts as a middleware for the applications to access several databases. The Type-3 configuration internally may use Type-2 configuration to connect to database. See Fig 11.4 that shows the configuration of this driver.

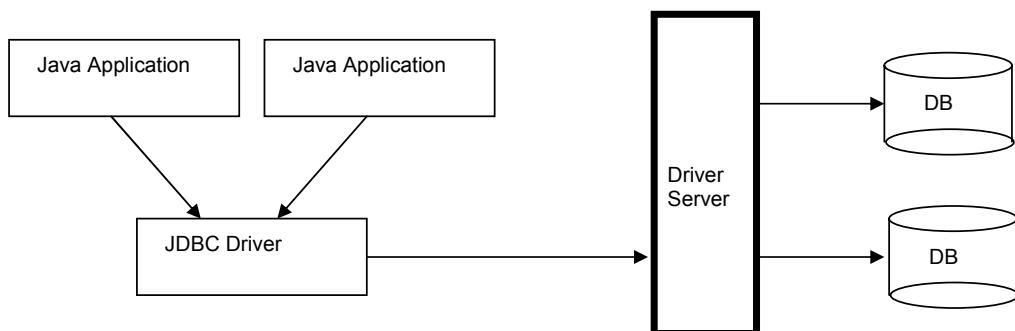


Fig 11.4 Driver Server

Pure Java Drivers

These are called as *Type-4* drivers. Because of the widespread usage of Java, to facilitate easy and faster access to databases from Java applications, database vendors built the driver itself using Java like good friends helping each other. This is really cool as it completely eliminates the translation process. JDBC is Java based technology, and with the driver also written in Java, it can directly invoke the driver program as if it were any other Java program. This is the driver that all the J2EE applications use to interact with databases. Because this is a Java to Java interaction, this driver offers the best performance. Fig 11.5 shows a Type-4 driver.

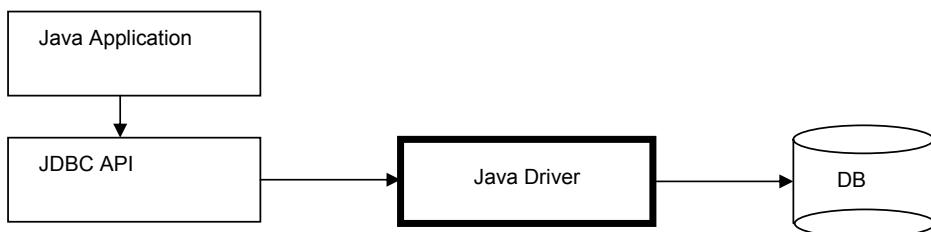


Fig 11.5 Pure Java Driver

Interview Question: What type of JDBC driver is normally used to access databases?

Answer: Type-4 driver which is also called as Pure Java driver.

To summarize, besides having a database, we need a Type-4 Java driver to work with databases. Very important point to remember.

Now the question is, where do we get this Type-4 driver? For most the databases, we get this driver program bundled in a JAR file when we download the database. Cool deal.

In this chapter, we will use MySQL database for all our applications. Assuming you have downloaded both MySQL database and MySQL driver with the instructions in the last chapter, we are now completely ready to write Java programs using JDBC API.

JDBC API

JDBC technology is nothing but an API. It is a set of classes and interfaces that our Java programs use to execute the SQL queries against the database. The fundamental idea behind JDBC is that, Java programs use JDBC API, and JDBC API will in turn use the

driver to work with the database. Therefore, to work with MySQL database, we need to configure the JDBC with all the MySQL information. This information is nothing but:

1. Name of the MySQL driver class
2. URL of the database schema.

In simple words, for a Java program to work with the database, we first need to configure JDBC with the above database info and then make the Java program use the JDBC.

We'll see how to configure JDBC later, but once configured, Table 11.2 lists the important JDBC classes and interfaces that we can use to build applications.

Table 11.2 JDBC API

Class/Interface	Description
DriverManager	This classes is used for managing the Drivers
Connection	It's an interface that represents the connection to the database
Statement	Used to execute static SQL statements
PreparedStatement	Used to execute dynamic SQL statements
CallableStatement	Used to execute the database stored procedures
ResultSet	Interface that represents the database results
ResultSetMetaData	Used to know the information about a table
DatabaseMetadata	Used to know the information about the database
SQLException	The checked exception that all the database classes will throw.

Let me tell you one thing here. JDBC programming is the simplest of all. There is a standard process we follow to execute the SQL queries. Following lists the basic steps involved in any JDBC program.

1. Load the Driver
2. Get the connection to the database using the URL
3. Create the statements
4. Execute the statements
5. Process the results
6. Close the statements
7. Close the connection.

By following the above process, we can have our Java programs successfully interact with databases. Please remember the above process. It's an interview question.

Starting the MySQL Database

Before we run any of the programs, we need to start the MySQL database. To do so, run the following command:

```
C:/>MySQL>bin>mysqld
```

The above command will start the MySQL database and listens at port 3306. You can close the command window if you want.

With the database up and running, using the above process and the JDBC API, let's write our first JDBC program. Our program simply establishes a connection to the MySQL database. See code in listing 11.1.

Listing 11.1 (`ConnectionTest.java`) Class establishing a connection with database.

```
package jdbc;  
import java.sql.*;  
public class ConnectionTest {  
    public static void main(String args[]) {  
        try {  
            // Step 1: Load the Driver  
            Class.forName("com.mysql.jdbc.Driver");  
  
            // Step 2: Get the connection by passing the URL  
            String url = "jdbc:mysql://localhost:3306/MyDB";  
            Connection con = DriverManager.getConnection(url);  
  
            System.out.println(con);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

If you look at the above program, the first thing we need to do is import the JDBC classes using the following import statement. This is required for all the programs that use JDBC.

```
java.sql.*;
```

To get a connection to the database, we just have to follow the first two steps. In Step 1, we load the JDBC driver using the following statement:

```
Class.forName("com.mysql.jdbc.Driver");
```

A JDBC driver is normally specified using fully qualified name of the driver class. For MySQL database, the driver class name is `com.mysql.jdbc.Driver`. For some other databases, the class name will be different. The second step is to get the connection to the database by specifying the URL. So, what is a URL? A URL is something that identifies the database. Its syntax as shown below:

```
jdbc:<database name>://<host name or IP address where mysql is installed>:<port number>/<schema Name>
```

If the database is running on the same computer, the host name will be `localhost`. With the above syntax, the MySQL database URL will look as

```
jdbc:mysql://localhost:3306/MyDB
```

In the above URL, `MyDB` is the name of the database that we created while installing the database. You can imagine schema as a something that contains all the tables in the database. Every schema will have a name, which in our case is `MyDB`.

To get the connection, we need to invoke the static `getConnection()` method in the `DriverManager` class by passing the above URL as a string. This method then returns a `Connection` object as shown below:

```
String url = "jdbc:mysql://localhost:3306/MyDB";
Connection con = DriverManager.getConnection(url);
```

Once we got the connection object, we simply displayed it. Save the program in the ‘`jdbc`’ directory and compile and execute the program as shown below:

```
C:/JavaTraining>chapter11>javac jdbc*.java
C:/JavaTraining>chapter11>java jdbc.ConnectionTest
```

If all goes well and a connection is established, you’ll see the output something like shown below.

```
Connection@jhg1234
```

If something went wrong, you would see an exception message. As I said before, don't loose your heart if the program doesn't run. As long as you understood what the program is doing, you are good. Trust me.

In all the JDBC programs, steps 1,2 and 7 are always needed. Instead of writing the same code again and again, let's write a utility class that defines these three steps, and we can use this utility in all the programs to eliminate the redundant code. So, let's write this class.See code listed in 11.2.

Listing 11.2 (DBUtils.java) Simple utility class for Steps 1,2,7.

```
package jdbc;

import java.sql.*;

public class DBUtils {

    static Connection con;

    // Static block to load the driver just once
    static {

        try {
            // Step 1: Load the Driver
            Class.forName("com.mysql.jdbc.Driver");

        } catch (Exception e) {

            e.printStackTrace();
            System.out
                .println("Unable to load the Driver. Please check your
Classpath");
        }
    }

    public static Connection getConnection() {

        // Step 2: Get the connection by passing the URL
        try {

            String url = "jdbc:mysql://localhost:3306/MyDB";
            con = DriverManager.getConnection(url);

        } catch (Exception e) {

            System.out.println("Unable to get connection");
            e.printStackTrace();
        }
        return con;
    }

    public static void closeConnection() {
        try {
```

```
// Step 7: Close the connection.  
con.close();  
  
} catch (Exception e) {  
    System.out.println("Failed to close the connection");  
}  
  
}  
}
```

If you look at the above utility class, we have a **static** block to load the driver. The reason we put this in the **static** block is that we only have to load the driver once. This class defines two methods namely `getConnection()` to return the `Connection` object and `closeConnection()` method to close the connection.

In all our programs from now onwards, we'll use the above utility program to get the connection at the beginning, and close the connection at the end.

Creating Table

Our next program uses JDBC to create a table named **Customers** in the database. We will use this created table for the rest of the programs. Let's first look at the code in listing 11.3.

Listing 11.3 (`CreateTable.java`) JDBC program for creating a table.

```
package jdbc;  
  
import java.sql.*;  
  
public class CreateTable {  
  
    public static void main(String args[]) throws Exception {  
        createTable();  
    }  
  
    private static void createTable() throws Exception {  
  
        // Create the DDL  
        String sql = "CREATE TABLE CUSTOMERS" + "(" + "FIRSTNAME VARCHAR(50),"  
        + "MIDDLENAME VARCHAR(2)," + "LASTNAME VARCHAR(50),"  
        + "AGE NUMERIC(2)," + "SSN NUMERIC(10)," + "CITY VARCHAR(32),"  
        + "STATE VARCHAR(2)," + "COUNTRY VARCHAR(40)" + ")";  
  
        // Get the connection using our utils.  
        Connection con = DBUtils.getConnection();  
        if (con != null) {  
  
            // Create statement from connection  
            Statement stmt = con.createStatement();
```

```
// Execute the statement by passing the sql
int result = stmt.executeUpdate(sql);

if (result != -1) {
    System.out.println("Table created sucessfully");
} else {
    System.out.println("Could'nt create table. Please
                      check your SQL syntax");
}

// Close the statements and Connections
stmt.close();
DBUtils.closeConnection();

} else {
    System.out.println("Unable to get the connection");
}
}
}
```

As you can see from the above code, we created a class with a **main** method. The main method invokes the `createTable()` method which is where all the JDBC logic is written.

The `createTable()` method first defines the SQL query for creating the table and then gets the connection using our utility class as shown below:

```
con = DBUtils.getConnection();
```

Once we got the connection, we need to create a statement (Step 3). For this, we need to invoke the `createStatement()` method on the `con` object obtained in Step 2 as shown below:

```
Statement stmt = con.createStatement();
```

The above method returns a statement object that we can use to execute the SQL query. To execute the sql query, we use the `executeUpdate()` method and pass the sql as a string parameter shown below (Step 4):

```
int result = stmt.executeUpdate(sql);
```

The above method returns an `int` that specifies the status of query execution. If the value of the `result` variable is anything other than -1, then it means the table is successfully created, otherwise, it means some thing went wrong during SQL query

execution. Once all this is completed, we finally need to close the statements and the connection as shown below:

```
// Close the statements and Connections  
stmt.close();  
DBUtils.closeConnection();
```

That's it. Compile and execute the program as shown below:

```
C:/JavaTraining>chapter11>javac jdbc/*.java  
C:/JavaTraining>chapter11>java jdbc.CreateTable
```

If all goes well you notice the following output:

```
Table Created Successfully.
```

To verify this, open the DbVisualiser tool we installed and connect to the database. You'll see the customers table created.

Note: For all the SQL operations that doesn't return any results like CREATE, INSERT, DELETE, UPDATE we always use the `executeUpdate()` method on the statement object which returns an `int`. The `int` value specifies the number of records that effected.

Inserting Data

The next thing we need to do is populate the above **Customers** table with all the data. To spice up the things, let's assume we have all the data in a file named **Data.txt** as shown below:

Data.txt

```
John,M,Smith,20,12345,Denver,CO,USA  
Joe,R,Roberts,30,567567,Denver,CO,USA  
Sam,P,Perry,33,7686456,Denver,CO,USA  
Rob,ALewis,22,123123,Denver,CO,USA  
Mike,H,Ball,43,890890,Denver,CO,USA
```

Copy the above file into the following directory as,

```
C:/JavaTraining/chapter11/jdbc/Data.txt
```

The above file has five records and every record has eight columns of data separated by a comma. What we will do is,

1. Read the file line by line

2. Break the line into individual pieces
3. Insert the data into the table columns
4. Repeat steps 1,2,3 until the end of file is reached.

Take a look at the code in listing 11.4.

Listing 11.4 (`InsertRecord.java`) JDBC program for inserting data.

```
package jdbc;

import java.io.*;
import java.sql.*;
import java.util.StringTokenizer;

public class InsertRecord {

    public static void main(String args[]) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(
            new FileInputStream("Data.txt")));
        // Read line by line from the file
        while (true) {
            // Read the record line
            String line = br.readLine();

            // End of file, then break out of the loop.
            if (line == null) {
                break;
            }

            //Otherwise, parse the line
            StringTokenizer st = new StringTokenizer(line, ",");

            String firstName = st.nextToken();
            String middleName = st.nextToken();
            String lastName = st.nextToken();
            String age = st.nextToken();
            String ssn = st.nextToken();
            String city = st.nextToken();
            String state = st.nextToken();
            String country = st.nextToken();

            String sql = "INSERT INTO CUSTOMERS VALUES(" + " '" + firstName
                + " ' , " + " '" + middleName + " ' , " + " '" + lastName
                + " ' , " + age + " , " + ssn + " , " + " '" + city + " ' , "
                + " '" + state + " ' , " + " '" + country + " ' )";

            // Pass the sql to JDBC method
            insertCustomer(sql);
        } // End of while loop
    }

    private static void insertCustomer(String sql) throws Exception {
```

```
// Get the connection using our utils.;
Connection con = DBUtils.getConnection();
if (con != null) {

    // Create statement from connection
    Statement stmt = con.createStatement();
    // Execute the statement by passing the sql
    int result = stmt.executeUpdate(sql);

    if (result != -1) {

        System.out.println("Inserted" + result + "Record(s) successfully");
    } else {

        System.out.println("Unable to insert record. Please check
                           your SQL syntax");
    }

    // Close the statements and Connections
    stmt.close();
    DBUtils.closeConnection();

} else {

    System.out.println("Unable to get the connection");
}
}
```

The above program though long, is very simple. All it does is, the `main()` method reads the file record by record, breaks the record using the `StringTokenizer` class, then assigns all the eight tokens in the record to appropriate variables, and constructs a query string in proper format. It then passes the SQL query to the `insertCustomer()` method which executes it. The code in this method is just the same as previous program. Compile and execute the code to see the following result.

```
Inserted 5 records successfully.
```

To verify this, open the DbVisualiser tool and connect to the database. Click on the **Customers** table, and you'll see the records.

Reading the Data

This is the most important and widely performed operation. In order to retrieve the data, we use the following method on the statement object:

```
public ResultSet executeQuery ( String sql )
```

The above method takes the query and returns a `ResultSet` object that contains all the records returned by the database. We should then *iterate* over it and display the data. To retrieve the data from the `ResultSet`, there are several *get* methods available based on the type of the data. Let's look at an example that retrieves all of the customers data and then see how it works. See listing 11.5.

Listing 11.6 (`ReadData.java`) JDBC program for reading data.

```
package jdbc;

import java.sql.*;

public class ReadData {

    static Connection con = null;
    static Statement stmt = null;
    static ResultSet records = null;

    public static void main(String args[]) throws Exception {
        readData();
    }

    private static void readData() throws Exception {

        // Create the SQL
        String sql = "SELECT * FROM CUSTOMERS";
        // Get the connection using our utils.
        con = DBUtils.getConnection();

        if (con != null) {

            // Create statement from connection
            stmt = con.createStatement();
            // Execute the statement by passing the sql.
            records = stmt.executeQuery(sql);

            // Iterate over all the returned records
            while (records.next()) {

                String firstName = records.getString(1);
                String middleName = records.getString(2);
                String lastName = records.getString(3);
                int age = records.getInt(4);
                int ssn = records.getInt(5);
                String city = records.getString(6);
                String state = records.getString(7);
                String country = records.getString(8);

                System.out.println(firstName + " , " + middleName + " , " + lastName
                    + " , " + age + " , " + ssn + " , " + city + " , " + state + " ,
                    + country);

            } // End of While
        } else {
    }
```

```
        System.out.println("Unable to get the connection");
    }
}
```

As you can see from the above code, we used the `executeQuery()` method to execute the `SELECT` query. This method will return a `ResultSet` object as shown below:

```
ResultSet records = stmt.executeQuery(sql);
```

A resultset object can be imagined like a table of records with a pointer at the beginning of the table as shown below:



FirstName	LastName	SSN	Age	City	State	Country
Joe	Roberts	123456	20	Vegas	CA	USA
Sam	Perry	34534543	22	LA	CA	USA
Serra	Bates	567657	67	SFO	CA	USA

To read the records, we need to move the pointer to the record using the `next()` method until the pointer reaches the end of the records. This is what the *while* loop does. Once we are in the loop, we need to read the record that the pointer points using the `get` methods. The `get` methods can either take the column number starting from 1 or the column name itself. In our example we used the column numbers. Following are the most commonly used `get` methods:

```
public String getString()    Used for VARCHAR columns
public int getInt()         Used for NUMERIC columns
public Date getDate()       Used for DATE columns.
```

Based on the above methods, since SSN and Age are declared as NUMERIC data types when we first created the table, we used `getInt()` method to read the data, and all the remaining are VARCHAR types, so we used `getString()` method to read the column data as shown below:

```
String firstName = rs.getString(1);
String middleName = rs.getString(2);
String lastName = rs.getString(3);
String age = rs.getInt(4);
String ssn = rs.getInt(5);
String city = rs.getString(6);
String state = rs.getString(7);
String country = rs.getString(8);
```

Instead of specifying the column numbers, we can also specify the column names as shown below:

```
String firstName = rs.getString("FirstName");
String lastName = rs.getString("LastName");
```

You are free to use which ever convention you like. Once we get the data, we can do what ever we want. In our case, we simply displayed it. The *while* loop will print all the records and terminates. Compile and execute the code. If all goes well, you'll see the following output:

```
John,M,Smith,20,12345,Denver,CO,USA
Joe,R,Roberts,30,567567,Denver,CO,USA
Sam,P,Perry,33,7686456,Denver,CO,USA
Rob,ALewis,22,123123,Denver,CO,USA
Mike,H,Ball,43,890890,Denver,CO,USA
```

If you noticed carefully, most of the code is same in all the programs. The only difference is the execute method that we call. Here is the trick to remember:

Trick: If the query is SELECT, use the `executeQuery()` method. For all other queries (CREATE, INSERT, DELETE, UPDATE etc) use the `executeUpdate()` method.

The DELETE and UPDATE operations work the same way as CREATE and INSERT. You can use the same program by just modifying the sql. Take this as home work. Use the code in listing 11.3, but use the following queries.

For updating data use the following query.

```
UPDATE CUSTOMERS SET Country = 'Canada' where Age < 30;
```

After running the code in listing 11.3 with the above query, run the code in listing 11.5 to see the updated data.

For deleting data use the following query.

```
DELETE FROM CUSTOMERS WHERE Age < 30;
```

After running the the code in listing 11.3 with the above query, run the code in listing 11.5 to see the updated data. (Some records will be deleted).

If you observe the SQL queries we used in the above examples, we hard coded all the data in the query itself as shown below:

```
SELECT * FROM CUSTOMERS WHERE NAME = 'sam';
INSERT INTO CUSTOMERS VALUE ( 'John', 'Smith', 12, 23432....);
```

In real world applications, data should be inserted or retrieved on the fly based on the data supplied at run time. As an example, let's say we want to write a search application that searches for customers based on last name. In this case, the SQL query must be dynamically built as shown below:

```
void search( String lastName ){  
    String sql = "SELECT * FROM CUSTOMERS WHERE LASTNAME ='" + lastName + "'";  
    // Execute the query  
}
```

Here we wrote a method that takes the lastname and dynamically constructs the query by appending it at the end. Though the above approach works, it gets cumbersome while formatting the sql. We need to carefully use all the punctuation marks in the sql like the commas, quotes etc as shown above. In JDBC, there is a better approach for executing the queries whose parameters are dynamic. Let's see what it is.

PreparedStatement

In real world applications, the data to the query comes from some other world. One good example is registering in a web site where we fill in a HTML form and submit it. The submitted data will then go to a JDBC program that reads the data and inserts it into the database. This is where prepared statements come in handy. These will not only simplify the process, but also improve the performance of the application.

Like the name suggests this is really a “prepared” statement. This class replaces the old Statement class that we used for executing the queries in the previous examples. The usage of this class is pretty simple. It involves three simple steps as listed below:

1. Create a PreparedStatement
2. Populate the statement
3. Execute the prepared statement.

The basic idea with this class is using the ‘?’ symbol in the query wherever the data is to be substituted and then when the data is available, replace the symbols with the data. Let see how we do this.

Step1: To create a prepared statement we use the following method on the Connection object as shown below:

```
PreparedStatement prepareStatement ( String sql );
```

The SQL query to be passed to the above method will look as shown below:

```
String sql = "INSERT INTO CUSTOMERS VALUE ( ?, ?, ?, ?, ?, ?, ?, ? )";
```

Since we need to insert data into eight columns, we used eight question marks. If you notice the query, there are no single quotes like we had in earlier examples. Once we have the above sql, we create the prepared statement as shown below:

```
PreparedStatement stmt = conn.prepareStatement ( sql );
```

Step 2: Now that we created a prepared statement, its time to populate it with the data. To do this, we use the *set* methods as shown below:

```
stmt.setString (1, name data);
stmt.setInt (3, ssn data);
```

Looks like the above *set* methods are familiar, right? These work opposite to the *get* methods we used with `ResultSet` class couple pages back. The first parameter to the *set* method denotes the position of '?' to replace and the second argument is the actual data. The trick here is, which *set* method to use. Look at the following table that tells which method to use:

Table 11.3 Prepared Statement methods.

If Database Column is	Method to use is
VARCHAR	<code>setString()</code>
NUMERIC or INTEGER	<code>setInt()</code>
DATE OR TIMESTAMP	<code>setDate()</code>

In the above sql, the first '?' represents 'firstName' which is a VARCHAR. So, we use `setString()` method. Let's write a simple example to get a clear picture. See the code in listing 11.7.

Listing 11.7 (`PreparedStatementDemo.java`) JDBC program using prepared statement.

```
package jdbc;

import java.sql.*;

public class PreparedStatementDemo {
    public static void main(String args[]) throws Exception {
        // Pass the data to the method.
```

```
insertCustomer("Brian", "B", "Smith", 20, 123456789, "Columbus", "OH",
    "USA");
}

private static void insertCustomer(String firstName, String middleName,
    String lastName, int age, int ssn, String city, String state,
    String country) throws Exception {
    PreparedStatement ps = null;

    // Create the SQL
    String sql = "INSERT INTO CUSTOMERS VALUES (?, ?, ?, ?, ?, ?, ?, ?);";

    // Get the connection using our utils.
    Connection con = DBUtils.getConnection();

    if (con != null) {
        // Create statement from connection
        ps = con.prepareStatement(sql);

        // Populate the data

        ps.setString(1, firstName);
        ps.setString(2, middleName);
        ps.setString(3, lastName);
        ps.setInt(4, age);
        ps.setInt(5, ssn);
        ps.setString(6, city);
        ps.setString(7, state);
        ps.setString(8, country);

        // Execute the prepared statement
        int result = ps.executeUpdate();

        if (result != -1) {
            System.out.println("Inserted " + result +
                " Record(s) successfully");
        } else {
            System.out
                .println("Unable to insert record.
                    Please check your SQL syntax");
        }
    }

    // Close the statements and Connections
    ps.close();
    DBUtils.closeConnection();
} else {
    System.out.println("Unable to get the connection");
}
}
```

If you look at the above code, you'll see all the JDBC steps we learned before. Always remember one thing. The numbers in the *set* methods denote the position of the question

mark symbol but not the column number in the table. The `main()` method simply passes the data to the `insertCustomer()` method. This method will then use its parameters to populate the prepared statement. Look how clean this program is. There is absolutely no query formatting at all. Compile and execute the code. If all goes well, you'll see the following output:

```
Inserted 1 record(s) successfully.
```

Following are some of the examples that demonstrate the usage of `PreparedStatement` class.

```
String sql = "SELECT * FROM CUSTOMER WHERE FIRSTNAME = ? ";
stmt.setString ( 1, "John");

String sql = "UPDATE CUSTOMERS SET FIRSTNAME= ? WHERE SSN = ?";
stmt.setString ( 1, "Smith");
stmt.setInt ( 2, 87690 ); // Used setInt since ssn is a number in the table
```

This completes all the important things that you need to know in JDBC. The above examples are what you'll see in most of the real world applications. The only difference is that you'll see several of these sql queries and several classes. If you understood the above examples, trust me, you can call yourself a champion JDBC developer. You can curse me if you see anything besides the above.

Batch Processing

The JDBC programs we wrote until now simply execute one SQL query at a time. If you have several of these queries, what you can do is store the SQL queries in a file, read one by one, and execute it. The only drawback with this approach is that, we need to call the `executeUpdate()` method for each and every query, right? So, if you have 1000 queries, you have to call the `executeUpdate()` method 1000 times.

Executing a single SQL query means, we are physically sending each query to the database possibly across the network (in real world applications, databases are distributed across the network), and making the database run it. This process takes some time and resources. It would rather be good, if we could send all the queries across the network in one go, and then have the database execute all the queries in one go. This is what is called as **batch processing** of queries.

To do batch processing using JDBC, we need to follow 2 simple steps:

1. Add all the SQL queries to the batch
2. Execute the batch.

The only limitation with batch processing is that we cannot add SELECT queries to the batch. Always use INSERT, UPDATE and DELETE statements with batch processing. Following are the two methods in Statement class that support batch processing:

```
public void addBatch(String sql);
int executeBatch();
```

The first method is used add a query to the batch, and the second one to execute the batch after all the queries are added. Let's do an example. See the code in listing 11.8.

Listing 11.8 (BatchDemo.java) JDBC program processing a batch.

```
package jdbc;

import java.sql.*;

public class BatchDemo {

    static Connection con = null;
    static Statement stmt = null;

    public static void main(String args[]) {
        try {
            runAsBatch();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            DBUtils.closeConnection();
        }
    }

    private static void runAsBatch() throws Exception {
        // Create the SQLs

        String sql1 = "INSERT INTO Customers
                      VALUES('Customer1','A','Smith',11,12345,'Columbus','OH','USA');");
        String sql2 = "INSERT INTO Customers
                      VALUES('Name1','A','Smith',12,12345,'Columbus','OH','USA');");
        String sql3 = "INSERT INTO Customers
                      VALUES('Name2','A','Smith',13,12345,'Columbus','OH','USA');");
        String sql4 = "INSERT INTO Customers
                      VALUES('Name3','A','Smith',14,12345,'Columbus','OH','USA');");
        String sql5 = "INSERT INTO Customers
                      VALUES('Name4','A','Smith',15,12345,'Columbus','OH','USA');");
        String sql6 = "INSERT INTO Customers
                      VALUES('Name5','A','Smith',16,12345,'Columbus','OH','USA');";

        // Get the connection using our utils.
```

```
con = DBUtils.getConnection();
if (con != null) {
    // Create statement from connection
    stmt = con.createStatement();

    // Add the SQL's to the Batch
    stmt.addBatch(sql1);
    stmt.addBatch(sql2);
    stmt.addBatch(sql3);
    stmt.addBatch(sql4);
    stmt.addBatch(sql5);
    stmt.addBatch(sql6);

    // Run the batch
    int[] status = stmt.executeBatch();

    for (int i = 0; i < status.length; i++) {
        if (status[i] != -1) {
            System.out.println("Processed SQL");
        } else {
            System.out.println("Something went wrong");
        }
    }
}

else {
    System.out.println("Unable to get the connection");
}
}
```

If you look at the `runAsBatch()` method, it defined six queries and then added them using the `addBatch()` methods. Once all the queries are added, we used the `executeBatch()` method to execute all the queries in one go. Compile and execute the code. If all goes well, you'll see the following output:

```
Processed SQL
Processed SQL
Processed SQL
Processed SQL
Processed SQL
Processed SQL
```

With the above knowledge of batch processing, let's move on to some thing that is really important while working with databases.

Database Transactions

In some situations, processing a single business transaction might involve executing more than one SQL queries in which *all* the queries must either succeed or fail. In such

scenarios what we need to do is, group all the queries as one *single unit of work* and execute this single unit as a whole. Such a unit is called as a transaction.

Let's look at a real world scenario. Two customers go to bank to transfer funds from one account to another. This financial transaction requires the following two basic operations.

1. Decrement the funds from A's account
2. Increment the funds in B's account

For the business transaction to succeed, both the above two operations must succeed. Even if one fails, the entire financial transaction must fail. Let's say, SQL1 and SQL2 perform the above two operations. If SQL1 succeeds and SQL2 fails for some reason, say, the database went down, from business standpoint this means we debited funds from A's account but failed to credit B's account. This is totally unacceptable and the customers will freak out.

To prevent this, what we need to do is, group SQL1 and SQL2 as one operation (we call this as Transaction) and if any of the SQL fails then we need to **rollback** the transaction, meaning, bring back A's and B's accounts to original state that existed before the transaction began. However, if both the SQL's succeeded, then we need to **commit** the transaction in which case changes should be made permanently in the database.

In the JDBC Connection interface, there are three methods for transaction support namely,

```
public void setAutoCommit()  
public void commit() and  
public void rollBack().
```

Using the above methods, let's write a program and then see how it works. See the code in listing 11.9.

Listing 11.9 (`TransactionDemo.java`) JDBC program processing transactions.

```
package jdbc;  
import java.sql.*;  
public class TransactionDemo {  
    public static void main(String args[]){  
        try {
```

```
        openJointAccount();

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
    DBUtils.closeConnection();
}
}

private static void openJointAccount() throws Exception {

    Connection conn = null;
    Statement stmt = null;

    // Create the SQLs
    String sql1 = "INSERT INTO Customer
                  VALUES ('Customer1','A','Smith',11,12345'Columbus','OH','USA');");
    String sql2 = "INSERT INTO Customer
                  VALUES ('Name1','A','Smith',12,12345,'Columbus','OH','USA');";

    // Get the connection using our utils.
    conn = DBUtils.getConnection();

    if (conn != null) {
        // Create statement from connection
        stmt = conn.createStatement();

        // Reset the transaction
        conn.setAutoCommit(false);

        // Execute the SQL's
        int status1 = stmt.executeUpdate(sql1);
        int status2 = stmt.executeUpdate(sql2);

        if (status1 != -1 && status2 != -1) {
            System.out.println("Joint account created");
            conn.commit();
        } else {
            System.out.println("Something went wrong!
                               Rolled back the transaction");
            conn.rollback();
        }
    }

    else {
        System.out.println("Unable to get the connection");
    }
}
```

Here is what the above program does. It first creates a `Statement` object. To initiate a database transaction, it need to turn off the auto-committing the data using the following statement:

```
conn.setAutoCommit(false);
```

The above statement will tell the database not to permanently make the table changes until the user informs it to do so. We then execute the two queries using the `executeUpdate()` method as shown below:

```
int status1 = stmt.executeUpdate(sql1);
int status2 = stmt.executeUpdate(sql2);
```

We then check if the values of `status1` and `status2` are positive or not and based on it we either commit or rollback the transaction as shown below:

```
if (status1 != -1 && status2 != -1) {
    System.out.println("Joint account created");
    conn.commit();
} else {
    System.out.println("Something went wrong!
                        Rolled back the transaction");
    conn.rollback();
}
```

In most of the real world applications, there will be multiple queries to execute as part of single transaction and are always run within a database transaction. This completes all the practicals that we need to do in this chapter. You can stop this chapter here and can go into the next chapter. However, there are some theoretical concepts that you need to now to be more comfortable. Read them like a story, ok. They are simple though :-)

Connection Pooling

In all the programs we wrote thus far, there is one thing that we are repeatedly doing all the times. Opening a connection at the beginning and closing the connection at the end when everything is done. If you noticed carefully, we are always closing the connection in the `finally` block. Can you guess why we are doing this? You guessed it right. We always want to close the connection irrespective of whether the program runs or not. There are two major problems that hunt us when we write database programs.

Problem 1: Usually, every database has **limited** number of connections. Let's say a database supports only 10 connections, and if your program doesn't close the connection every time it ran, the 11th time you run, database will refuse to give you a connection. If you don't close a connection after using it, from database point of view somebody is still using the connection, and it will not reuse this connection until the connection is closed. You need to shutdown the database and restart it to release all the connections which is not good.

Solution: Always make sure you close the database connection.

Problem 2: Let's say we have a database that can support a maximum of 200 connections at a time and the application we wrote supports 1000 customers at a time. So, if the 200 connections are used for 200 customers, what will happen to the remaining 800 customers? They are simply kicked off. This is no good because you are loosing business.

Solution: Need to have a sophisticated mechanism to manage the connections effectively and efficiently by optimizing the usage of database connections.

Such a mechanism is what we call as **connection pooling**. Now the question is who implements connection pooling? To implement the connection pooling we need to know all the intricacies of database **connection**. Who else other than the database vendors know better about connection details? This is the reason why database vendors themselves implement connection pooling mechanism in Java using certain standards. This is a big relief for us, right? The connection pooling program comes along when we download the drivers for a particular database. Sweet! Connection pooling mechanism addresses both the above problems.

The way the connection pooling works is,

1. It initializes a pool of connections with the database.
2. When a connection is requested by the application, it returns a connection from the pool.
3. The application after using the connection returns it back to the pool.

Take a look at Fig 11.6 to get a clear idea.

Connection Pool Program

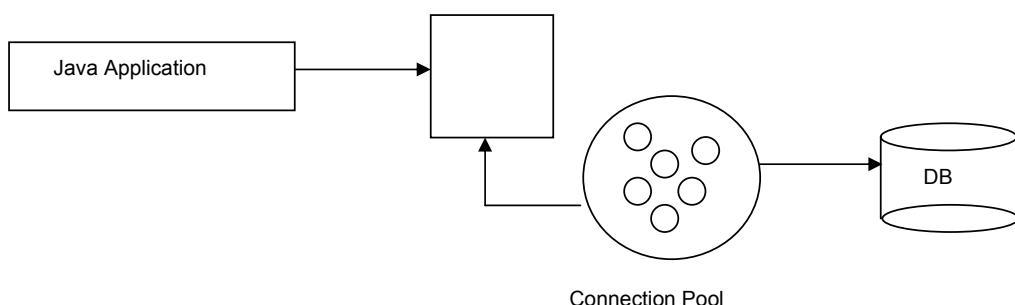


Fig 11.6 Connection Pooling

Connection pooling will be implemented in such a way that all the details are hidden behind the scenes. The only change we as application developers need to make is the way we retrieve the connection object. Once we get a connection object, the rest of the program will be the same.

Connection pooling is usually used with large scale enterprise applications where thousands of processes need to access the database at the same time. In such situations the application server will be configured to use the connection pooling. Since we are not yet there, we will revisit this topic in the later chapters. For now understanding the details of connection pooling is good enough.

Database Isolation levels

I will conclude JDBC with this one last topic. Since we are working with databases, we need to know some of the general problems that any database can have, and check whether the database vendors addressed the problems in their database or not. These problems may play a significant role in paralyzing your enterprise application. Hmm. So, before using a particular database say MySQL, we need to check whether it addressed the problems or not.

The 3 most common problems with any database are:

1. Dirty Reads
2. Repeatable Reads
3. Phantom Reads

Let's see what these are, and find out how much damage they may cause to our application.

Dirty Read

Quite often in database processing, we come across a situation wherein one transaction can change a value, and a second transaction can read this value before the original change has been committed or rolled back. This is known as dirty read because there is always a possibility that the first transaction may rollback the change, resulting in the second transaction reading an invalid value.

Ex: Let's say we have a joint account with \$500 in it. Jim goes to the bank to debit \$200, and initiates a transaction. Now the account will have \$300 in it. At this point of time, Tom goes to ATM to withdraw \$400. His transaction sees only \$300 in the account and

gets rejected saying "Insufficient funds" which is OK. Now, Jim changes his mind and cancels his transaction at the bank, so his 200\$ is credited back which leaves the account back at \$500. Few minutes later, Tom checks the account balance and to his surprise sees \$500 in it and freaks out. What's going on? The good thing happened is the funds didn't get lost. All this is to say that dirty reads don't cause damages to data, but they confuse the users.

Repeatable Read

'B' performs a read, but 'A' modifies or deletes that data later. If 'B' reads the same row again, he will get different data.

Phantom Reads

'A' does a SELECT query on a table and gets 10 matching records. 'B' modifies the table such a way that when 'A' queries the table again it would get different number of records back.

In all the above three problems, what is happening is that the transactions are getting overlapped with each other. So the solution is, to isolate the individual transactions. Every database will have something called *isolation level* that indicates which of the above problems are addressed. All we need to do is, check this isolation level and then decide whether or not we want to continue with our database transactions.

The isolation level is a number from 0-3 (inclusive). These are represented as CONSTANTS in the Connection interface as follows.

```
public static final int TRANSACTION_READ_UNCOMMITTED = 0;
public static final int TRANSACTION_READ_COMMITTED = 1;
public static final int TRANSACTION_REPEATABLE_READ = 2;
public static final int TRANSACTION_SERIALIZABLE = 3;
```

Following table lists various isolation levels and what problems are addressed by various isolation levels:

Table 11.4 Isolation Levels

Isolation Level	DirtyRead Addressed	Repeatable Read Addressed	Phantom Read Addressed
0	No	No	No
1	Yes	No	No
2	Yes	Yes	No
3	Yes	Yes	Yes

As you can see from the above table, the best isolation level is 3 which address all the above problems. Following is the method in the Connection interface that returns the isolation level of the database.

```
public int getIsolationLevel();
```

Without wasting any further time, let's write a small program to display the isolation of MySQL database. See the code in listing 11.10.

Listing 11.10 (`IsolationTest.java`) JDBC program checking the isolation levels.

```
package jdbc;

import java.sql.*;

public class IsolationTest {

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

    private static void checkIsolation() {
        Connection con = null;

        try {
            if (con.getTransactionIsolation() ==
                    Connection.TRANSACTION_READ_COMMITTED) {
                System.out.println("MYSQL database addresses DIRTY READS ONLY");
            } else if (con.getTransactionIsolation() ==
                    Connection.TRANSACTION_REPEATABLE_READ) {
                System.out.println("MYSQL database addresses DIRTY READS and
                    REPEATABLE READS ONLY");
            } else if (con.getTransactionIsolation() ==
                    Connection.TRANSACTION_SERIALIZABLE) {
                System.out.println("MYSQL database addresses DIRTY READS, REPEATABLE
                    READS and PHANTOM READS.Good");
            } else if (con.getTransactionIsolation() ==
                    Connection.TRANSACTION_READ_UNCOMMITTED) {
                System.out.println("MYSQL database does not address anything.");
            }
        // Most of the databases will address all the three problems. So the
        // isolation level will always be 3.
        // But many a times, we want to set our own isolation level. we can
        // use the following to only address "Dirty reads".
        con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
    }
}
```

```
// Here you do your SQL operations, like creating statements and all  
// that good stuff.OK  
  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    DBUtils.closeConnection();  
}  
  
}  
}
```

In the above code, we got the isolation level using the `getIsolationLevel()` method and then checked which level of isolation is supported by the database using the conditional statements. Compile and execute the above program to see which isolation level is supported by your database.

Most of the databases address all the three problems. So, when you try to get the isolation level using the above code, it always returns 3. Isolation level 3 is termed as "very strong isolation" and poses serious performance problems. 99.9% of the enterprise applications are fine with isolation level 2 even if the database supports isolation level 3. In such cases, we need to select our own isolation level to improve the application performance.

To change the isolation level, use the following method in the `Connection` interface.

```
public void setTransactionIsolation(int isolationLevel);
```

In our example we used the above method and changed the isolation level to 1.

This completes all the topics in this chapter. The important things that you need to understand is how to execute the SQL queries using `Statement` and `PreparedStatement` classes. I am sure you are now an expert in writing JDBC programs to work with the databases. If the concept like isolation levels is confusing, don't worry about it. It is seldom seen in real world applications.

Summary

- ✓ JDBC technology is used by Java applications to connect with database.
- ✓ JDBC is a universal API for accessing any database.
- ✓ Every database will have something called as **Driver** using which external applications can connect to database.
- ✓ There are four different types of database drivers namely, Type-1, Type-2, Type-3 and Type-4.

- ✓ JDBC uses Pure Java Driver (Type-4) driver to connect with databases.
- ✓ JDBC applications need to load the driver before getting a connection to the database.
- ✓ Using JDBC API such as Statement and PreparedStatement classes, we can execute different SQL queries.
- ✓ We use `executeQuery()` method to execute SELECT statements, and `executeUpdate()` method for all other SQL queries.
- ✓ The main difference between Statement and PreparedStatement classes is that Statement object always compiles the SQL before executing it, while the later compiles the query just once and execute 'n' number of times. Because of this, PreparedStatement offers better performance.
- ✓ PreparedStatement provides more flexibility to bind the data.
- ✓ We use CallableStatement to execute stored procedures.
- ✓ To optimize the usage of connections in enterprise applications, we use connection pooling.
- ✓ Connection pooling maintains a pool of connections with the database. An application retrieves the connection from the pool, and upon using it returns back to the pool instead of closing the connection.
- ✓ Application Servers usually configure the connection pooling program.
- ✓ Database transactions are used to group multiple queries into one single operation.
- ✓ The transaction will either **commit** or **rollback** based on the query results.
- ✓ Every database have problems like **Dirty Reads, Repeatable Reads, Phantom Reads**.
- ✓ We use the Isolation Level of the database to check whether the above problems are addressed by the database vendors or not.

Time to play 50-50

1. Which of the following is used to load the database driver?
 - a) `Class.forName("com.mysql.jdbc.Driver")`
 - b) `Class.loadClass("com.mysql.jdbc.Driver");`
2. Which of the following is used to retrieve a connection to database?
 - a) `DriverManager.loadConnection();`
 - b) `DriverManager.getConnection();`
3. Which of the following is a valid database URL?

- a) http:mysql://localhost:3306/MyDB
 - b) jdbc:mysql://localhost:3306/MyDB
4. Which of the following interface represents a JDBC connection?
- a) JDBCConnection
 - b) Connection
5. Which of the following class is used for executing SQL queries?
- a) Statement
 - b) SQLStatement
6. Which of the following method returns a Statement object?
- a) createStatement()
 - b) getStatement()
7. Which of the following method returns a PreparedStatement object?
- a) createPreparedStatement()
 - b) prepareStatement();
8. Which of the following method is used to execute an UPDATE query?
- a) executeQuery()
 - b) executeUpdate()
9. Which of the following method is used to execute SELECT statement?
- a) executeQuery()
 - b) executeUpdate()
10. Which of the following driver JDBC normally uses?
- a) Type – 1 Driver
 - b) Type – 4 Driver

Interview Questions

Question: What are the different types of JDBC drivers?

Answer: JDBC-ODBC bridge driver, Partly Native Partly Java driver, Intermediate Server Driver, Pure Java Driveer. For details, refer to page 280.

Question: Explain the various steps that to execute SQL queries using JDBC.

Answer: Refer to page 284.

Question: What is the difference between Statement and PreparedStatement?

Answer: Statement object always compiles the SQL before executing it, while the later compiles the SQL just once and execute 'n' number of times. Because of this, PreparedStatement offers better performance. PreparedStatement is used for executing dynamic SQL queries where the data is plugged in at run time.

Question: What is a database transaction?

Answer: A database transaction represents a group of SQL's which either succeed or fail as a whole. Based on the results, the transaction is either committed or rolled back.

This completed the JDBC chapter. I am sure you enjoyed this chapter more than anything else. We are now ready to learn another simple technology used to build web applications. The fun begins. Let's move forward.

Chapter 12

XML And Java

By the end of this chapter, you should be able to know what XML is and its usage in Java. This chapter will also give you an idea on parsing XML documents using Java. Understanding this chapter is very important from J2EE point of view as XML is widely used in almost all the J2EE applications.

Chapter Goals

- ✓ Understand what XML is
- ✓ Understand the XML standards SAX and DOM
- ✓ XML parsing using SAX and DOM
- ✓ Understanding JAXP API for parsing XML documents

Environment Setup

1. Create directory ‘xml’ as shown below:

C:/JavaTraining/chapter12/xml

Save all the programs in this chapter in “xml” directory.

2. Copy C:\sun\AppServer\lib\xercesImpl.jar file into the already created j2eelib directory.
3. Create a batch file named “env.bat” as shown below

C:/JavaTraining/chapter12>env.bat

and copy the following contents into it

```
set CLASSPATH=.;  
C:\JavaTraining\j2eelib\j2ee.jar;C:\JavaTraining\j2eelib  
\mysql-connector-java-5.0.3-bin.jar;c:\JavaTraining\  
j2eelib\xercesImpl.jar;
```

Note: Whenever you start working with XML, the first thing you need to do is execute the batch file as shown below to setup the CLASSPATH:

```
c:/JavaTraining>chapter12>env.bat
```

Once you execute the above command you are ready to compile and execute any programs in this chapter.

Introduction

XML technology has become a buzz word every where in the IT community. Ever since its inception, XML technology has made leaps and bounds and completely changed the way enterprise computing is done. Its adoption by the software industry is such that you'll hardly find any software application that doesn't use XML. So, let's see what this is and how we can use XML in the world of J2EE.

What is XML?

XML stands for "eXtensible Markup Language". The notion of XML is based on something called XML Document.

What is an XML Document?

We all know that there are several ways for storing data and information. For instance, we use a text file to store the data line by line, we use database where tables are used to store data etc., XML document is just another way for storing data and information but uses a *tree* like structure.

An XML document/file comprises of several *tags* that represent the data or information. These tags are also referred to as nodes. Following is how a XML tag looks like:

```
<lastName>John</lastName>
```

In the above tag, `lastName` is the name of the tag. Every tag has a beginning and an end. The beginning of the tag is denoted by the tag name in between '`<`' and '`>`' and the end of the tag is denoted with the tag name in between '`</`' and '`>`' symbols. All the characters or text in between represent the tag data

The above tag is the simplest and smallest tag you can see in an XML document. However, we can also have complex tags which are nothing but tags within tags as shown below:

```
<name>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
</name>
```

Here, name is a complex tag that contains two simple tags namely firstName and lastName. The good thing with XML is that, the tag names can be anything. However, there is just one rule that we need to strictly follow.

Rule: Every tag that is opened must be closed.

If the above rule is followed, then we say that the XML document as *well formed*. Following shows a well formed XML document:

```
<?xml version="1.0"?>
<customer>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <age>20</age>
    <ssn>23324</ssn>
    <address>
        <addressline1>Apt 2222</addressline1>
        <city>Columbus</city>
        <state>OH</state>
        <country>USA</country>
    </address>
</customer>
```

The above XML represents customer information. If you notice, every tag has a closing tag which is why we say that it is well formed. Don't you see a tree like representation here? I am sure you did. In this document, customer is the root node. This has five child nodes namely firstName, lastName, age, ssn and address. The address child node in turn has 4 child nodes. There is no limit on the number of child nodes a node can have. Following figure shows the tree representation of the above xml document:

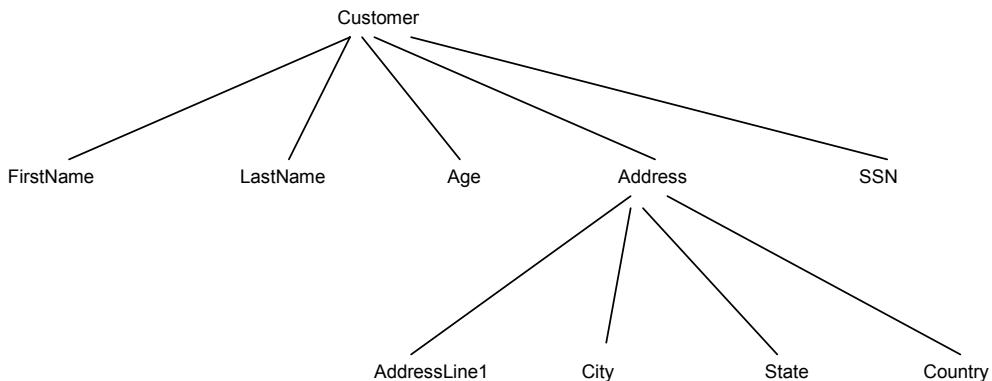


Fig 12.1 XML Tree

I know you guys would be thinking what the following line is

```
<?xml version="1.0"?>
```

We call this as the *prolog* of the XML document. It represents the version of the XML we are using.

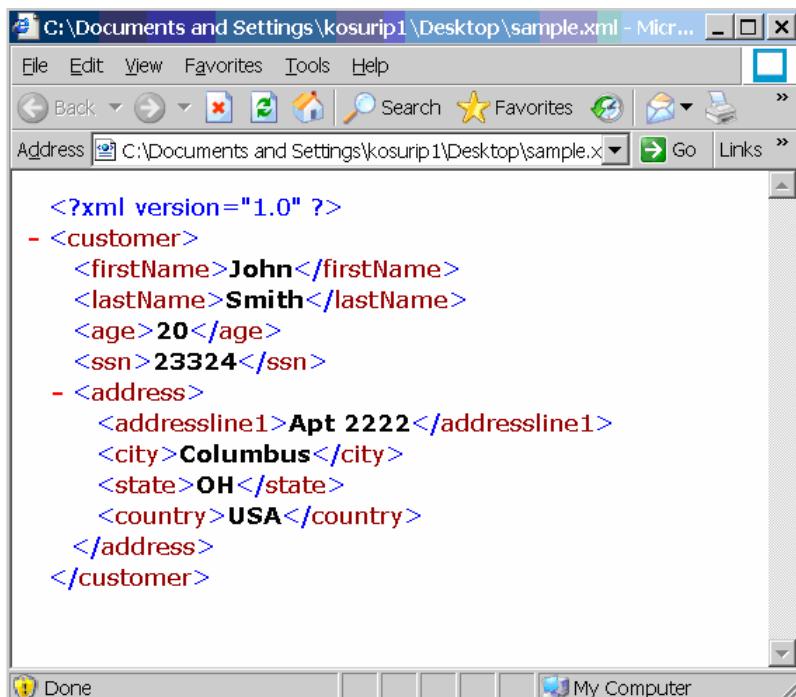
In XML, a node can also have *attributes* as shown below:

```
<customer email = "john.smith@abc.com">
```

In the above node, *email* is the attribute of tag whose value is *john.smith@abc.com*. A tag can have any number of attributes as shown below:

```
<book author="john" isbn="HG76876" pages="200">
    <publisher>Wrox</publisher>
</book>
```

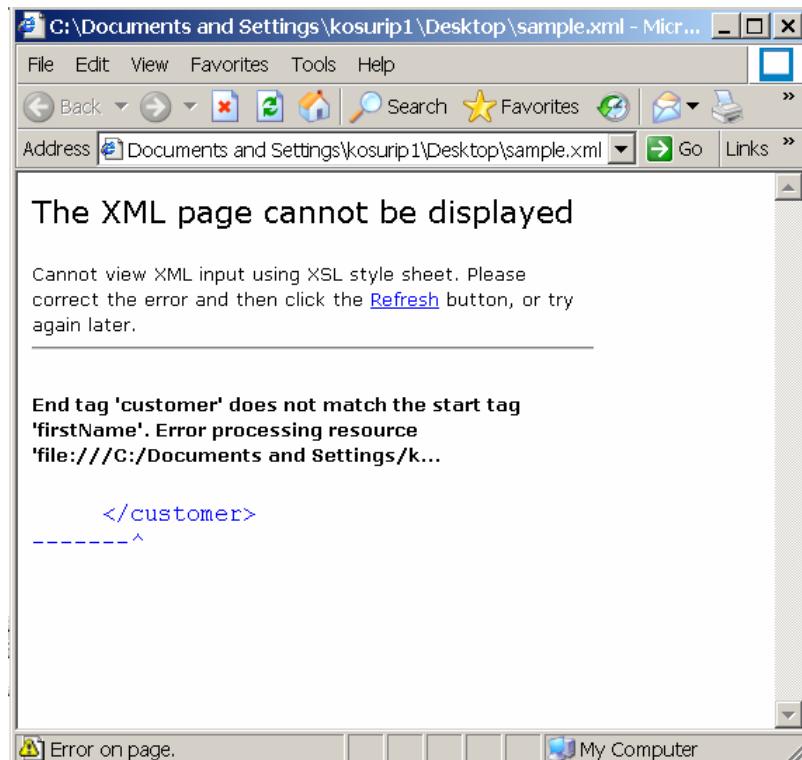
An XML document is saved with the extension ".xml". One simple way of verifying whether an XML document is well-formed or not is by opening the xml file using any browser like Internet Explorer. If the document is well-formed, you'll see the complete XML in the browser as shown below:



Chapter 12

XML And Java

If we fail to close some tag, the browser will display an error as shown below indicating that the XML document is not well formed. The error will also list the element name where the violation occurred.



Why XML is important?

The main reason why XML has tasted unprecedented success is because it is 100% language independent and platform independent. It can be used to represent any complex data or information without any limitations on size with *user defined* tags. The only rule is that the document must be well formed. This is the reason why XML though very simple, is yet so powerful.

Since enterprise applications involve complex data representation and processing, the flexibility that XML offers make it an ideal candidate for using in such situations. Over the course of next several chapters, you'll see how J2EE extensively uses XML in various ways for simplifying things.

XML Validation

As I said before XML is used to represent data or information. Whenever we have any data, the first thing we need to do before processing the data is to verify whether the data is valid or not, right? So the question is how to validate the data represented by XML document? The simplest way is by using a DTD which stands for Document Type Definition. So, let see how to use DTD to validate data.

Document Type Definition (DTD)

A DTD basically defines the following things:

1. The order of elements in the XML
2. The valid list child elements of a particular element
3. The list of valid attributes for a particular element

All the validations defined using DTD are stored in a file with the extension “.dtd” and is referenced by the XML document. A DTD defines the validation rules for elements in an XML document using the ELEMENT declaration. Let's see how we define an xml element using a DTD declaration.

The syntax for the element declarations in a DTD is shown below:

```
<!ELEMENT element name content-model>
```

The content-model basically tells the type of content the element can have. There are four types of content models an element can have as listed below:

1. *EMPTY*: This indicates the element will not have any content

As an example, look at the following DTD for an element, and the element usage in the XML based on the DTD declaration

DTD

```
<!ELEMENT format EMPTY>
```

XML

```
<format></format> or <format/>
```

2. ANY: This indicates that the element can have anything.

3. *Children Only*: This indicates that the element can only have child elements in the specified order. Look at the following DTD and equivalent XML.

DTD

```
<!ELEMENT account (accountNumber, accountBalance) >
```

XML

```
<account>
    <accountNumber>1234</accountNumber>
    <accountBalance>100.23</accountBalance>
</account>
```

4. *Text with mixed children*: This indicates that an element can have text as well as specified children in any order. See the following DTD and equivalent XML.

DTD

```
<!ELEMENT description (#PCDATA|b|code)* >
```

where asterisk(*) indicates that the elements in parenthesis can occur 0 or more times within *description*. Following lists the various symbols.

- + means 1 or more
- ? means 0 or 1
- * means 0 or more

XML

```
<description>
    This is a test <b> description </b>
    The child elements can be <code> in </code> in any <b>order</b>
</description>
```

Using the above declarations, following is the DTD for the customer data we defined before.

```
<!ELEMENT customer (firstName,lastName,age,ssn,address) >
<!ELEMENT firstName #PCDATA >
<!ELEMENT lastName #PCDATA >
<!ELEMENT age #PCDATA >
<!ELEMENT ssn #PCDATA >
<!ELEMENT address (addressLine1, city, state, country) >
<!ELEMENT addressLine1#PCDATA >
```

```
<!ELEMENT city #PCDATA >
<!ELEMENT state #PCDATA >
<!ELEMENT country #PCDATA >
```

The above DTD tells that the `customer` element should have the five child nodes namely `firstName`, `lastName`, `age`, `ssn`, `address` in the same order. It then defines the content types of every element. All the text elements are defined as `PCDATA` which stands for Parsed Character DATA. We'll see what parsing is in the next section. The `address` element in turn defines its child elements along with the order of the elements.

As we learned before, an XML element can also have attributes. Let's see how we can define the validation rules for attributes using DTD. Following is the syntax for attribute declarations in a DTD

```
<! ATTLIST element-name
      (attribute-name attribute-type default-declaration) *
>
```

Consider the following DTD definition for an element named `book`.

```
<!ELEMENT book (author, publisher) >
<! ATTLIST book
      isbn          CDATA      #REQUIRED
      pages         CDATA      #IMPLIED
>
```

The above declaration tells that `book` element can have two attributes namely `isbn` which is required and `pages` attribute which is optional. So, the xml will look as shown below:

XML

```
<book isbn="SD34324" pages="100">
  <author>James</author>
  <publisher>BPB</publisher>
</book>
```

These are the basic things you need to know to write a DTD for an XML document. Once we have the DTD in a file say `customer.dtd`, we finally need to link it with the XML file using the `DOCTYPE` element as shown below:

```
<!DOCTYPE customer SYSTEM "customer.dtd">
<?xml version="1.0" ?>
<customer>
  .....
</customer>
```

It's important that you have the DOCTYPE element before the XML prolog. Once the XML is linked with the DTD, our applications can start validating the XML document before processing it. Though DTD provides a simple way of validating an XML document, it has certain limitations listed below.

1. It can only validate the order of the elements. It cannot validate the list of valid values an element can have.
2. Less flexible. A DTD cannot be extended using inheritance.
3. No support for validating numeric and boolean data

To overcome the above limitations, a new validation scheme is created which is called as *XML Schema*. Let's see what this is and how we can use xml schema to validate xml documents in a better way.

XML Schema

Unlike a DTD, an XML Schema is itself an XML document with elements, attributes etc. XML Schemas overcame all the limitations of DTD and had now become the standard for validating XML documents. The good thing about XML Schema is that it is closely associated with Object Oriented data models. One of the major concerns with DTD is the lack of support of various data types. There is no way that using a DTD we can validate the type of data an element can have. This is where schema comes in real handy. It contains several built in primitive data types such as string, integer, float etc for validating the data.

XML Schema can also be used to build complex data types using simple data types. First, let's look at the important simple data types listed in the following table.

Table 12.1 XML Schema Data types

Data Type	Description
string	Used for text data
boolean	Used for boolean data (True/False)
float	Used for 32 bit decimal numbers
double	Used for 64 bit decimal numbers

Using the above data types, an element named score will be defined as shown below:

```
<xsd:element name="score" type="xsd:int"/>
```

Following are some of the examples using different data types:

```
<xsd:element name="firstName" type="xsd:string" />
<xsd:element name="expiration" type="xsd:date"/>
<xsd:element name="financialIndicator" type="xsd:boolean"/>
```

All the above defines data types for simple elements. However, using XML schema we can build complex data structures from simple data structures. This is where XML schema exhibits its true power. It allows us build complex data structures by defining the order of elements, valid values for different elements and so on.

Complex Data Structures using XML Schema

Let's first write a sample XML and then define the corresponding XML Schema that includes all the validation rules.

Sample XML

```
<?xml version="1.0" ?>

<rentalcompany name="Lake View Apartments">
    <type>Residential</type>
    <owner>Lake View Corporation Inc</owner>
    <address>
        <addressLine1>111 S St</addressLine1>
        <city>Lincoln</city>
        <state code="NE" />
        <zip>43081</zip>
        <country>USA</country>
    </address>
    <apartments>
        <apartment type="Studio">
            <aptnumber>234</aptnumber>
            <rooms>1</rooms>
            <rent>500</rent>
            <garageIndicator>N</garageIndicator>
            <garageNumber>N/A</garageNumber>
            <leasecontract>
                <leasingperiod>12 Months</leasingperiod>
                <lessee-name>John</lessee-name>
                <deposit>250</deposit>
            </leasecontract>
        </apartment>
    </apartments>
    <garages>
        <garage id="001" rate="50.00" />
        <garage id="002" rate="75.00" />
    </garages>
</rentalcompany>
```

Let's define the schema for the above XML. The XML defines several complex data elements namely `garage`, `garages`, `leasecontract`, `apartment` etc. Let's look at one by one.

`garage` : Following is the schema definition for this element.

```
<xsd:element name="garage">
    <xsd:complexType>
        <xsd:attribute name="id" type="xsd:string"/>
        <xsd:attribute name="rate" type="xsd:float"/>
    </xsd:complexType>
</xsd:element>
```

The `<xsd:element>` tag defines the name of the element as `garage`. The `<xsd:complexType>` element tells the schema that the element named `garage` is a complex element that can have both child elements as well as attributes. In our case, it only has attributes which are defined by the `<xsd:attribute>` elements.

`leasecontract`: Following is the schema definition for this element.

```
<xsd:element name="leasingperiod" type="xsd:int"/>
<xsd:element name="lesse-name" type="xsd:string"/>
<xsd:element name="deposit" type="xsd:float"/>

<xsd:element name="leasecontract">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="leasingperiod"/>
            <xsd:element ref="lesse-name"/>
            <xsd:element ref="deposit" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

This element first defined the simple elements namely `leasingperiod`, `deposit` etc. It then built the complex data type named `leasecontract` by referencing the simple elements. The `sequence` element tells the schema about the order of the elements in the XML document. The `minOccurs` attribute makes the element optional. The absence of this attribute means the element is required. Following similar lines, complete the definitions for other elements namely `address`, `apartment` etc. Let's now define definition for the root element `rentalcompany`.

`rentalcompany`: Following is the schema definition for this element.

```
<xsd:element name="rentalcompany">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="type"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

```
<xsd:element ref="owner"/>
<xsd:element ref="address"/>
<xsd:element ref="apartments"/>
    <xsd:element ref="garages" minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

As you can see from the above chunk, we defined all the child elements under rentalcompany element. Once all the elements are defined, we need to assemble them into one big chunk of document called as schema document as shown below:

```
<?xml version="1.0" ?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

<xsd:element name="rentalcompany">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="type"/>
      <xsd:element ref="owner"/>
      <xsd:element ref="address"/>
      <xsd:element ref="apartments"/>
      <xsd:element ref="garages" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
.....
.....
.....
</xsd:schema>
```

The important thing to note in the above schema document is the `<xsd:schema>` element which defined the namespace for all the elements in the schema. A namespace uses a prefixing strategy that associates element names with a Uniform Resource Identifier where the actual vocabulary for the elements will be defined. A namespace is defined using the attribute `xmlns:xsd`. This tells the schema to look for the vocabulary for all the elements prefixed with `xsd` at the resource specified by the URI. You are free to use which ever prefix you want. Namespaces come very handy especially when combining more than one XML document together resulting in namespace collisions.

I think with all this basic idea about DTD and XML Schema, we are now ready to look at the most important aspect of XML, XML Parsing.

Even if you don't understand the above XML schema and DTD, you'll still be fine. Usually, there will be several tools that automatically do it for you. You don't have to remember the syntax and all that. You can take my word. The only thing you need to

know is writing a well formed XML document, which I am sure you have absolutely no problem in writing it. Good.

XML Parsing

Representing data using XML is one side of the coin. Though XML provides infinite flexibility to represent data of any complexity, it's of no use if we cannot read the data back from XML file. This is where parsing an XML document comes into picture. Parsing an XML document is nothing but reading the data back from the document. This is the other side of XML. The application that parses an XML document is called an *XML parser*.

The purpose of an XML parser is to make some interfaces available to an application so that it can modify and read the contents of an XML document. The generation of these interfaces is based on two XML standards namely SAX and DOM.

SAX

SAX is abbreviated for Simple API for XML. SAX parsing is based on event model in which sequences of events are generated for each and every tag in the XML document. Based on the type of events, the application can take appropriate action. The various events a SAX based parser generates are:

1. Start of Document
2. Start of Tag
3. End Of Tag
4. End of Document etc.

SAX based parsing can be used only for reading the data from XML and not for modifying the contents. Moreover, SAX parsing is more efficient in situations where the XML document is huge and we only want to extract a small piece of data from it.

DOM

DOM stands for Document Object Model. In this model, an XML document is represented as a tree of nodes. A parser based on this model, can traverse the through the nodes to read the data or modify the data by removing the nodes. In DOM parsing, the entire XML must be loaded into the memory before reading or modifying the document. Because of this reason, DOM based parsing should be used only when the XML document is small enough not to cause any memory issues.

Any XML parser that is constructed will be based on either SAX or DOM model. There are several XML parsers available in the market for free of cost. Most notable ones are the parsers from Apache, BEA and Sun Microsystems. In this chapter, we use the parser from Apache called as *Xerces* parser to parse the XML documents.

Now that we know what XML parsing is, and the two parsing standards SAX and DOM, let's see how we can use an XML parser it to parse XML documents from Java applications. This is where we need to know something about JAXP.

JAXP

JAXP stands for Java API for XML Parsing. This API is created by Sun Microsystems as a universal way for parsing XML documents from Java. One important thing to understand with JAXP is that it provides classes and interfaces that have zero implementation. If this is the case, how can we use JAXP to parse the XML documents? Good question. JAXP uses the classes that are implemented by some other vendors to parse the documents. However, there is one limitation here. The vendors who implement the parsers must adhere to JAXP specification. This is like if B adheres to A's rules, then A can use B. Here B is the vendor XML parser and A is JAXP.

In this chapter, we will use sun's *xerces* parser which adheres to JAXP specification. This parser supports both SAX and DOM parsing. Sun distributes this parser in the form of a Jar file named *xercesimpl.jar*. To use this jar file, all we need to do is put it in the classpath. This is shown the environment setup at the beginning of the chapter.

Once the xerces parser is loaded in the classpath, we can use the JAXP API to write Java programs and parse XML documents using SAX and DOM.

In this chapter, we will see examples using both SAX as well as DOM.

SAX Parsing using JAXP

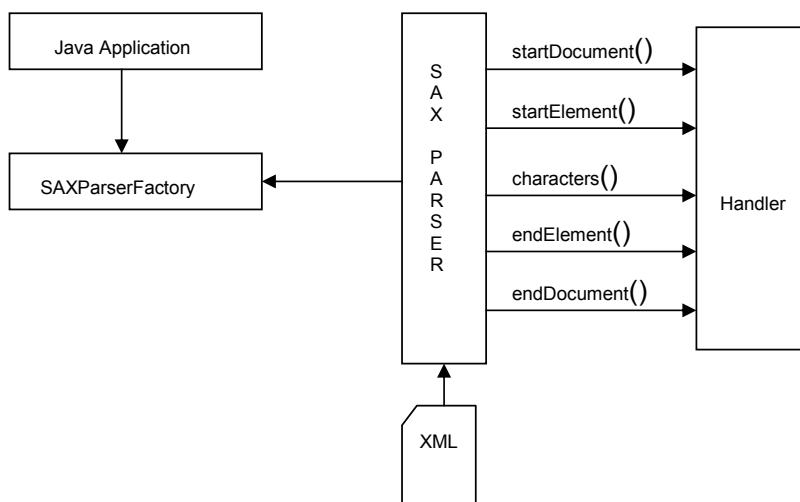
As said before, SAX based parsing is based on event model in which a sequence of events are generated for processing the tags. The five most important events are listed in the following table.

Table 12.2 SAX Events

Event	Description
startDocument	This event indicates the beginning of the XML document

startElement	This event indicates the beginning of a tag or a node
characters	This represents the data within the tag
endElement	This indicates the end of the tag
endDocument	This event indicates the end of the XML document

Following figure gives a complete idea of SAX Parsing.



From the above figure, we observe the following things:

1. Java application uses a SAXParserFactory to get a SAXParser
2. The SAXParser takes the XML document to parse, and begins generating the events and send them to a handler that handles the events.

Steps for parsing a document with SAX Parser

1. Write a handler class to handle events.
2. Write a class to initiate parsing with the following steps
 - a) Get an instance of SAXParserFactory class
 - b) Get a SAXParser from the factory class
 - c) Invoke the parse() method by passing the XML file and a reference to handler object.

Following the above steps, let's write an example to parse the following XML:

```
<?xml version="1.0"?>

<customer>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <age>20</age>
    <ssn>23324</ssn>
    <address>
        <addressline1>Apt 2222</addressline1>
        <city>Columbus</city>
        <state>OH</state>
        <country>USA</country>
    </address>
</customer>
```

As I said before, SAX parsing should be used in situations where we need to extract only few details from XML data. In this example, our goal is to retrieve the `firstName` and `SSN` of the customer. Take a look at the code in listing 12.1.

Listing 12.1 (`Customer.java`) A simple Java Bean for storing the data.

```
package xml;

public class Customer {

    private String name;
    private String ssn;

    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public void setSSN(String ssn) {
        this.ssn = ssn;
    }
    public String getSSN() {
        return this.ssn;
    }
}
```

Listing 12.1b (`MySAXHandler.java`) A simple SAX handler class.

```
package xml;

import org.xml.sax.AttributeList;
import org.xml.sax.HandlerBase;
import org.xml.sax.SAXException;

public class MySAXHandler extends HandlerBase {

    private Customer customer = new Customer();
    String data;
```

```
public void startDocument() throws SAXException {
    System.out.println(" Begin of Document");
}
public void startElement(String name, AttributeList attrs)
    throws SAXException {
}
public void characters(char buf[], int offset, int len)
    throws SAXException {
    data = new String(buf, offset, len);
}
public void endElement(String name) throws SAXException {
    if ("firstName".equals(name)) {
        customer.setName(data);
    }
    if ("ssn".equals(name)) {
        customer.setSSN(data);
    }
}
public void endDocument() throws SAXException {
    System.out.println(" End of Document");
}
public Customer getCustomer() {
    return customer;
}
```

Listing 12.1c (`SAXParserDemo.java`) A simple SAX parser class.

```
package xml;

import org.xml.sax.*;
import javax.xml.parsers.*;

public class SAXParserDemo {

    public static void main(String args[]) throws Exception {
        // Create the SAX Factory
        SAXParserFactory factory = SAXParserFactory.newInstance();

        // Get the sax parser
        SAXParser parser = factory.newSAXParser();

        // Create the handler object
        MySAXHandler handler = new MySAXHandler();

        // Parse the XML by passing the handler object
        parser.parse("customer.xml", handler);

        // Once the parsing is done, get the customer bean
```

```
Customer customer = handler.getCustomer();

// Print the data in the bean
System.out.println(customer.getName());
System.out.println(customer.getSSN());
}
}
```

The above code comprises of 3 classes. A `Customer` bean for storing the data (`firstName` and `SSN`), a handler class `MySAXHandler` that handles the events and finally the `SAXParserDemo` that initiates the parsing. The bean class is a simple class that defines the getters and setters for storing the data that is retrieved from the xml document. To write a handler class, we need to do the following 2 things.

1. The class must extend the `HandlerBase` class. This is one of the JAXP built-in library classes.
2. The handler class should overwite the methods namely `startDocument()`, `startElement()` etc. These methods are known as **callback** methods that get invoked automatically when the parsing begins.

The only methods we use in this example are the `characters()` and `endElement()`. Whenever the parser comes across a tag, it first calls the `startElement()` method, followed by `characters()` and then followed by the `endElement()` method. The `startElement()` and `endElement()` methods have a `name` parameter which represents the name of the tag that is currently parsed. So what we will do is, check whether the name of the tag is either `SSN` or `firstName` and populate the `customer` bean with the `data` read in the `characters()` method. When the end of the document is reached, the `customer` bean will have both the `SSN` and the `name` populated.

The `SAXParserDemo` class simply follows the steps outlined before, and initiates the parsing by invoking the `parse()` method. This method takes two parameters. One is the name of the XML file, and the other is the object of the handler class. When all the parsing is completed it retrieves the `customer` bean and displays the data.

Compile and execute the programs using the following commands:

```
C:/>JavaTraining>chapter12>javac xml\*.java
C:/>JavaTraining>chapter12>java xml.SAXParserDemo
```

If all goes well, the output of the above program will be

```
John
23324
```

This is how we do SAX parsing. If you need any other data from XML, you will add getters and setters in the `Customer` bean, and then add another `if` block in the `endElement()` method to populate the `beans` property. Most of the real world applications implement SAX parsing just like the above example. For the first time it might seem confusing, but go through it 2-3 times and you'll get it. You can use the above code as a reference to write any Java program that needs to parse XML using SAX.

The main disadvantage with SAX is that, if the XML has repeating elements, then extracting the data becomes really complicated. For instance, look at the following XML:

```
<customer>
    <name>John</name>
    <ssn>12345</ssn>
</customer>
<customer>
    <name>Sam</name>
    <ssn>4543</ssn>
</customer>
```

The above XML represents data for two customers. Here the `customer` tag is repeating twice. If we need to extract this customer data using SAX parser, the application logic will get more complicated due to duplicate elements. In such situations, DOM parsing offers more flexibility. So, let's see what it is.

DOM Parsing using JAXP

In DOM Parsing, the entire XML is loaded in the memory as a tree of nodes. We call this entire tree as the `Document`. Once the XML is loaded, we can traverse through the entire tree and get the information about every node. In DOM, there are different types of nodes corresponding to the component types of an XML document. These are:

1. Element Node
2. Attribute Node
3. CDATA Node etc.

The most important interfaces we use in DOM parsing are the `Document` and `Node`. The basic process to implement DOM parsing is shown below:

1. Get an instance of `DocumentBuilderFactory` class.
2. Create a `DocumentBuilder` object using the factory class.
3. Invoke the `parse()` method on the builder object by passing the XML file. This method then returns a `Document` object.

4. Use the Document object to process the XML.

Let's write an example based on the above steps. In this example, we will retrieve customer information from the following XML, and store it in the database. This example uses the JDBC API we discussed in the previous chapter.

```
<?xml version="1.0"?>

<customers>
    <customer>
        <firstName>John</firstName>
        <middleName>John</middleName>
        <lastName>Smith</lastName>
        <age>20</age>
        <ssn>23324</ssn>
        <city>Columbus</city>
        <state>OH</state>
        <country>USA</country>
    </customer>
    <customer>
        <firstName>Rob</firstName>
        <middleName>D</middleName>
        <lastName>Smith</lastName>
        <age>30</age>
        <ssn>56767</ssn>
        <city>Dayton</city>
        <state>OH</state>
        <country>USA</country>
    </customer>
</customers>
```

Let's first write the Java Bean for storing the above data. See the code in listing 12.2a.

Listing 12.2a (`CustomerInfo.java`) A simple POJO.

```
package xml;

public class CustomerInfo {

    private String firstName;
    private String middleName;
    private String lastName;
    private int age;
    private int ssn;
    private String city;
    private String state;
    private String country;

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getCity() {
```

```
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    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 String getMiddleName() {
        return middleName;
    }
    public void setMiddleName(String middleName) {
        this.middleName = middleName;
    }
    public int getSSN() {
        return ssn;
    }
    public void setSSN(int ssn) {
        this.ssn = ssn;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
}
```

As you can see the above code, we simply defined the bean with getters and setters for every element in the customer element. Nothing special. Now, look at the parser class shown in listing 12.2b.

Listing 12.2b (DOMParserDemo.java) A simple DOM Parser.

```
package xml;

import java.util.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.parsers.*;
import java.sql.*;
```

```
public class DOMParserDemo {  
  
    public static void main(String args[]) throws Exception {  
  
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  
        DocumentBuilder db = factory.newDocumentBuilder();  
        Document doc = db.parse("customerdata.xml");  
  
        // Get all the customer nodes  
  
        NodeList list = doc.getElementsByTagName("customer");  
  
        // The list will have 2 customer nodes.  
  
        List customers = new ArrayList();  
  
        for (int i = 0; i < list.getLength(); i++) {  
  
            Node node = list.item(i);  
            Element ele = (Element) node;  
  
            String fn = getData(ele, "firstName");  
            String mn = getData(ele, "middleName");  
            String ln = getData(ele, "lastName");  
            String age = getData(ele, "age");  
            String ssn = getData(ele, "ssn");  
            String city = getData(ele, "city");  
            String state = getData(ele, "state");  
            String country = getData(ele, "country");  
  
            // Populate the customer bean and add it to the list  
            CustomerInfo info = new CustomerInfo();  
  
            info.setFirstName(fn);  
            info.setMiddleName(mn);  
            info.setLastName(ln);  
            info.setAge(new Integer(age).intValue());  
            info.setSsn(new Integer(ssn).intValue());  
            info.setCity(city);  
            info.setState(state);  
            info.setCountry(country);  
  
            // Add to the list  
            customers.add(info);  
        }  
  
        saveToDatabase(customers);  
  
        System.out.println(" XML data saved in the database");  
    }  
  
    private static String getData(Element element, String name) {  
  
        String value = null;  
  
        NodeList nl = element.getElementsByTagName(name);  
        if (nl != null && nl.getLength() > 0) {  
  
            Element el = (Element) nl.item(0);  

```

```
        value = el.getFirstChild().getNodeValue();
    }
    return value;
}

private static void saveToDatabase(List customers) {
    Connection con = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        con = DriverManager.getConnection("");
        String sql = "INSERT INTO CUSTOMERS VALUES(?,?,?,?,?,?,?,?,?,?)";
        PreparedStatement stmt = con.prepareStatement(sql);
        for (int i = 0; i < customers.size(); i++) {
            CustomerInfo customer = (CustomerInfo) customers.get(i);
            stmt.setString(1, customer.getFirstName());
            stmt.setString(2, customer.getMiddleName());
            stmt.setString(3, customer.getLastName());
            stmt.setInt(4, customer.getAge());
            stmt.setInt(5, customer.getSsn());
            stmt.setString(6, customer.getCity());
            stmt.setString(7, customer.getState());
            stmt.setString(8, customer.getCountry());
            stmt.executeUpdate();
        }
    } catch (Exception e) {
        System.out.println("Exception while saving to database");
        e.printStackTrace();
    } finally {
        try {
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

I know the above program is really lengthy, but you know what, it is very simple. Following is what we did:

1. Created a bean named `CustomerInfo` to store the customers data.
2. Created a `DOMParserDemo` class to parse the XML file. This class does several things:

- a) Using the DOM API, it gets a `Document` object. It will then retrieve the two **customer** nodes.
- b) It will iterate over the node list using `for` loop, get the customers information using the `getData()` method, populate the customer bean and finally store the bean in an `ArrayList`.
- c) It then passes the list of customers to `saveToDatabase()` method, which uses the JDBC API, and inserts the customers beans into the database.

This is one of the widely used in real world applications. You can use this as a reference to parse an XML document using DOM. Save both the programs and compile them as shown below:

```
c:/>JavaTraining>chapter12>javac xml/*.java  
c:/>JavaTraining>chapter12>java xml.DOMParserDemo
```

If all goes well, you'll see the following output, and see two records inserted into the **Customers** table in the database.

```
XML Data saved in the Database
```

To verify, open DBVisualizer tool, and check the contents of the **customers** table.

This is all you need to know about XML and XML parsing. Most of the real world applications use either SAX parsing or DOM parsing based on the application. Even if you didn't thoroughly understood the above two examples, you can use them as references. When you work in a company, no one will ever ask you to write code with a closed book. You are free to use references available online or some other place and parse the XML documents as you wish.

From interview standpoint, all you need to know is the difference between SAX parsing and DOM parsing. That's it. Read the following differences between the two, and you are good to go.

Difference between SAX and DOM

SAX

- ✓ SAX is based on event model.
- ✓ SAX will never load the XML into memory.
- ✓ SAX generates various events sequentially like `startDocument`, `startElement`, `characters`, `endElement`, `endDocument`.
- ✓ SAX uses a handler to handle the events.

- ✓ SAX parsing is used for reading XML documents only and cannot be used to modify its contents.
- ✓ SAX is used for reading small portion of information from large XML documents.

DOM

- ✓ DOM is tree based.
- ✓ DOM will load the entire XML into memory before parsing.
- ✓ DOM is used to read and modify XML data.
- ✓ DOM is usually used with small XML documents.

I am sure this chapter may have helped you to know some basics about XML and its intricacies. So, let's summarize what we learned in this chapter.

Summary

- ✓ XML is an industry standard for representing complex data.
- ✓ XML is 100% language and platform independent.
- ✓ Any application can use XML to exchange data.
- ✓ An XML document is built using tags.
- ✓ An XML document is said to be well-formed when every opened tag has a closing tag.
- ✓ XML parser is used to parse XML documents.
- ✓ SAX and DOM are two standards for parsing XML documents.
- ✓ JAXP API is used by Java applications for parsing XML documents.
- ✓ JAXP is a specification. It uses parser implementations provided by third party vendors to parse documents. Xerces parser from Apache is one of JAXP compliant XML parser that supports both SAX and DOM parsing.
- ✓ SAX parsing is based on event driven model and DOM is tree based.

Interview Questions

Question: What is an XML parser?

Answer: A parser is an application used for reading and modifying the contents of an XML file.

Question: What are two XML parsing standards?

Answer: SAX and DOM

Question: What is the difference between SAX and DOM?

Answer: Refer to the last section.

Question: When do you use SAX and DOM?

Answer: We use SAX when a small portion of information is to be retrieved from a large XML document. We use DOM when we want to modify or read the entire content of the XML file. DOM is normally used with smaller XML files.

Question: Which API is used for parsing XML documents?

Answer: Java API for XML Processing (JAXP)

Question: Which is the popular JAXP complaint XML parser ?

Answer: xerces parser from Apache Software foundation.

This completes all the basic XML that we need to know to get on with J2EE. However, if you want to know more about XML, please refer to the books and resources that teach XML and XML related technologies in detail. Let's continue with J2EE for now.

Chapter 13

Servlet Programming

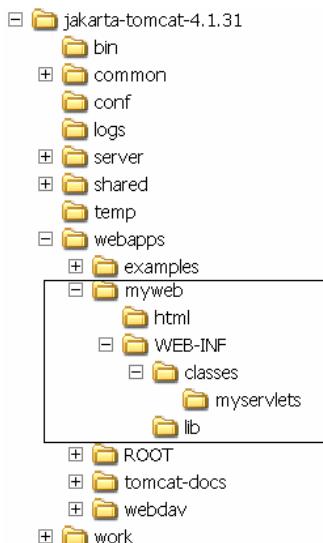
This chapter introduces you to the world of Web Applications using Java. Since Web Applications form the basis for enterprise internet applications, starting with this chapter, the next few chapters give you all the details about Web application development using J2EE and open-source technologies. This first chapter introduces you to basic web application development using Servlet technology and also gives you an idea about Web Containers.

Chapter Goals

- ✓ Understand the evolution of Web Applications
- ✓ Understand the basics of HTTP
- ✓ Understand the notion of Web Containers
- ✓ Servlet API
- ✓ Structure of J2EE Web Applications
- ✓ Building Web Applications using Servlet API
- ✓ Understand Web deployment descriptor

Environment Setup

1. Install Tomcat Server following the instructions at the end of the book.
2. Create the directory structure shown in the rectangle in Tomcat installation directory



3. Create “env.bat” file with the following contents:

```
set CLASSPATH=.;  
C:\JavaTraining\j2eelib\j2ee.jar;C:\JavaTraining\j2eelib\  
mysql-connector-java-5.0.3-bin.jar;C:\JavaTraining\j2eelib  
\xercesImpl.jar
```

and copy it into the following directory

C:/jakarta-tomcat-4.1.31/webapps/**myweb**/WEB-INF/classes/env.bat

4. Save the following into a file named **web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web  
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">  
  
<web-app>  
    <display-name>myweb</display-name>  
  
    <!-- Servlet Definitions goes here -->  
  
    <!-- Servlet Mappings goes here -->  
  
</web-app>
```

Save the file in the WEB-INF directory as shown below:

C:/jakarta-tomcat-4.1.31/webapps/myweb/WEB-INF/web.xml

5. Add the following variable to the system environment variables.

CATALINA_HOME = c:\jakarta-tomcat-4.1.31

This completes the setup for running the web applications:

Starting the Server

To start the server, click on the “**startup.bat**” file in the following directory.

c:\jakarta-tomcat-4.1.31\bin

A command window opens and keeps listening.

Introduction

Several years ago, client-server applications have become very popular for building enterprise applications. In this model, client application is usually installed on the client's personal computer, which then sends requests to the server application via network. In such a model, client applications usually contained most of the presentation and application logic and the interaction with the server application is done through a GUI. Such application clients are referred to as *fat* clients since the application logic is also included in the client application. Any change in the application logic requires re-installation of the client application on all the computers.

With the advent of internet, application clients are completely replaced with web clients. In this model, both the application logic and presentation logic are physically separated from the clients PC and moved to the server side. The Web client delegates all the user interactions to the server side application logic which then processes the requests and uses the presentation tier components to send the response. Since the application logic is no longer present on the client PC, web clients are referred to as **thin** clients. Since the presentation logic is also moved to the server side, the layout of the user interface can also be controlled from the server side. A web client in this case is nothing but a browser application like Internet Explorer, FireFox etc. Applications that use web clients for user interaction are called as Web Applications.

Web clients to interact with server side applications, use the following:

1. A general purpose browser like Internet Explorer to send requests and display the response.
2. HTML markup for defining the user interfaces for interaction with the server
3. HTTP protocol to send and receive requests and responses.

The way the web applications work is,

1. Web client like IE sends a request using HTTP protocol.
2. The server side program takes the HTTP request, processes it and sends a HTTP response back to the web client. The response includes body content in HTML format.
3. Web client then reads the HTTP response, formats the HTML and displays it to the user.

Since HTTP plays an important role in web applications, let's first know some basics about it and then look at the server side details.

HTTP

HTTP stands for Hyper Text Transfer Protocol. Following are some of the important properties of HTTP:

1. It is a *stateless* protocol, meaning that every HTTP request is independent of each other.
2. It can send *data* in the request. The server side program reads the data, processes it and sends the response back. This is the most important feature of HTTP, the ability to send data to server side program.

Based on how the data is sent in the request, HTTP requests are categorized into several types, but the most important and widely used are the GET requests and POST requests.

A typical HTTP request is identified by something called URI (Uniform Resource Identifier) as shown below:

```
http:// <host name>:<port number>/<request details>
```

GET Request

GET request is the simplest of all the requests. This type of request is normally used for accessing server-side static resources such as HTML files, images etc. This doesn't mean that GET requests cannot be used for retrieving dynamic resources. To retrieve dynamic resources, some data must be sent in the request and GET request can send the data by appending it in the URI itself as shown below:

```
http://somedomain.com?uid=John&role=admin
```

In the above URI, the HTTP request sends two pieces of data in the form of name value pairs. Each name-value pair must be separated by '&' symbol. The server side program will then read the data, process it, and send a dynamic response.

Though GET request is simplest of all, it has some limitations:

1. All the data in the request is appended to the URI itself making the data visible to every one. When secure information need to be sent, GET request is not the solution.
2. GET requests can only send text data and not the binary data. Therefore in situations where you need to upload image files to server, GET request cannot be used.

These limitations are addressed by the HTTP POST request. Let's see what this is.

POST Request

POST request is normally used for accessing dynamic resources. POST requests are used when we need to send large amounts of data to the server. Moreover, the data in the POST request is hidden behind the scenes therefore making data transmittal more secure. In most of the real world applications, all the HTTP requests are sent as POST requests.

Now that we know the basics of HTTP, let's move on to the server side of the web applications.

Server Side of the Web Application

The server- side is the heart of any web application as it comprises of the following:

- ✓ Static resources like HTML files, XML files, Images etc
- ✓ Server programs for processing the HTTP requests
- ✓ A runtime that executes the server side programs
- ✓ A deployment tool for deploying the programs in the server

In order to meet the above requirements, J2EE offers the following:

- ✓ *Servlet and JSP technology* to build server side programs and
- ✓ *Web Container* for hosting the server side programs.
- ✓ *Deployment descriptor* which is an XML file used to configure the web application.

Let's see each of them one by one.

Web Container

A Web container is the heart of Java based web application which is a sophisticated program that hosts the server side programs like Servlets. Once the Servlet programs are deployed in this container, the container is ready to receive HTTP requests and delegate them to the programs that run inside the container. These programs then process the requests and generate the responses. Also a single web container can host several web applications. Look at the following figure that shows how a typical web container looks like:

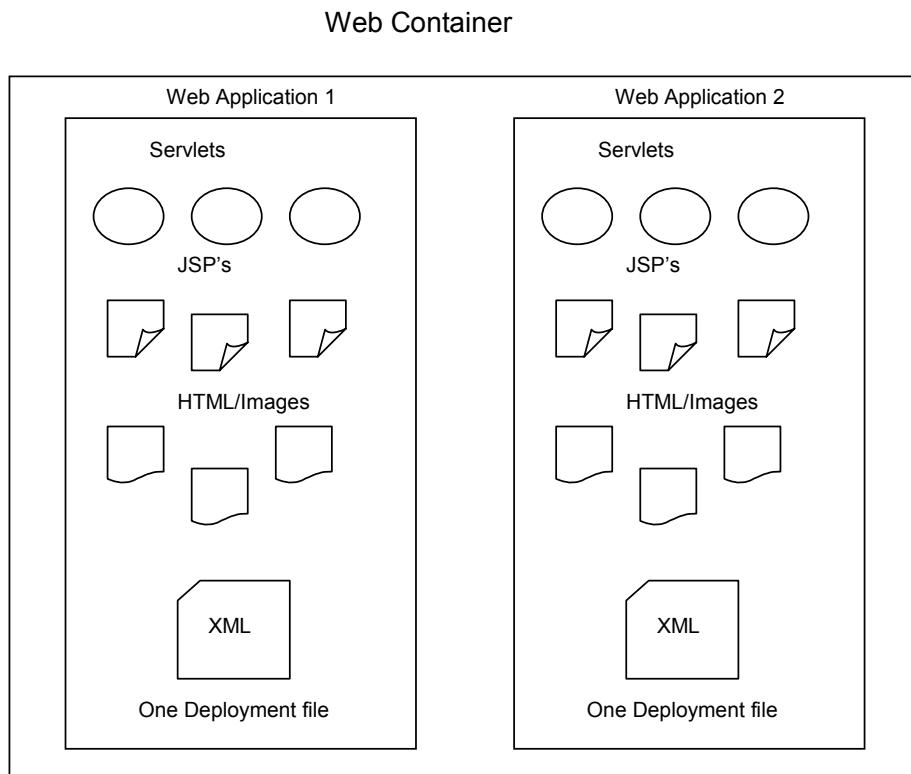


Fig 13.1 A typical web container

There are several J2EE Web Containers available in the market. One of the most notable one is the Apache's Tomcat container which is available for free of cost. Sweet! This is what we will install for running web applications. Installing Tomcat Web container is one of the steps in the environment setup page at the beginning of the chapter. Having just the web container is not sufficient. We need to have a programming model to build server side programs that run within this container.

J2EE supports two key technologies to build server side components in Java. These are Servlet and JSP technologies. In this chapter, we will use Servlet technology to build web applications.

I know you might be wondering about the need for several technologies to build web applications. Don't worry; you'll get the answer by yourself as we move on.

Note: At this point, since we know what a web container is, please install Tomcat Web container.

Structure of a Web Application

For the web container to run the web application, the web application must follow a standard directory structure within the container. Let's say we want to build a web application named **myweb**. The directory structure for this application must be as shown below:

```
webapps
    myweb
        WEB-INF
            web.xml
            classes
                myservlets
                    Servlet1.class
                    Servlet2.class
                    .
                    .
                    .
            lib
            index.html
            ....
```

where **webapps** is a standard directory within the web container (Tomcat installation directory). It is from this directory, you need to start creating the directory structure.

Following table lists the purpose of each directory:

Table 13.1 Web application directories

Directory	Description
myweb	This directory represents the entire web application
WEB-INF	This directory contains the configuration file like <code>web.xml</code>
classes	This directory must contain all the Java class files in the web application

All the static resources namely html files, image files can be stored directly under **myweb** directory. Creating this directory structure is one of the steps in the environment configuration section at the beginning of the chapter.

With the above directory structure, all the components within **myweb** application should be accessed with the URL starting with:

`http://localhost:8080/myweb/`

If you want to create another web application like **mywebproject**, you must again create the same directory structure with **mywebproject** as the root directory and should access the components using the URL starting with:

`http://localhost:8080/mywebproject/`

At this point please make sure you created the directory structure as outlined in environment configuration section.

Servlet Technology

Servlet technology is a standard J2EE technology used for building dynamic web applications in Java. Using Servlet technology is again nothing but using the standard classes and interfaces that it comes with. These classes and interfaces form what we call as Servlet API.

Definition of Servlet

A server side Java program that uses the above API is called as **Servlet**. In simple terms, a Servlet is a server side Java program that processes HTTP requests and generates HTTP response.

Servlet API

Though there are several classes and interfaces in this API, we are interested in the most important ones shown in Table 13.2.

Table 13.2 Servlet API

Class/Interface	Description
HttpServlet	The base class that represents a HTTP Servlet
HttpServletRequest	The class that encapsulates all the HTTP request details
HttpServletResponse	The class used for sending HTTP response back to the browser
ServletConfig	Class used for dynamically initializing the servlet
HttpSession	Class used for session management
RequestDispatcher	Class used by the servlet to dispatch the request to various resources.

At this point, we have the following two important things to build the web applications in Java:

1. Servlet Technology to build the server-side programs.
2. A Web Container like Tomcat to run the server-side programs.

Are the above two things good enough to build applications? The answer is big No. We are missing the most important component here. This is nothing but the deployment descriptor. Once we have a Web container and several Servlets, it is the deployment descriptor that tells the web container what servlets to run. So, this is the key component of any web application. Let's see what this is.

Deployment Descriptor

A deployment descriptor is a standard XML file named **web.xml** that is used by the web container to run the web applications. Every web application will have one and only one deployment descriptor (web.xml file). This file defines the following important information pertaining to a Servlet:

1. Servlet name and Servlet class
2. The URL mapping used to access the Servlet

Let's assume we wrote a servlet named `FormProcessingServlet` in a package named `myservlets`. The definition for this servlet in the `web.xml` will be as shown below:

```
<servlet>
    <servlet-name>FormProcessingServlet</servlet-name>
    <servlet-class>myservlets.FormProcessingServlet</servlet-class>
</servlet>
```

The servlet name can be any arbitrary name, but it's a good practice to have the class name as the servlet name. However, the servlet class tag must represent the fully qualified name of the servlet which includes the package name as shown above. This completes Step 1.

The next thing we need to define is the URL mapping which identifies how the servlet is accessed from the browser. For the above servlet, the url mapping will be as shown below:

```
<servlet-mapping>
    <servlet-name>FormProcessingServlet</servlet-name>
```

```
<url-pattern>/FormProcessingServlet</url-pattern>
</servlet-mapping>
```

If you noticed carefully, the servlet name in both the XML snippets is the same. This is how we bind the url mapping with a servlet. The url pattern defines how the servlet will be accessed. With the above mapping, the FormProcessingServlet should be accessed with the following URL:

`http://localhost:8080/myweb/FormProcessingServlet`

The web container then delegates the request to `myservlets.FormProcessingServlet` class to process the request.

Steps for Writing a Servlet

Writing a servlet is very simple. Trust me. You just have to follow a standard process as shown below:

1. Create a class that extends `HttpServlet`
2. Define 3 methods namely `init()`, `doGet()` and `doPost()`.

That's it. Following is how a typical servlet looks like.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        // This method is called before the following methods are called
        //and gets called only ONCE. This is like a
        // constructor. We do all the initialization here.
    }

    public void doGet(HTTSPortletRequest req, HTTSPortletResponse res)
    throws ServletException, IOException {
        // This method gets called automatically for GET requests. This
        //method can only process GET
    }

    public void doPost(HTTSPortletRequest req, HTTSPortletResponse res)
    throws ServletException, IOException {
        // This method gets called automatically for POST requests. This
        //method can only process POST
    }
}
```

If you look at the above servlet, it does the following things

1. Imports all the servlet classes
2. The class extends HttpServlet
3. Defined 3 methods `init()`, `doGet()` and `doPost()`

These three methods are the standard methods for any servlet. These are called as callback methods that the web container invokes automatically when a HTTP request comes to this servlet. For all the GET requests, the web container invokes the `doGet()` method, and for POST requests it invokes the `doPost()` method.

LifeCycle of a Servlet

The life cycle of a servlet represents how the web container uses the servlet to process the requests. Following is what a web container does with a servlet:

1. Loads the Servlet
2. Instantiates the Servlet
3. Initializes the servlet by executing the `init()` method.
4. Invokes the `doGet()` or `doPost()` methods to process the requests.
5. Repeats Step 4 until all the requests are processed
6. Destroy the servlet

Before we start writing examples, remember the following three important points:

1. Any request sent by typing the URL in the browser is always a GET request
2. When a hyperlink is clicked, it's always a GET request
3. When a html form is submitted it can either be GET or POST

Assuming that you completed all the steps in the environment setup section, let's begin writing servlet examples.

The first example we are going to write simply displays a greeting in a web browser. Before we understand the details about how it works, take a look at the code in listing 13.1.

Listing 13.1 (`GreetingServlet.java`) A simple servlet that echos a greeting

```
package myservlets;  
  
import javax.servlet.*;  
import javax.servlet.http.*;
```

```
import java.io.*;

public class GreetingServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        res.setContentType("text/html");

        PrintWriter pw = res.getWriter();

        // Send the message
        pw.println("<h1>Welcome to Servlets </h1>");

    }
}
```

This is a simple servlet which defined just the `doGet()` method since we will only send a request from the browser which is a GET request. This method takes two arguments namely `HttpServletRequest` and `HttpServletResponse`. The former is used to read the data from the request and the later is used to send the response back to the browser. In this example we only use the response object to send a greeting message to the browser. To send a response, we need to do three things as outlined below:

1. Define the type of the response data. This is done using the following statement:

```
res.setContentType("text/html");
```

The above line specifies that we are sending HTML response back.

2. Open a stream/channel to the browser to send the response. This is done using the following statement:

```
PrintWriter pw = res.getWriter();
```

3. Send the html message to the browser. To send response message to the browser, we use the `println()` method as shown below:

```
pw.println("<h1>Welcome to Servlets </h1>");
```

The first two steps are common to any servlet. Let's now run the servlet.

Steps to run the Servlet

1. Save the servlet program in the following directory as

/myweb/WEB-INF/classes/myservlets/GreetingServlet

2. Compile the servlet as shown below:

```
C:/>jakarta-tomcat-4.1.31>webapps>myweb>WEB-INF>classes>javac  
myservlet*.java
```

3. Update and save the web.xml file in the WEB-INF directory as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web  
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">  
  
<web-app>  
  
    <display-name>myweb</display-name>  
  
    <!-- Servlet Definitions -->  
  
    <servlet>  
        <servlet-name>GreetingServlet</servlet-name>  
        <servlet-class>myservlets.GreetingServlet</servlet-class>  
    </servlet>  
  
    <!-- Servlet Mappings -->  
  
    <servlet-mapping>  
        <servlet-name>GreetingServlet</servlet-name>  
        <url-pattern>/GreetingServlet</url-pattern>  
    </servlet-mapping>  
  
</web-app>
```

4. Start the server by clicking the startup.bat in the bin directory.
5. Type the following URL in the browser

<http://localhost:8080/myweb/GreetingServlet>

The above URL displays the following.



See how simple it is to write a servlet. All we did is followed a simple process. The program is very simple, right?

The next example demonstrates servlet initialization. Our servlet reads the initialization parameters defined in the web.xml, and displays them in the browser. The two parameters are driver and url for the MySQL database.

Servlet Initialization

Initializing a servlet is one of the most common practices. Servlet initialization is done in the `init()` method of the servlet. Defining initialization parameters for servlets in `web.xml` is a good practice as it eliminates hard coding in the servlet. The web container while loading the servlet, invokes the `init()` method of the servlet as part of the life cycle and passes the parameters defined in the `web.xml` to it. The other advantage with this is, in the future if the initialization values need to be changed, you only have to change the `web.xml` without having to recompile the entire web application.

Initialization parameters to a servlet are defined in the `web.xml` using the `<init-param>` element as shown below:

```
<init-param>
    <param-name>driver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
</init-param>
```

With the above definition, a servlet can access the parameter in the `init()` method as shown below:

```
public void init(ServletConfig config) {
```

```
    String drivername = config.getInitParameter("driver");
}
```

As you can see from the above code, the `ServletConfig` class defines a method named `getInitParameter()` that takes the name of the initialization parameter defined in the `web.xml` for that servlet, and returns the value. Therefore `drivername` will be initialized with `com.mysql.jdbc.Driver`. To verify this, take a look at the code in listing 13.2.

Listing 13.2 (`InitServlet.java`) Servlet reading initialization parameters.

```
package myservlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class InitServlet extends HttpServlet {

    String drivername;
    String databaseURL;

    public void init(ServletConfig config) throws ServletException {
        drivername = config.getInitParameter("driver");
        databaseURL = config.getInitParameter("URL");
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");

        PrintWriter pw = res.getWriter();

        // Send the response
        pw.println("<h1>Driver Name is : </h1><br/>" + drivername);
        pw.println("<h1>DB URL is : </h1>" + databaseURL);
    }
}
```

If you look at the above code, the `init()` method reads two parameters namely `drivername` and `databaseURL` using the `ServletConfig` object. The parameter keys namely **driver** and **URL** will be defined in the `web.xml` in the servlet definition as shown below:

```
<servlet>
    <servlet-name>InitServlet</servlet-name>
    <servlet-class>myServlets.InitServlet</servlet-class>
```

```
<init-param>
    <param-name>driver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
</init-param>
<init-param>
    <param-name>URL</param-name>
    <param-value>jdbc:mysql://localhost:3306/MyDB</param-value>
</init-param>

</servlet>
```

Once the `init()` method is completed, the container will invoke the `doGet()` method which displays the values in the browser.

Steps to run the Servlet

1. Save the servlet program in the following directory as

/myweb/WEB-INF/classes/myservlets/InitServlet

2. Compile the servlet as shown below:

```
C:/>jakarta-tomcat-4.1.31>webapps>myweb>WEB-INF>classes>javac
myservlet\*.java
```

3. Add the following snippet in the web.xml file

```
<!--  Servlet Definitions  -- >

<servlet>
    <servlet-name>InitServlet</servlet-name>
    <servlet-class>myServlets.InitServlet</servlet-class>

    <init-param>
        <param-name>driver</param-name>
        <param-value>com.mysql.jdbc.Driver</param-value>
    </init-param>
    <init-param>
        <param-name>URL</param-name>
        <param-value>jdbc:mysql://localhost:3306/MyDB</param-value>
    </init-param>

</servlet>

<!--  Servlet Mappings  -- >

<servlet-mapping>
    <servlet-name>InitServlet</servlet-name>
    <url-pattern>/InitServlet</url-pattern>
</servlet-mapping>
```

4. Start the server by clicking the `startup.bat` in the `bin` directory.

5. Type the following URL in the browser

`http://localhost:8080/myweb/InitServlet`

The above URL displays the following



Reading HTML Form Data

HTML form processing is one of the most common things that any web application does. I think you might have noticed on various websites where you fill in html form with all the information and submit it for processing. The application then responds with a confirmation message something like “information is successfully processed”. Registration pages, Email composing etc are some of the examples.

Following are the steps for processing a HTML form using a Servlet

1. User fills the data in the html page and submits it.
2. The form data will then be sent to Servlet
3. Servlet reads the form data, processes it and send a confirmation.

The form data will be sent in the HTTP request object as name value pairs as shown below:

`http://localhost:8080/myweb/SomeServlet?name=John&age=20`

The servlet then reads the above request data using the `HttpServletRequest` object as shown below:

```
String fn = request.getParameter("name");
String age = request.getParameter("age");
```

`fn` will now have John and `age` will have 20. The servlet can do whatever it wants with the data, and send a confirmation message back. Let's do a quick example shown in listing 13.3.

Listing 13.3a (`CustomerForm.html`) Simple HTML form.

```
<html>
<head>
<title>Customer Form</title>
</head>
<body>
<h3>Please fill in the following details and submit it</h3>
<br/>

<form action="/myweb/FormProcessingServlet" method="GET">
<table>
    <tr>
        <td>First Name</td>
        <td><input type="text" name="firstName" size=20 />
    </tr>
    <tr>
        <td>Middle Name</td>
        <td><input type="text" name="middleName" size=5 />
    </tr>
    <tr>
        <td>Last Name</td>
        <td><input type="text" name="lastName" size=20 />
    </tr>
    <tr>
        <td>Age</td>
        <td><input type="text" name="age" size=20 />
    </tr>
    <tr>
        <td>SSN</td>
        <td><input type="text" name="ssn" size=20 />
    </tr>
    <tr>
        <td>City</td>
        <td><input type="text" name="city" size=20 />
    </tr>
    <tr>
        <td>State</td>
        <td><input type="text" name="state" size=20 />
    </tr>
    <tr>
        <td>Country</td>
        <td><input type="text" name="country" size=20 />
    </tr>
<tr>
```

```
<td></td>
<td><input type="submit" name="Submit" size=20 />
</tr>
</table>
</form>
</body>
</html>
```

The above code represents a simple HTML form with the customer fields. The data in this form will be submitted to a servlet using the following form element:

```
<form action="/myweb/FormProcessingServlet" method="GET">
```

The above form element sends the form data as a GET request to the FormProcessingServlet. Now, let's see how to write this servlet. Look at the code in listing 13.3b.

Listing 13.3b (`FormProcessingServlet.java`) Servlet processing a form.

```
package myservlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class FormProcessingServlet extends HttpServlet {

    String drivername;
    String databaseURL;

    public void init(ServletConfig config) throws ServletException {
        // This method is called before the following methods are called
        // and gets called only ONCE. This is like a
        // constructor. We do all the initialization here.

        drivername = config.getInitParameter("driver");
        databaseURL = config.getInitParameter("URL");
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");

        PrintWriter pw = res.getWriter();

        // Read the form data

        String responseMsg = "";

        String fName = req.getParameter("firstName");
        String mName = req.getParameter("middleName");
        String lName = req.getParameter("lastName");
```

```
String age = req.getParameter("age");
String ssn = req.getParameter("ssn");
String city = req.getParameter("city");
String state = req.getParameter("state");
String country = req.getParameter("country");

// Remember in our database, age and ssn are integers. So lets
//convert the strings to int and then store it to database
int iage = 0;
int issn = 0;
try {

    iage = Integer.parseInt(age);
    issn = Integer.parseInt(ssn);

    // Call the method that inserts into database
    int status = insertCustomer(fName, mName, lName,
                                iage, issn, city, state, country);

    if (status != -1) {
        responseMsg = "Sucessfully processed the form";
    } else {
        responseMsg = "Failed to process the form. Please try again";
    }
} catch (Exception e) {
    responseMsg = "Error while Parsing Age and SSN.
                    Make sure you enter numbers";
}
// Send the response
pw.println("<h1" + responseMsg + "</h1>");

}

private int insertCustomer(String firstName, String middleName,
    String lastName, int age, int ssn, String city, String state,
    String country) {
    // Please write the JDBC logic to insert the data into the
    //Customers table. Take this as a home work.
    // This method should return an int that the executeUpdate
    //methods returns. Note: the driver name and the URL are
    //already available in the init() method.

    return 1;
}
}
```

The FormProcessingServlet does the following things:

1. In the `doGet()` method, it reads all the form data using the `getParameter()` method. All the data will be returned as strings. When the form is submitted, the URL will look as shown below:

`http://localhost:8080/myweb/FormProcessIngServlet?firstName=Smith&....`

The `getParameter()` method takes the `firstName` as the key, and returns

Smith as the value.

2. Converts the age and ssn into integers.
3. Invokes the `insertCustomer()` method and passes all the data to it. This method should insert the data into the database. Take this as a home work.
4. Finally it sends the response message back to the browser.

Steps to run the Servlet

1. Save the two files as

/myweb/WEB-INF/classes/myservlets/FormProcessingServlet.java
/myweb/htmls/CustomerForm.html

2. Compile the servlet as shown below:

```
C:/>jakarta-tomcat-4.1.31>webapps>myweb>WEB-INF>classes>javac  
myservlet*.java
```

3. Add the following snippet in the web.xml file

```
<!-- Servlet Definitions -->  
  
<servlet>  
    <servlet-name>FormProcessingServlet</servlet-name>  
    <servlet-class>myservlets.FormProcessingServlet</servlet-class>  
    <init-param>  
        <param-name>driver</param-name>  
        <param-value>com.mysql.jdbc.Driver</param-value>  
    </init-param>  
    <init-param>  
        <param-name>URL</param-name>  
        <param-value>jdbc:mysql://localhost:3306/MDIT</param-value>  
    </init-param>  
</servlet>  
  
<!-- Servlet Mappings -->  
  
<servlet-mapping>  
    <servlet-name>FormProcessingServlet</servlet-name>  
    <url-pattern>/FormProcessingServlet</url-pattern>  
</servlet-mapping>
```

4. Start the server by clicking the `startup.bat` in the bin directory.
5. Type the following URL in the browser

<http://localhost:8080/myweb/htmls/CustomerForm.html>

Fill in the form and click the submit button. If you entered valid data, you will see the message

Form Processed Sucessfully.

If you entered characters for age or ssn, you'll see the following message:

Error while Parsing Age and SSN. Make sure you enter numbers

Session Management

The examples we saw until now just deal with one servlet. However, a typical web application comprises of several servlets that require collaborating with each other to give a complete response. For instance, if you go online to purchase a book, you need to go through multiple pages like search page, shopping page, billing page etc before you can complete the transaction. In situations like this, it is important that one servlet shares information or data with other servlet.

In web application terminology we call the shared data or information as *state*. Having **state** is just not sufficient. Someone must be able to pass this state from one servlet to other servlet so that they can share the data. So, who does this state propagation? Can HTTP do this? No, because HTTP is a stateless protocol which means it cannot propagate the state. So, is there anyone to help us out here to make the state available to all the servlets? Yes, there is one guy who is always there for our rescue and it's none other than web container (Tomcat).

A web container provides a common storage area called as *session* store the state and provides access to this session to all the servlets within the web application. For instance, servlet A can create some state (information) and store it in the session. Servlet B can then get hold of the session and read the state.

Since the state (data or information) in the session is user specific, the web container maintains a **separate** session for each and every user as shown in the following diagram.

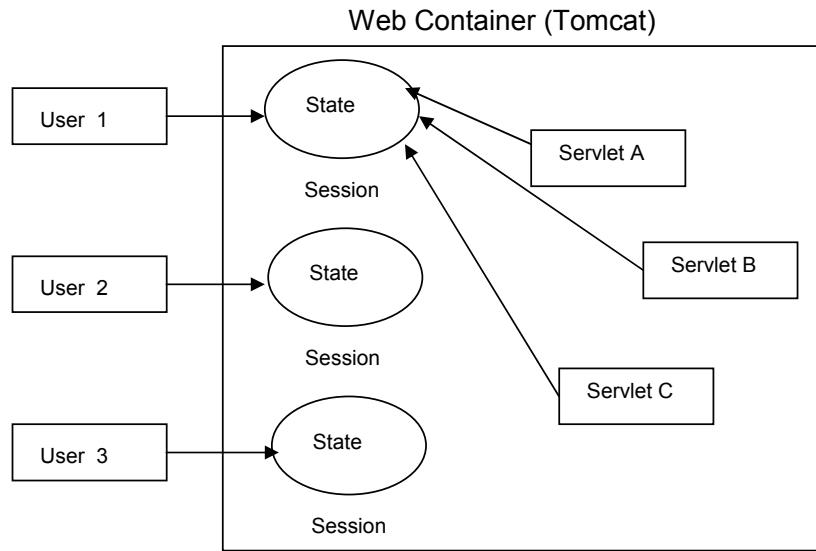


Fig 13.2 Session Management

If you look at the above figure, every user will have his own session object for storing the state pertaining to his transaction and all the servlets will have access to the same session. Also notice, the **session** objects are present within the container.

Now that we know what a session is, let's see how a servlet uses the session for sharing the data across multiple pages in a web application. A servlet can do the following four most important operations to work with sessions.

1. Create the session
2. Store the data in the session
3. Read the data from the session
4. Destroy the session or invalidate the session.

The above four operations are called as Session Management operations. Let's see each of them one by one.

Creating a Session

The servlet API provides us with a class called `HttpSession` to work with sessions. To create a session, we do the following:

```
HttpSession session = request.getSession(true);
```

The above method returns a new session object if one is not present, otherwise it returns the old session object that is already created before.

Storing the data in Session

Data in session is stored as key-value pair just like in `HashMap` or `Hashtable`. The value can be any Java object and the key is usually a `String`. To store the data we use the `setAttribute()` method as shown below:

```
session.setAttribute("price", new Double("12.45"));
```

Reading the data from the Session

To read the data, we need to use the `getAttribute()` method by passing in the **key** as shown below which then returns the value object:

```
Double d = (Double)session.getAttribute("price");
```

Notice that we need to do proper casting here. Since we stored `Double` object, we need to cast it again as `Double` while reading it.

Destroying the Session

A session is usually destroyed by the last page or servlet in the web application. A session is destroyed by invoking the `invalidate()` method as shown below:

```
session.invalidate();
```

Before we write an example using session management, let's see a simple concept called Request Dispatching.

Request Dispatching

Request dispatching is the ability of one servlet to dispatch or delegate the request to another servlet for processing. In simple words, let's say we have a servlet A which doesn't know how to completely process the request. Therefore, after partially processing the request, it should forward the request to another servlet B. This servlet then does the rest of the processing and sends the final response back to the browser.

The class used for dispatching requests is the `RequestDispatcher` interface in Servlet API. This interface has two methods namely `forward()` and `include()`.

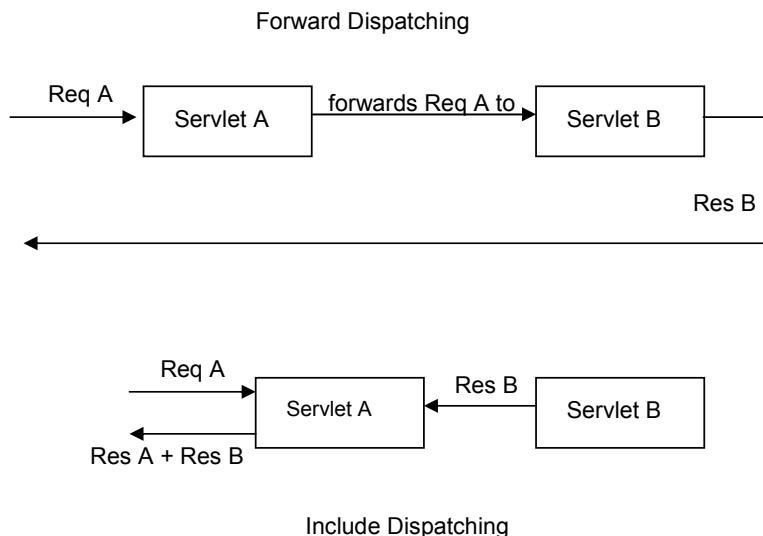
The `forward()` method

This method is used for forwarding request from one servlet to another. Consider two servlets A and B. Using this method, A gets the request, which then forwards the request to B, B processes the request and sends the response to the browser. This method takes `HttpServletRequest` and `HttpServletResponse` as parameters.

The `include()` method

With this method, one servlet can include the response of other servlet. The first servlet will then send the combined response back to the browser. This method also takes `HttpServletRequest` and `HttpServletResponse` as parameters.

Following two figures demonstrate dispatching using `forward()` and `include()` methods.



Now that we also know something about request dispatching let's build a mini web application using session management and request dispatching.

The following figure shows the structure of the web application we are going to build.

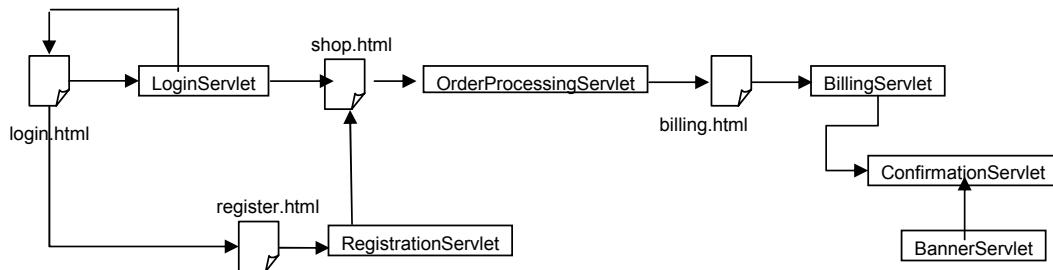


Fig 13.3 A sample web application

Following is how the application works:

1. The `login.html` sends the login information to the `LoginServlet`.
2. The `login.html` also sends the user to `register.html` to register
3. The `LoginServlet` will process the login information. If login fails, it redirects the user back to `login.html` page. Otherwise, it takes the user to `shop.html` page.
4. The `register.html` will take the customers information and sends it to `RegistrationServlet`. This servlet saves all the data in the database and sends him to the `shop.html`.
5. The user will shop the products in `shop.html`. Upon checkout, shopping cart information will be sent to `OrderProcessingServlet` which processes the order and saves all the billing details in the session. It then forwards the request to `billing.html` page.
6. The `billing.html` page will take the credit card information and sends the data to `BillingServlet`.
7. `BillingServlet` uses the billing data in the session and bills the customer. It then forwards to `ConfirmationServlet` that displays the confirmation message with a thank you greeting.
8. The response of `BannerServlet` will be included in the response of `ConfirmationServlet`.

Let's build the components one by one.

login.html

Listing 13.4a (`login.html`) Login Form

```
<HTML>
<BODY>
<P align="center"><FONT size="6" color="#008040" face="Verdانا">Welcome
to BuyForLess</FONT></P>
<P><BR>
</P>
<DIV align="center">

<form action="/myweb/LoginServlet">
<TABLE border="0">
    <TBODY>
        <TR>
            <TD width="109"><FONT size="3" face="verdana">Login ID</FONT></TD>
            <TD width="255"><INPUT type="text" name="userid" size="35"></TD>
        </TR>
        <TR>
            <TD width="109"><FONT size="3" face="verdana">Password</FONT></TD>
            <TD width="255"><INPUT type="password" name="password" size="35"></TD>
        </TR>
        <TR>
            <TD width="109"></TD>
            <TD width="255"><INPUT type="submit" name="submit" value="Login"></TD>
        </TR>
    </TBODY>
</TABLE>
<BR>
<FONT size="3" face="verdana">If you are visiting for the first time,
please <A href="/myweb/register.html">register here</A></FONT></DIV>
</BODY>
</HTML>
```

LoginServlet.java

Listing 13.4b (LoginServlet.java) Login Servlet

```
package myservlets;

import java.io.IOException;

import javax.servlet.*;
import javax.servlet.http.*;

public class LoginServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Read the parameters
        String userid = req.getParameter("userid");
        String password = req.getParameter("password");

        if (isAuthenticated(userid, password)) {
            // Go to the shopping page
            RequestDispatcher rd = req.getRequestDispatcher("/htmls/shop.html");
            rd.forward(req, res);
        } else {
    }}
```

```
// Go back the login page
RequestDispatcher rd = req.getRequestDispatcher("/htmls/login.html");
rd.forward(req, res);
}

protected void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    doGet(req, res);
}

private boolean isAuthenticated(String userid, String password) {
    // You need to write JDBC loginc here to verify from the datababse
    return true;
}
}
```

shop.html

Listing 13.4c (shop.html) A simple shopping page

```
<html>
    <head>
        <title>Customer Form</title>
    </head>
    <body>
        <p align="center">
            <font size="6" color="#008040" face="Verdana">
                Welcome to BuyForLess
            </font>
        </p>
        <hr />
        <h4 align="center">
            <font size="3" face="verdana">
                Please Select the following items and click Buy to
                purchase
            </font>
        </h4>
        <form action="/myweb/OrderProcessingServlet">

            <table border="1" cellspacing="1" cellpadding="1">
                <tr>
                    <td align="center">Book Name</td>
                    <td align="center">Distributer</td>
                    <td align="center">Price</td>
                    <td align="center">Quantity </td>
                </tr>
                <tr>
                    <td align="center">Advanced J2EE</td>
                    <td align="center">Hall Publications</td>
                    <td align="center">$10.99</td>
                    <td align="center">
```

```
<SELECT name="j2eeQty">
    <OPTION value="0">0</OPTION>
    <OPTION value="1">1</OPTION>
    <OPTION value="2">2</OPTION>
    <OPTION value="3">3</OPTION>
</SELECT>

        </td>
    </tr>
    <tr>
        <td align="center">Perl Scripting</td>
        <td align="center">Wrox Publications</td>
        <td align="center">$9.99</td>
        <td align="center">
            <SELECT name="perlQty">
                <OPTION value="0">0</OPTION>
                <OPTION value="1">1</OPTION>
                <OPTION value="2">2</OPTION>
                <OPTION value="3">3</OPTION>
            </SELECT>
        </td>
    </tr>
    <tr>
        <td align="center">Red Roses</td>
        <td align="center">CoolFlowers.com</td>
        <td align="center">$1.99</td>
        <td align="center">
            <SELECT name="flowerQty">
                <OPTION value="0">0</OPTION>
                <OPTION value="1">1</OPTION>
                <OPTION value="2">2</OPTION>
                <OPTION value="3">3</OPTION>
            </SELECT>
        </td>
    </tr>
    <tr>
        <td></td>
        <td align="center">
            <input type="submit" value="Checkout"
                  size="20" />
        </td>
    </tr>
</table>
</form>
<br />
<i>Copyright 2001 BuyForLess.com</i>
</body>
</html>
```

register.html

Listing 13.4d (register.html) A simple registration page

```
<HTML>
<BODY>
```

```
<P align="center"><FONT size="6" color="#008040" face="Verdana"> Welcome  
to BuyForLess </FONT></P>  
<P align="center"><BR />  
Please fill in the following details for registration</P>  
<DIV align="center">  
<hr />  
<form action="/myweb/RegistrationServlet">  
<TABLE border="0">  
    <TBODY>  
        <TR>  
            <TD width="142">Name</TD>  
            <TD width="222"><INPUT name="name" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142">Email</TD>  
            <TD width="222"><INPUT name="email" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142">Address Line 1</TD>  
            <TD width="222"><INPUT name="addressline1" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142">City</TD>  
            <TD width="222"><INPUT name="city" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142">State</TD>  
            <TD width="222"><INPUT name="state" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142">Country</TD>  
            <TD width="222"><INPUT name="country" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142">UserName</TD>  
            <TD width="222"><INPUT name="username" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142">Password</TD>  
            <TD width="222"><INPUT name="password" size="35" /></TD>  
        </TR>  
        <TR>  
            <TD width="142"></TD>  
            <TD width="222"><INPUT type="submit" name="submit" value="Register" />  
        </TD>  
    </TR>  
    </TBODY>  
</TABLE>  
</form>  
</DIV>  
<hr />  
</BODY>  
</HTML>
```

RegistrationServlet.java

Listing 13.4e (`RegistrationServlet.java`) A simple registration servlet

```
package myservlets;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;

public class RegistrationServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Read the data
        String name = req.getParameter("name");
        String email = req.getParameter("email");
        String line1 = req.getParameter("addressline1");
        String city = req.getParameter("city");
        String state = req.getParameter("state");
        String country = req.getParameter("country");
        String userid = req.getParameter("username");
        String password = req.getParameter("password");

        insertToDatabase(name, email, line1, city, state,
                         country, userid, password);

        // Forward to Shopping page
        req.getRequestDispatcher("/htmls/shop.html").forward(req, res);
    }

    private void insertToDatabase(String name, String email, String line1,
                                 String city, String state, String country,
                                 String userid, String password) {
        // Write JDBC logic to insert data to REGCUSTOMERS table
    }
}
```

OrderProcessingServlet

Listing 13.4f (`OrderProcessingServlet.java`) Servlet to process the order.

```
package myservlets;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;

public class OrderProcessingServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
throws ServletException, IOException {  
  
    // Read the form data.  
  
    String j2eeQty = req.getParameter("j2ee");  
    String perlQty = req.getParameter("perl");  
    String flowersQty = req.getParameter("flowers");  
  
    // Compute the price  
    int q1 = Integer.parseInt(j2eeQty);  
    int q2 = Integer.parseInt(perlQty);  
    int q3 = Integer.parseInt(flowersQty);  
  
    double totalPrice = q1 * 10.99 + q2 * 9.99 + q3 * 1.99;  
  
    // Get the session  
    HttpSession session = req.getSession(true);  
  
    // Store the data in the session  
    session.setAttribute("TotalPrice", new Double(totalPrice));  
  
    // Forward to Billing page  
    req.getRequestDispatcher("/htmls/billing.html").forward(req, res);  
  
}  
}
```

billing.html

Listing 13.4g (billing.html) A simple billing page.

```
<HTML>  
<BODY>  
  <P align="center"><FONT size="6" color="#008040" face="Verdana">Welcome  
  to BuyForLess</FONT></P>  
  <P><BR>  
  </P>  
  <DIV align="center">  
    <hr />  
    <form action="/myweb/BillingServlet">  
      <TABLE border="0">  
        <TBODY>  
          <TR>  
            <TD width="225">Card Type</TD>  
            <TD width="347"><INPUT type="radio" name="cardtype"  
               value="MasterCard" checked> MasterCard  
               <INPUT type="radio" name="cardtype" value="Visa">  
               VISA  
            </TD>  
          </TR>  
          <TR>  
            <TD width="225">Credit Card Number</TD>  
            <TD width="347">  
              <INPUT type="text" name="number" size="35"></TD>  
            </TR>  
            <TR>
```

```
<TD width="225">Expiration Date</TD>
<TD width="347"><INPUT name="date" size="15"></TD>
</TR>
<TR>
    <TD width="225"></TD>
    <TD width="347">
        <INPUT type="submit" name="submit" value="Submit"></TD>
    </TR>
</TBODY>
</TABLE>
</BODY>
<hr />
</DIV>
</HTML>
```

BillingServlet.java

Listing 13.4h (`BillingServlet.java`) A simple billing Servlet.

```
package myservlets;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;

public class BillingServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Read the card details

        String cardType = req.getParameter("cardType");
        String number = req.getParameter("number");
        String date = req.getParameter("date");

        // Get the session
        HttpSession session = req.getSession(true);

        // Generate a confirmation number
        long conf = (long) (Math.random() * 9999999);

        // Store the data in the session
        session.setAttribute("Confirmation", new Long(conf));

        // Forward to Billing page
        req.getRequestDispatcher("/ConfirmationServlet").forward(req, res);
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        doGet(req, res);
    }
}
```

}

ConfirmationServlet.java

Listing 13.4i (ConfirmationServlet.java) A simple confirmation Servlet.

```
package myservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ConfirmationServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Get the session
        HttpSession session = req.getSession(true);

        // Read the price and confirmation number

        Double price = (Double) session.getAttribute("TotalPrice");
        Long conf = (Long) session.getAttribute("Confirmation");

        // Compute the price including tax
        double totalPrice = price.doubleValue() + 0.07 * price.doubleValue();
        PrintWriter pw = res.getWriter();

        pw.println("<h3> Your Order has been sucessfully processed</h3><br/>");
        pw.println("<h3>Your credit card is charged for $ " + totalPrice
                  + "</h3><br/>");
        pw.println("<h3>The tracking Number for this order is " +
                  conf.longValue()
                  + "<br/>");

        // Incude the response of Banner Servlet

        req.getRequestDispatcher("/BannerServlet").include(req, res);
    }
}
```

BannerServlet.java

Listing 13.4j (BannerServlet.java) A simple banner Servlet.

```
package myservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BannerServlet extends HttpServlet {
```

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    PrintWriter pw = res.getWriter();
    pw.println("<hr/>");
    pw.println("Thank You for Shopping at BuyForLess.com. Please Visit
    again. Have a nice day!");
}
```

Steps to run the application

1. Save the html files in the following directory

/myweb/htmls/

2. Save all the Java files in the following directory

/myweb/WEB-INF/classes/myservlets

3. Compile the servlets as shown below:

C:/>jakarta-tomcat-4.1.31>webapps>myweb>WEB-INF>classes>javac
myservlet*.java

4. Add the following snippet in the web.xml file

```
<!-- Servlet Definitions -->
<servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>myservlets.LoginServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>RegistrationServlet</servlet-name>
    <servlet-class>myservlets.RegistrationServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>OrderProcessingServlet</servlet-name>
    <servlet-class>myservlets.OrderProcessingServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>BillingServlet</servlet-name>
    <servlet-class>myservlets.BillingServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>ConfirmationServlet</servlet-name>
    <servlet-class>myservlets.ConfirmationServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>BannerServlet</servlet-name>
    <servlet-class>myservlets.BannerServlet</servlet-class>
```

```
</servlet>

<!--  Servlet Mappings  -- >

<servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/LoginServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>RegistrationServlet</servlet-name>
    <url-pattern>/RegistrationServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>OrderProcessingServlet</servlet-name>
    <url-pattern>/OrderProcessingServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>BillingServlet</servlet-name>
    <url-pattern>/BillingServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ConfirmationServlet</servlet-name>
    <url-pattern>/ConfirmationServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>BannerServlet</servlet-name>
    <url-pattern>/BannerServlet</url-pattern>
</servlet-mapping>
```

5. Start the server by clicking the `startup.bat` in the `bin` directory.
6. Type the following URL in the browser

`http://localhost:8080/myweb/htmls/login.html`

Type in any username and password and you'll enter in to the application. In the shopping page select the products and checkout. This will send you to a billing page that takes the credit card details and gives you a confirmation message.

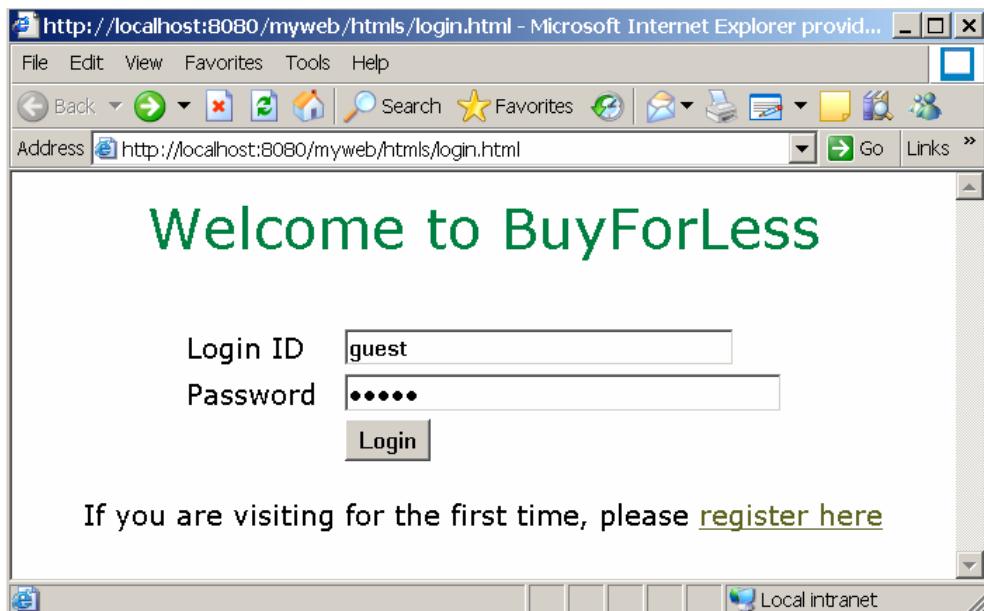
Following is how each of the Servlet classes work.

LoginServlet

This servlet reads the parameters posted by `login.html`. It passes the username and password to a method named `isAuthenticated()` which returns either `true` or `false`. For now, this method is defaulted to always return `true`. In reality, you need to write database logic to verify whether the user exists in the database or not. If not, then return `false`.

If the method returns true, which it does, it uses the RequestDispatcher and forwards the request to the shop.html page. Otherwise, it forwards back to the login.html page. Following is the code that does this.

```
if( isAuthenticated( userid, password) ) {
    // Go to the shopping page
    RequestDispatcher rd = req.getRequestDispatcher("/htmls/shop.html");
    rd.forward(req,res);
}
else{
    // Go back the login page
    RequestDispatcher rd = req.getRequestDispatcher("/htmls/login.html");
    rd.forward(req,res);
}
```



OrderProcessingServlet

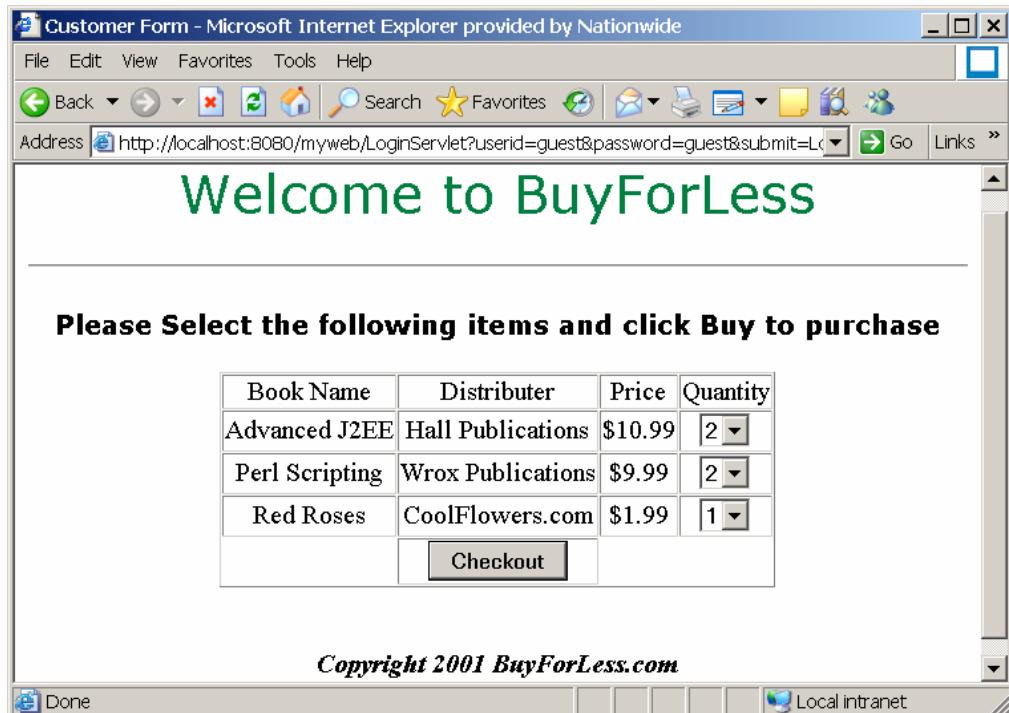
This servlet reads all the data submitted by the form in the shop.html, and computes the price of the order. It then stores the price in the session as shown below:

```
// Get the session
HttpSession session = req.getSession(true);

// Store the data in the session
session.setAttribute("TotalPrice",new Double(totalPrice));
```

The price will then used by the `BillingServlet` later on. Finally it forwards the request to the billing page as shown below:

```
// Forward to Billing page  
req.getRequestDispatcher("/htmls/billing.html").forward(req,res);
```



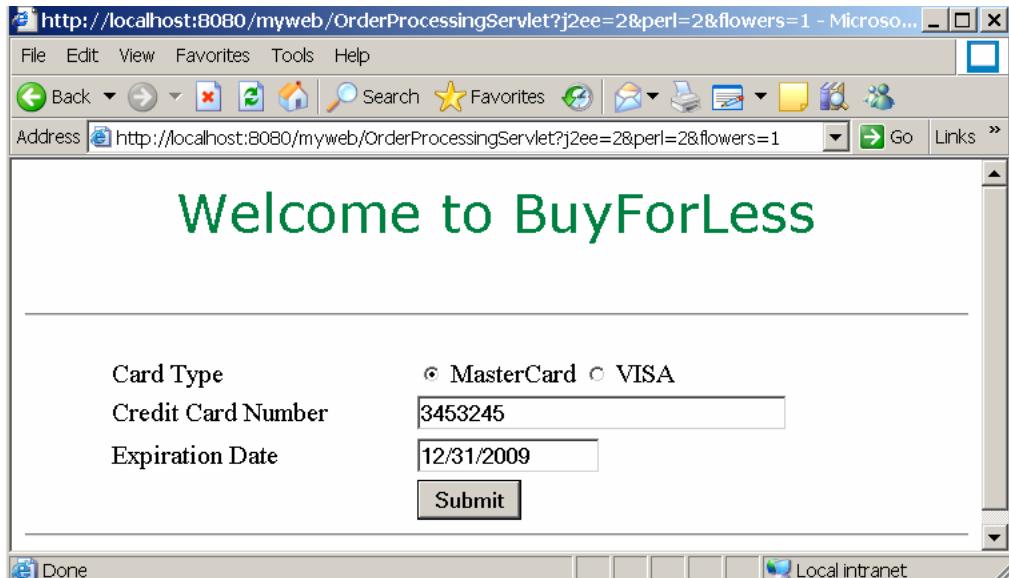
BillingServlet

This servlet reads the billing information posted by `billing.html` page and generates a confirmation number. It then stores this confirmation number in the session as shown below:

```
// Get the session  
HttpSession session = req.getSession(true);  
  
// Generate a random confirmation number  
long conf = (long)(Math.random()*9999999);  
  
// Store the data in the session  
session.setAttribute("Confirmation",new Long(conf));
```

Finally it forwards the request to `ConfirmationServlet` as shown below:

```
// Forward to confirmation page  
req.getRequestDispatcher("/ConfirmationServlet").forward(req,res);
```



ConfirmationServlet

This servlet reads both the price and confirmation number from the session, and computes the overall price including the taxes (7%) as shown below:

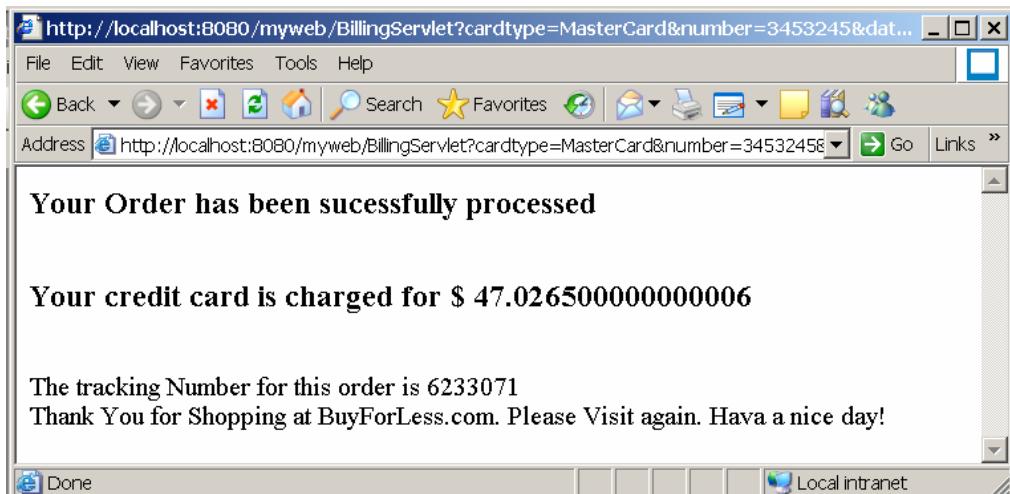
```
// Get the session  
HttpSession session = req.getSession(true);  
  
// Read the price and confirmation number  
  
Double price = (Double) session.getAttribute("TotalPrice");  
Long conf = (Long) session.getAttribute("Confirmation");  
  
// Compute the price including tax  
double totalPrice = price.doubleValue() + 0.07*price.doubleValue();
```

It then combines its own response message along with the response of BannerServlet as shown below:

```
pw.println("<h3> Your Order has been sucessfully processed</h3><br/>");  
pw.println("<h3>Your credit card is charged for $ "+totalPrice+"</h3><br/>");  
pw.println("<h3>The tracking Number for this order is "+  
conf.longValue()+"<br/>");  
  
// Incude the response of Banner Servlet
```

```
req.getRequestDispatcher("/BannerServlet").include(req, res);
```

Notice that we used the `include()` method as opposed to `forward()` method to include the response of `BannerServlet`.



This completes all the important concepts in servlets. All the examples we did until now are more than enough using Servlets. These days, most of the web applications are moving away from using servlet technology and using JSP and open source technologies like Struts to build web applications. This doesn't mean we can completely ignore Servlets. The reason is simple. The underlying technology for both JSP as well as Struts is again Servlets. So, it's always good to know how servlets work.

One of the primary drawbacks with servlets is the way they generate the html response. If you noticed, we use whole lot of `println()` method to send the html response. This is really tedious and time consuming task. A typical web application will have 90% of HTML code and 10% dynamic content. So all our Java code will be cluttered with these `println` statements which makes the servlet hard to understand and manage. Practically, it's tedious to write HTML markup in a Java program and I am sure you'll agree with me.

For a sec, let's try to reverse the situation here. How will it be if we rather write Java code in a HTML file? Hmm, sounds interesting, right? If this is the case, we don't have to write a single `println` statement. We just have to write the Java code in HTML file where ever needed. This reverse technology is nothing but what we call as JSP technology. We will discuss this in the next chapter. JSP makes our life very simple, but

again, the underlying technology for JSP is Servlets. So, learning Servlets is worth the time.

The beauty of J2EE is that it simplifies life as we move forward. I know you might be thinking why not jump right away to simplified technologies. If we do this, even the simplest things will get tough. So, let's move step by step.

Summary

- ✓ Web Applications use Internet for enterprise computing.
- ✓ In a web application, a browser sends requests to server side programs.
- ✓ The server side programs processes the requests and send the responses.
- ✓ The protocol used to send requests is HTTP. HTTP is a stateless protocol.
- ✓ The most widely used HTTP requests are GET and POST.
- ✓ In GET request, all the query data is appended to the URI itself. GET requests are using for sending smaller amounts of data that need not be secure.
- ✓ POST requests can send large amounts of data. It also hides the data from being visible when sent to a server program.
- ✓ In J2EE, the server side programs run in a Web Container. The most popular one is the Tomcat container from Apache.
- ✓ A Servlet is a server side program that runs in a web container to process the HTTP requests and send HTTP response.
- ✓ The web container uses **web.xml** to identify which servlets it need to load. The **web.xml** file is called as the deployment descriptor.
- ✓ To write a servlet we use Servlet API. One of the main applications of servlet is form processing.
- ✓ A typical web application will have several servlets that collaborate with each other.
- ✓ Servlets share the data using **session**. A **session** is created and maintained by the web container. The web container uses a separate session for every client.
- ✓ Servlets use RequestDispatcher object to delegate requests.
- ✓ The main difference between **include** and **forward** is that, in the former a servlet uses the **include()** method to include the response of another servlet in its response while in the later, a servlet uses the **forward()** method to dispatch the request to another servlet.

Time to play 50-50

1. Which of the following protocol is used by web applications

- a) HTML
b) HTTP
2. Which of the following is used to read the data in a HTTP request
 - a) HttpServletRequest
 - b) HttpServletRequest
3. Which of the following file is used as a deployment descriptor for web applications?
 - a) servlet.xml
 - b) web.xml
4. Which of the following is used for managing sessions in a Servlet?
 - a) HttpSession
 - b) HtmlSession
5. Which of the following class is used to dispatch requests?
 - a) RequestDispatcher
 - b) ServletDispatcher
6. Which of the following method is used by a servlet to delegate a request?
 - a) forward()
 - b) dispatch()
7. Which of the following is a method in RequestDispatcher to add the response of another servlet
 - a) forward
 - b) include

Interview Questions

Question: What is a Servlet?

Servlet Programming

Answer: Server is a server side Java program that runs in a web container for processing HTTP requests and sending HTTP responses.

Question: What is the name of the deployment descriptor a web application uses?

Answer: web.xml

Question: What are the various ways of managing sessions?

Answer: Cookies, Hidden Form fields and URL rewriting

Question: What is URL rewriting?

Answer: URL rewriting is a technique used for session management. In this approach, a web container generates a unique identifier called as **SESSIONID** and appends to every URL in the page. When the URL is clicked, the browser sends the **SESSIONID** back to the server which it uses to identify the client.

Question: What is the difference between include and forward dispatching?

Answer: Refer to the Request Dispatch section

Chapter 14

JSP Programming

This chapter introduces you to the most widely used web technology, the Java Server Pages (JSP). By the end of this chapter, you'll become familiar in building web applications using JSP pages. JSP technology is the coolest and simplest of all web technologies and I am sure you'll enjoy reading it.

Chapter Goals

- ✓ Understand the fundamentals of JSP
- ✓ Understand the JSP directives
- ✓ Understand JSP scriptlets
- ✓ Using Implicit Objects
- ✓ Using Java Beans
- ✓ Custom Tag Anatomy
- ✓ Building custom tags

Environment Setup

1. Create a directory named `jsp`s in the following directory in Tomcat

`C:/...../.../webapps/myweb/jsps`

2. Create a directory named `beans` in the following directory in Tomcat

`C:/...../.../webapps/myweb/WEB-INF/classes/beans`

3. Create a directory named `customtags` in the following directory in Tomcat

`C:/...../.../webapps/myweb/WEB-INF/classes/customtags`

Introduction

JSP is yet another J2EE technology for building web applications using Java. This technology is built to overcome all the cons of Servlet technology, one of which is the way it renders the html response. Since most of the web pages have static content with little dynamic content, Sun Microsystems decided to build a technology that completely simplifies the development of web applications by embedding Java code inside html markup as opposed to html markup inside Java code used by servlets. JSP technology has become an instant hit and has become the de facto standard for building web applications using Java. In a single sentence, JSP is the past, present and will be the future technology for any web application built using Java and J2EE. This is the reason why we are seriously interested to learn and digest this cool technology.

Let me tell you one thing here. One of the primary reasons for the success of any technology is the simplicity and ease that it offers. The simpler a technology gets, more is the success it tastes. This is applicable to everything, not just Java and J2EE. Trust me, keep your life as simple as possibly can, and you'll truly taste success.

One thing you must know at this point is that JSP technology is built on top of servlet technology. This is why we can call JSP as an abstraction over servlet. What does abstraction mean? In simple terms, abstraction is a simplified fine grained layer over a slightly more complex layer that makes the development faster and easier. More the abstraction, more the simplicity.

With this brief introduction, let's jump into the real world of JSP and see how it justifies all the good things about it that we stated thus far.

JSP Basics

A typical JSP page very much looks like html page with all the html markup except that you also see Java code in it. So, in essence,

HTML + Java = JSP

Therefore, JSP content is a **mixed** content, with a mixture of HTML and Java. If this is the case, let me ask you a quick question. Can we save this mixed content in a file with ".html" extension? You guessed it right. No we can't, because the html formatter will also treat the Java code as plain text which is not what we want. We want the Java code to be executed and display the dynamic content in the page. For this to happen, we need to use a different extension which is the ".jsp" extension. Good. To summarize, a html

file will only have html markup, and a jsp file will have both html markup and Java code. Point to be noted.

Now, look at the following figure:

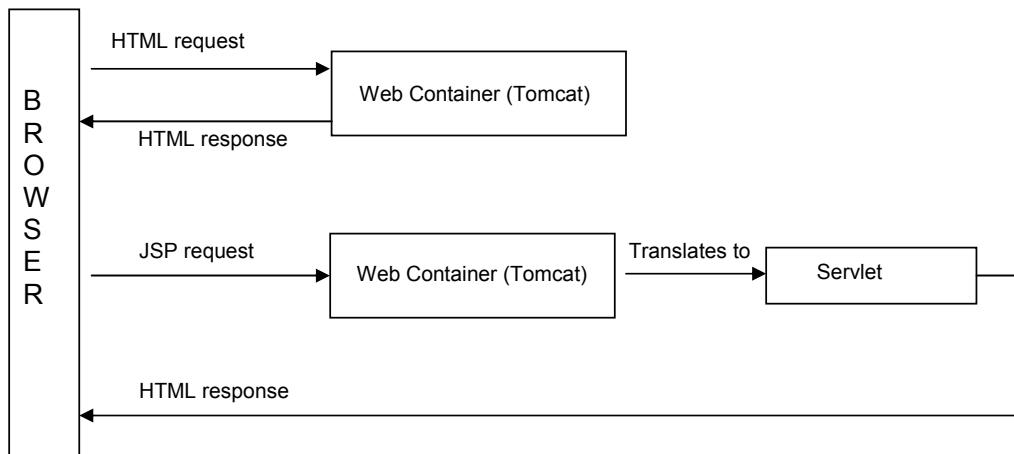


Fig 14.1 HTML and JSP Requests

The above picture should give you some idea of how a web container processes html requests and JSP requests. When the browser requests for html file, the web container simply responds with a html response without any processing. However, when the browser sends a JSP page request, the web container assumes that the JSP page might include Java code, and *translates* the page into a *Servlet*. The servlet then processes the Java code, and returns the complete html response including the dynamic content generated by the Java code. I am sure the picture is now 100% clear. Don't think anything beyond this.

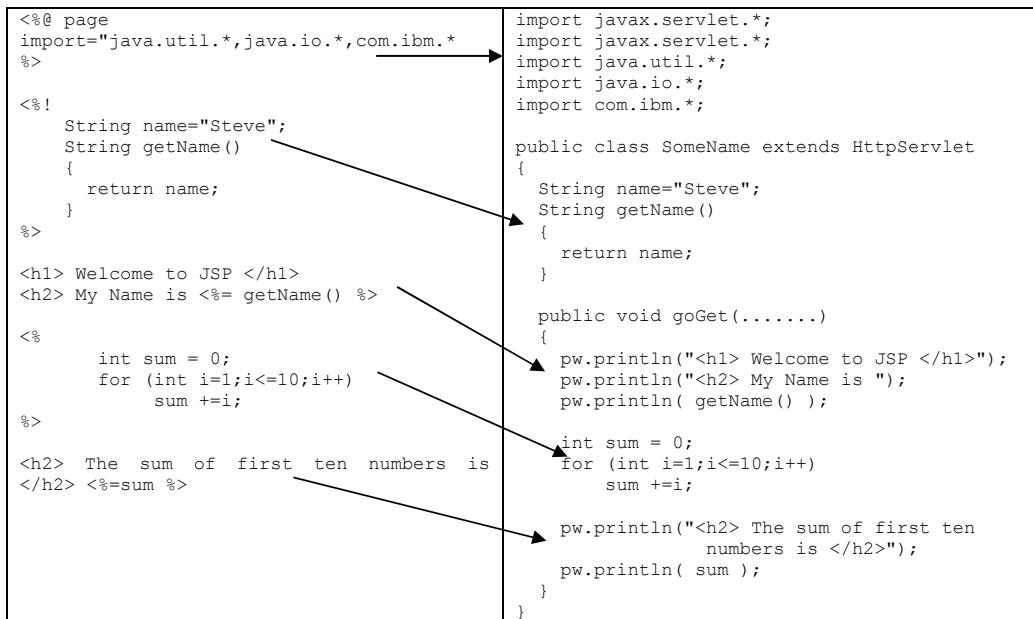
Though the web container does the translation of JSP to Servlet, we need to know how it does this translation. For a web container to translate JSP, it needs to identify from the JSP page, as to which is HTML markup and which is Java code. According to J2EE specification, a JSP page must use **special symbols** as placeholders for Java code. The web container instead of scratching its head to identify Java code, simply uses the special symbols to identify it. This is a contract between JSP and the Web container. This is an example of making life easy.

Let's understand what these special symbols are. In JSP, we use four different types of placeholders for Java code. Following table shows these placeholders along with how the web container translates the Java code with them.

Table 14.1 JSP to Servlet Translation

JSP Code	Translated to
<%! Java Code %>	Global Variables and Methods in a Servlet
<% Java Code %>	doGet { Java Code } doPost { Java Code }
<%= Java Code %>	PrintWriter pw = res.getWriter(); pw.println(Java Code);
<%@ %>	Mostly as imports in Servlet

To better understand the above four place holders, let's take a sample JSP page and see how the web container translates it into a Servlet. See the following JSP and Servlet code snippets.



The above table shows exactly how the Java code in various placeholders is translated into different places in a servlet. Look at the rules in previous table and every thing will make sense. Even if you don't understand the above translation, it's still fine because we don't do this translation. We are just trying to understand what the web container does. After all, who cares what it does. So, don't worry. All you just have to know as a JSP

developer is, how and for what purposes we use various placeholders. Understanding this hardly takes few minutes. So, without wasting any further time, let's see what these are.

JSP Directives

Directives are used for declaring classes, session usage etc., and does not produce any response to the client. A directive uses attributes for declarations. There are 3 types of directives as listed below:

- ✓ page directive
- ✓ include directive
- ✓ taglib directive

The page directive

This directive is used to declare things that are important to the entire JSP page. The syntax for this directive is shown below

```
<%@ page attribute list %>
```

Following table lists the most widely used attributes with this directive:

Table 14.2 Page directive attributes

page Attribute	Description
import	Used by the current JSP for importing the resources
session	Used by the current JSP for session management
errorPage	Used to specify the error pages for the current JSP
isErrorPage	Used to specify the current JSP as an error page to some other JSP

Ex 1: If the JSP page needs to import the core library classes, it uses the page directive as:

```
<%@ page import="java.util.*, java.io.*" %>
```

All the resources must be separated by a comma as shown above.

Ex 2: If the JSP page needs to use the session, then its uses page directive attribute as shown below:

```
<%@ page session="true" %>
```

If the session attribute is set to true, then the page will have access to session.

Ex 3: If a JSP page should forward to a custom error page for any exceptions within the page, the page directive will be as shown below:

```
<%@ page errorPage="Error.jsp" %>
```

The web container will then forward the request to Error.jsp if it encounters exceptions within the page.

Ex 4: If a JSP page should allow itself as an error page for other JSP pages, it should use the following page attribute:

```
<%@ page isErrorPage="true" %>
```

Instead of defining one attribute per page directive, we can define all the attributes at a time as shown below:

```
<%@ page import="java.util.*" session="true" errorPage="Error.jsp" %>
```

I know you are waiting to see an example, so take a look at the code in listing 14.1 that uses a page directive.

Listing 14.1 (PageDirectiveDemo.jsp) JSP using page directive

```
<%@ page import="java.util.*" session="true" isErrorPage="false"%>

<HTML>
<BODY>
<h4>Welcome to the world of JSP</h4>

This JSP uses the page directive
</BODY>
</HTML>
```

Steps to run the JSP

1. Save the code as “/myweb/jsp/PageDirectiveDemo.jsp”.
2. Start the server, and type the following URL in the browser

```
http://localhost:8080/myweb/jsp/PageDirectiveDemo.jsp
```

Since we are writing Java code in JSP, there is 100% chance that the Java code may throw an exception. So, what happens when the Java code in JSP throws some exception? You'll see the very familiar *page not found* error. Such pages really freak the customers

who visit the web site. Good web applications usually display custom error page rather than blank pages. By displaying error pages, we can let the users know what might have caused the error and give them a help desk number for customer support.

To display custom error pages, JSP uses the page directive attribute `errorPage`. Let's first write a piece of code to demonstrate its usage and then see how it works. See the code in listing 14.2.

Listing 14.2a (`ErrorPageDemo.jsp`) JSP using page directive

```
<%@ page errorPage="Error.jsp" %>  
<%  
    int i=0;  
    int k = 10/i; // This throws Arithmetic Exception  
%>
```

For now, don't worry about the code in `<%` and `%>` symbols. We'll see about it later.

Listing 14.2b (`Error.jsp`) JSP using page directive

```
<%@ page isErrorPage="true" %>  
<h3> An exception is thrown by the page you're trying to access.  
Please don't panic and call 1-800-888-9999 for help desk  
</h3>
```

Steps to run the JSP

1. Save the above two files as

`/myweb/jsp/`
`ErrorPageDemo.jsp` and
`/myweb/jsp/`
`Error.jsp`

2. Start the server, and type the following URL in the browser:

`http://localhost:8080/myweb/jsp/PageDirectiveDemo.jsp`

Look at the `ErrorPageDemo.jsp`. It declares the page directive attribute `errorPage` to use `Error.jsp` as an error page incase any exception pops up. The `Error.jsp` page grants permission to use it as an error page by using the page directive attribute `isErrorpage` with the value set to `true`. This is like mutual understanding between the pages. If A needs to use B, then B should grant the permission to A. This is exactly what both the above JSP's are doing.

When the above URL accesses the `ErrorPageDemo.jsp`, it throws `ArithmaticException` due to division by zero. Therefore, the web container gracefully forwards it to the linked error page `Error.jsp` which displays the custom error message. Following is what you'll see in the browser when you access the page.



The include directive

This directive is used to include the response of another resource (JSP or html) at the point where the directive is placed in the current JSP. Its usage is shown below.

```
<%@ include file="file name" %>
```

See the code in listing 14.3.

Listing 14.3 (`IncludeDirectiveDemo.jsp`) JSP using include directive

```
<HTML>
<BODY>

    <h4> This is the response from the current JSP page</h4>
    <h3> Following is the response from another JSP </h3>
    <hr/>
    <%@ include file="/jps/PageDirectiveDemo.jsp" %>
    <hr/>

</BODY>
</HTML>
```

Steps to run the JSP

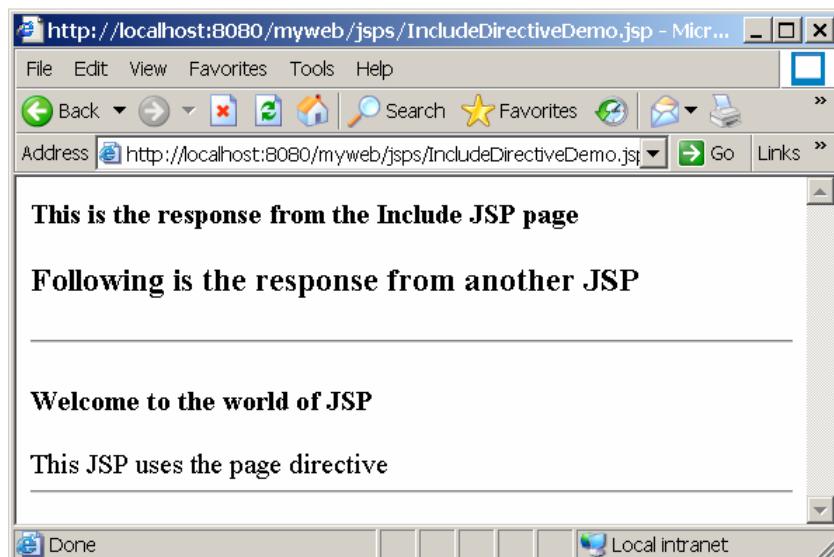
1. Save the code as

/myweb/jsp/IncludeDirectiveDemo.jsp

2. Start the server, and type the following URL in the browser

`http://localhost:8080/myweb/jsp/IncludeDirectiveDemo.jsp`

The above URL will produce the following output:



The taglib directive

This directive allows the JSP page to use custom tags written in Java. Custom tag definitions are usually defined in a separate file called as Tag Library Descriptor. For the current JSP to use a custom tag, it needs to import the tag library file which is why this directive is used. Following is how `taglib` directive is used.

```
<%@ taglib uri="location of definition file" prefix="prefix name" %>
```

Custom tags and tag libraries are explained in detail in the later pages and we'll see how to use this directive at that point. For now, don't worry about it.

Now that we know how and when to use JSP directives, let's see the usage of another type of placeholder.

JSP Declarations

JSP declarations are used to declare global variables and methods that can be used in the entire JSP page. A declaration block is enclosed within <%!> and <%> symbols as shown below:

```
<%!
    Variable declarations
    Global methods
%>
```

Take a look at the code in listing 14.4.

Listing 14.4 (JSPDeclarationDemo.jsp) JSP Declarations

```
<%@ page import = "java.util.Date" %>
<%!
    String getGreeting( String name){
        Date d = new Date();

        return "Hello " + name + "! It's "+ d + " and how are you doing today";
    }

%>

<h3> This is a JSP Declaration demo. The JSP invokes the
      global method to produce the following
</h3>

<hr/>

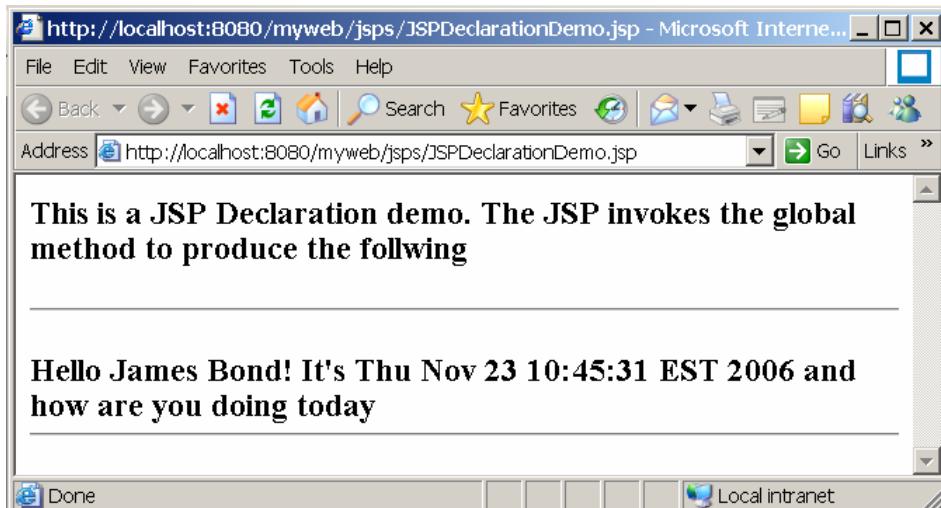
<h3> <%= getGreeting("James Bond") %>
<hr/>
```

Steps to run the JSP

1. Save the code as
`/myweb/jsp/JSPDeclarationDemo.jsp`
2. Start the server, and type the following URL in the browser:

`http://localhost:8080/myweb/jsp/JSPDeclarationDemo.jsp`

The above URL will produce the following output:



JSP Expressions

Expressions in JSP are used to display the dynamic content in the page. An expression could be a variable or a method that returns some data or anything that returns a value back. Expressions are enclosed in `<%=` and `%>` as shown below

`<%= Java Expression %>`

Take a look at the code in listing 14.5

Example 4: Look at the following JSP code

Listing 14.5 (JSPExpressionDemo.jsp) JSP expressions

```
<%!
String name="John Smith";
String address = "1111 S St, Lincoln, NE, USA";

String getMessage() {
    return "Your shipment has been sent to the following address.Thank you
           for shopping at BuyForLess.com";
}
%>

<h3> Your order is sucessfully processed. The confirmation number is 876876
```

```
</h3>
<hr/>
<h3> <%= getMessage() %> </h3>
<h4>
<%= name %>
<%= address %>
</h4> <hr/>
```

Steps to run the JSP

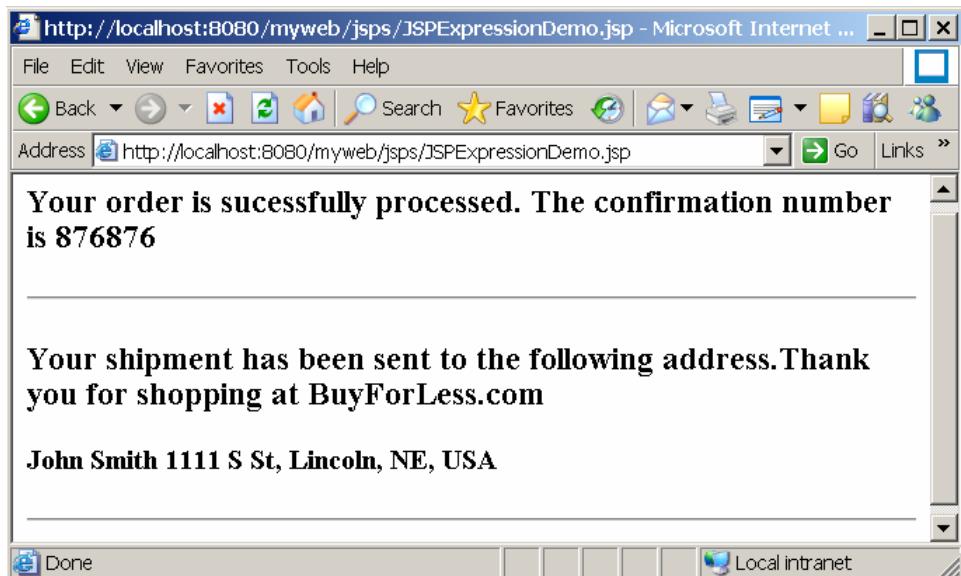
1. Save the code as

/myweb/jsp/JSPExpressionDemo.jsp

2. Start the server, and type the following URL in the browser:

<http://localhost:8080/myweb/jsp/JSPDeclarationDemo.jsp>

The above URL will produce the following output:



JSP Scriptlets

A Scriptlet is a piece of Java code that represents processing logic to generate and display the dynamic content where ever needed in the page. Scriptlets are enclosed

between <% and %> symbols. This is the most commonly used placeholder for Java code. Look at the code in listing 14.6.

Listing 14.6 (JSPScriptletDemo.jsp) JSP scriptlets

```
<HTML>
    <HEAD><TITLE>Tag - Methods</TITLE></HEAD>

    <h3> This is an example using Scriptlets </h3>

    <%
        int sum=0;
        for(int i=1;i<=100;i++) {
            sum+=i;
        }
    %>
    <hr/>

    The sum of first 100 numbers is <%= sum %>
    <hr/>
    <h3> Following is generated by the Loop

    <%
        for(int i=2;i<=5;i++) {

            if( i % 2 == 0){
                <br/><%=i %> is an even number

            } else{
                <br/><%= i %> is an odd number
            }
        } // End of for loop
    %>
</HTML>
```

Steps to run the JSP

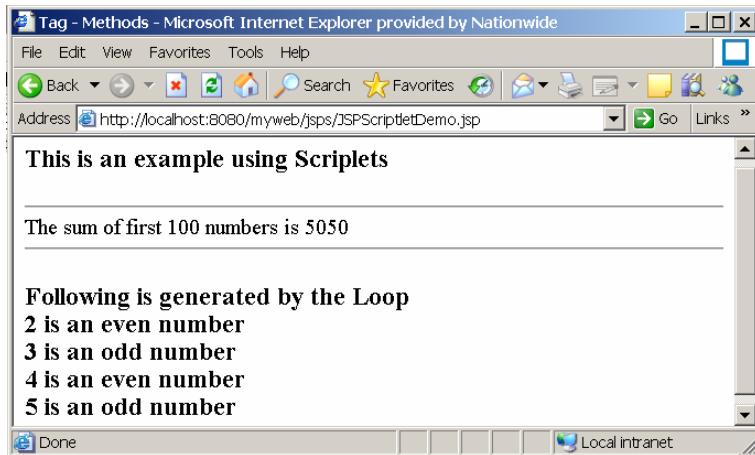
1. Save the code as

/myweb/jsp/JSPScriptletDemo.jsp

2. Start the server, and type the following URL in the browser:

<http://localhost:8080/myweb/jsp/JSPScriptletDemo.jsp>

The above URL will produce the following output:



The above four symbols are all the Java placeholders that we use in a typical JSP page. You can use them as you wish to build complex web pages. The most important one is the JSP scriptlet. Let me tell you a small trick here. When you write a scriptlet, follow the following two steps.

1. Write Java code and HTML without using <% and %>.
2. Once you finished writing the scriptlet code, then divide the Java code and html code using <% and %>. This makes life easy.

All the concepts we learned so far are very simple, right? This is why JSP is the coolest web technology. Remember we processed HTML forms using servlet? Let's see how JSP handles this aspect of web applications. Before we get to this, let's look at something called *Implicit Objects*.

Implicit Objects

As the name suggests every JSP page has some implicit objects that it can use without even declaring them. The JSP page can readily use them for several purposes. Following table lists the implicit objects.

Table 14.3 JSP Implicit Objects

Implicit Object	Description
request	This is an object of <code>HttpServletRequest</code> class. Used for reading request parameters (Widely used)
response	This is an object of <code>HttpServletResponse</code> class. Used for

	displaying the response content. This is almost never used by the JSP pages. So, don't worry about it.
session	This is an object of HttpSession class used for session management. (Widely Used)
application	This is an object of ServletContext. Used to share data by all Web applications. Rarely used.
out	This is an object of JspWriter. Similar to PrintWriter in Servlet. Rarely used.

Out of all the above implicit objects, only `request` and `session` objects are widely used in JSP pages. The `request` object is used for reading form data and `session` object is used for storing and retrieving data from session.

Let's see an example using `request` and `session` implicit objects. In this example, we will have html form post some data to a JSP page with reads the form parameters and stores them in the session. We will then write another JSP that reads the data from the session and displays them in the browser. Take a look at the code in listing 14.7

Listing 14.7 (Login.jsp) Simple login page

```
<html>
<body>

<form action="StoreData.jsp">
<table border=1>
<tr>
    <td> Login Name </td>
    <td> <input type="text" name="username" size="20"/></td>
</tr>
<tr>
    <td> Password </td>
    <td> <input type="password" name="password" size="20"/></td>
</tr>
<tr>
    <td> <input type="submit" value="Submit"/></td>
</tr>
</table>
</form>
</html>
```

Listing 14.7b (StoreData.jsp) JSP that stores the form data in session

```
<%@ page session="true" %>

<%
// Read the data from the request
String name = request.getParameter("username");
```

```
String pwd =    request.getParameter("password");  
  
// Store the data in the session  
session.setAttribute("userid",name);  
session.setAttribute("password",pwd);  
  
%>  
  
<h3> This page read the form data and stored it in the session.</h3>  
  
<a href="RetrieveData.jsp">Click Here</a> to go to the page that displays  
the data.
```

Listing 14.7c (ReadData.jsp) JSP that reads the form data in session

```
<%@ page session="true" %>  
  
<%  
    // Read the data in the session  
    String name = (String)session.getAttribute( "userid");  
    String pwd      = (String)session.getAttribute("password");  
  
%>  
  
<h3>  
The username is <%= name %> <br/>  
The password is <%= pwd %>
```

Look at the code in Login.jsp. This page simply has a html form that posts the data to StoreData.jsp as shown below:

```
<form action="StoreData.jsp">
```

The StoreData.jsp does two things listed below:

1. Read the form data using the `request` implicit object.

```
String name = request.getParameter("username");  
String pwd =    request.getParameter("password");
```

2. Store the form data in the session using the `session` implicit object.

```
session.setAttribute( "userid",name);  
session.setAttribute("password",pwd);
```

This page then provides a link to the RetrieveData.jsp.

The RetrievePage.jsp reads the data stored in the session as shown below and displays it.

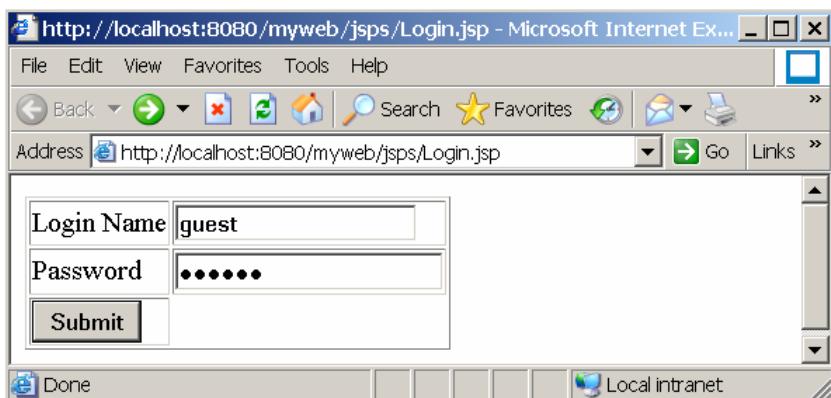
```
String name = (String)session.getAttribute( "userid");  
String pwd      = (String)session.getAttribute("password");
```

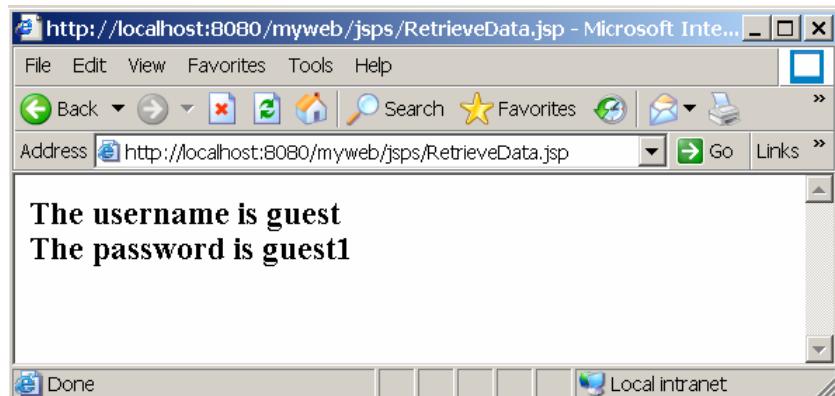
Steps to run the JSP

1. Save the code as
/myweb/jsp/Login.jsp
/myweb/jsp/SaveData.jsp
/myweb/jsp/ReadData.jsp
2. Start the server, and type the following URL in the browser:

<http://localhost:8080/myweb/jsp/Login.jsp>

The above URL will produce the following output:





All the examples we've seen until now demonstrate some of the basic capabilities of JSP. The real power of JSP is its ability to use Java beans and the properties of the beans. Let's see what this is all about.

Java Beans in JSP

One of the good practices while writing JSP is to isolate the presentation logic (HTML) from business logic (Java code). A typical JSP page should have as minimum business logic as possibly can. If this is the case, where should the business logic be? Good question. The business logic will be moved into external entities which will then be accessed from within JSP page. These external entities are nothing but Java beans.

A Java bean as we learned at the beginning of the book is a simple class with getters and setters. Using Java beans in JSP offers whole lot of flexibility and avoids duplicate business logic. JSP technology uses standard actions for working with beans. So, let's see how we can use beans in JSP and simplify our life.

Following are the three standard actions for working with Java beans:

1. <jsp:useBean>
2. <jsp:setProperty>
3. <jsp:getProperty>

Let's see the above standard actions one by one.

jsp:useBean

This action is used by the web container to instantiate a Java Bean or locate an existing bean. The web container then assigns the bean to an `id` which the JSP can use to work with it. The Java Bean is usually stored or retrieved in and from the specified scope. The syntax for this action is shown below:

```
<jsp:useBean id="bean name" class="class name" scope="scope name"/>
```

where,

`id` - A unique identifier that references the instance of the bean

`class` - Fully qualified name of the bean class

`scope` - The attribute that defines where the bean will be stored or retrieved from; can be `request` or `session` (widely used)

Consider the following declaration.

```
<jsp:useBean id = "cus" class="beans.Customer" scope="session" />
```

With the above declaration, following is what the web container does.

1. Tries to locate the bean with the name `cus` in `session` scope. If it finds one, it returns the bean to the JSP.
2. If the bean is not found, then the container instantiates a new bean, stores it in the session and returns it to the JSP.

If the scope is `session`, then the bean will be available to all the requests made by the client in the same session. If the scope is `request`, then the bean will only be available for the current request only.

jsp:setProperty

This action as the name suggests is used to populate the bean properties in the specified scope. Following is the syntax for this action.

```
<jsp:setProperty name ="bean name" property ="property name" value= "data" />
```

For instance, if we need to populate a bean whose property is `firstName` with a value `John` we use this action as shown below:

```
<jsp:setProperty name "cus" property ="firstName" value= "John" />
```

jsp:getProperty

This standard action is used to retrieve a specified property from a bean in a specified scope. Following is how we can use this action to retrieve the value of `firstName` property in a bean identified by `cus` in the `session` scope.

```
<jsp:getProperty name="cus" property="firstName" scope="session" />
```

Don't worry even if you are confused with the above actions. A quick example will clear all the confusions.

There are two common scenarios with using JavaBeans in JSP.

Scenario 1: JSP collects the data from the client, populate the bean's properties and stores the bean in `request` or `session` scope. This bean will then be used by another server side component to process the data.

Scenario 2: A Server side component loads a Java Bean with all the information and stores it in the `request` or `session`. The JSP will then retrieve the bean and displays the information to the client.

In scenario 1, JSP uses the bean to **collect** the data into it, while in scenario 2, it uses the bean to **read** the data from it to display.

The following example demonstrates scenario 1 in which a html form posts the data to a JSP page. The JSP page will then collect the data and store it in a Java Bean named `Customer` in `session` scope. It then provides a link to a servlet which retrieves the bean from the session and process the data in it. Fig 14.1 depicts this example.

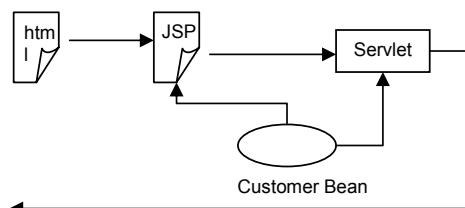


Fig 14.1 Scenario 1 of beans usage

Based on the above figure, look at the code in listing 14.8

Listing 14.8a (`CustomerData.html`) Simple html form that posts info to a JSP

```
<html>
  <head>
    <title> Customer Form </title>
  </head>
  <body>
    <h3>Please fill in the following details and submit it </h3>
    <br/>

    <form action="/myweb/jsp/CustomerInfoGatherer.jsp" method="GET">
      <table>
        <tr>
          <td>First Name</td>
          <td><input type="text" name="firstName" size=20/></td>
        </tr>
        <tr>
          <td>Middle Name</td>
          <td><input type="text" name="middleName" size=5/></td>
        </tr>
        <tr>
          <td>Last Name</td>
          <td><input type="text" name="lastName" size=20/></td>
        </tr>
        <tr>
          <td>Age</td>
          <td><input type="text" name="age" size=20/></td>
        </tr>
        <tr>
          <td> SSN</td>
          <td><input type="text" name="ssn" size=20/></td>
        </tr>
        <tr>
          <td> City</td>
          <td><input type="text" name="city" size=20/></td>
        </tr>
        <tr>
          <td>State</td>
          <td><input type="text" name="state" size=20/></td>
        </tr>
        <tr>
          <td>Country</td>
          <td><input type="text" name="country" size=20/></td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" name="Submit" size=20/></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

Listing 14.8b (CustomerInfoGatherer.jsp) JSP reading form data and populating a Java bean.

```
<HTML>
  <HEAD>
    <TITLE></TITLE>
```

```
</HEAD>
<BODY>
    <H3>Reading the form data</H3>

    <H3>Populating the bean and Storing in session </H3>

    <jsp:useBean id="userInfo" class="beans.Customer" scope="session"
        />

    <jsp:setProperty name="userInfo" property="firstName"
        value="<% request.getParameter("firstName") %>" />
    <jsp:setProperty name="userInfo" property="middleName"
        value="<% request.getParameter("middleName") %>" />
    <jsp:setProperty name="userInfo" property="lastName"
        value="<% request.getParameter("lastName") %>" />
    <jsp:setProperty name="userInfo" property="age"
        value="<% request.getParameter("age") %>" />
    <jsp:setProperty name="userInfo" property="ssn"
        value="<% request.getParameter("ssn") %>" />
    <jsp:setProperty name="userInfo" property="city"
        value="<% request.getParameter("city") %>" />
    <jsp:setProperty name="userInfo" property="state"
        value="<% request.getParameter("state") %>" />
    <jsp:setProperty name="userInfo" property="country"
        value="<% request.getParameter("country") %>" />

    <H3>Finished storing in the session </H3>

    <br/>
    <a href="/myweb/CustomerInfoProcessor" >Click Here </a> to invoke
        the servlet that processes the bean data in session.

</BODY>
</HTML>
```

Listing 14.8c (Customer.java) A simple customer bean.

```
package beans;
public class Customer implements java.io.Serializable {
    String firstName;
    String middleName;
    String lastName;
    String age;
    String ssn;
    String city;
    String state;
    String country;
    public String getAge() {
        return age;
    }
    public void setAge(String age) {
        this.age = age;
    }
    public String getCity() {
        return city;
    }
}
```

```
public void setCity(String city) {
    this.city = city;
}
public String getCountry() {
    return country;
}
public void setCountry(String country) {
    this.country = country;
}
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 String getMiddleName() {
    return middleName;
}
public void setMiddleName(String middleName) {
    this.middleName = middleName;
}
public String getSSN() {
    return ssn;
}
public void setSSN(String ssn) {
    this.ssn = ssn;
}
public String getState() {
    return state;
}
public void setState(String state) {
    this.state = state;
}
```

Listing 14.8d (*CustomerInfoGatherer.java*) Servlet processing a bean.

```
package myservlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

import beans.*;

public class CustomerInfoProcessor extends HttpServlet {

    String drivername;
    String databaseURL;

    public void init(ServletConfig config) throws ServletException {
        // This method is called before the following methods are called and
```

```
// gets called only ONCE. This is like a
// constructor. We do all the initialization here.

drivername = config.getInitParameter("driver");
databaseURL = config.getInitParameter("URL");

}

public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    String responseMsg = "";

    // Get the bean from session
    Customer customer = (Customer) req.getSession(true)
        .getAttribute("userInfo");

    String fName = customer.getFirstName();
    String mName = customer.getMiddleName();
    String lName = customer.getLastName();
    String age = customer.getAge();
    String ssn = customer.getSSN();
    String city = customer.getCity();
    String state = customer.getState();
    String country = customer.getCountry();

    // Remember in our database, age and ssn are integers. So lets convert
    // the strings to int and then store it to database
    int iage = 0;
    int issn = 0;
    try {
        iage = Integer.parseInt(age);
        issn = Integer.parseInt(ssn);

        // Call the method that inserts into database

        int status = insertCustomer(fName, mName, lName, iage, issn, city,
            state, country);

        if (status != -1) {
            responseMsg = "Successfully processed the form";
        } else {
            responseMsg = "Failed to process the form. Please try again";
        }
    } catch (Exception e) {
        responseMsg = "Error while Parsing Age and SSN. Make sure you enter
            numbers";
    }
    res.setContentType("text/html");

    PrintWriter pw = res.getWriter();

    // Send the response

    pw.println(fName + "      " + mName + "      " + lName + "      " + age +
```

```
        "      "+ ssn + "      " + city +
        "      " + state + "      " + country);
    pw.println("<br/><h3>" + responseMsg + "</h3>");
}

private int insertCustomer(String firstName, String middleName,
    String lastName, int age, int ssn, String city, String state,
    String country) {

    // Please write the JDBC logic to insert the data into the Customers
    // table. Take this as a home work.
    // This method should return an int that the executeUpdate methods
    // returns. Note: the driver name and the URL are already available in
    // the init()method.

    return 1;
}
}
```

Following is how the above code works:

The form in the `CustomerData.html` will post all the customer information to the `CustomerInfoGatherer.jsp`. This JSP will first create the `Customer` bean using the following action:

```
<jsp:useBean id="userInfo" class="beans.Customer" scope="session" />
```

The above action creates a customer bean with the id `userInfo` and stores it in the session. The class attribute must specify the fully qualified name of the bean class. Since the class is stored in `beans` package, the class attribute should be `beans.Customer`.

The customer bean simply defines all the fields (properties) along with the getters and setters for every property. The `CustomerInfoGatherer.jsp` after declaring the bean, will populate all the bean properties from the request parameters as shown below:

```
<jsp:setProperty name="userInfo" property="firstName"
    value="<% request.getParameter("firstName") %>" />
```

The above action will populate the `firstName` property in the bean with the value in the request parameter came from the html form. Let me ask you a question here. How does the web container use the above action and populate the actual bean property? Good question. Internally, the web container invokes the `setFirstName()` method and stores the data. This is why we need to define the setters in the bean class to set the property values. Similarly, we populate the rest of the bean properties. At this point the

customer bean is fully populated with the data and is available in the session. The JSP page will then provide a link to a servlet that processes the bean.

When the hyperlink is clicked, the web container invokes the `doGet()` method of the `CustomerInfoProcessor` servlet. This servlet first reads the customer bean from the session as shown below:

```
Customer customer = (Customer)req.getSession(true).getAttribute("userInfo");
```

Notice that the session attribute name must be `userInfo`, since this is the name the JSP defined for the bean. Also notice the casting we did. Since the object in the session belongs to `Customer` class, we need to cast it as shown. Once the bean is retrieved from the session, the servlet simply invokes the bean getters and get the bean data. From here onwards, I am sure you can understand what we did.

Steps to run the JSP

1. Save the html files in the following directory

```
/myweb/htmls/CustomerData.html  
/myweb/jsp/CustomerInfoGatherer.jsp
```

2. Save all the Java files in the following directories

```
/myweb/WEB-INF/classes/beans/Customer.java  
/myweb/WEB-INF/classes/myservlets/CustomerInfoProcessor.java
```

3. Compile the classes as shown below:

```
c:/>JavaTraining>..>..>classes>javac beans\*.java  
c:/>JavaTraining>..>..>classes>javac myservlets\*.java
```

4. Add the following snippet in the web.xml file

```
<!-- Servlet Definitions -- >  
  
<servlet>  
  <servlet-name>CustomerInfoProcessor</servlet-name>  
  <servlet-class>myservlets.CustomerInfoProcessor</servlet-class>  
  <init-param>  
    <param-name>driver</param-name>  
    <param-value>com.mysql.jdbc.Driver</param-value>  
  </init-param>  
  <init-param>  
    <param-name>URL</param-name>  
    <param-value>jdbc:mysql://localhost:3306/MDIT</param-value>  
  </init-param>  
</servlet>  
  
<!-- Servlet Mappings -- >
```

```
<servlet-mapping>
  <servlet-name>CustomerInfoProcessor</servlet-name>
  <url-pattern>/CustomerInfoProcessor</url-pattern>
</servlet-mapping>
```

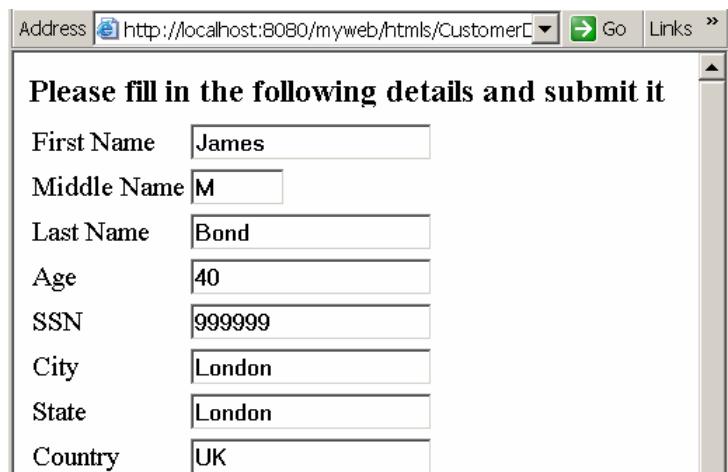
5. Start the server using the following command and wait for few seconds.

C:/>jakarta-tomcat-4.1.31>webapps>myweb>WEB-INF>classes>startup

6. Type the following URL in the browser

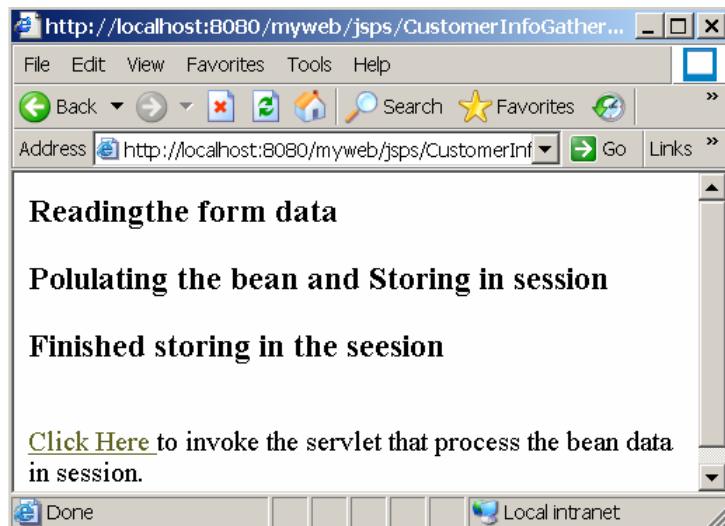
<http://localhost:8080/myweb/htmls/CustomerData.html>

The above URL will produces the following result.



A screenshot of a web browser window. The address bar shows the URL <http://localhost:8080/myweb/htmls/CustomerData.html>. The page content is a form titled "Please fill in the following details and submit it". The form fields and their values are:

First Name	James
Middle Name	M
Last Name	Bond
Age	40
SSN	999999
City	London
State	London
Country	UK



For a second, go back and observe the code in `CustomerInfoGatherer.jsp`. The bulk of the code is occupied with the `setProperty` actions, right? What if my html form has 100 fields? Should I write 100 `setProperty` actions in my JSP? No if you follow a simple trick. The trick is:

1. Make all the html forms field names same as bean property names as shown below:

In the html file : `<input type="text" name="firstName" size=100/>`

In the Customer bean class: `String firstName;`

2. Use the following one and only one `setProperty` action with the property attribute set to asterisk(*) to populate all the form data in all the bean properties in one step as shown below:

```
<jsp:setProperty name="userInfo" property="*" />
```

That's it. All the form data will be stored in the bean in one go. In this example, we already implemented Step 1 (matching names and properties). So, let's remove all the setProperty actions and replace all of them with the above setproperty action. The updated `CustomerInfoGatherer.jsp` will now look as shown in listing 14.8e.

Listing 14.8e (`CustomerInfoGatherer.jsp`) Updated JSP

```
<HTML>
  <HEAD>
    <TITLE></TITLE>
  </HEAD>

  <BODY>
    <H3>Reading the form data</H3>

    <H3>Populating the bean and Storing in session </H3>

    <jsp:useBean id="userInfo" class="beans.Customer"
      scope="session" />

    <jsp:setProperty name="userInfo" property="*" />

    <H3>Finished storing in the session </H3>

    <br/>
    <a href="/myweb/CustomerInfoProcessor" >Click Here </a>
    to invoke the servlet that process the bean data in session.

  </BODY>
</HTML>
```

Run the example again, and you'll notice the same result. Isn't this cool? I am sure it is. This is why JSP is really a cool technology. Let's now look at scenario 2 of JavaBeans application.

In this scenario, we will have a JSP that sends customer name to a servlet. The servlet based on the name, will create, populate and store the bean in the session. The servlet then forwards the request to a JSP page which retrieves the bean from the session and displays it to the client. Fig 14.2 depicts this.

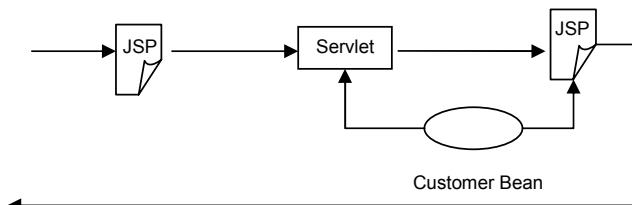


Fig 14.2 Scenario 2 of beans usage

See the listing in 14.9 to implement the above scenario.

Example 9: Look at the following Code

Listing 14.9a (`Search.jsp`) A simple search page

```

<html>
<body>

<form action="/myweb/SearchServlet">

<table border=1>
  <tr>
    <td> Customer Name </td>
    <td> <input type="text" name="customername" size="20"/></td>
  </tr>
  <tr>
    <td> <input type="submit" value="Submit"/></td>
  </tr>
</table>
</form>
</html>
  
```

Listing 14.9b (`SearchServlet.java`) Servlet creating a bean and storing it in session

```

package myservlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

import beans.*;

public class SearchServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Read the customer name
        String customerName = req.getParameter("customername");

        Customer customer = getCustomer(customerName);
  
```

```
// store it in the session
req.getSession(true).setAttribute("searchresult", customer);

// Forward it to a SearchResult page

RequestDispatcher rd =
    req.getRequestDispatcher("/jsps/SearchResult.jsp");
rd.forward(req, res);

}

private Customer getCustomer(String customerName) {

    // This is where you need to write the JDBC logic for the following SQL
    // SELECT * from Customers where LASTNAME = ?
    // Then use preprepared stament to set the customername in the place of
    //
    // Refer to the JDBC chapter that has this example
    // Finally return a Customer object
    // For now Iam mockin two customers

    Customer customer = null;

    if (customerName.trim().length() > 0
        && customerName.equalsIgnoreCase("john")) {

        customer = new Customer();

        customer.setFirstName("John");
        customer.setMiddleName("M");
        customer.setLastName("Smith");
        customer.setAge("20");
        customer.setSsn("324324");
        customer.setCity("Lincoln");
        customer.setState("NE");
        customer.setCountry("USA");
    } else if (customerName.trim().length() > 0
        && customerName.equalsIgnoreCase("James")) {

        customer = new Customer();

        customer.setFirstName("James");
        customer.setMiddleName("P");
        customer.setLastName("Bond");
        customer.setAge("40");
        customer.setSsn("999999");
        customer.setCity("London");
        customer.setState("London");
        customer.setCountry("UK");
    }

    return customer;
}
}
```

Listing 14.9c (SearchResult.java) Search result JSP page

```
<html>
<body>

<jsp:useBean id="searchresult" class="beans.Customer" scope="session" />

<table border=1>
<tr>
    <td> First Name </td>
    <td><jsp:getProperty name="searchresult" property="firstName" /></td>
</tr>
<tr>
    <td> Middle Name </td>
    <td><jsp:getProperty name="searchresult" property="middleName" /></td>
</tr>
<tr>
    <td> Last Name </td>
    <td><jsp:getProperty name="searchresult" property="lastName" /></td>
</tr>
<tr>
    <td> Age </td>
    <td><jsp:getProperty name="searchresult" property="age" /></td>
</tr>
<tr>
    <td> SSN </td>
    <td><jsp:getProperty name="searchresult" property="ssn" /></td>
</tr>
<tr>
    <td> City </td>
    <td><jsp:getProperty name="searchresult" property="city" /></td>
</tr>
<tr>
    <td> State </td>
    <td><jsp:getProperty name="searchresult" property="state" /></td>
</tr>
<tr>
    <td> Country </td>
    <td><jsp:getProperty name="searchresult" property="country" /></td>
</tr>

</table>
</body>
</html>
```

Following is how the above code works.

The Search.jsp page will post the customer name to SearchServlet. This servlet will read the customer name and builds the customer object by calling the getCustomer() method. Ideally, this method should use the JDBC logic and retrieve the customer data from a database. Take this as a home work. For now, this method mocked two customer objects whose names are James and John and returns the

appropriate customer object based on name. The servlet then stores the returned customer object with the name `searchresult` in the session as shown below:

```
req.getSession(true).setAttribute("searchresult",customer);
```

Finally, it forwards the request to a `SearchResult.jsp` to display the search results. If you look at the `SearchResult.jsp` page, it first retrieves the customer bean from the session as shown below:

```
<jsp:useBean id="searchresult" class="beans.Customer" scope="session" />
```

Notice that the `id` must be the same name that the servlet used to store the bean in the session. Also, the `scope` attribute must be set to `session` as this is where our servlet stored the bean. Once it gets the bean, it uses the `getProperty` actions as shown below to display the properties of the bean.

```
<tr>
    <td> First Name </td>
    <td><jsp:getProperty name="searchresult" property="firstName" /></td>
</tr>
```

Steps to run the JSP

1. Save the html files in the following directory

```
/myweb/jsp/
```

```
/myweb/jsp/Search.jsp
```

```
/myweb/jsp/SearchResult.jsp
```

2. Save all the Java files in the following directories

```
/myweb/WEB-INF/classes/myservlets/SearchServlet.java
```

3. Compile the classes as shown below:

```
c:/>JavaTraining>..>..>classes>javac myservlets\*.java
```

4. Add the following snippet in the web.xml file

```
<!-- Servlet Definitions -->

<servlet>
    <servlet-name>SearchServlet</servlet-name>
    <servlet-class>myservlets. SearchServlet </servlet-class>
    <init-param>
        <param-name>driver</param-name>
        <param-value>com.mysql.jdbc.Driver</param-value>
    </init-param>
```

```
<init-param>
    <param-name>URL</param-name>
    <param-value>jdbc:mysql://localhost:3306/MDIT</param-value>
</init-param>
</servlet>

<!--  Servlet Mappings  -- >

<servlet-mapping>
    <servlet-name>SearchServlet </servlet-name>
    <url-pattern>/SearchServlet </url-pattern>
</servlet-mapping>
```

5. Start the server using the following command and wait for few seconds.

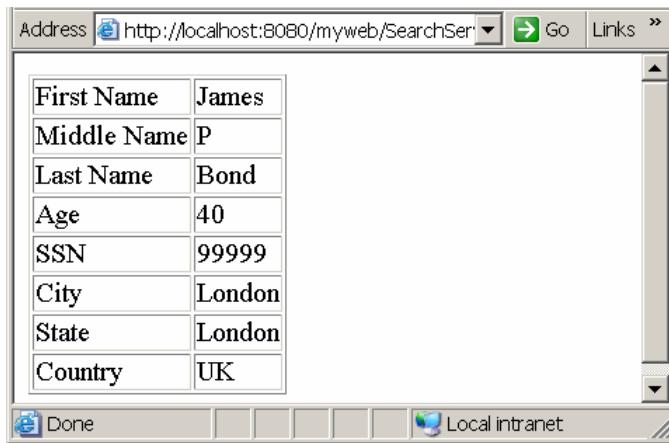
C:/>jakarta-tomcat-4.1.31>webapps>myweb>WEB-INF>classes>startup

6. Type the following URL in the browser

<http://localhost:8080/myweb/jsp/Search.jsp>

The above URL will produce the following output:





In the previous two examples if you notice the customer bean class, all it has is getters and setters for storing and retrieving the data. Such beans are one side of the coin. The other side of the coin is the beans that encapsulate business logic. This is where the real power of JavaBeans is exposed. To demonstrate this, let's look at an example.

Let's write an example with and without using the beans and you'll notice the beauty of beans. See the code in listing 14.10a which doesn't use the beans.

Listing 14.10a (`BadCalculator.jsp`) Simple JSP without beans.

```
<HTML>
<%
    int sum=0;
    for(int i=1;i<=100;i++)  {
        sum+=i;
    }
%>
<h3> The sum of first 100 numbers is <%= sum %> </h3>
</HTML>
```

If you look at the above code, the JSP uses a scriptlet to compute the sum of 100 numbers, which is good. The above code suffers from one serious drawback not in terms of functionality but in terms readability of the code. If someone comes and gaze though the page, he will simply be stumped because he doesn't know what the heck is going on in the page. All this is because of the presence of the Java code in the JSP. A JSP is fundamentally a presentation component which should have presentation logic instead of business logic. Let's see how we can improve the code using beans. See listing 14.10b which uses beans.

Listing 14.10b (GoodCalculator.jsp) Simple JSP with beans.

```
<HTML>
<body>
<jsp:useBean id="calculator" class="beans.Calculator" />
<jsp:setProperty name="calculator" property="count" value="100" />
The sum of first <jsp:getProperty name="calculator" property="count" />
numbers is <jsp:getProperty name="calculator" property="sum" />
</body>
</HTML>
```

Listing 14.10c (Calculator.java) A calculator bean

```
package beans;

public class Calculator {

    String count;
    String sum;

    public String getCount() {
        return count;
    }
    public void setCount(String count) {
        this.count = count;
    }
    public String getSum() {

        int s = 0;
        int maxCount = Integer.parseInt(count);

        for (int i = 1; i <= maxCount; i++) {
            s += i;
        }

        sum = s + "";
    }
    return sum;
}
public void setSum(String sum) {
    this.sum = sum;
}
}
```

Look at the above calculator bean class. It defined two properties namely `count` and `sum` along with the getters and setters. The interesting piece is the `getSum()` method. This method uses the `count` variable, computes the sum and returns it to the JSP.

The JSP page will define the bean and set the `count` property at the first place. It then retrieves the `sum` property which computes the sum and displays the result. The

computing logic is now moved from JSP page to the bean method. The good thing with this implementation is that, if the JSP need to compute the sum of other numbers, it can simply do so as shown below:

Listing 14.10b (GoodCalculator.jsp) Simple JSP with beans.

```
<HTML>
<body>

<jsp:useBean id="calculator" class="beans.Calculator" />
<jsp:setProperty name="calculator" property="count" value="100" />
The sum of first <jsp:getProperty name="calculator" property="count" />
numbers is <jsp:getProperty name="calculator" property="sum" />
<jsp:setProperty name="calculator" property="count" value="150" />
The sum of first <jsp:getProperty name="calculator" property="count" />
numbers is <jsp:getProperty name="calculator" property="sum" />

</body>
</HTML>
```

If you noticed the above JSP code, we completely avoided duplicate business logic, right? This is another beauty of using beans. Start the server and type the following URL

<http://localhost:8080/myweb/jsp/GoodCalculator.jsp>

The above URL produces the following result:



This completes most of the important things you need to know about using JavaBeans in JSP pages. Notice the GoodCalculator.jsp code again. It eliminated duplicate business logic and at the same time improved the readability of code several times.

However, a careful insight into the code reveals another problem. This is duplicate HTML code as shown below highlighted in bold.

```
The sum of first <jsp:getProperty name="calculator" property="count" /> numbers is
<jsp:getProperty name="calculator" property="sum" />
```

We eliminated duplicate Java code using JavaBeans. Now you might be wondering if there is another solution to also eliminate the duplicate html content. Are we greedy here? If we do have one, then our JSP will be a killer and you know what, we have a solution. It's nothing but using custom tags. Let's see what custom tags are and how they aid in the development of simplified JSP pages. This is the last concept in this chapter.

Custom Tags

Custom Tags are introduced from JSP 1.1 specification. A custom tag is a user defined tag, which looks just like an XML tag in terms of representation, but conveys a special meaning to the web container when it comes across it. Custom tags are extremely powerful and offers a whole lot of flexibility. The notion of custom tag is such a big hit that you literally see them in almost every single JSP page in any real world J2EE based web application.

To write a custom tag and use it in a JSP page, we need to follow simple process. This is a very small price compared to the benefits it offers. So, without wasting any further time, let's get into the details of custom tag anatomy.

A custom tag like any standard html tag will have the following things:

1. A tag name
2. One or more tag attributes
3. Body Content
4. Nested tags

Following is how a typical custom tag looks like.

```
<bean:write name="some name" property="some prop">
    This is the body content
</bean:write>
```

In the above custom tag, `write` is the name of the tag and `name` and `property` are the attributes of the tag. With custom tags it's very important that every tag that is opened **must** also be closed. Failing to do so will result in a JSP translation error.

Following is the standard process for writing a custom tag.

1. Write a Tag class
2. Define the tag class in a tag library descriptor
3. Import the tag library descriptor into JSP page
4. Finally, use the tag.

Just remember the above process even if you don't understand the custom tags. You'll be good. Let's now see the details of above steps one by one.

Step 1: Writing a Tag class

A custom tag can be viewed as a "representative" of some Java code in a JSP page. To write a custom tag, we need to write a Java class using the standard tag extension API. This API has few built-in classes that we need to use to write the Tag class. Trust me; it's very easy to write this class.

To write a basic custom tag,

1. Write a class that extends TagSupport class.
2. Declare one instance variable per attribute of the tag, and define a getter and setter method.
3. Overwrite two methods namely doStartTag() and doEndTag().

Using the above process, let's write the following custom tag that displays the sum of numbers.

```
<calc:sum count="100"/>
```

The web container should compile the above tag and produce the following html markup. This is our first goal.

```
<h3>The Sum of 100 numbers is 5050</h3>
```

See listing 14.11a that's shown a tag class for the above tag.

Listing 14.11a (CalculatorTag.java) A simple tag class

```
package customtags;  
  
import java.io.IOException;  
import javax.servlet.jsp.JspException;  
import javax.servlet.jsp.tagext.TagSupport;
```

```
import beans.*;

public class CalculatorTag extends TagSupport {

    // This method will be called when the JSP encounters the start of the
    // tag implemented by this class.

    String count;

    public String getCount() {
        return count;
    }

    public void setCount(String count) {
        this.count = count;
    }

    public int doStartTag() throws JspException {

        // This means the JSP must evaluate the contents of any child tags
        // in this tag;

        return EVAL_BODY_INCLUDE;
    }

    // This method is called when the JSP encounters the end of the tag
    // implemented by this class

    public int doEndTag() throws JspException {

        // Use the bean that we already have to calculate the sum.
        Calculator calc = new Calculator();
        calc.setCount(count);
        String sum = calc.getSum();

        try {
            pageContext.getOut().write(
                "The Sum of first " + count + " numbers is " + sum);
        } catch (IOException e) {

            throw new JspException(
                "Fatal Error: HelloTag could'nt write to the JSP out");
        }

        // This return type tells the JSP page to continue processing
        // the rest of the page
        return EVAL_PAGE;
    }
}
```

If you observe the above code, we created a class named `CalculatorTag` that inherits from `TagSupport` class, a built-in class that supports custom tags development (Rule 1). Since our custom tag takes `count` as an attribute, we defined a property (instance variable) named `count` and implemented both the getter and setter method (Rule 2). We then overwrote two methods namely `doStartTag()` and `doEndTag()` (Rule 3).

When the web container comes across the custom tag in the JSP, it follows a systematic approach. It first calls all the setters and assigns the attribute values to the tag class properties. It then calls the `doStartTag()` method. The `doStartTag()` method is usually empty and always returns `EVAL_BODY_INCLUDE`. Where did this come from? Don't worry. This is a constant defined in the parent class. Returning this constant means, "*Hey container, please evaluate the body content if there is some, otherwise go ahead with the next step*".

Since our tag doesn't have any body content, the web container gracefully goes to the next step which is the invocation of `doEndTag()` method. Entering in to this method means, the web container is at the end of the tag. This is where we need to do whatever we want to have the custom tag render the desired html to JSP page. If you notice the implementation of this method, it first uses the `Calculator` bean class, and computes the sum as shown below:

```
calculator calc = new Calculator();
calc.setCount(count);
String sum = calc.getSum();
```

Once the sum is computed, it should return it back to the JSP in proper format. For the custom tag to return anything to the JSP, it needs to use the `pageContext` object inherited from the parent class and use the `write()` method as shown below:

```
pageContext.getOut().write("<h3>The Sum of first "+count+" numbers is " + sum +
"</h3>");
```

Now that it finished sending the desired html markup to the JSP, it must return the constant `EVAL_PAGE`. This means, "*Hey container, I am done processing the tag, so please go ahead with the rest of the page*". That's it guys. We finished writing the tag class. Save the above file in the following directory

/myweb/WEB-INF/classes/customtags/CalculatorTag.java

Step 2: Define the tag class in the tag library descriptor.

This is the most important step. A tag library descriptor is a standard XML file that defines all the details of the custom tag that the web container needs to process it. It defines the following information:

1. Name of tag
2. Class of the tag
3. Attributes of the tag

See the following tag library descriptor for our custom tag shown below:

```
<calc:sum count="100"/>
```

Listing 14.11b (mytags.tld) The custom tag library descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>examples</shortname>
    <info>Simple Library</info>
    <tag>
        <name>sum</name>
        <tagclass>customtags.CalculatorTag</tagclass>
        <bodycontent>JSP</bodycontent>
        <attribute>
            <name>count</name>
            <required>true</required>
            <rteprvalue>false</rteprvalue>
        </attribute>
    </tag>
</taglib>
```

The only important element in the above XML is the one highlighted in bold. The elements like `tlibverion`, `jspversion`, `shortname` and `info` are used to convey some basic information to the container. These are the required elements. The important one is the `tag` element which defined the name, the tag class, and the list of attributes the tag will have. The above code is pretty obvious and conveys a clear picture.

This file though contains xml content, must be saved to a file with a ".tld" extension. Save this file in the following directory as:

/myweb/WEB-INF/tlds/mytags.tld

Step 3: Importing the TLD file into the JSP page

Remember we talked about JSP directives at the beginning and delegated the usage if third directive named `taglib` to later section. Now is the time to use this. This directive is used to import the TLD file into the JSP page as shown below:

```
<%@ taglib uri="/WEB-INF/tlds/mytags.tld" prefix="calc" %>
```

The above directive will tell the web container something like, *"Hey container, this JSP uses custom tags that are defined in mytags.tld file which has all the information to process the*

tags. Moreover, all the tags defined by this TLD file will be prefixed with `calc`" as shown below:

```
<calc:sum count="100"/>
```

Step 4: Using the custom tag in the JSP page.

There is nothing to explain here guys. Simply use the above custom tag in the JSP page. Look at the following code that covers both Step 3 and 4.

Listing 14.11c (`CalculatorTag.jsp`) JSP using a custom tag

```
<%@ taglib uri="/WEB-INF/tlds/mytags.tld" prefix="calc" %>  
<calc:sum count="100" />
```

Steps to run the JSP

1. Save all files in the following directories

```
/myweb/jsp/CalculatorTag.jsp  
/myweb/WEB-INF/classes/customtags/CalculatorTag.java  
/myweb/WEB-INF/classes/tlds/mytags.tld
```

2. Compile the classes as shown below:

```
C:/>JavaTraining>..>..>classes>javac customtags\*.java
```

3. Start the server using the following command and wait for few seconds.

```
C:/>jakarta-tomcat-4.1.31>webapps>myweb>WEB-INF>classes>startup
```

4. Type the following URL in the browser

```
http://localhost:8080/myweb/jsp/CalculatorTag.jsp
```

The above URL produces the following result:

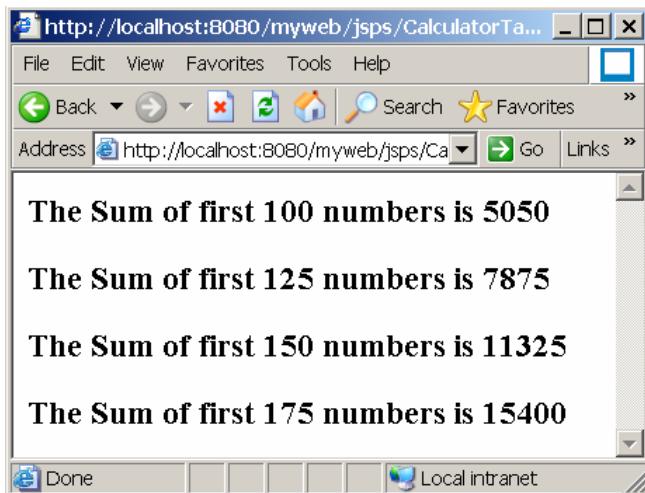


You can now update the JSP to use the custom tag again and again as shown below:

```
<%@ taglib uri="/WEB-INF/tlds/mytags.tld" prefix="calc" %>

<calc:sum count="100" />
<calc:sum count="125" />
<calc:sum count="150" />
<calc:sum count="175" />
```

The above code produces the following result:



We can also specify the body content for a custom tag. The body content can either be **html** or **JSP** as shown below:

HTML body content

```
<calc:sum count="100">Hello John</calc:sum>
```

JSP body content

```
<calc:sum count="100"> Hello <%= name %> </calc:sum>
```

All you need to do is in the TLD file, specify the body content for the tag as html or JSP as shown below:

```
<tag>
  <name>sum</name>
  <tagclass>customtags.CalculatorTag</tagclass>
  <bodycontent>JSP</bodycontent>
</tag>
```

Recognizing the important of custom tags in JSP pages, to simplify our life from building custom tags from scratch, companies like SUN and Apache Software came up with technologies with rich set of ready made custom tags. These companies identified the most common custom tags that any JSP page requires and provided the implementations for the tags by themselves. Most of the real world applications uses these technologies and leverage the built-in custom tags to use in JSP pages. However, you are free to write your own custom tags just in case if the functionality you are expecting is not available with the built-in ones. In such cases, follow the above procedure. The two most popular technologies that had rich set of custom tags are **JSTL** which stands for Java Standard Tag Libraries and **Struts** created by Apache. These are our next chapters.

At this point, we finished learning all the important concepts in JSP technology. I am sure you really enjoyed JSP programming and learnt to use several cool features it offered. You can now use combination of Servlets and JSP to build any complex web application. Always remember that, simple concepts are easy to forget. So, read this chapter twice before you go to the next one, ok.

Summary

- ✓ JSP is a reverse technology to Servlets.
- ✓ In JSP technology, Java code is embedded in html code.
- ✓ JSP technology is built over servlet technology. The web container translates JSP into a Servlet
- ✓ JSP uses special placeholders for Java code.
- ✓ The JSP directives namely page, include and taglib defines the elements that effects the overall structure of the page.
- ✓ JSP declarations are used for declaring global methods and variables.
- ✓ Scriptlets are used for generating the dynamic content.

- ✓ JSP expressions are used for echoing the dynamic content.
- ✓ A JSP page can use the implicit objects namely request, out and session. These are the most widely used objects.
- ✓ JSP uses standard actions to work with JavaBeans.
- ✓ JavaBeans are used to isolate business logic from presentation logic.
- ✓ JSP uses standard actions to store and retrieve the data in and from the JavaBeans.
- ✓ JSP uses Custom tags to build html markup to be used anywhere in the page.
- ✓ JSTL and Struts are popular technologies that include several built-in custom tags.

Time to play 50-50

1. Which of the following directive is used for importing the resources?
 - a) include
 - b) page
2. Which of the following directive is used for importing the custom tags descriptor?
 - a) include
 - b) taglib
3. Which of the following page attribute allows session management?
 - a) session = "true"
 - b) scope="session"
4. Which of the following placeholder is used to define a scriptlet
 - a) <% %>
 - b) <%= %>
5. Which of the following standard action is used for defining a Java bean?
 - a) <jsp:getBean>
 - b) <jsp:useBean>
6. Which of the following is used to retrieve the bean property?

- a) <jsp:property>
b) <jsp:getProperty>
7. Which of the following is an implicit object?
 - a) request
 - b) HttpServletRequest
8. Which of the following directive is used to include the response of another resource?
 - a) getResource
 - b) include
9. Which of the following file is used to define custom tags?
 - a) TLD file
 - b) Text file
10. Which of the following tag technology is created by Sun?
 - a) Struts
 - b) JSTL

Interview Questions

Question: What are the implicit objects in a JSP page?

Answer: request, response, session, out, context

Question: What is the difference between include directive and standard action <jsp:include>

Answer: Directive inclusion provides static inclusion of content and <jsp:include> action provides dynamic inclusion of content. The content can be either HTML or JSP.

Question: What is a custom tag?

Answer: Custom tag is a user defined tag that conveys a special meaning to the container.

Question: List the steps involved in writing a custom tag

Answer: Refer to Custom tags section.

Question: Where do you configure custom tags?

Answer: Tag library descriptor, which is an xml file, but uses ".tld" extension.

Question: What is a scriptlet?

Answer: A scriptlet is a piece of Java code embedded in JSP page to generate the dynamic content.

Chapter 15

JSTL

This chapter introduces to a utility custom tag library called JSTL for building JSP pages. By the end of this chapter you'll get acquainted in using standard JSTL tags in JSP pages.

Chapter Goals

- ✓ Understanding JSTL
- ✓ JSTL Core Tags
- ✓ JSTL SQL Tags
- ✓ JSTL XML Tags

Environment Setup

1. Download JSTL library at the following link.

<http://people.apache.org/builds/jakarta-taglibs/nightly/>

2. Copy the following TLD files in JSTL installation directory (jakarta-taglibs\standard-1.0\tld) into the **myweb** project in Tomcat.

c.tld,c-rt.tld,fmt.tld,fmt-rt.tld,sql.tld,sql-rt.tld,x.tld,x-rt.tld into
/myweb/WEB-INF/tld directory

3. Copy the following two Jar files from JSTL installation directory (jakarta-taglibs\standard-1.0\lib) into **myweb** project

jstl.jar and standard.jar into
/myweb/WEB-INF/lib directory.

Introduction

This chapter introduces you to JSTL which stands for Java Standard Tag Library. In the last chapter we saw how to develop custom tags for JSP pages. Though we know how to build custom tags on our own, the only concern is that we have to develop them from scratch. Since we are in competitive world in which time is very precious, it would be really nice if we have ready made useful custom tags that we can use in our JSP pages without having to develop them. This is what JSTL is all about. It is a set of cool ready made custom tags.

JSTL provides us with custom tags that can be used for most of the common actions in JSP pages such as iteration, conditions etc. Using these tags has become a common practice in most of the real world applications due to the following reasons:

1. Simple to use
2. Enhances JSP page readability
3. Highly powerful and
4. Faster JSP development

This chapter gives you an idea about how to use different variety of JSTL tags in JSP pages. Let's see what these are.

JSTL tags are broadly classified into four groups as shown below:

1. Core Tags
2. SQL Tags
3. XML Tags
4. Format Tags

Out of the above, core tags are the widely used ones followed by SQL tags. Let me tell you one thing here. Each category has like 5-6 tags and are very simple to use and hardly takes few minutes to master them. What we will do is, learn all the tags in each category and then do an example using the same. So, without wasting any further time let's see what these are.

Core Tags

These tags are used for general purpose page actions like conditions, iterations, displaying and manipulating variables in specified scope etc. All the core tags will be prefixed with "c". Following are the important ones.

```
<c:out value="" default="" />
```

This tag will display whatever you pass into value attribute.

Example 1:

```
<c:out value="Hello World" /> prints "Hello World" in the browser.
```

Example 2:

```
<%
    String str="Cool";
    str += " JSTL";
%>
<c:out value="${str}" />
```

The above snippet prints the value in the scripting variable str which is Cool
JSTL

Use this tag wherever you want to display dynamic content. I know you are seriously wondering about what "\${ }" is for. We need to use this to display the value of a page variable or a bean property. The symbol \$ should be read as "*Value of variable or bean property*" and the variable/property name should be enclosed within curly braces. The default attribute specifies the default value to be displayed if the value attribute evaluates to null.

Example 3:

```
<c:out value="${data}" default="NO DATA" />
```

In the above tag, if the value of data variable is null, then it displays NO DATA in the browser.

```
<c:set var="" value="" target="" property=""
      scope="request/session" />
```

This tag is used to store data in the specified scope as name-value pair. Usually we store the data in request or session scope.

Example 4:

To store a value something like Steve in session with the key user, we can use this tag as shown below:

```
<c:set var="user" value="John" scope="session" />
```

Example 5:

To store some value in a Java Bean property we use it as

```
<c:set target="customer" property="ssn" value="12345" scope="session" />
```

The above tag stores the value 12345 in the ssn property of customer bean in session.

```
<c:forEach var="" items="" begin="" end="" ></c:forEach>
```

This is the most widely used tag for iterating through a collection of objects. Look at the following implementation for this tag.

```
<c:forEach var="item" items="${list}" >
    <c:out value="${item}" />
</c:forEach>
```

The items attribute take the collection to iterate and assigns every object in the collection to item variable. We can then print the object using the c:out tag as shown above. The above tag should be read as, *for each object in list taken into item, display item.* The above tag is equivalent to the following Java code:

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

As you can see, the custom tag representation is more readable than the equivalent Java code representation. This is the beauty of custom tags.

```
<c:if test="" />
```

This tag is used to test a condition. If the condition is evaluated to true then the statements inside the tag will be evaluated.

Example 6:

```
<c:if test="${accountExists}" >
    Total Balance in the account is ${c:out value="${totalFunds}" />
</c:if>
```

The above tag is equivalent to the following Java code:

```
if ( accountExists == true){  
    System.out.println ("Total Balance in the account is $ " + totalFunds);  
}  
  
<c:choose></c:choose>
```

This tag is used in conjunction with `<c:if>` tag and `<c:when>` tag. This tag is normally used like a typical *if-else* block.

Example 7:

```
<c:choose>  
    <c:if test="${accountExists}">  
        Total Balance in the account is ${totalFunds} />  
    </c:if>  
    <c:otherwise>  
        Account not found.  
    </c:otherwise>  
</c:choose>
```

In the above snippet, if the account doesn't exist then the content enclosed in the `c:otherwise` tag will be evaluated.

```
<c:url var="" value="">  
<c:param name="" value="">
```

The above two tags are used to create a URL and append query parameters as shown below:

```
<c:url var="nextpage" value="http://localhost:8080/myweb/PagingServlet">  
    <c:param name="page" value="2"/>  
<c:url>
```

The above code snippet will produce the following URL and stores it in `nextpage` variable.

`http://localhost:8080/myweb/PagingServlet?page=2`

We can then use the above URL in a hyperlink element as shown below:

```
<a href='<c:out value="${nextpage}" />'> Click Here to go to next Page </a>
```

This completes all the important core tags. As I promised before, let's do an example using all the above core tags. See the code in listing 15.1.

Listing 15.1 (CoreTagsDemo.jsp) JSP using a JSTL core tags

```
<%@ taglib prefix="c" uri="/WEB-INF/tlds/c.tld" %>
<%@ taglib prefix="sql" uri="/WEB-INF/tlds/sql.tld" %>

Setting the value: <b>Hello World! </b> in a variable named <b>hello</b>
<c:set var="hello" value="Hello World!"%>

<br/>

The value in <b>hello</b> variable is :

<c:out value="${hello}"%>

<%
    java.util.List list = new java.util.ArrayList();

    list.add("Hello");
    list.add("Welcome");
    list.add("JSTL");

    pageContext.setAttribute("list", list);
%>
<br/>
Following are the names in a collection.
<br/>

<c:forEach var="name" items="${list}" %>
    <c:out value="${name}" /><br/>
</c:forEach>

<br>
Following is a demo of c:choose tag
<br/>

<c:set var="test" value="Hello" />

<c:choose>
    <c:when test="${test=='Hello'}">
        <c:out value="The value is Hello"/>
    </c:when>
    <c:otherwise>
        <c:out value="The value is not Hello"/>
    </c:otherwise>
</c:choose>
```

Steps to run the above JSP

1. Save the JSP in the following directory

/myweb/jsp/

2. Start the server and type the following URL

<http://localhost:8080/myweb/jsp/CoreTagsDemo.jsp>

The above URL produces the following output



SQL Tags

I am sure you already guessed what these tags have to offer. Good. The SQL tags as the name suggests are used for database operations. These tags allow the page developers to execute SQL queries from the page itself with minimum effort. All the SQL tags are usually prefixed with the word `sql`. Let's see the important ones here.

`<sql:setDataSource>`

Following is the syntax for this tag.

```
<sql:setDataSource var="" url="" driver="" username="" password="" />
```

This tag is used to establish a connection with the database with the given information. The connection object will be returned in a variable specified by the `var` attribute.

Example 1:

```
<sql:setDataSource var="myDatSrc" url="jdbc:mysql://localhost:3306/MyDB"
                   driver = "com.mysql.jdbc.Driver"/>
```

The above tag gets a connection to the MyDB database and stores it in `myDatSrc` variable. We can now use this variable to execute the SQL queries.

<sql:update>

This tag is used for executing all the SQL queries except the SELECT query. The query for this tag can be specified as the body content or through the `sql` attribute. The syntax for this tag is shown below:

```
<sql:update sql="" dataSource="" /> or  
<sql:update dataSource="">SQL </sql:update>
```

Use the first convention if the SQL is static and doesn't require setting any parameters dynamically as shown below:

```
<sql:update sql="INSERT INTO CUSTOMERS VALUES ('James', 'Bond')"  
           dataSource="${myDatSrc}"/>
```

If the parameters are required to be passed dynamically which is usually the case, we use the second convention as shown below

```
<sql:update dataSource="${myDatSrc}">  
    INSERT INTO CUSTOMERS VALUES (?,?)  
    <sql:param value="${firstName}" />  
    <sql:param value="${lastName}" />  
</sql:update>
```

The important thing with this convention is the way we set the query parameters. We need to use the `<sql:param>` tag with the `value` attribute as shown above. In our query we plugged in the `firstName` and `lastName` as two parameters.

<sql:query>

This tag is used to execute the SELECT queries. The syntax for this is shown below

```
<sql:query var="" sql="" dataSource="" />
```

Like the update tag, the select query can be supplied as an attribute or body content. This tag will then return the results object that we can iterate and display the results. Following snippets show both the cases.

```
<sql:query var="results" sql="Select * from Customers"  
           dataSource="${myDatSrc}"/>
```

(or)

```
<sql:query var="results" dataSource="${myDatSrc}">
    Select * from Customers where ssn=?
    <sql:param value="${ssn}" />
</sql:query>
```

Let us assume that the above SQL returned the results as shown in the following table.

FirstName	LastName	SSN
James	Bond	99999
Rob	Smith	12344

The `results` variable should be iterated as shown below to display the data.

```
<c:forEach var="record" items="${results.rows}">
    <c:out value="${record.firstName}">
    <c:out value="${record.lastName}">
    <c:out value="${record.ssn}">
<c:forEach>

<sql:transaction>
```

This tag is used to specify multiple queries in a single transaction. Its usage is pretty simple as shown below:

```
<sql:transaction>
    <sql:update sql="" dataSource="${myDatSrc}"/>
    <sql:update sql="" dataSource="${myDatSrc}"/>
</sql:transaction>
```

All we need to do is enclose all the queries that are to be part of the transaction in the body of the tag. This completes all the important SQL Tags. Let's do an example. See the code in listing 15.2.

Listing 15.2 (SQLTagsDemo.jsp) JSP using a JSTL sql tags

```
<%@ taglib prefix="sql" uri="/WEB-INF/tld/sql.tld"%>
<%@ taglib prefix="c" uri="/WEB-INF/tld/c.tld"%>

<%
    int count = 1;

    %
    count++;
    String fn = "James " + count;
    String ln = "Bond" + count;
```

```

pageContext.setAttribute("fn", fn);
pageContext.setAttribute("ln", ln);

%>

<sql:setDataSource var="myDatSrc" url="jdbc:mysql://localhost:3306/MyDB"
                   driver="com.mysql.jdbc.Driver" />

<sql:update dataSource="${myDatSrc}">
    INSERT INTO CUSTOMERS VALUES (?, 'V', ?, 20, 12345,
                                    'London', 'London', 'UK')
    <sql:param value="${fn}" />
    <sql:param value="${ln}" />
</sql:update>

<table border=1>

    <sql:query var="results" dataSource="${myDatSrc}">
        SELECT * from CUSTOMERS
    </sql:query>

    <c:forEach var="record" items="${results.rows}">
        <tr>
            <td><c:out value="${record.firstName}" /></td>
            <td><c:out value="${record.middlename}" /></td>
            <td><c:out value="${record.lastName}" /></td>
            <td><c:out value="${record.age}" /></td>
            <td><c:out value="${record.ssn}" /></td>
            <td><c:out value="${record.city}" /></td>
            <td><c:out value="${record.state}" /></td>
            <td><c:out value="${record.country}" /></td>
        </tr>
    </c:forEach>
</table>

```

The above JSP code first inserts a record and then displays all the records in the **Customers** table.

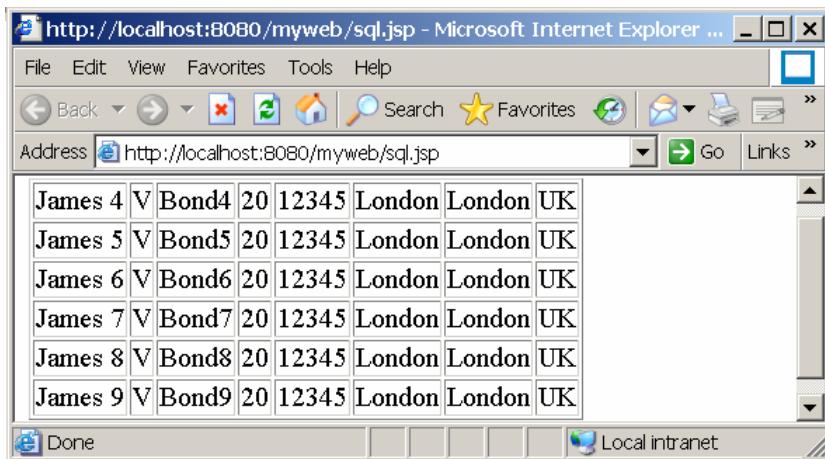
Steps to run the above JSP

1. Copy the MySQL driver Jar file into the following directory

/myweb/WEB-INF/lib
2. Save the code as “myweb/jsp/SQLTagsDemo.jsp”
3. Assuming that you already have the **Customers** table, start the server and type the following URL

<http://localhost:8080/myweb/jsp/SQLTagsDemo.jsp>

The above URL produces the following output



XML Tags

XML Tags are used for processing XML documents. These tags basically use the XPATH structure to retrieve the data from xml. XML tags are used for parsing the XML content and then displaying the nodal information. Most the XML tags except the `parse` tag are similar to core tags with just a small difference in the sense these tags use the attribute `select` instead of `value` or `items` attributes. Usually XML tags are prefixed with "x".

Following are important XML tags.

<x:parse>

This tag is used to parse the given XML document. The syntax for this tag is shown below:

```
<x:parse xml="xml source" var="variable name" />
```

The XML source can be the URI to an xml file on the server, or the variable name that contains the xml content. For instance, look at the following usage.

```
<x:parse xml="/xmls/Messages.xml" var="data" />
```

The above tag parses the `Messages.xml` file in the directory named `xmls`. If the xml content is stored in a page variable, then we can use this tag as shown below:

```
<x:parse xml="${xmlData}" var="data"/>
```

Once the XML parsed, it stores all the nodal structure with the reference variable specified by the var attribute (data). Using the reference variable, we can retrieve the values in various nodes.

<x:forEach>

This tag is used to iterate over the XML elements using the XPATH representation. Consider the following XML.

```
<response>
    <messages>
        <message>
            <text>This is message 1</text>
            <id>1234</id>
            <status>Processed</status>
        </message>
        <message>
            <text>This is message 2</text>
            <id>12345</id>
            <status>Processed</status>
        </message>
        <message>
            <text>This is message 3</text>
            <id>12346</id>
            <status>Failed</status>
        </message>
    </messages>
</response>
```

When the above XML is parsed and referenced with the variable data, we can use the **forEach** tag to display the contents of repetitive elements (message) as shown below:

```
<x:forEach var="element" select="$data/response/messages">
    <td> <x:out select="$element/text" /> </td>
    <td> <x:out select="$element/id" /> </td>
    <td> <x:out select="$element/status" /> </td>
<x:forEach>
```

<x:if>

This tag is used to evaluate the body of the tag based on the XPATH expression as shown below:

```
<x:forEach var="element" select="$data/response/messages">
    <x:if select="$element/status == Processed">
        <td> <x:out select="$element/text" /> </td>
    </x:if>
<x:forEach>
```

The above code only displays the processed messages. The other XML tags like `<x:when>`, `<x:choose>` work just the same way as the equivalent core tags. Let's write a quick example using XML Tags. See the code in listing 15.3.

Listing 15.3a (`Messages.xml`) Simple XML

```
<response>
    <messages>
        <message>
            <text>This is message 1</text>
            <id>1234</id>
            <status>Processed</status>
        </message>
        <message>
            <text>This is message 2</text>
            <id>12345</id>
            <status>Processed</status>
        </message>
        <message>
            <text>This is message 3</text>
            <id>12346</id>
            <status>Failed</status>
        </message>
    </messages>
</response>
```

Listing 15.3b (`XMLTagsDemo.jsp`) JSP using a JSTL xml tags

```
<%@ taglib prefix="c" uri="/WEB-INF/tld/c.tld"%>
<%@ taglib prefix="x" uri="/WEB-INF/tld/x.tld"%>

<c:url var="url" value="/myweb/xmls/Messages.xml" />
<x:parse xml=?${url}? var="data" />

<table>
    <x:forEach var="element" select="$data/response/messages">
        <tr>
            <x:if select="$element/status= Processed ">
                <td> <x:out select="$element/text" /> </td>
            </x:if>
        </tr>
    <x:forEach>
</table>
```

Steps to run the above JSP

1. Copy the XPATH driver Jar files into the following directory

/myweb/WEB-INF/lib

2. Save the XML as “/myweb/xmls/messages.xml”

3. Save the JSP code as “myweb/jsp/XTMLTagsDemo.jsp”
4. Start the server and type the following URL

`http://localhost:8080/myweb/jsp/XTMLTagsDemo.jsp`

This completes all the JSTL tags that we need to know for JSP page development. JSTL tags are extensively used in real world applications and using them makes the page look much cleaner and simple. I am sure you found the JSTL tags very helpful and easy to use them in JSP pages. Just read the core tags twice, since these are the widely used ones. Let's summarize this chapter.

Summary

- ✓ JSTL is a standard library of Custom tags that can be readily used in JSP pages without building from scratch.
- ✓ JSTL tags are classified as Core tags, SQL Tags, XML tags and Format Tags.
- ✓ The most frequently used ones are the core tags. These tags provide the basic actions in JSP such as iterations, conditions etc.
- ✓ SQL tags are used for working with databases by executing SQL queries.
- ✓ XML tags are used for processing xml content. These tags use XPATH syntax for retrieving the XML content.

Interview Questions

Question: What the different types of JSTL tags?

Answer: Core Tags, SQL Tags, XML Tags and Format Tags.

Chapter 16

Struts

This chapter introduces to the most popular and widely used web technology called Struts. This is an open source framework developed by Apache Software foundation. By the end of this chapter, you'll understand the important features of Struts and how web applications are built using it.

Chapter Goals

- ✓ Understand MVC Design Pattern
- ✓ Using Struts components.
- ✓ Using Struts Validator
- ✓ Using Struts Tiles
- ✓ Internationalization using Struts

Environment Setup

1. Download the Struts framework at the following URL.

<http://struts.apache.org/download.cgi>

2. Copy all the Jar files in the Struts installation directory (`jakarta-struts\lib*.jar`) to the following “lib” directory in your web project.

`/myweb/WEB-INF/lib`

3. Create the following directories to store Struts components.

`/myweb/jsp/struts, /myweb/WEB-INF/classes/formbeans
/myweb/WEB-INF/classes/actions, /myweb/WEB-INF/classes/properties`

Introduction

Struts is an open source framework developed by Apache Software foundation for building dynamic web applications. What, one more framework to build web applications? Thankfully Yes. The reason I used the word ‘thankfully’ is because this is such a good framework and makes life much easier with web application development. You'll agree with me by the end of this chapter.

Generally the term framework means some kind of ready made infrastructure based on a very good design using which one can build web applications at faster pace and that are flexible and easy to maintain. Struts is one such framework for building web applications in Java. A framework typically doesn't reinvent the wheel, but it uses a good design to build a better and reliable wheel. What I mean to say is, a framework doesn't introduce you to new type of programs, but it will use the same old technologies, but in a different style. By following its style (process) we can build some cool web applications. Struts framework also has its own style (process) based on a design called MVC and we follow this process to build robust dynamic web applications in Java.

In earlier chapters we looked at Servlet and JSP technologies for building web applications. The good thing is that struts framework is built using same technologies but uses a different approach. Therefore, we will still continue to write the same old JSP pages, simple Java programs etc, but using its new approach or process. For instance, we will write a Java program that is derived from so and so class, store it in some directory, then define the class in an XML file and store it so and so directory and so on. The process that struts follow is very simple. You'll like it, because it makes life easy.

Two important to remember here:

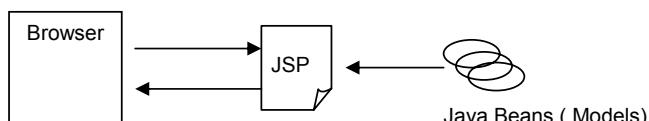
Point number one: Struts framework comes with several built in classes and interfaces. However, we don't need to know about all of them. But there are few classes that we need to use to build web applications. They are very simple, trust me. We'll see what these classes are, and how they help us and all that good stuff later.

Point number two: Like there is woman behind a successful man, there is always a good design behind a successful framework.

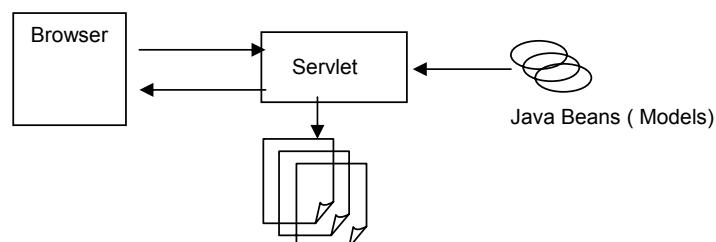
The design behind Struts is the popular design pattern, “The Model View Controller” a.k.a MVC. Let's see what MVC design is before learning something about Struts

Model View Controller Design

When we learned Servlets and JSP technology, unknowingly we used two types of design processes namely Model 1 design and Model 2 design. To better understand these two designs, look at the following two figures.



Model View Design (MVC 1)



Model View Controller Design (MVC 2)

Fig 16.1 MVC1 and MVC2 designs

The top figure shows the Model 1 design in which a JSP (View) takes the requests from the browser, processes the request using the Java Beans (Models) and sends the response back to the browser. In this design JSP acts as both the controller component as well as the view component. As a controller, it intercepts and processes the requests by dispatching to helper classes, and as a view it also displays the information to the user. This design is referred to as MVC1 design.

Now look at the second figure which shows the Model 2 design. I am sure you already guessed how the design works. The browser can no longer send requests to JSP pages. It can only send requests to a servlet, which will use its own expertise or helper classes to process the request. Once all the data is processed and models are prepared to encapsulate all the information, it will select the appropriate JSP page to display the information in the browser. In this design, all we did is cleanly separated controller duties from the view component into a servlet. Moreover, there will only be one controller servlet in this design and this why we say that the controller as a centralized component. This design is popularly known as MVC2.

The only difference between the two designs is that, in MVC1 design, JSP acts as both controller and view, and in MVC2 design, a Servlet acts as a centralized controller and JSP acts as a View component. This is an interview question.

To summarize, following are the key components of MVC2 design.

Model: An entity that stores the data (A Java Bean, a POJO - Plain Old Java Object)

View: A component that sends queries or requests to update the model data, or to read the model data.

Controller: A centralized component that reads the requests from the browser or a view. It acts as a single point of entry and exit to/from the application. It manages all the interactions within the application by coordinating with other components. Its main duties are:

- 1) Processing the requests and preparing the models
- 2) Select the View to display the models.

In MVC2 design, there will be "n" number of models to store different variety of data, "n" number of Views that queries or updates different models, and "1 and only 1" Controller. Note this point. This is the design that Struts framework use to build dynamic web applications in Java. Now that we know what this design is and how it works, let's get into serious Struts business.

Why Struts is important to us?

Here are the few reasons why we are interested in Struts framework.

1. Allows building web applications that are flexible and easy to maintain.
2. Follows a systematic process to build applications.
3. Comes with several cool and ready made custom tags to be used in JSP pages.
4. Uses XML for configuring the applications.
5. It has excellent support and ready made infrastructure for data validation.
6. It comes with Tiles framework that we can leverage to build composite pages with flexible page layouts.
7. It has inherent support for internationalization making web applications run in different locales.
8. Excellent support for error handling.

With all the above good things that Struts framework has to offer, I am sure you can't wait to learn Struts. The above features are one of the reasons why Struts has tasted instant success and became a de facto standard for all the real world web applications built in Java.

How Struts work?

Look at the following figure.

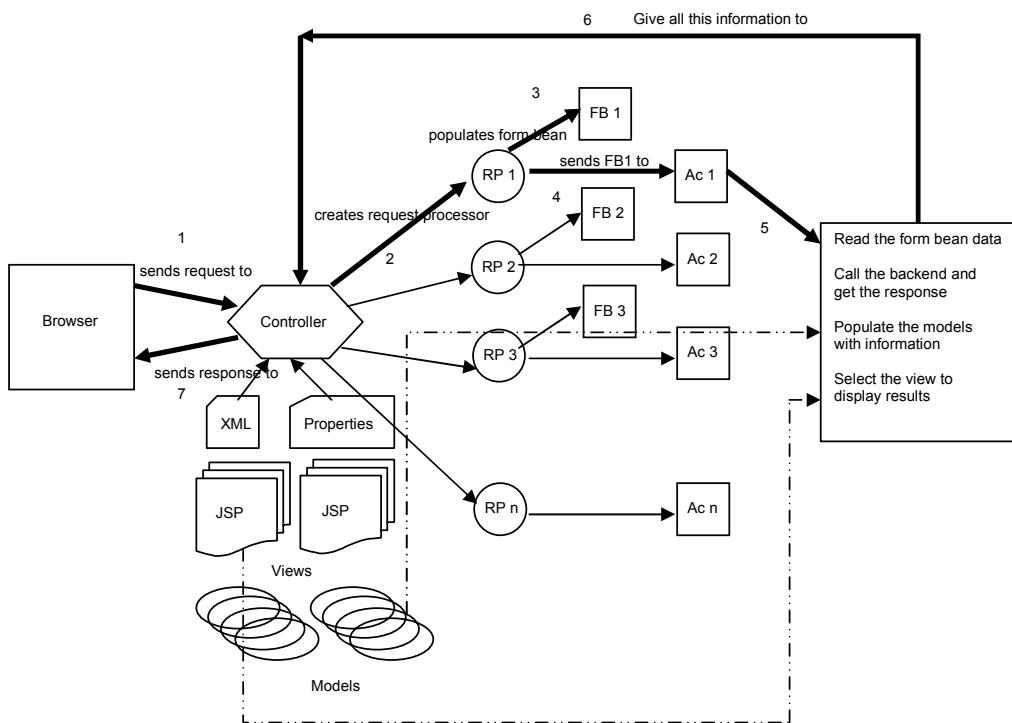


Fig 16.2 Struts flow of control

I hope the above picture is clear and give you some idea about how Struts work. Follow the bolded arrows starting from 1, and read the text in the boxes and along the arrows. If you read carefully, this is how the story builds up:

1. Browser sends request to Controller (ActionServlet class)
2. Controller loads the information related with request from the XML file and

creates a RequestProcessor by passing the information to it (RP 1)

3. Based on the information the request processor (RP1) got from controller, it does two things.

- 1) Read the data in the request and populate the Form Bean (FB1)
- 2) Send the form bean as a parameter to Action class (Ac 1)

4. Action class after receiving the form bean, does the following

- 1) Read the data from the Form bean
- 2) Process the data, get the response and
- 3) Populate the models (Java Beans) and store them in request scope or session scope.
- 4) Select the JSP that will use the models.

5. The action class will supply all the information to the Controller which will then use the selected JSP to send the response to the browser.

In the above figure, some components are ready made and some of them are not. The ready made components are controller and request processors. We don't have to create them or write them. They are internally used by Struts framework. The form beans and action classes are the ones that we need to create as part of development. Writing these classes hardly takes couple of minutes. Besides these, we also have to write the JSP pages. Once we finished developing all these components, we define them in a configuration file named `struts-config.xml`. The struts framework will take care of the rest.

Now that we know how Struts work, let's see some of the components that we use to build web applications.

Struts API

The Struts API is a set of classes and interfaces that we normally use to build application components. Following table lists the most important and widely used classes.

Table 16.1 Struts API

Class/Interface	Description
ActionForm	This class is used to create form beans

Action	This class processes http requests
ActionMapping	Class used for selecting the view components
ActionError	Class used for defining an error
ActionErrors	Class that represents a collection of errors
DynaFormBean	Class that is used for working with forms and validation

The best way to learn the API is by using the API. I will explain the usage of the above classes when we start writing example programs. There are two important files we need to know for Struts development. These are:

struts-config.xml: This file is where we configure all the Struts related components.

property file: This file defines all the struts related and application related message properties which is used for displaying messages to the user.

Struts Requests

Struts framework follows some conventions. One of the conventions that stand on top of all, is the way the HTTP requests are send by the browser. As we learned couple pages ago, the controller servlet intercepts all the requests sent by the browser to a struts based web application. The controller usually expects all the requests from the browser to end with a ".do" extension as shown below:

`http://localhost:8080/myweb/welcome.do`

The struts controller component will intercept the above request and interpret it in the following manner.

1. Get the name of the request. The name if the request is the part of the URL after the context name myweb and before .do extension. Therefore it gets /welcome as the name of the request.
2. See if there is information defined in `struts-config.xml` with the name /welcome. If the information is found, it will read the information and initiates processing the request.

The controller component is a servlet named `ActionServlet`. Since this is a servlet, can you tell me the name of the file in which this servlet will be configured? Good. It's `web.xml` file we used to configure servlets.

Typical Struts Scenarios

In Struts based applications, we commonly see the following two scenarios for processing the requests.

Scenario 1: This is the scenario *without* action class. In this case, the controller will use its own expertise with the information defined in the configuration file to process the request.

Scenario 2: This is the scenario *with* action class. In this case, the controller will delegate the request to a RequestProcessor object which will use the formbean (if any) and action class to process the request. The action class finally returns the response back to the controller which will then send it to the browser.

These are the only two cases you'll normally see in any struts based application. All the examples we do in this chapter also fall in one of the above cases.

The struts configuration file

As said before, the configuration file that struts uses is `struts-config.xml` just like servlets use `web.xml` file. Let's see what configuration information this file defines.

Following is the typical structure of the `struts-config.xml` file

```
<struts-config>
    <!-- Data Sources -->
    <data-sources>
    </data-sources>

    <!-- Form Beans -->
    <form-beans>
    </form-beans>

    <!-- Global Exceptions -->
    <global-exceptions>
    </global-exceptions>

    <!-- Global Forwards -->
    <global-forwards>
    </global-forwards>

    <!-- Action Mappings -->
    <action-mappings>
    </action-mappings>

    <!-- Message Resources -->
    <message-resources parameter="props.MessageResources"/>
```

```
<!-- Message Resources -->
<plug-in>
</plug-in>

</struts-config>
```

<data-sources>

This tag is used for defining the data sources to use within the application. This tag is rarely used.

<form-beans>

This element is used to define all the form beans within the application. This element uses the child element **<form-bean>** for defining individual beans as shown below:

```
<form-beans>
    <form-bean name="customer" type="formbeans.CustomerBean" />
</form-beans>
```

The above defines a single formbean with the name `customer` that belongs to a class `CustomerBean` in the `formbeans` package.

<global-exceptions>

This element is used to define error pages for various exceptions thrown by the application components. Its usage is as shown below:

```
<global-exceptions>
    <exception type="java.io.IOException" path="Error.jsp"/>
</global-exceptions>
```

If any of the application components throws `IOException`, then the controller will forward the request to the `Error.jsp` page.

<global-forwards>

This element is used to declare the global forward pages that can be used by all the action classes. Its usage is shown below:

```
<global-forwards>
    <forward name="success" path="/jspots/Main.jsp"/>
    <forward name="failure" path="/jspots/Error.jsp"/>
</global-forwards>
```

With the above forwards, if the action class selects the success forward, then the controller will forward the request to `Main.jsp`.

<action-mappings>

This is the most important element in the configuration file. It defines all the action classes and related information like exceptions, forwards etc. We'll see the details of this element when we do examples.

<message-resources>

This element is used for internationalization support. It defines the list of property files and resource bundles to be used by the framework to display messages on the browser based on locale.

<plug-in>

This element is used to plug in external frameworks into struts framework. This is one of the key features of struts, the ability to integrate with other frameworks.

With this basic understanding of what a struts configuration file contains, let's begin our journey into Struts.

Two simple tricks

1. If the request sent from the browser doesn't contain any data, then *we don't need to write formbean and action class*. We just have to define an *action mapping* in the configuration file with information sufficient for the controller to take care of the request.
2. If the request does include data, then *we must write an action class*. If the data comes by submitting a HTML form, then we also need to write a formbean class. Otherwise, no form bean is required.

Let's write a simple program to get started. This example simply takes a ".do" request from a browser and forwards it to a welcome page. Since no data will be sent in the request, we use trick 1. See the code in listing 16.1.

Listing 16.1a (Welcome.jsp) Simple JSP page

```
<HTML>
<HEAD>
```

```
<TITLE>Tag - Methods</TITLE>
</HEAD>

<h1>Welcome to Struts</h1>

</HTML>
```

Listing 16.1b (struts-config.xml) Struts configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

    <!-- Data Sources -->
    <data-sources></data-sources>

    <!-- Form Beans -->
    <form-beans></form-beans>

    <!-- Global Exceptions -->
    <global-exceptions></global-exceptions>

    <!-- Global Forwards -->
    <global-forwards></global-forwards>

    <!-- Action Mappings -->
    <action-mappings>

        <action path="/welcome" forward="/jsps/struts/Welcome.jsp" />

    </action-mappings>

    <!-- Message Resources -->

```

Steps to run the program

1. Save the JSP page in the following directory as

/myweb/jsps/struts/Welcome.jsp

2. Save the configuration as

/myweb/WEB-INF/struts-config.xml

3. Start the server.

4. Type the following URL in the browser

```
http://localhost:8080/myweb/welcome.do
```

When the above URL is sent to the controller servlet, it checks for action mapping whose path is /welcome. Since we defined the following as the action mapping in the struts-config.xml file,

```
<action path="/welcome" forward="/jsps/struts/Welcome.jsp"/>
```

the controller will forward all the /welcome requests to Welcome.jsp page. The above URL will therefore produce the following result:



The above example is very simple and I am sure you got it in few seconds. In the next example what we will do is, send a request parameter called page with two possible values. Based on its value, we will have the controller forward it to appropriate page. In this example, we will create two JSP pages namely Sales.jsp and Inventory.jsp. When the browser sends the following maintenance request, the controller should display Sales.jsp content:

```
http://localhost:8080/myweb/maintenance.do?page=sales
```

Similarly, for the following URL, it must forward it to Inventory.jsp

```
http://localhost:8080/myweb/maintenance.do?page=inventory
```

This example uses trick 2 listed before. See the code in listing 16.2.

Listing 16.2a (Sales.jsp) Simple JSP.

```
<HTML>
```

```
<h1>Welcome to Sales Page</h1>
</HTML>
```

Listing 16.2b (`Inventory.jsp`) Simple JSP.

```
<HTML>
<h1>Welcome to Inventory Page</h1>
</HTML>
```

Listing 16.2c (`MaintenanceAction.java`) Simple action class.

```
package actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class MaintenanceAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        String nextPage = "";
        try {
            // No From bean. So read the data from request.
            String page = request.getParameter("page");

            // If page is "sales", then it goes to the page whose forward is named
            // with "sales"

            nextPage = page;
        } catch (Exception e) {
        }
        // Finish with
        return mapping.findForward(nextPage);
    }
}
```

We'll see how the above action class works after running the example.

Steps to run the program

1. Save all the files as shown below

```
/myweb/jsp/struts/Sales.jsp  
/myweb/jsp/struts/Inventory.jsp  
/myweb/WEB-INF/classes/actions/MaintenanceAction.java
```

2. Add the following action mapping in the action mappings section in the struts-config.xml (Next to previous mapping)

```
<action name="/maintenance" type="actions.MaintenanceAction">  
    <forward name="sales" path="/jsp/struts/Sales.jsp" />  
    <forward name="inventory" path="/jsp/struts/Inventory.jsp" />  
</action>
```

3. Compile the struts classes as shown below:

```
C:/>JavaTraining>..>..>classes>javac actions/*.java
```

4. Start the server.

5. Type the following URL in the browser

```
http://localhost:8080/myweb/maintenance.do?page=sales
```

The above URL produces the following result



Now change the URL to send inventory as the page parameter as shown below:

```
http://localhost:8080/myweb/maintenance.do?page=inventory
```



Since the example worked the way it should, let's understand how to build action class.

Action Class

Action class is a core component in the struts framework that we need to write to process the request. To create an action class,

1. Write a class that extends `org.apache.struts.Action`.
 2. Overwrite the following method in the Action class.

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception
```

The above `execute()` method takes the following four arguments:

1. ActionMapping - Used by the action class to select the view (JSP page)
 2. ActionForm - This argument represents the formbean if there is one. Since our example doesn't have any formbean class, it will be **null** and we don't use this for now.
 3. HttpServletRequest - This parameter is used to read request parameters and get a handle to session object.
 4. HttpServletResponse - This is a response object, but we never use it. So, we will ignore this argument completely.

The `execute()` method returns an object of `ActionForward` class. This object is used to return the selected view information back to the controller. Following is what we normally do inside the `execute` method:

Read the request data

If a formbean is not defined by the application, we use the following methods to read the data based on where the data is present.

```
request.getParameter() (or)  
request.getSession().getAttribute()
```

If there is form bean defined, say CustomerBean, then we need to do the following:

- a) Get the form bean using the `form` argument as shown below:

```
public ActionForward execute ( ActionMapping mapping, ActionForm form, ... ) {  
    CustomerBean bean = (CustomerBen) form;  
}
```

- b) Read the data from the above bean using its `get` methods.

Process the Data

This is the step where we process all the data and generate dynamic content by calling whichever backend systems we want to. If there is no dynamic content, we simply select the next view (page) to go to. However, if there is dynamic content (which is normally the case) we store it in session and then select the next view. Make sense?

Return the View

Once the above step chooses the view, the execute method returns it back to the controller using the `mapping` argument as shown below:

```
mapping.findForward(nextPage);
```

In our example, we just followed the above process. Since we don't have any form bean, we read the data as,

```
String page = request.getParameter("page");
```

We then assigned the value of the above variable to `nextPage`. This variable can have either `sales` or `inventory` based on what value is sent in the request, right? We then returned the view using the following statement.

```
mapping.findForward(nextPage);
```

The above statement will return either sales or inventory back to the controller to select the view. Can you tell me how the controller associates the above mapping to the actual JSP pages? Look at the following action mapping we added in the configuration file:

```
<action name="/maintenance" type="actions.MaintenanceAction">
    <forward name="sales" path="/jspots/struts/Sales.jsp" />
    <forward name="inventory" path="/jspots/struts/Inventory.jsp" />
</action>
```

I am sure you figured it now. If the MaintenanceAction selects either sales or inventory view, the controller tries to locate the JSP in the *forwards* defined in the action mapping as shown above. What if the above forwards are not defined? You'll get a "Page Not Found" error.

In this example, our action class can only return sales or inventory as the forward. In the next example, instead of sending a request from the browser, let's have a html form post the login information to a login action. This example will use a form bean to store the form data. Let's first write the code and then look at the details. See the code in listing 16.3.

Listing 16.3a (Login.jsp) Simple JSP

```
<html>
<BODY>
<P>Please Login into the application.<BR>
<BR>
</P>
<form action="/myweb/login.do">
<TABLE border="1">
    <TBODY>
        <TR>
            <TD>UserName</TD>
            <TD>
                <INPUT type="text" name="username" size="20">
            </TD>
        </TR>
        <TR>
            <TD>Password</TD>
            <TD><INPUT type="password" name="password" size="20"></TD>
        </TR>
        <TR>
            <TD colspan="2" align="center"><INPUT type="submit" value="Submit"></TD>
        </TR>
    </TBODY>
</TABLE>
</form>
</BODY>
</html>
```

Listing 16.3b (Main.jsp) Simple JSP

```
<%@ page session="true"%>  
  
<%  
    String name = (String) session.getAttribute("Name");  
>  
<h3>Hello Mr <%= name%>. How are you doing?</h3>
```

Listing 16.3c (LoginForm.java) Simple form bean

```
package formbeans;  
  
import org.apache.struts.action.ActionForm;  
  
public class LoginForm extends ActionForm {  
  
    private String password = null;  
    private String username = null;  
  
    public String getPassword() {  
        return password;  
    }  
  
    public void setPassword(String p) {  
        this.password = p;  
    }  
  
    public String getUsername() {  
        return username;  
    }  
  
    public void setUsername(String u) {  
        this.username = u;  
    }  
}
```

Listing 16.3d (LoginAction.java) Simple login action.

```
package actions;  
  
import java.util.HashMap;  
import javax.servlet.http.*;  
import org.apache.struts.action.*;  
import formbeans.LoginForm;  
  
public class LoginAction extends Action {  
  
    public ActionForward execute(ActionMapping mapping, ActionForm form,  
        HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
  
        LoginForm loginForm = (LoginForm) form;  
        String nextPage = "";
```

```
try {  
    String username = loginForm.getUsername();  
    String password = loginForm.getPassword();  
  
    String name = getName(username);  
  
    // Store it in the session  
    request.getSession(true).setAttribute("Name", name);  
  
    nextPage = "success";  
}  
catch (Exception e) {  
}  
  
// Finish with  
return (mapping.findForward(nextPage));  
}  
  
private String getName(String username) {  
    HashMap data = new HashMap();  
    data.put("john", "John Smith");  
    data.put("james", "James Bond");  
  
    String name = (String) data.get(username);  
    if (name == null)  
        name = "Guest";  
  
    return name;  
}
```

Steps to run the program

1. Save all the files as shown below

/myweb/jsp/struts/Login.jsp
/myweb/WEB-INF/classes/formbeans/LoginForm.java
/myweb/WEB-INF/classes/actions/LoginAction.java

2. Add the following action mapping in the action mappings section in the struts-config.xml (Next to previous mappings)

```
<form-beans>  
    <form-bean name="loginForm" type="formbeans.LoginForm">  
    </form-bean>  
</form-beans>  
<action-mappings>
```

```
<action name="/login" type="actions.LoginAction">
    <forward name="success" path="/jsps/struts/Main.jsp" />
</action>

</action-mappings>
```

3. Compile the struts classes as shown below:

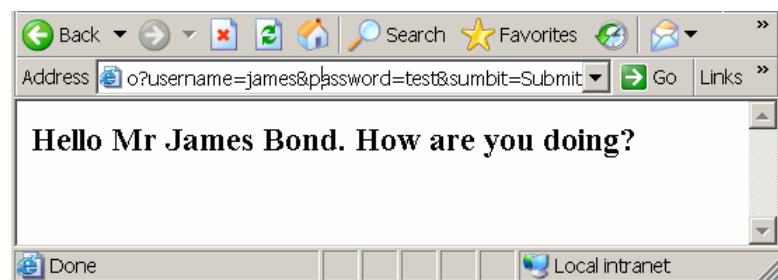
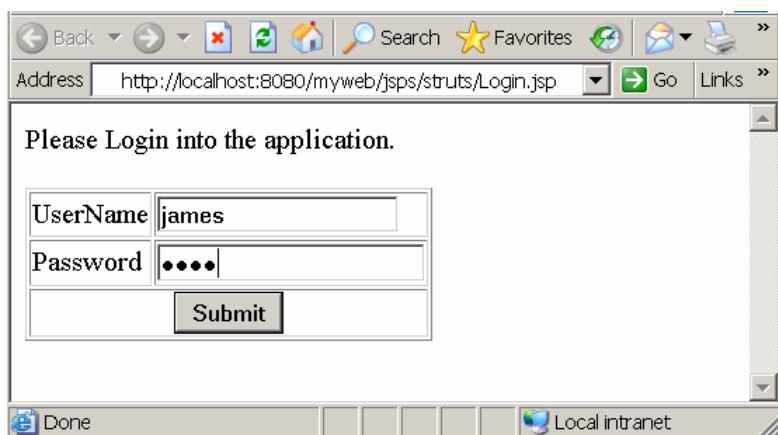
```
C:/>JavaTraining>..>..>classes>javac formbeans\*.java
C:/>JavaTraining>..>..>classes>javac actions\*.java
```

4. Start the server.

5. Type the following URL in the browser

```
http://localhost:8080/myweb/jsps/struts/Login.jsp
```

The above URL produces the following result



Now that you successfully executed the program, let's see how it works.

The login page posts the login information to /login action as shown below:

```
<form action="/myweb/login.do">
```

The controller will then retrieve all the information from struts-config.xml for this action. Following is what it finds:

```
<action path="/login" name="loginForm" type="actions.LoginAction">
    <forward name="success" path="/jsps/struts/Main.jsp" />
</action>
```

The above action mapping is almost same as the previous one but includes a name attribute. This attribute will tell the controller that the LoginAction class is using a formbean named loginForm to collect the data. The controller goes to the form beans mappings in the configuration file, and looks for the bean definition with the same name and finds the following:

```
<form-bean name="loginForm" type="formBeans.LoginForm"></form-bean>
```

Now look at the form bean class LoginForm. Two simples rules to write this class:

1. This class should inherit from ActionForm class
2. Define the form properties (names must match the html field names) with getters and setters.

This is exactly what we did in the LoginForm class. When the login page is submitted, the controller will populate the formbean properties with the form data and passes the formbean as a parameter to LoginAction class. The action class will first read the formbean as shown below:

```
LoginForm loginForm = (LoginForm) form;
```

Once the bean is retrieved, it will invoke the getter methods to read the data.

```
String username = loginForm.getUsername();
String password = loginForm.getPassword();
```

At this point, you can do whatever you want with the data. In our case, we want to get the actual name of the user from the username to display it in the next page. For this, we called the getName() method passing the username. This method should ideally connect to database and get the name, but we just mocked up couple of names using HashMap object. If no name is found, it returns Guest. After the name is returned back,

we just can't leave it alone. Instead, we must store it in the session so that the next page can read it from the session and display it. This is exactly what we did with the following statements.

```
request.getSession(true).setAttribute("Name", name);  
nextPage = "success";
```

Now if you look at the action mapping for this action class, **success** will be forwarded to **Main.jsp** as shown below:

```
<forward name="success" path="/jsps/struts/Main.jsp" />
```

Finally, the main page will read the session attribute and displays the greeting. If the above explanation is confusing, here is the simple process that outlines the above:

1. HTML form is submitted to action class
2. Action class uses the form bean to collect data. So, the controller populates the bean and sends it to action class
3. Action class will read the form bean data, process it and selects the next page

I am sure the above process is pretty straightforward.

Important Trick: Whenever there is a html form, write a form bean with the form properties and use the formbean within the action class. This is also the reason why it got the name formbean.

Now that you understood the basics of Struts, it time to concentrate on JSP page development using struts style. Once we learned to build JSP pages using struts style, we will again get back to normal struts business. Ok.

JSP development using Struts Tags

As I said before, Struts comes will several built in custom tags to assist in view (JSP) development. One of the main goals of Struts is to completely change the style of JSP development by using custom tags for everything including the standard HTML elements like textfields, hyperlinks, textareas etc. Let's now see how we can use Struts custom tags to build JSP pages.

All the custom tags in struts are categorized into four types as shown below:

1. html tags (Used to build HTML elements)

2. bean tags (Used to process the models or Java beans)
3. logic tags (Used to do some logical operations like looping)
4. tiles tags (Used for page layout)

The struts installation will include the TLD files for all the above tags. Following are the four TLD files for the above four categories of tags:

1. struts-html.tld
2. struts-bean.tld
3. struts-logic.tld
4. struts-tiles.tld

To use the custom tags defined in the above TLD files, we import them in the JSP pages as shown below:

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-tiles" prefix="tiles" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
```

Based on the above taglib directives, you can easily guess that all the html tags will be prefixed with html, all the bean tags will be prefixed with bean etc., as shown below:

```
<html:link.....
<bean:define .....
<logic:iterate....
<tiles:insert..... and so on.
```

Let me tell you one thing here. There are several custom tags in each category unlike in JSTL where we have few tags. I know it's tough to memorize all of them, so we will only look at the most important ones here. Ok. The usage of these tags is pretty simple and again hardly takes few minutes to master them.

There is good documentation available in Apache Struts website with complete information about each and every tag and how to use it. Following is the URL to the tags documentation:

<http://struts.apache.org/1.2.9/userGuide/>

Bean Tags

These tags as the name suggests are used for reading and populating the bean properties. All the bean tags are normally prefixed with the word bean. To better understand how the bean tags work, let's consider a bean structure as shown below:

```
Account
    accountDetail: AccountDetail
        accountType
        accountBalance
    address: Address
        addressLine1
        city
        state
        country
    creditCards[]: CreditCard
        name
        number
    bankName: String
```

The Account bean has three properties namely accountDetail, address, array of creditcards and a bankName. The AccountDetail bean in turn has two properties, Creditcard has two properties and Address bean has four properties. The Account bean is the top level bean that represents all the information. Based one the above bean structure, given an account bean, the account type is retrieved with the property key accountDetail.accountType and so on. Now, let's look at the important tags.

<bean:define>

This tag is used to define a scripting variable in a JSP page. The syntax is shown below:

```
<bean:define id="" value="" />
<bean:define id="" name="" property="" scope="" />
```

Example 1:

The following defines a scripting variable named userSSN and assigns the value 12345. You can then use userSSN variable any where in the JSP page.

```
<bean:define id="userSSN" value="12345" />
```

Example 2:

The following defines a scripting variable named type and assigns the value of accountType property of account bean stored in the session.

```
<bean:define id="type" name="account" property="accountDetail.accountType"
    scope="session"/>
```

```
<bean:write>
```

This tag echoes the value stored in a bean property on the page. Its syntax is shown below:

```
<bean:write name="" property="" />
```

Example 3:

The following usage prints the value in bankName property of account bean stored in session.

```
<bean:write name="account" property="bankName" scope="session" />
```

Example 4:

The following usage prints the value in name and number properties of first creditcard in account bean stored in session.

```
<bean:write name="account" property="creditcards[0].name" scope="session" />
<bean:write name="account" property="creditcards[0].number" scope="session" />
```

```
<bean:message>
```

This tag reads the property value from the properties file based on the specified key. Its syntax is shown below:

```
<bean:message key="" />
```

Example 5:

The following displays the property value of error.username key from the property file.

```
<bean:message key="error.username" />
```

Now, let's write an example using the above tags. What we will do is,

1. Create the above Account bean structure
2. Write an action class to populate the Account bean and forward it to JSP page named BeanTagsDemo.jsp
3. Use the bean tags in the JSP and display the Account bean properties.

See the code in listing 16.4.

Listing 16.4a (`CreditCard.java`) Simple bean.

```
package beans;

public class CreditCard {

    String name;
    String number;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }
}
```

Listing 16.4b (`AccountDetail.java`) Simple bean.

```
package beans;

public class AccountDetail {

    String accountType;
    String accountBalance;

    public String getAccountBalance() {
        return accountBalance;
    }

    public void setAccountBalance(String accountBalance) {
        this.accountBalance = accountBalance;
    }

    public String getAccountType() {
        return accountType;
    }

    public void setAccountType(String accountType) {
        this.accountType = accountType;
    }
}
```

Listing 16.4c (`Address.java`) Simple bean.

```
package beans;

public class Address {

    String addressLine1;
    String city;
    String state;
    String country;

    public String getAddressLine1() {
        return addressLine1;
    }

    public void setAddressLine1(String addressLine1) {
        this.addressLine1 = addressLine1;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
```

Listing 16.4d (`Account.java`) Simple bean.

```
package beans;

import java.util.List;

public class Account {

    AccountDetail accountDetail;
    Address address;
    List creditCards;
    String bankName;
```

```
public String getBankName() {
    return bankName;
}

public void setBankName(String bankName) {
    this.bankName = bankName;
}

public AccountDetail getAccountDetail() {
    return accountDetail;
}

public void setAccountDetail(AccountDetail accountDetail) {
    this.accountDetail = accountDetail;
}

public Address getAddress() {
    return address;
}

public void setAddress(Address address) {
    this.address = address;
}

public List getCreditCards() {
    return creditCards;
}

public void setCreditCards(List creditCards) {
    this.creditCards = creditCards;
}

public CreditCard getCreditCards(int pos) {
    return (CreditCard) creditCards.get(pos);
}

public void setCreditCards(int pos, CreditCard card) {
    this.creditCards.add(pos, card);
}
}
```

The above four bean classes are pretty straight forward. Look at the `Account` bean. This bean defines four properties as shown below:

```
AccountDetail accountDetail;
Address address;
List creditCards;
String bankName;
```

Since we want to store multiple creditcards, we defined a `List` of `creditCards` as shown above. Typically, if the parameter is of type `List` (repeating element), then besides the following two regular getter and setter methods,

```
public List getCreditCards() {
    return creditCards;
```

```
}

public void setCreditCards(List creditCards) {
    this.creditCards = creditCards;
}
```

we also need to define two additional *indexed* methods with the actual object contained in the list as shown below:

```
public CreditCard getCreditCards(int pos) {
    return (CreditCard)creditCards.get(pos);
}
public void setCreditCards(int pos, CreditCard card) {
    this.creditCards.add(pos,card);
}
```

Now, let's write an action class to populate the Account bean and store it in the session scope.

Listing 16.4e (BeanAction.java) Action class storing the bean in session

```
package actions;

import java.util.ArrayList;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.*;
import beans.*;

public class BeanAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        ActionErrors errors = new ActionErrors();
        ActionForward forward = new ActionForward();

        String nextPage = request.getParameter("page");

        try {

            CreditCard cc1 = new CreditCard();
            cc1.setName("CapitalOne");
            cc1.setNumber("12345");

            CreditCard cc2 = new CreditCard();
            cc2.setName("AMEX");
            cc2.setNumber("999999");

            List cards = new ArrayList();
            cards.add(cc1);
            cards.add(cc2);

        }
    }
}
```

```
AccountDetail detail = new AccountDetail();
detail.setAccountType("Checkings");
detail.setAccountBalance("2000.30");

Address address = new Address();
address.setAddressLine1("1111 S St");
address.setCity("Lincoln");
address.setState("NE");
address.setCountry("USA");

Account account = new Account();
account.setAccountDetail(detail);
account.setAddress(address);
account.setCreditCards(cards);
account.setBankName("Chase");

request.getSession(true).setAttribute("account", account);

} catch (Exception e) {
}

// If a message is required, save the specified key(s)

return (mapping.findForward(nextPage));

}
}
```

As you can see from the above code, we first created and populated the child beans, and then added the child beans to the `Account` bean using the `set` methods. Finally, we stored the `Account` bean in the session with the name `account` as shown below:

```
request.getSession(true).setAttribute("account", account);
```

In the request from the browser, let's send a parameter named `page` whose value is used to select the view. Following is the JSP that uses the bean tags to display the `account` bean properties.

Listing 16.4f (`BeanTagsDemo.jsp`) JSP page using bean tags

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<html>

<BODY>
<P>

<h2> Accessing the Account bean properties from the session </h2></BR>
<bean:define id="ssn" value="324324" />

<bean:write name="account" property="accountDetail.accountType"
scope="session"/>
<bean:write name="account" property="accountDetail.accountBalance"
scope="session"/>
```

```
<br/>

<bean:write name="account" property="address.addressLine1" scope="session"/>
<bean:write name="account" property="address.city" scope="session"/>
<bean:write name="account" property="address.state" scope="session"/>
<bean:write name="account" property="address.country" scope="session"/>

<br/>

<bean:write name="account" property="creditCards[0].name" scope="session"/>
<bean:write name="account" property="creditCards[0].number"
    scope="session"/> <br>
<bean:write name="account" property="creditCards[1].name" scope="session"/>
<bean:write name="account" property="creditCards[1].number"
    scope="session"/>

<bean:write name="ssn" />
<h2>Messages from property files</h2><br/>
<bean:message key="title" /><br/>
<bean:message key="error.username" /><br/>
<bean:message key="error.password" />

</P>
</BODY>
</html>
```

This JSP uses the bean tags to retrieve the Account bean information. For instance, to retrieve the account type, it uses the following bean tag:

```
<bean:write name="account" property="accountDetail.accountType"
    scope="session"/>
```

Since we stored the bean with the name account in session scope, the above tag must also use the same name to access it. It also specifies where the bean can be found using the scope attribute. The property attribute is the one that specifies the complete path of the property. We also used the message tags to read the properties from the following property file:

```
MyStruts.properties

title = Welcome Page
error.username = Invalid UserName. Must contain atleast 2 numbers
error.password = Invalid Password. Must be atleast 8 characters
```

To display the above property values in the browser, we use the message tag as shown below:

```
<bean:message key="error.password" />
```

The above tag will retrieve the property value of `error.password` from the property file and display it in the browser. Let's execute the program.

Steps to run the program

1. Save all the files as shown below

```
myweb/WEB-INF/classes/beans/Address.java  
myweb/WEB-INF/classes/beans/AccountDetail.java  
myweb/WEB-INF/classes/beans/CreditCard.java  
myweb/WEB-INF/classes/beans/Account.java  
myweb/WEB-INF/classes/actions/BeanAction.java  
myweb/jsp/struts/BeanTagsDemo.jsp  
myweb/WEB-INF/classes/myprops/MyStruts.properties
```

2. Add the following action mapping in the action mappings section in the `struts-config.xml` (Next to previous mappings)

```
<action-mappings>  
    . . . . .  
    <action path="/customTags" type="actions.BeanAction">  
        <forward name="bean" path="/jsp/struts/BeanTagsDemo.jsp" />  
    </action>  
  
</action-mappings>  
<!-- Message Resources -->  
<message-resources parameter="myprops.MyStruts"/>
```

3. Compile the struts classes as shown below:

```
C:/>JavaTraining>..>..>classes>javac beans\*.java  
C:/>JavaTraining>..>..>classes>javac actions\*.java
```

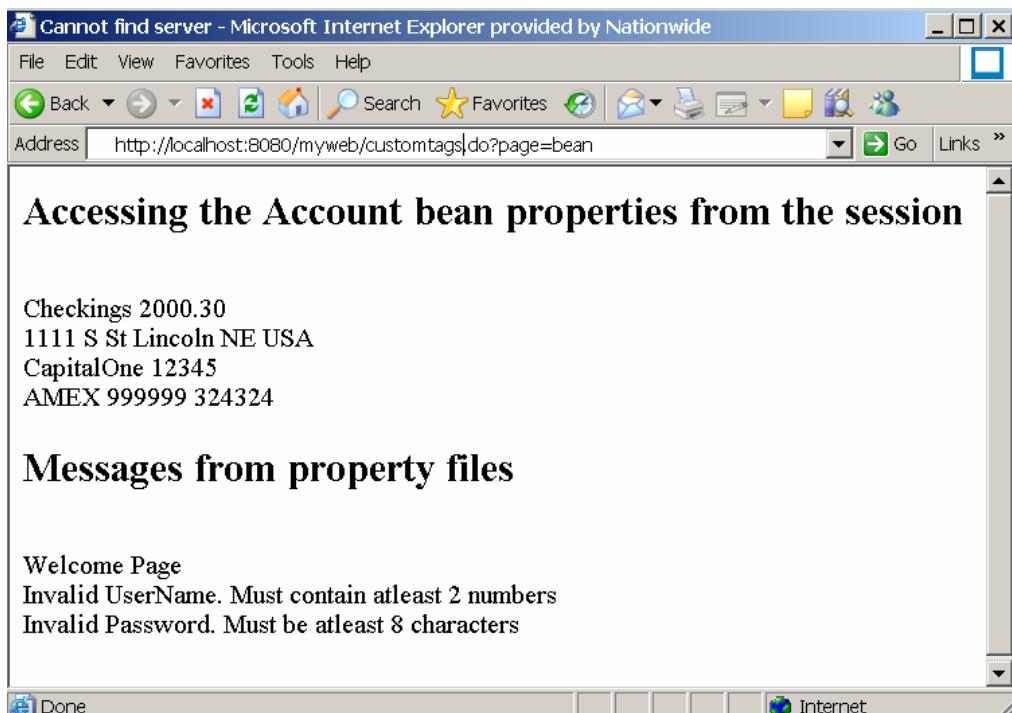
4. Start the server.
5. Type the following URL in the browser

```
http://localhost:8080/myweb/customtags.do?page=bean
```

Notice the `page` parameter in the request. The action class will use this parameter to select the view. This is the reason why we have the forward with the name `bean` in the configuration file as shown below:

```
<forward name="bean" path="/jsps/struts/BeanTagsDemo.jsp" />
```

The above URL produces the following result.



Now that we understood how the bean tags work, let's look at how logic tags work.

Logic Tags

These tags are used for basic operations in the JSP pages like iterations, conditions etc. These tags are very much similar to the JSTL core tags that we read in the previous chapter. These tags are normally prefixed with the word "logic". Following are the most important tags in this category.

```
<logic:empty>
```

This tag is used to check the emptiness of a collection referenced by a JSP page variable or a bean property. The body of the tag is evaluated based on the outcome. The syntax for this tag is shown below:

```
<logic:empty name="" />
```

```
<logic:empty name="" property="" scope="">
```

Example 1:

```
<%  
    List data = new ArrayList();  
%>  
  
<logic:empty name="data">  
    Some content  
</logic:empty>
```

The above usage will evaluate the body of the tag if the collection referenced by data is empty

Example 2:

The following usage of the tag will evaluate the body if the collection referenced by creditCards property in accounts bean is empty.

```
<logic:empty name="accounts" property="creditCards" scope="session">  
    Some Content  
</logic:empty>  
  
<b><logic:notEmpty></logic:notEmpty></b>
```

The behavior of this tag is exactly reverse of the above empty tag. This tag will evaluate the body if the collection is *not empty*. The syntax for this tag is shown below:

```
<logic:notEmpty name="" />  
<logic:notEmpty name="" property="" />
```

<logic:equal>

This tag as the name suggests will evaluate the body of the tag if the value of a page variable or bean property is *equal* to the specified value. Its syntax is shown below:

```
<logic:equal name="" value="" />  
<logic:equal name="" property="" value="" scope="" />
```

Example 3:

```
<logic:equal name="firstName" value="John">  
    Some content here  
</logic:equal>
```

The above usage of this tag will evaluate the body if the value in `firstName` variable is equal to John.

Example 4:

```
<logic:equal name="account" property="bankName" scope="session" value="Swiss">
    Thank you for choosing Swiss.
</logic:equal>
```

The above usage of the tag will check the value of `bankName` property in `account` bean and then evaluate the content accordingly.

<logic:iterate>

This tag is used to iterate over a collection of objects. This is one of the most widely used logic tag. Following is the syntax for this tag:

```
<logic:iterate id="" name="" />
```

Example 5:

```
<logic:iterate id="customer" name="account" property="creditCards"
    scope="session">
    <bean:write name="customer" property="name" />
    <bean:write name="customer" property="number" />
</logic:iterate>
```

This above usage of the tag will iterate over the `creditCards` collection in the `account` bean. It uses the `id` attribute to reference every object in the collection. Inside the tag, we use the `id` value and read the properties as shown above. The iteration will display all the credit card names and numbers stored in the `account` bean.

Using the above logic tags let's write an example. What we will do is, use the same bean classes and action class of the previous example and simply create a new JSP file called `LogicTagsDemo.jsp` and use the `account` bean stored in the session class.

See the code in listing 16.5.

Listing 16.5 (LogicTagsDemo.jsp) JSP page using logic tags

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic"%>

<html>
```

```
<BODY>
<P>

<logic:present name="account" property="bankName" scope="session">
    <h3> The name of the bank is
        <bean:write name="account" property="bankName"
                    scope="session"/>
    </h3>
</logic:present>

<logic:notPresent name="account" property="branchNumber" scope="session">
    <h3> The branch number is not available </h3>
</logic:notPresent>

<logic:equal name="account" property="accountDetail.accountType"
             scope="session" value="Checkings">
    <h3> The total checkings balance is $2000.20 </h3>
</logic:equal>

<logic:notEqual name="account" property="accountDetail.accountType"
                 scope="session" value="Checkings">
    <h3> No Checkings account </h3>
</logic:notEqual>

<logic:notEmpty name="account" property="creditCards" scope="session">
    <logic:iterate id="card" name="account" property="creditCards"
                   scope="session">
        <bean:write name="card" property="name"/>
        <bean:write name="card" property="number"/>
    </logic:iterate>
</logic:notEmpty>

</P>
</BODY>
</html>
```

Steps to run the program

1. Save the JSP file as

myweb/jsp/struts/LogicTagsDemo.jsp

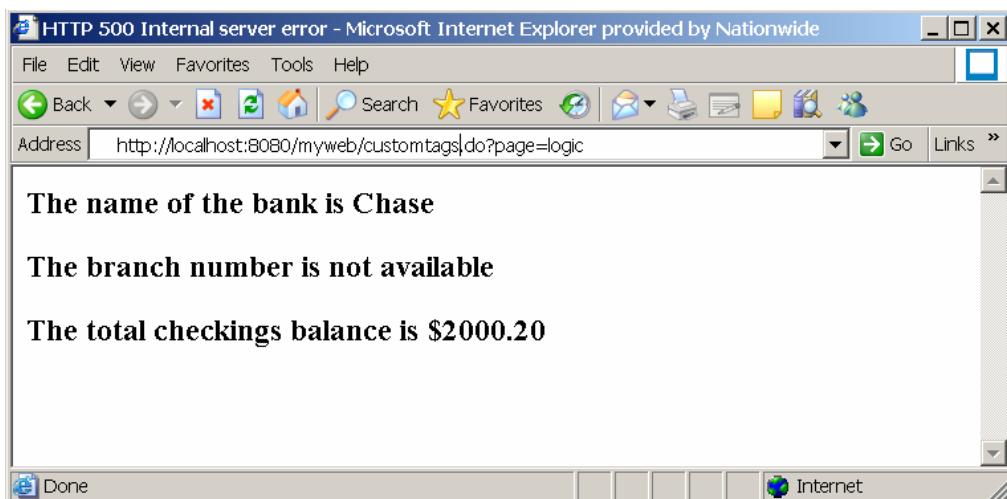
2. Update the action mapping used in the previous example (/customtags) to add the following new forward:

```
<action path="/customTags" type="actions.BeanAction">
    <forward name="bean" path="/jsps/struts/BeanTagsDemo.jsp" />
    <forward name="logic" path="/jsps/struts/LogicTagsDemo.jsp" />
</action>
```

3. Start the tomcat server and type the following URL

```
http://localhost:8080/myweb/customtags.do?page=logic
```

The above URL produces the following result



HTML tags

The Struts html tags are used to represent the standard html elements in the JSP page. Though these tags work much the same way, it's a good practice to use them to be consistent. These tags are normally prefixed with the word "html". Following are the important ones.

```
<html:form action="">
```

This tag is used to create a html form element. To submit a html form to an action mapping named "/welcome.do" we use this tag as shown below:

```
<html:form action="welcome">
```

Notice that we need not specify the ".do" extension.

```
<html:text property="" size="" maxlength="" />
```

This tag is used to create html text field.

Example 1:

To create a text field with the name `firstName` of size 20, and with maximum character limitation of 50, we use this tag a shown below:

```
<html:text property="firstName" size="20" maxlength="50" />  
  
<html:textarea property="" rows="" cols="" />
```

This tag is used to create html text area with specified name, rows and columns.

Example 2:

To create a text area with the name `description` with dimensions 5 by 40, we use this tag as shown below:

```
<html:textarea property="description" rows="5" cols="40" />  
  
<html:link action="" name="" />
```

This tag defines a hyperlink. The name attribute is optional and if present must be a Map which will get converted to `queryString`.

Example 3:

```
<%  
    Map queryString = new HashMap();  
    queryString.put("firstName", "john");  
    queryString.put("lastName", "Smith");  
%>  
  
<html:link action="register" name="queryString"/>
```

The above hyperlink gets manifested as a URL shown below:

`http://localhost:8080/myweb/register.do?firstName=john&lastName=Smith`

```
<html:select collection="" property="" />
```

This tag is used to represent a drop down box for selecting the items.

Example 4:

```
<%
```

```
ArrayList books = new ArrayList();
books.add("Java");
books.add("C Programming");
books.add("Perl Scripting");
%>
<html:select collection="books" property="book" />
```

When the form is submitted, the selected item in the drop down will be sent in the request with the property name book as shown below:

```
http://localhost:8080/myweb/select.do?book=Java
<html:errors />
```

This tag is used to display all the Struts errors. The usage of this tag is explained in the later section.

This completes all the basic custom tags that we need to know to build JSP pages. With this knowledge, let's get back to the real struts business.

Struts Validation

Validating the form data is one of the common things we do in web applications. Struts support data validation through two mechanisms listed below:

1. Formbean Validation
2. Struts Validator

Let's look at both the mechanisms one by one and see which one offers more flexibility.

Validation using FormBeans

The example that we are going to develop now will cover all the concepts that we learned so far. In this example, we will build a JSP page that uses the struts custom tags and post the customer information to an action class. Since we have a form, we will create a form bean to collect the data. This example also introduces you to data validation using form beans. Following is what we will do in this example:

1. Send a request from the browser to access the JSP page.
2. The customer form in the JSP will be submitted to an action class.
3. The controller will use the form bean to store and *validate* the data. If all the data is valid, controller will send the formbean to the action class.

4. If the validation fails, the controller will display the form listing the errors.
5. The action class will insert the data into database and displays a confirmation message.

Let's first write the code and then look at the details. See the code in listing 16.6.

Listing 16.6a (*CustomerDetails.jsp*) JSP page using struts tags

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>

<html:html>

<BODY>

<html:errors/>

<P>Please fill in the following details</P>

<html:form action="customer">

<TABLE border="1">
    <TBODY>
        <TR>
            <TD>FirstName</TD>
            <TD><html:text property="firstName"></html:text></TD>
        </TR>
        <TR>
            <TD>MiddleName</TD>
            <TD><html:text property="middleName"></html:text></TD>
        </TR>
        <TR>
            <TD>LastName</TD>
            <TD><html:text property="lastName"></html:text></TD>
        </TR>
        <TR>
            <TD>Age</TD>
            <TD><html:text property="age"></html:text></TD>
        </TR>
        <TR>
            <TD>SSN</TD>
            <TD><html:text property="ssn"></html:text></TD>
        </TR>
        <TR>
            <TD>City</TD>
            <TD><html:text property="city"></html:text></TD>
        </TR>
        <TR>
            <TD>State</TD>
            <TD><html:text property="satte"></html:text></TD>
        </TR>
        <TR>
            <TD>Country</TD>
            <TD><html:text property="country"></html:text></TD>
        </TR>
        <TR>
```

```
<TD colspan="2"><html:submit></html:submit></TD></TR>
</TBODY>
</TABLE>
</html:form>
</BODY>
</html:html>
```

Listing 16.6b (`Confirmation.jsp`) Simple confirmation JSP page

```
<html>
  <h2> Thanks for registering with us.</h2>
</html>
```

Listing 16.6b (`CustomerForm.java`) Simple form bean with validation

```
package formbeans;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionError;

public class CustomerForm extends ActionForm {

    private String ssn = null;
    private String age = null;
    private String country = null;
    private String state = null;
    private String middleName = null;
    private String firstName = null;
    private String city = null;
    private String lastName = null;

    public String getAge() {
        return age;
    }
    public void setAge(String age) {
        this.age = age;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    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 String getMiddleName() {
    return middleName;
}
public void setMiddleName(String middleName) {
    this.middleName = middleName;
}
public String getSSN() {
    return ssn;
}
public void setSSN(String ssn) {
    this.ssn = ssn;
}
public String getState() {
    return state;
}
public void setState(String state) {
    this.state = state;
}
public ActionErrors validate(ActionMapping mapping, HttpServletRequest
request) {

    ActionErrors errors = new ActionErrors();

    // Validate the fields in your form
    // Lastname and SSN are always required

    if ((lastName == null) || (lastName.length() == 0)) {
        errors.add("lastName", new ActionError("error.lastName.required"));
    }
    if ((ssn == null) || (ssn.length() == 0)) {
        errors.add("ssn", new ActionError("error.ssn.required"));
    }

    // Validation for valid values for SSN and Age
    try {
        int ssn1 = Integer.parseInt(ssn);
    } catch (Exception e) {
        // If it comes here, then the user entered non-numeric data for SSN
        errors.add("ssn_number", new ActionError("error.ssn.invalid"));
    }
    try {
        int age1 = Integer.parseInt(age);
    } catch (Exception e) {
        errors.add("age_number", new ActionError("error.age.invalid"));
    }
    return errors;
}
}
```

Listing 16.6c (`MyStruts.properties`) Struts property file

```
title = Welcome Page
error.username = Invalid UserName. Must contain atleast 2 numbers
error.password = Invalid Password. Must be atleast 8 characters

error.lastName.required = LastName is required
error.ssn.required = SSN is required
error.ssn.invalid = SSN must be a 10 digit number
error.age.invalid = Age must be a 2 digit number
```

Listing 16.6d (`CustomerAction`) Customer action class.

```
package actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import formbeans.CustomerForm;

public class CustomerAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        ActionErrors errors = new ActionErrors();
        CustomerForm customerForm = (CustomerForm) form;

        try {

            insertCustomerRecord(customerForm.getFirstName(),
                customerForm.getMiddleName(), customerForm.getLastName(),
                customerForm.getAge(), customerForm.getSsn(), customerForm.getCity(),
                customerForm.getState(), customerForm.getCountry());

        } catch (Exception e) {

            // Report the error using the appropriate name and ID.
            errors.add("name", new ActionError("id"));

        }

        // Finish with
        return (mapping.findForward("success"));

    }

    private void insertCustomerRecord(String firstName, String middleName,
        String lastName, String age, String ssn, String city, String state,
```

```
        String country) {  
    // Use JDBC Logic here to insert the data into the Customers table.  
}  
}
```

Steps to run the program

1. Save all the files as shown below

```
myweb/jsp/struts/CustomerDetails.jsp  
myweb/jsp/struts/Confirmation.jsp  
myweb/WEB-INF/classes/formbeans/CustomerForm.java  
myweb/WEB-INF/classes/actions/CustomerAction.java  
myweb/WEB-INF/classes/myprops/MyStruts.properties
```

2. Add the following action mapping in the action mappings section in the struts-config.xml (Next to previous mappings)

```
<action-mappings>  
    . . . . .  
    <action name="customerForm" path="/customer"  
        validate="true"  
        input="/jsp/struts/CustomerDetails.jsp" scope="request"  
        type="actions.CustomerAction">  
        <forward name="success" path="/jsp/struts/Confirmation.jsp" />  
    </action>  
  
</action-mapings>  
<!-- Message Resources -->  
<message-resources parameter="myprops.MyStruts"/>
```

3. Compile the struts classes as shown below:

```
C:/>JavaTraining>..>..>classes>javac formbeans\*.java  
C:/>JavaTraining>..>..>classes>javac actions\*.java
```

4. Start the server.
5. Type the following URL in the browser

```
http://localhost:8080/myweb/CustomerDetails.jsp
```

Let's now look at the details of each of the files.

CustomerDetails.jsp

This page uses the struts html tags to build a customer form and posts the data to /customer.do action as shown below:

```
<html:form action="customer">
```

The configuration file defines the customer action as shown below:

```
<action path="/customer" name="CustomerForm"
       validate="true"
       input="/jsps/struts/CustomerDetails.jsp" scope="request"
       type="actions.CustomerAction">
  <forward name="success" path="/jsps/struts/Confirmation.jsp" /
</action>
```

The above mapping has three important attributes namely name, validate and input.

- ✓ The name attribute specifies the form bean to use.
- ✓ The validate attribute tells the controller to use the formbean and validate the data.
- ✓ The input attribute tells the controller to forward to the specified page in the event of validation errors. Usually, the JSP page specified by this attribute will be the same page from where the form is submitted.

CustomerForm.java

This formbean simply defines all the properties of the form along with the getters and setters. The important thing to understand here is how the controller uses the bean to validate the data. Here is the trick. To validate the data inside formbean, it should implement the validate() method as shown below:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest
                           request) {

    ActionErrors errors = new ActionErrors();

    // Validate the fields in your form
    // Lastname and SSN are always required

    if ((lastName == null) || (lastName.length() == 0)) {
        errors.add("lastName", new ActionError("error.lastName.required"));
    }
    if ((ssn == null) || (ssn.length() == 0)) {
        errors.add("ssn", new ActionError("error.ssn.required"));
    }

    // Validation for valid values for SSN and Age
```

```
try {
    int ssn1 = Integer.parseInt(ssn);
} catch (Exception e) {
    // If it comes here, then the user entered non-numeric data for SSN
    errors.add("ssn_number", new ActionError("error.ssn.invalid"));
}
try {
    int age1 = Integer.parseInt(age);
} catch (Exception e) {
    errors.add("age_number", new ActionError("error.age.invalid"));
}

return errors;
}
```

This method is a standard method for validating the data. Since the validation is turned on using the validate attribute in the action mapping, the controller after populating formbean properties will invoke the validate() method. This method will return ActionErrors object which is a collection of ActionError objects. For every failed validation, we need to add an ActionError object to the ActionErrors list as shown below:

```
if ((lastName == null) || (lastName.length() == 0)) {
    errors.add("lastName", new ActionError("error.lastName.required"));
}
```

The above code checks for lastName, and does the following two things:

1. If lastName is not populated, it creates an action error object as,

```
new ActionError ("error.lastName.required");
```

The constructor usually specifies the key defined in the property file whose value will be echoed when the validation fails. With the following property defined in the property file,

```
error.lastName.required = LastName is required
```

when the errors are displayed on the page, the actual message from the property file will be displayed; in our case the message will be "LastName is required".

2. The above action error will then be added to the ActionErrors list as

```
errors.add("lastName", new ActionError ("error.lastName.required"));
```

Similarly, we add the errors for other fields. By the end of the `validate()` method, if the `ActionErrors` list is empty (no errors), then the form bean will be sent to action class. Otherwise, the controller will go back to the page specified by the `input` attribute in the action mapping which happens to be the same JSP page that submitted the form. Since the formbean is performing four validations, we will have four property messages defined in the properties file as shown below:

```
error.lastName.required = LastName is required  
error.ssn.required = SSN is required  
error.ssn.invalid = SSN must be a 10 digit number  
error.age.invalid = Age must be a 2 digit number
```

CustomerAction.java

The `execute()` method of this action class simply reads the formbean data and invokes the method to insert the data into the database. Take this part as assignment to implement the JDBC logic. It finally selects the `success` forward which takes us to the confirmation page using the following forward definition:

```
<forward name="success" path="/jsps/struts/Confirmation.jsp"/>
```

There is one more thing we need to know. Assuming that validation errors if any are returned back to the page, how should we display them? Very simple. We simply need to add the following html errors tag in the jsp page where ever we want to display the errors.

```
<html:errors/>
```

In our example, let's display the errors at the top of the page by placing the above tag as shown below:

```
<BODY>  
<html:errors/>  
<P>Please fill in the following details</P>
```

Run the example by entering the following URL in the browser:

```
http://localhost:8080/myweb/jsps/struts/CustomerDetails.jsp
```

Enter non-numeric data for Age and SSN and you'll see the following result:

The screenshot shows a Microsoft Internet Explorer window with the URL <http://localhost:9080/Struts/customer.do>. The page displays validation errors:

SSN must be a 10 digit number Age must be a 2 digit number

Please fill in the following details

FirstName	Test
MiddleName	T
LastName	Test
Age	jhghj
SSN	jhgjhg
City	Test
State	Test
Country	IT - IT

The browser's status bar at the bottom right shows "Local intranet".

Congratulations. Now you know how to validate form data using formbeans. Though formbeans give us the flexibility for custom validations, it has some serious drawbacks too. Let's see this negative side of formbeans.

1. Since a single formbean is associated with a single html form, if the web application has like 100 html forms, we need to have 100 formbean classes, right? This poses code maintenance problems and the project size also gets lengthy.
2. If the html form undergoes changes by adding or removing fields, then the formbean properties must be changed accordingly, which requires recompilation of entire project.

To overcome the above two problems, does struts have a solution by which we can completely eliminate the formbean classes and also eliminate the recompilation of the project due to form changes? Yes, we have a solution in the form of *dynaformbeans*. So, without wasting any time, let's see what this is all about.

Formbeans using DynaActionForm class

If you noticed the formbeans we wrote until now, they all belong to **ActionForm** class because the formbean inherits from it as shown below:

```
public class CustomerForm extends ActionForm
```

Such formbeans suffer from the above two problems. The new type of formbeans use **DynaActionForm** class instead of **ActionForm** class to address the problems. This is

why we call them as dynaformbeans. Using `DynaActionForm` we can eliminate all the formbean classes by declaring the formbean properties in the struts configuration file (`struts-config.xml`) itself. The formbean definition will go right in the form beans section of the xml file. By moving the entire formbean definition into the xml itself, we no longer have to recompile the project in the event of changes. We just have to restart the server. Following is how the formbean definition for a customer form looks like:

```
<form-bean name="dynamicCustomerForm"
           type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="firstName" type="java.lang.String" />
    <form-property name="middleName" type="java.lang.String" />
    <form-property name="lastName" type="java.lang.String" />
    <form-property name="age" type="java.lang.String" />
    <form-property name="ssn" type="java.lang.String" />
    <form-property name="city" type="java.lang.String" />
    <form-property name="state" type="java.lang.String" />
    <form-property name="country" type="java.lang.String" />
</form-bean>
```

There are two important things we need to understand with the above definition. The `type` attribute should always be `org.apache.struts.validator.DynaValidatorForm` and we need to use nested `<form-property>` tags to define the bean properties. This element should also define the class of the property using the `type` attribute as shown below:

```
<form-property name="firstName" type="java.lang.String" />
```

The above defines `firstName` as a `String`. Similarly, if you want to define a property to store numbers, it will be defined as shown below:

```
<form-property name="ssn" type="java.lang.Integer" />
```

Though the above definition solved the two problems, what about validation? Clearly, we did not move the validation logic in the `validate()` method to the xml, right? So, if we want to do data validation, what should we do? Good question. There are two solutions. One solution is create a class and just define the `validate()` method. What? a new class again? This is what we want to eliminate, right? Using this solution brings us back to square one. So, this is definitely not the solution. The solution is XML based validation using Struts Validation framework. Every time we have a problem, struts is there for our rescue. This is the beauty of Struts.

Now that we know how to define a dynaformbean in the configuration file, we need to know how to read the data from this bean in the action class. If you recall from the previous example, action class first casts the formbean into the appropriate class and

then invoke the getter methods to read the data. We do almost exactly the same with dynaformbeans too. Following is how the code looks like in the action class:

```
DynaValidatorForm bean = (DynaValidatorForm) form;  
  
String firstName = bean.get("firstName");  
String middleName = bean.get("middleName");
```

As you can see from the above code, we cast the form to DynaValidatorForm class and use the get methods passing the property names to read the data. Let's stop this theory crap and look at an example using the dynaformbeans. In this example we will use the same JSP code from the previous example, but post it to a new action class that uses dynaformbean. See the code in listing 16.7.

Listing 16.7a (ConfirmationPage2.jsp) Confirmation page

```
<%@ page session="true" %>  
  
<h3> Thank you <%= session.getAttribute("Name") %> for visiting the WebSite  
</h3>
```

Listing 16.7b (DynamicCustomerAction.java) Action class using dyna form beans

```
package actions;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
import org.apache.struts.action.Action;  
import org.apache.struts.action.ActionForm;  
import org.apache.struts.action.ActionForward;  
import org.apache.struts.action.ActionMapping;  
import org.apache.struts.validator.DynaValidatorForm;  
  
public class DynamicCustomerAction extends Action {  
  
    public ActionForward execute(ActionMapping mapping, ActionForm form,  
        HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
  
        String nextPage = "success";  
  
        try {  
  
            // We have a form bean in this case  
            DynaValidatorForm bean = (DynaValidatorForm) form;  
  
            String firstName = (String) bean.get("firstName");
```

```
String middleName = (String) bean.get("middleName");
String lastName = (String) bean.get("lastName");
String age = (String) bean.get("age");
String ssn = (String) bean.get("ssn");
String city = (String) bean.get("city");
String state = (String) bean.get("state");
String country = (String) bean.get("country");

// Remember in our database, age and ssn are integers. So lets convert
// the strings to int and then store it to database

int iage = 0;
int issn = 0;

iage = Integer.parseInt(age);
issn = Integer.parseInt(ssn);

request.getSession(true).setAttribute("Name", firstName + " " +
lastName);

// Call the method that inserts into database
/*
 * int status = insertCustomer(firstName,middleName,lastName,
 * iage,issn,city,state,country);
 */

} catch (Exception e) {

}

// Finish with
return mapping.findForward(nextPage);

}

private int insertCustomer(String firstName, String middleName,
String lastName, int age, int ssn, String city, String state,
String country) {

Connection conn = null;
try {

Class.forName("com.mysql.jdbc.Driver");
conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/MyDB");

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

int result = -1;
// Create the DDL
String sql = "INSERT INTO CUSTOMERS VALUES (?,?,?,?,?,?,?,?,?);";

PreparedStatement ps = null;
try {
if (conn != null) {
// Create statement from connection
ps = conn.prepareStatement(sql);

// Populate the data
```

```

        ps.setString(1, firstName);
        ps.setString(2, middleName);
        ps.setString(3, lastName);
        ps.setInt(4, age);
        ps.setInt(5, ssn);
        ps.setString(6, city);
        ps.setString(7, state);
        ps.setString(8, country);

        // Execute the prepared statement
        result = ps.executeUpdate();
    }
} catch (Exception e) {
    result = -1;
}
return result;
}
}

```

Don't worry by the length of the code. We also included the JDBC logic to insert the customer details to the database. If you look at the code in the above action class, we used the dynaformbean to read the form data. This is the only change from the previous example.

Steps to run the example

1. Save the JSP and the action class as

```
/myweb/jsp/struts/Confirmation2.jsp
/myweb/WEB-INF/classes/actions/DynamicCustomerAction.java
```

2. Add the following xml snippet in the formbeans section in struts-config.xml file next to the previous one.

```

<form-bean name="dynamicCustomerForm"
            type="org.apache.struts.validator.DynaValidatorForm">

    <form-property name="firstName" type="java.lang.String" />
    <form-property name="middleName" type="java.lang.String" />
    <form-property name="lastName" type="java.lang.String" />
    <form-property name="age" type="java.lang.String" />
    <form-property name="ssn" type="java.lang.String" />
    <form-property name="city" type="java.lang.String" />
    <form-property name="state" type="java.lang.String" />
    <form-property name="country" type="java.lang.String" />
</form-bean>

```

3. Add the following snippet in the action mappings section next to the previous mapping.

```
<action path="/dynaCustomer" name="dynamicCustomerForm"
```

```
    type="actions.DynamicCustomerAction">
    <forward name="success" path="/jsps/struts/Confirmation2.jsp"/>
</action>
```

Notice that our new formbean name is “dynaCustomerForm” which uses the dynaformbean defined in Step 2. It also defines a forward to Confirmation2.jsp.

4. In the CustomerDetails.jsp page created in the previous example, change the form action as shown below to point to the above action mapping.

```
<html:form action="dynaCustomer">
```

5. Compile the action class as shown below

```
c:/>JavaTraining.....>myweb>WEB-INF>classes>javac actions\*.java
```

Let's see how the code works. The JSP will post the form data to dynaCustomer action. Since this action mapping uses dynaformbean, the controller will populate the bean properties with the form data and passes it as an argument to action class. The action class simply reads all the data and stores the Name of the customer in the session. For now we commented the method to insert the data into database. If you have the table ready, then you can uncomment this code to store the data in the table. The action class then selected the success forward which forwards it to confirmation page defined in the action mapping. This JSP will read the Name field from the session and displays a greeting to the user.

Start the server, and type the following URL in the browser.

```
http://localhost:8080/myweb/jsps/struts/CustomerDetails.jsp
```

The above URL will produce the following result:

The screenshot shows a web browser window with the address bar set to `http://localhost:9080/Struts/jspstruts/Cus`. The page title is "Please fill in the following details". The form contains the following data:

FirstName	James
MiddleName	M
LastName	Bond
Age	40
SSN	22222
City	London
State	London
Country	UK

A "Submit" button is located at the bottom of the form.



Now that we know how to use dynamic formbeans for capturing the form data, it's time to understand how we can do data validation using the same. Since we already ruled out the usage of additional class, the only solution left is the "Validation Framework". Let's see what this is.

Struts Validator Framework

Struts validation framework is a built in validation model for data validation. This framework is based on XML and offers several benefits over the regular formbean validation. Here is one thing that you should always keep in mind. XML based utilities are always powerful and offers immense flexibility.

The validation framework is created by David Winterfeld and it now integrated into the Struts framework as part the distribution. This validator completely eliminates the coding of formbean classes and declaratively configures the validation logic in the XML files. This framework uses the following two XML files for validation:

1. validation-rules.xml
2. validation.xml

The validation-rules.xml is a standard XML file that comes along with the Struts distribution. The validation.xml file is the one that we create and declare all the validations our application needs. Let's understand the usage of these two XML files in detail.

validation-rules.xml

This XML as I said before comes along with the Struts distribution. This file defines all the standard validation functions that are most widely and commonly used in typical web applications. Seldom web applications go beyond the validation rules defined in this file. Following is the basic structure of this XML file:

```
<form-validation>
    <global>
        <validator/>
        <validator/>
    </global>
</form-validation>
```

As shown above, the root element of this XML document is the `form-validation`. This element can have one or more `global` elements, but usually have just one `global` element. The `global` element will have several `validator` elements each defining a validation rule. There is absolutely no limit on the number of validation rules it can have. If the standard rules don't meet your requirements, you are free to add new validation rules. The `validator` element forms the heart of this XML. So, let's see how this element typically looks like.

The `validator` element supports six attributes as shown below:

```
<validator      name="required"
                  classname="org.apache.struts.util.StrutsValidator"
                  method="validateRequired"
                  methodParams="java.lang.Object,
                               org.apache.commons.validator.ValidatorAction,
                               org.apache.commons.validator.Field,
                               org.apache.struts.action.ActionErrors,
                               javax.servlet.http.HttpServletRequest"
```

```
msg="errors.required"
depends="numeric,mask" />
```

Following table lists the purpose of each attribute.

Table 16.2 Validator Attributes

Attribute	Description
name	A unique name that identifies the validation rule. In this case, the <u>name if the rule is required</u>
classname	This represents the class that contains the actual validation logic. This will always be <code>org.apache.struts.util.StrutsValidator</code> which is a standard built-in struts class.
method	This attribute represents the name of the method in the above class that does the validation. Once such method is <code>validateRequired</code> . This method does the required validation for a form field.
methodParams	This represents a comma delimited list of parameters to the above method. It takes five parameters.
msg	This represents the key in the property file whose value should be used in the event of validation error
depends	This attribute defines names of other validation rules that should be fired before firing the current rule. We can specify more than one dependent rules here using a comma delimiter.

This validation-rules XML file will define several such validator elements one for each validation rule. So, don't get scared by the length of the XML file. All the validation rules in this XML file uses the `msg` attribute that defines the property key to get the actual error message as shown below:

```
msg="errors.required"
```

Therefore, for the validation rule to work, we need to define the above key in the properties file with the actual error message as shown below:

```
errors.required = Last Name is required.
```

Every validation rule will have its own message key and therefore we need to define each an every key in the property file. The good thing is, the XML file defines all the message properties in the comment section. We just have to copy paste them into our property file. Following are the validation keys used by the validation rules in this XML file. We sincerely need to copy paste them into the property file.

```

errors.required={0} is required.
errors.minLength={0} can not be less than {1} characters.
errors.maxLength={0} can not be greater than {1} characters.
errors.invalid={0} is invalid.
errors.integer={0} must be an integer.
errors.date={0} is not a date.
errors.range={0} is not in the range {1} through {2}.
errors.creditcard={0} is an invalid credit card number.
errors.email={0} is an invalid e-mail address.

```

Don't worry about the numbers in the message text. We'll look at their significance in the later section. For now, remember them as arguments to the message. For every validation rule in the XML file, there will be a corresponding method defined in the `StrutsValidator` class. Following table lists the rules and corresponding methods.

Table 16.3 Validation functions and equivalent methods

Validation Rule Name	Method in the StrutsValidator
required	validateRequired()
date	validateDate()
creditcard	validateCreditCard()
integer	validateInteger()
minLength	validateMinLength()
Range	validateRange()
email	validateEmail()

The most important validation rules that we frequently use are:

Table 16.4 Important Validation functions

Validation Rule Name	Description
required	Used for required validation for a field (Ex: LastName)
integer	Used for numeric validation for a field (Ex: age, ssn)
minLength	Used for validating minimum length for a field
maxLength	Used for validating maximum length for a field
email	Used for valid email addresses
phone	Used for valid phone numbers

This is all you need to know about the `validation-rules.xml` file. Now, let's look at the second XML file, the `validation.xml`.

validation.xml

This file defines which validation rules in the *validation-rules.xml* are required to be fired for various formbean fields. The XML structure for this file is shown below:

```
<form-validation>
    <global>
        <constant/>
        <constant/>
        -----
    </global>

    <formset>
        <form>
            <field/>
            <field/>
            -----
        </form>

        <form>
            <field/>
            <field/>

            </form>
        </formset>
    </form-validation>
```

As you can see from the above xml fragment, the root `form-validation` element has two child elements namely `global` and `formset`. The `global` element defines one or more `constant` elements and `formset` element defines one or more `form` elements. The `form` element is the most important one that specifies which validation rules need to be fired for various form fields. Consider the following form element

```
<form name="customerForm">
    <field property="lastName" depends="required">
        <arg0 key="customerForm.lastName"/>
    </field>

    <field property="ssn" depends="required,integer">
        <arg0 key="customerForm.ssn"/>
    </field>

    <field property="age" depends="required,range">
        <arg0 key="customerForm.age"/>
        <arg1 name = "range" key="${val:limit1}" resource="false"/>
        <arg2 name = "range" key="${val:limit2}" resource="false"/>
        <var>
            <var-name>limit1</var-name>
            <var-value>10</var-value>
        </var>
        <var>
            <var-name>limit2</var-name>
            <var-value>20</var-value></var>
        </var>
    </field>
</form>
```

The above form element defines validations for three form fields in a formbean named customerForm whose names are lastName, ssn and age respectively. Let's look at the field elements one by one.

Following field defines the validation for lastName.

```
<field property="lastName" depends="required">
    <arg0 key="customerForm.lastName"/>
</field>
```

This field uses the required validation as specified by the depends attribute. This validation uses the following property in the property file to display the error message:

```
errors.required={0} is required.
```

In the above message “{0}” represents the 0th argument which should be replaced with the actual field name. Therefore, we need to pass the field name for 0th argument as shown below:

```
<arg0 key="customerForm.lastName"/>
```

where customerForm.lastName is the property key defined in the property file as shown below:

```
customerForm.lastName = Last Name
```

With all the above, the error message produced for this field is “Last Name is required”.

Look at the following field definition for SSN.

```
<field property="ssn" depends="required,integer">
    <arg0 key="customerForm.ssn"/>
</field>
```

The only difference with this field is that we added integer validation in the depends attribute. For this field, the struts validator first performs the required validation. If a value is entered for this field, then it fires the integer validation that checks whether the value entered in the field is a number or not. If it is not a number, then it generates an error message. Following two properties are used for this validation:

```
errors.required={0} is required.
errors.integer={0} must be an integer.
```

Let me ask you a question. How many arguments do we need to pass for the above two messages? One or two? Good. We have to pass only ONE argument. This is because each of the above messages only need one argument.

Trick: If the “depends” attribute uses say, three validations as “required,integer,range”, then follow a two step process to determine the number of arguments to pass:

1. Identify the error messages for the three functions. This gives us the following

```
errors.required={0} is required.  
errors.integer={0} must be an integer.  
errors.range={0} is not in the range {1} through {2}.
```

2. The total number of arguments is the largest argument number + 1. From the above messages it is $(2 + 1) = 3$.

0th argument will be used by first three messages, and 1st and 2nd arguments will be used by third message. So, if I have three arguments as shown below,

```
<field property="test" depends="required,integer,range">  
  <arg0 key="key1"/>  
  <arg1 key="key2"/>  
  <arg2 key="key3"/>  
</field>
```

and if the following keys are defined in my property file as,

```
key1 =Age  
key2=20  
key3=30
```

then the possible three error messages will be

```
Age is required  
Age must be integer  
Age is not in the range 20 through 30
```

Sometimes, users prefer to define the actual values in the XML itself as opposed to defining them in property file. In the above example, if we want to define values 20 and 30 in the field definition itself, we can do it as shown below:

```
<field property="age" depends="required,range">  
  <arg0 key="customerForm.age"/>  
  <arg1 name = "range" key="${val:limit1}" resource="false"/>  
  <arg2 name = "range" key="${val:limit2}" resource="false"/>  
  <var>  
    <var-name>limit1</var-name>
```

```
<var-value>10</var-value>
</var>
<var>
    <var-name>limit2</var-name>
    <var-value>20</var-value>
</var>
</field>
```

In the above snippet, we defined two variables namely `limit1` and `limit2` with values 10 and 20. Now look at 1st argument. It says, *for range validation (name=range), don't use the property file (resource = false), use the value of limit1 variable (key="\${val:limit1}")*. I am sure the picture is clear. If at any time you want to declare variable values that should be plugged in the error message, you can do so using the above convention.

I think with all the above theory crap, you now know how the struts validator works. To conclude, let's do an example and you'll be an expert. We will use the previous dyna formbean example and add the following validations for `lastName`, `ssn` and `age` fields:

```
lastName: required
ssn: required, integer
age: required, integer, range
```

Take a look at the code in listing 16.8

Listing 16.8 (validation.xml) Validation xml file

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons Validator Rules
    Configuration 1.0//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>

    <formset>

        <form name="dynamicCustomerForm">
            <field property="lastName" depends="required">
                <arg0 key="customerForm.lastName" />
            </field>
            <field property="ssn" depends="required,integer">
                <arg0 key="customerForm.ssn" />
            </field>
            <field property="age" depends="required,range">
                <arg0 key="customerForm.age" />
                <arg1 name="range" key="${var:minLength}"
                    resource="false" />
                <arg2 name="range" key="${var:maxLength}"
                    resource="false" />
            <var>
```

```
<var-name>minLength</var-name>
<var-value>10</var-value>
</var>
<var>
    <var-name>maxLength</var-name>
    <var-value>20</var-value>
</var>
</field>
</form>
</formset>
</form-validation>
```

The above validation xml defines validation rules for a form named `dynamicCustomerForm` as shown below:

```
<form name="dynamicCustomerForm">
```

The form name defined by this element must be exactly same as the formbean name used in the `struts-config.xml` file. Once the above field definitions are defined, we are ready to execute the program.

Steps to run the program

1. Save the `validation-rules.xml` and `validation.xml` file in

```
/myweb/WEB-INF/validation-rules.xml  
/myweb/WEB-INF/validation.xml
```

(Note: The rules xml file comes with struts installation. Just copy it)

2. Add the following properties in "MyStruts.properties" file.

```
errors.required={0} is required.  
errors.minLength={0} can not be less than {1} characters.  
errors.maxLength={0} can not be greater than {1} characters.  
errors.invalid={0} is invalid.  
errors.byte={0} must be a byte.  
errors.short={0} must be a short.  
errors.integer={0} must be an integer.  
errors.long={0} must be a long.  
errors.float={0} must be a float.  
errors.double={0} must be a double.  
errors.date={0} is not a date.  
errors.range={0} is not in the range {1} through {2}.  
errors.creditcard={0} is an invalid credit card number.  
errors.email={0} is an invalid e-mail address.  
  
customerForm.lastName = Last Name  
customerForm.ssn = SSN  
customerForm.age = Age
```

3. Add the following xml snippet in struts-config.xml to turn on the validation framework. Add it next to the message resources section

```
<!-- Plugin Details -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
        value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml" />
</plug-in>
```

4. Update the previous action mapping to validate the data as shown below:

```
<action path="/dynaCustomer" name="dynamicCustomerForm"
    validate="true"
    input="/jsps/struts/CustomerDetails.jsp"
    type="net.nwie.struts.actions.DynamicCustomerAction">
    <forward name="success" path="/jsps/struts/Main.jsp" />
</action>
```

In the above process, Step 3 deserves some attention. Struts validation framework by default will not be enabled. We need to ask the struts framework to enable it by defining it as a plugin as shown above. If you noticed in the plugin declaration, we also need to specify the locations of the two XML files *validation.xml* and *validation-rules.xml*. You don't have to remember the syntax of this element. Just copy and paste it as we only do it once. Ok. In this example, we are using all the code from the previous example. Now that we completed all the steps, start the server and test the following URL

<http://localhost:8080/myweb/jsps/struts/CustomerDetails.jsp>

Try the following cases.

1. Don't enter lastName, SSN and Age. You'll see the errors

```
Last Name is required
SSN is required
Age is required
```

2. Enter lastName, with invalid SSN and Age. You'll see the errors

```
SSN must be integer
Age must be integer
```

3. Enter 40 for Age and you'll see

```
Age is not in the range 20 through 30
```

The screenshot shows a Microsoft Internet Explorer browser window with the URL <http://localhost:9080/Struts/dynaCustomer.do>. The page displays a form with several input fields and error messages. The error messages are:

- Last Name is required. SSN is required. Age is required.
- Please fill in the following details

The form fields are as follows:

FirstName	James
MiddleName	V
LastName	
Age	
SSN	
City	London
State	London
Country	USA

At the bottom of the form is a **Submit** button.

There is one more interesting feature with the struts validator framework. With simple modifications, we can display error messages using JavaScript dialog boxes. Let's do this, and I am sure you'll like it.

Java Script Validation

This is an extension to the validator framework to display the error messages using Javascript dialog boxes. Follow the process below:

1. Update the CustomerForm.jsp page as shown below

```
<form action="dynaCustomer"
      onSubmit="return validateDynamicCustomerForm(this);">
```

2. Also add the following line in the JSP

```
<html:javascript formName="dynamicCustomerForm"/>
```

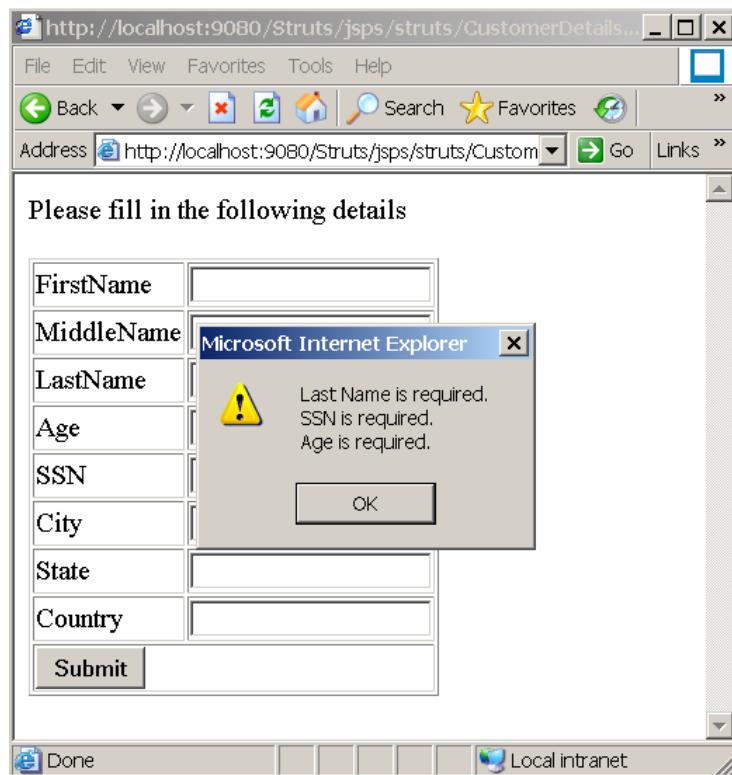
Since our formbean name is `dynamicCustomerForm`, in step 1, we need to specify the `onsubmit` attribute with the JavaScript function name using the following syntax:

validate <form bean Name> (this);

In the above function name, the formbean name should start with an uppercase letter and the function should be terminated by a semicolon as shown below:

```
onSubmit="return validateDynamicCustomerForm(this);"
```

In Step 2, we need to use one of the html tags with the `formName` attribute specifying the formbean name. No changing case here, ok. Now run the JSP again with the same URL as before, and you'll see the error messages as shown below:



This completes all the details of Struts validator framework. Using this framework, we can completely eliminate the formbean classes and make use of xml for data validation which offers more flexibility and ease of use. There is one more utility framework in

Struts just like the validator framework, but this is for a different purpose. It is used for web page layout. Let's see what this framework has to offer.

Tiles Framework

In real world applications there will be several web pages. If you noticed carefully, most of the pages will have certain things in common like headers, footers, navigation bars etc. A typical web page can be viewed as composite page built from several small pages (tiles) arranged according to a fixed layout called master layout. By using master layouts to build composite pages, we can increase the flexibility of the web pages. Tomorrow, if you want to use a different layout for the page, all you have to do is change the master layout, and all your web pages will take the new form automatically. This is cool, right? You don't have to manually change each and every page to adjust to the new layout. Struts tiles framework supports building such composite pages using master layouts.

Using struts tiles framework is very simple. There is a standard and simple process to follow as depicted below:

1. Create a master layout JSP page with different tiles. Every tile will have a unique name to identify it.
2. Create smaller JSP pages that will be used to build the composite page. The smaller pages are referred to as tiles.

The tiles framework like the validation framework uses an XML file to define the master layouts and tile information. The name of this file can be anything, but we usually name it as `layout-tiles-defs.xml`. The details of this file are deferred to a later section.

To use Struts tiles framework, we need to configure the tiles *plugin* just like we plugged in the struts validator in the configuration file. Following is the plugin information that we will be adding to the `struts-config.xml` file to turn on the tiles framework.

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
    <set-property property="definitions-config"
        value="/WEB-INF/layout-tiles-defs.xml" />
    <set-property property="moduleAware" value="false"/>
    <set-property property="definitions-parser-validate" value="true"/>
</plug-in>
```

If you notice the above plugin element, it takes the location of the tiles XML file as a parameter as shown below:

```
<set-property property="definitions-config"
    value="/WEB-INF/layout-tiles-defs.xml" />
```

Once we added the above plugin in the configuration file, we are ready to build applications using tiles. At the beginning of the chapter, we learned three categories of struts custom tags namely html tags, bean tags and logic tags. There is one more category of tags which is exclusively used with tiles framework. These are referred to as tiles tags. The tags are normally prefixed with the word "tiles". Following are the important ones.

```
<tiles:definition id="" page="" />
```

This tag is used by the composite page to specify the master layout to use.

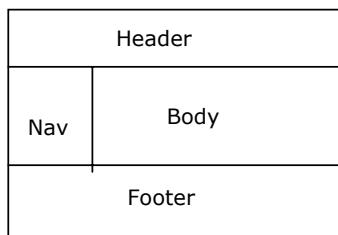
```
<tiles:insert attribute="" />
```

This tag is used to insert the tile in the master layout.

```
<tiles:put name="" value="" />
```

This tag is used by the composite page to associate the actual tile page in to be used in the master layout.

The above three tags are more than enough to develop JSP pages using tiles. Let's see a quick example to build a composite JSP page based on the following master layout.



Take a look at the code in listing 16.9.

Listing 16.9a (masterLayout.jsp) Master layout JSP page.

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>

<html>
<body>

<TABLE border=1>
<TR>
    <TD colspan=2 width=1001 height=93><div >
```

```
<td>
    <tiles:insert attribute="header" ignore="true" />
</td>
</tr>
<tr>
    <td width=227 height=604>
        <tiles:insert attribute="navigator" ignore="true" />
    </td>
    <td width=774 height=604>
        <tiles:insert attribute="body" ignore="true" />
    </td>
</tr>
<tr>
    <td colspan=2 width=1001 height=107>
        <tiles:insert attribute="footer" ignore="true" />
    </td>
</tr>
</table>
</body>
</html>
```

If you look at the JSP code, we used the tiles insert tags to define various tiles in the master layout . Following is how this tag is used.

```
<tiles:insert attribute="header" ignore="true"/>
```

As you can see from the above tag, every tile will have a unique name like header in the above tile. When we actually build the composite page using the master layout, we'll associate the header tile with the actual JSP content. Let's now build the smaller pages also referred to as tiles.

Listing 16.9b Individual tile pages.

Header.jsp

```
<html>
    <body>
        <h1> This is a Header. You can build a fancy one </h1>
    </body>
</html>
```

Footer.jsp

```
<html>
    <body>
        <h1> This is a Footer. You can build a fancy one </h1>
        <br/><h3> <i>Copyright HelloWorld Inc, 2006 </i></h3>
    </body>
</html>
```

Body.jsp

```
<html>
    <body>
        <h1> This is a Body. You can build a fancy one </h1>
```

```
</body>
</html>

Navigator.jsp

<html>
<body>
    <h1> This is a Navigator. You can build a fancy one </h1>
</body>
</html>
```

The above code forms the smaller pages or tiles that be used in building the composite page. Let's now build the composite page.

Listing 16.9c A composite JSP page.

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>

<tiles:insert page="/jsps/struts/layouts/masterLayout.jsp" flush="true">

    <tiles:put name="header" value="/jsps/struts/Header.jsp"/>
    <tiles:put name="footer" value="/jsps/struts/Footer.jsp"/>
    <tiles:put name="navigator" value="/jsps/struts/Navigator.jsp"/>
    <tiles:put name="body" value="/jsps/struts/Body.jsp"/>

</tiles:insert>
```

Look at the above composite page. There are two important things it does as listed below:

1. Uses the masterLayout.jsp using the tiles:insert tag.
2. Associates the actual tile pages to different tiles in the master layout. For instance, Header.jsp page will replace the tile identified by header in the master layout page. That's it.

Steps to run this example

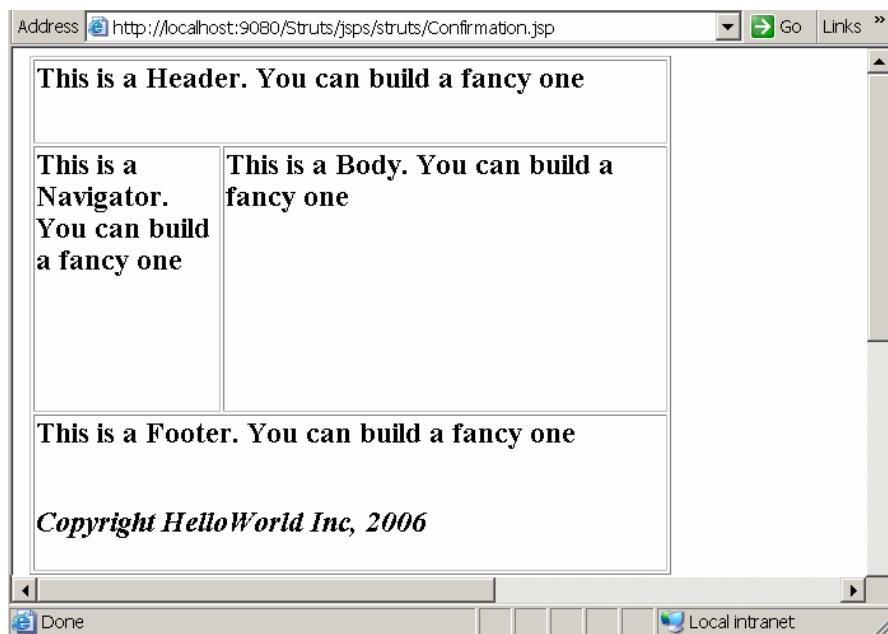
1. Save the JSP files as

```
myweb/jsps/struts/layouts/masterLayout.jsp
myweb/jsps/struts/Header.jsp
myweb/jsps/struts/Footer.jsp
myweb/jsps/struts/Body.jsp
myweb/jsps/struts/Navigator.jsp
myweb/jsps/struts/Composite1.jsp
```

Start the server and type the following URL to access the Composite page:

http://localhost:8080/myweb/jsp/struts/Composite1.jsp

The above URL produces the following result:



As you can see from the above result, the smaller pages got assembled and formed up a composite page using the master layout. Though the above example succeeded in building composite page, I see small maintenance problem. Let's say want to build 10 composite pages using the same master layout, I need to physically build ten different JSP files as Compsite1.jsp, Composite2.jsp and so on, right? If you carefully observe the composite page, the content is just the tile definition as shown below:

```
<tiles:insert page="/jsp/struts/layouts/masterLayout.jsp" flush="true">  
    <tiles:put name="header" value="/jsp/struts/Header.jsp"/>  
    <tiles:put name="footer" value="/jsp/struts/Footer.jsp"/>  
    <tiles:put name="navigator" value="/jsp/struts/Navigator.jsp"/>  
    <tiles:put name="body" value="/jsp/struts/Body.jsp"/>  
</tiles:insert>
```

Of course, each will have its own definition though. But still, do we really need to create a separate JSP just for the above definition? If there is a solution where we can store all the definitions in one file and permanently eliminate creating composite JSP pages then I guess it is the best solution, just like we eliminated the formbeans classes using

dynaformbeans. So the question is, do we have such solution? Thankfully Yes. We can store all the tiles definitions for *all* the composite pages in a single XML file and then have the tiles framework assist us in selecting a suitable definition. This XML is nothing but the `layout-tiles-defs.xml`.

After creating this XML file, we need to copy paste the tiles definition we had in the JSP page in to the XML file and make few modifications. The definition in the XML will look as shown below:

```
<!DOCTYPE tiles-definitions PUBLIC
  "-//Apache Software Foundation//DTD Tiles Configuration//EN"
  "http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">

<tiles-definitions>

  <definition name="composite1"
    path="/jsps/struts/layouts/masterLayout.jsp">

    <put name="head" value="/jsps/struts/Header.jsp" />
    <put name="body" value="/jsps/struts/Body.jsp"/>
    <put name="navigator" value="/jsps/struts/Navigator.jsp"/>
    <put name="footer" value="/jsps/struts/Footer.jsp" />
  </definition>

</tiles-definitions>
```

Can you notice the difference? We use the `definition` element with two attributes. The `name` attribute identifies the composite page, and the `path` attribute specifies the master layout page to use for that definition. The `body` elements will use the `put` element instead of `tiles:put` tag. That's it. Now you can permanently delete the `Composite1.jsp` file.

Let me ask you a question here. Earlier, since we had the `Composite1.jsp` file, we called it directly from the browser to see the response. Now that we deleted it and moved the definition to an XML file with the name `composite1`, the question is, how can we call this definition which is in the XML file from the browser? The answer is simple. You can't call the definition *directly* from the browser. This logically means we can *indirectly* call it, right? This is done by simply writing an action mapping in `struts-config.xml` fine that forwards the request to the tiles definition as shown below:

```
<action path="/compl" forward="composite1" />
```

Once we have the above mapping, we will call the action from the browser as shown below:

```
http://localhost:8080/myweb/compl.do
```

The action mapping will then forward the request to the tiles definition named composite1 from the tiles XML file. You can also define the tiles definition in the action forwards section as shown below:

```
<forward name="success" path="composite1"/>
```

Implement the following steps to use XML definition for the previous example:

1. Copy and save the above XML in the following file as
/WEB-INF/layout-tiles-defs.xml
2. Add the following tiles plugin in the struts-config.xml file next to the validator plugin.

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">  
    <set-property property="definitions-config"  
        value="/WEB-INF/layout-tiles-defs.xml" />  
    <set-property property="moduleAware" value="false"/>  
    <set-property property="definitions-parser-validate" value="true"/>  
</plug-in>
```

3. Add the following action mapping in the struts-config.xml file

```
<action path="/comp1" forward="composite1" />
```

3. Start the server and type the following URL. You'll see the same result.

```
http://localhost:8080/myweb/comp1.do
```

Let's say I want to build another composite page that should have the exact same header, footer, navigator but different body. Normally my new definition in the tiles xml file will look as shown below:

```
<definition name="composite2" path="/jsps/struts/layouts/masterLayout.jsp">  
    <put name="head" value="/jsps/struts/Header.jsp" />  
    <put name="body" value="/jsps/struts/Body1.jsp" />  
    <put name="navigator" value="/jsps/struts/Navigator.jsp" />  
    <put name="footer" value="/jsps/struts/Footer.jsp" />  
</definition>
```

If you observe the above definition, all we changed is the name attribute and the body element with the new JSP page Body1.jsp. Though the above definition works perfectly fine, I see some redundant code. The other three put elements are duplicated in my definition, right? The best solution will be to reuse the put elements from the composite1 definition. To do this, we simply use extends attribute as shown below:

```
<definition name="composite2" extends="composite1" >
    <put name="body" value="/jsps/struts/Body1.jsp"/>
</definition>
```

All we need to do is remove the path attribute, and add extends attribute with the name of the definition from which it should inherit the tiles. Now you can also remove the three duplicate put elements. This is tiles definition inheritance in which one definition inherits tiles from other definitions. This way of writing definitions is what we normally do in real world applications.

Implement the following steps to create a new composite page:

1. Copy the above composite2 definition into the “layout-tiles-defs.xml” file.
2. Create a new Body2.jsp how ever you want and save it as

myweb/jsps/struts/Body2.jsp

3. Add a new action mapping in the struts-config.xml file as shown below:

```
<action path="/comp2" forward="composite2" />
```

4. Start the server and type the above URL.

<http://localhost:8080/myweb/comp2.do>

This completes all the things we need to know in tiles framework. We now arrived at the last and final concept, internationalization. You’ll love this one. Trust me.

Internationalization

Internationalization feature in Struts allows your web applications to run in different languages. Let’s say we are trying to build a global web application. Though we can have all the web pages in the universal language “English”, some people prefer their regional languages. As a business person, our goal is to reach all possible customers all over the world, right? So, if you only build the application in one language, are you not losing customers and business? So, let’s see how we can tune our web application to run in different country say France in French language.

The key idea behind internationalization feature is something called resource bundles. A resource bundle is nothing but a collection of several property files. Java and Struts uses the following two things to achieve internationalization:

1. Locale
2. Resource Bundles

Locale

This is a class in Java library that specifies the locale of the country in which the application is running. Following are some of the standard locales:

en - English
fr - France
de - German
es_US - Spanish in US
ru - Russian

Resource Bundles

As the name suggests, it's a bundle of resources and resources are nothing but the property files that have messages based on the locale of the country. Every locale will have its own property file as shown below:

MyStruts_en.properties
MyStruts_fr.properties
MyStruts_de.properties

Notice the names of the property files. They must end with an “_locale”.

Struts and Internationalization

The trick to build universal web applications is to define the same messages in several property files, one per locale or language. So, if I want my web application to run in English, French and German, I will have three property files as shown below:

MyStruts_en.properties
MyStruts_fr.properties
MyStruts_de.properties

Every property file will have the same keys defined in English, but the values will be in different language as shown below:

```
MyStruts_en.properties
error.username= User Name is Invalid
MyStruts_fr.properties
```

```
error.username= Usero Un Invalido  
MyStruts_de.properties  
error.username= La De User Ine
```

(The above messages are not really French and German texts. It's just some crap I used to simulate :-)

Once we defined the above property files, Struts will smartly use the messages based on the country in which the application is running. Let's write a small application that runs in French language. What we will do is create some property files and run our previous examples to see different messages.

Steps to implement

1. Create a property file named "MyStruts_fr.properties" and copy the following properties.

```
error.username = Invalid Da Le User Nam  
error.password = Invalido Un Passwordino  
error.lastName.required = LastName is required <br/>  
error.ssn.required = SSN is required</br>  
error.ssn.invalid = Invalid Da Length for un SSN  
error.age.invalid = Invalid Na Value. Age Da La must Na Numeric.  
customerForm.lastName = Last Name  
customerForm.ssn = SSN  
customerForm.age = Age  
errors.required={0} la de requiro  
errors.integer={0} integero un presento  
errors.range={0} is not in the range {1} through {2}.
```

Notice that the keys will be exactly same, but the values will be different.

Save this file as myweb/WEB-INF/classes/myprops/MyStruts_fr.properties

2. Make sure the following message definition is present in the struts-config.xml (No "_locale" extension)

```
<message-resources parameter="myprops.MyStruts" />
```

3. Change your system locale to French as indicated below:

Start->Settings->Control Panel>Regional And Language Options

Select the language as French (France) in the drop down and click OK.

4. Start the server and run the dynaformbean example with the following URL

`http://localhost:8080/myweb/jsp/struts/CustomerDetails.jsp`

The above URL will produce the following result



Look at the error messages now. Do you like them? I am sure you do. Now go back and change the language back to English. From the developer standpoint all we need to do is create a new property file and store the messages in different language. The Struts framework will take care of the rest. This is the beauty of Struts. I am sure you now figured out why Struts has become a de facto standard for J2EE based web applications.

This completes all the concepts we need to know in Struts. You are now an expert in building web applications and let me say this, you are now a professional Struts, JSP or even better a J2EE developer.

Summary

- ✓ Struts is an open source framework for building web applications.
- ✓ Struts is based on Model View Controller architecture.
- ✓ The major components in Struts are the formbeans and action classes.

- ✓ Formbeans represent the HTML form data and also used to validate the data.
- ✓ Action classes are used to process the requests.
- ✓ All the Struts components are defined in a configuration file named struts-config.xml.
- ✓ Dynaformbeans eliminate the formbean classes by defining the form properties in the configuration file itself.
- ✓ Dynaformbeans use struts validator to validate the form data.
- ✓ Struts includes several built-in custom tags. These are categorized into html, bean, logic and tiles tags.
- ✓ HTML tags are used to create standard html elements like forms, textfields etc.
- ✓ Bean tags are used to work with JavaBeans and their properties.
- ✓ Logic tags are used to general purpose operations like iterations, conditions etc in the JSP page.
- ✓ Tiles tags are used for page layout.
- ✓ Struts messages are usually defined in property files which allow internationalization of the application.
- ✓ Struts validator framework is an XML based framework for validating the form data. This framework also supports messages displayed using Java Script dialog boxes.
- ✓ Struts tiles framework is used for building composite pages using master layouts. This framework use XML to define tiles definitions.

Time to play 50-50

1. Which of the following design is used by Struts framework?
 - a) MVC1 Design
 - b) MVC2 Design
2. Which of the following class represent controller component in struts?
 - a) ControllerServlet
 - b) ActionServlet
3. Which of the following class is used to store the form data?
 - a) Action
 - b) ActionForm
4. Which of the following class is used to process the request?

- a) Action
 - b) ActionForm
5. Which of the following component is used to validate the form data?
- a) ActionForm
 - b) Action
6. Which of the following file is used to configure struts applications?
- a) web.xml
 - b) struts-config-xml
7. Which of the following element is used to define global forwards in the configuration file?
- a) <global-forwards>
 - b) <action-mappings>
8. Which of the following element is used to define the formbeans in the configuration file?
- a) <action-mappings>
 - b) <form-beans>
9. Which of the following eliminates writing form bean class?
- a) ActionForm
 - b) DynaActionForm
10. Which of the following element is used to define action classes in the configuration file?
- a) <action-mappings>
 - b) <actions>

Interview Questions

Question: Which design does Struts framework use?

Answer: Model View Controller (MVC) design.

Question: Explain how MVC design works?

Answer: Refer to page 447.

Question: What is the difference between MVC1 and MVC2 architecture?

Answer: In MVC1 architecture, a JSP page acts as both View and Controller component while in MVC2 architecture, servlet is used as controller and JSP is used as View component.

Question: List the important Struts components.

Answer: Request Processors, Form beans, Action classes, Action Forwards, Action Mappings, Action Errors etc.

Question: What is the name of the configuration file used in Struts?

Answer: struts-config.xml

Question: What are the various elements defined in the configuration file?

Answer: Form beans, Global Forwards, Global Exceptions, Action Mappings, Message resources, Plugins (Validator and Tiles).

Question: What is the difference between a local forward and a global forward?

Answer: A local forward can be used by a single action class where as global forwards can be used by all the action classes.

Question: What is the name of the XML file in which all the validation rules are defined?

Answer: validation-rules.xml

This completes everything you need to know about Struts. I am sure you enjoyed this chapter. The next few chapters are smaller ones and should be a cakewalk once you arrived at this point.

Chapter 17

Java Messaging Service (JMS)

This chapter introduces you to asynchronous messaging using JMS. By the end of this chapter you'll know the basics of messaging systems and how synchronous and asynchronous messaging is implemented.

Chapter Goals

- ✓ Understand the notion of messaging systems.
- ✓ Understand JMS architecture
- ✓ Understand various JMS messaging models

Environment Setup

1. Download the ActiveMQ messaging system at the following URL.
<http://incubator.apache.org/activemq/activemq-32-release.html>
2. Copy the activemq-3.2.jar file from C:\ActiveMQ\lib directory into j2ee\lib directory. Create C:\JavaTraining\chapter17\env.bat with the following contents.

```
set CLASSPATH=.;  
C:\JavaTraining\j2ee\lib\j2ee.jar;C:\JavaTraining\j2ee\lib\mysql  
-connector-java-5.0.3-bin.jar;C:\JavaTraining\j2ee\lib\  
activemq-3.2.X.jar
```

Run the env.bat file before running the examples. This completes the environment setup.

Introduction

This chapter introduces you to yet another powerful J2EE technology called JMS. JMS stands for Java Messaging Service. Before we understand what JMS is all about, we need to understand the basics of enterprise level communication. In a typical enterprise application most of the communication is synchronous, which is based on request – response paradigm. In this type of communication, sender sends a request message and receives a response message instantly from the recipient of the message without having to wait for an indeterminate amount of time. This is a two-way communication. Most of the enterprise communication falls in this category. The simplest example of synchronous communication is HTTP communication in which we send a HTTP request from the browser and the server instantly responds with a HTTP response message. In this case both the parties namely sender and receiver are always active.

Besides synchronous messaging, there will be several cases where the communication is required to be asynchronous in nature. This is one way communication in which sender sends a message to the recipient and doesn't wait for any response or confirmation from the recipient. The sender doesn't even care whether the recipient received the message or not. This is where the challenge comes. What if the recipient of the message is *inactive* at the time when the sender sends the message? The message is lost in the air, right? Yes it will be lost. So, you might wonder why enterprise applications really need to do asynchronous communication when it is unreliable. Before we discuss a solution for this, let's first understand the importance of asynchronous communication.

Importance of Asynchronous messaging

One of the primary benefits of asynchronous messages is that the sender and receiver of the messages can be time independent. This means the sender can send messages at any time, and the recipient can read the messages at some other time. Typical examples include email systems, message boards, discussion forums etc.

Let's consider a typical eCommerce application which sells wide variety of products from different manufacturers. One of the main components of such application is the inventory management system. This system is required to update the inventory of products based on sales and availability and this has to be in real time. Let's say it has to send orders to three vendors namely A, B and C. If the inventory system uses synchronous messaging to send orders one by one, then tell me what happens if vendor A is down? Should the system wait until A comes up? If this the case, the orders to B and C will also be delayed, right? Why should B and C vendors suffer due to vendor A being down? One solution is, the inventory system can temporarily ignore vendor A,

process B and C and come back to A at some other time. This is not a good approach because the inventory system should now keep a track of which orders are placed and which are not. This is an additional burden on the system, right? The simple solution is to have the inventory system send all the orders to an “**always running middle ware system**” and then have the three vendors read the orders from the middle ware system. This will totally eliminate the dependency of inventory system on vendors A, B and C. From inventory system point of view, all it has to do is send the appropriate messages to the middle ware, that’s it. It is vendor’s responsibility to read messages from the middle ware and update the inventory. The communication between inventory system and the middleware is 1-way which is *asynchronous* messaging. Similarly, the communication between vendors and middleware is also asynchronous. Following figure demonstrates this form of messaging.

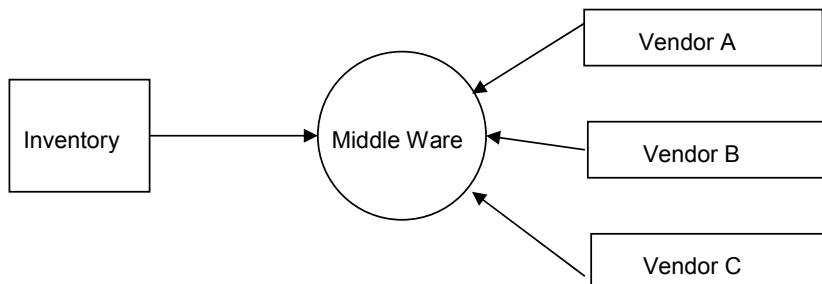


Fig 17.1 Typical Asynchronous Messaging

The middleware is now the heart of the application. All we did is broke the synchronous communication into asynchronous communication by introducing the middleware. This middleware is what we call as MoM system (Message oriented Middleware).

Remember I said at the beginning about a solution to reliable asynchronous messaging? Here it is. The presence of MoM in between the parties is the solution. The MoM will be built in such a way that it 100% guarantees the delivery of the messages to recipients (vendors A , B and C). This is referred to as *quality of service*.

MoM (Message oriented Middleware)

Following are the key features of any MoM system:

1. Infrastructure that asynchronously connects multiple systems.
2. Supports reliable and fast communication
3. Guaranteed message delivery, recipient notification and transaction control.

4. Eliminates the need for all systems to be available simultaneously.
5. Hides all the underlying communication details from producers and consumers of messages.

MoM can be viewed as a database of messages. As an application developer, our goal is to send and receive messages to/from the MoM system. There are several free and commercial MoM systems available in the market. The popular one is the MQSeries from IBM. This is a commercial product, but is available as 60 day trial version. In this chapter, I am going to use ActiveMQ product from Apache as the MoM system. Please see the environment page for installing this product. This is an open source messaging broker. Following picture shows a typical MoM with producers and consumers of messages.

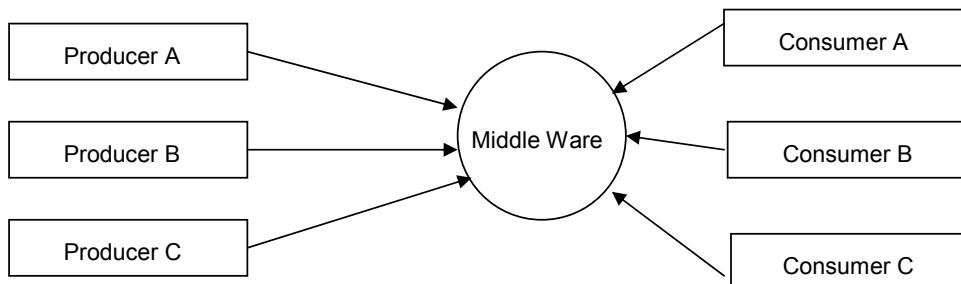


Fig 17.2 A typical MoM based application

Let me ask you a question here. Does MoM only support asynchronous communication (1-way)? No, MoM systems also support synchronous communication. Now that we know some basics about messaging systems, let's see the different messaging models these systems support.

Messaging Models

There are two popular messaging models that every MoM system supports. These are:

1. Publish/Subscribe model
2. Point-to-Point model

Publish/Subscribe Model

As the name suggests, in this model, the producer publishes the messages to something called *Topics*. The consumers of the messages will subscribe with the published topics

and receive the messages. This is *one-to-many* messaging model in which multiple consumers can subscribe to the same topic and receive the messages. A single consumer can also subscribe with several topics. The examples of this model are newsletters where we subscribe with topics called “Sports” to receive all the sports related news.

Fig 17.3 shows a publish subscribe messaging model.

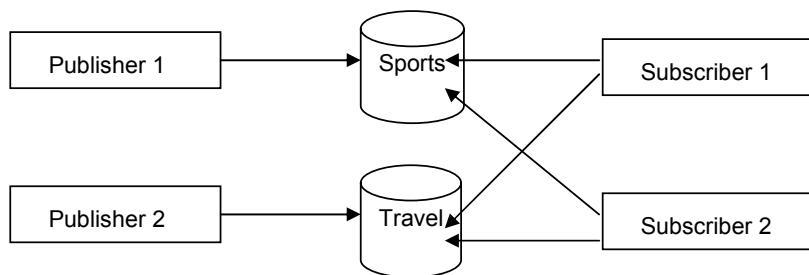


Fig 17.3 Publish-Subscribe model

As you can see from the above figure, one publisher publishes the messages to a topic named sports and both the subscribers subscribed to this topic to receive messages. In this model, *Topic* is an administered object.

Point-to-Point Model

In this messaging model, a producer puts a message into something called a *Queue*, and the consumer reads the message from the queue. This is a one-to-one messaging model. Unlike in pub/sub model, only one consumer can receive the message that is put in the queue. Once the message is consumed by the consumer it will no longer be available in the queue. In this model, *Queue* is an administered object. Fig 17.4 shows this model.

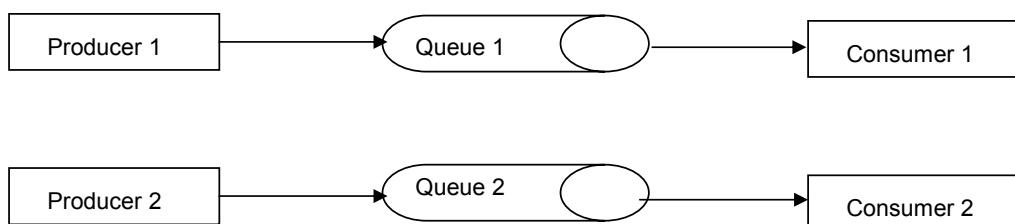


Fig 17.4 Point-to-Point messaging model

Most of the real world applications use point-to-point model due to the flexibility it offers. However, there will also be some applications that use pub/sub model. In this chapter, we will see example using both the messaging models. Fair enough?

Let me ask you a question here. Since MoM is a proprietary messaging system that can be implemented in any language, how do we access it from Java? The answer is simple. Recall how we used JDBC to access any type of database. Similarly we also have a J2EE technology that provides a universal way of accessing any MoM system. This technology is nothing but JMS, which stands for Java Messaging Service. So, to build synchronous/asynchronous applications in Java using MoM systems, we need to use JMS API. Good.

JMS API

JMS API is a set of simple classes we use to send and receive messages to and from MoM systems. In this chapter, we will use ActiveMQ MoM broker from Apache. There is a standard process we follow while working with MoM systems, no matter what the messaging model is. Fig 17.5 in the next page outlines the process.

As you can see from the figure, there are three steps we usually do to work with messaging system.

1. An Admin tool will bind the administered objects (Topics,Queues etc) in the JNDI namespace.
2. Java application will then use JNDI to lookup the admin objects.
3. Using the admin objects, the application establishes a connection with the messaging system and performs the messaging.

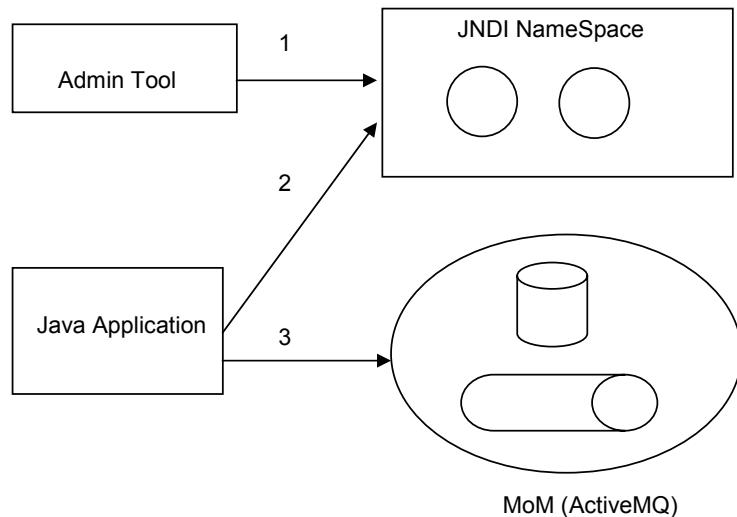


Fig 17.5 JMS and Messaging Systems

For now, don't worry about the details of what JNDI is. Just think of it as a Java based solution for storing and retrieving the objects from a common storage place (a.k.a registry) for Java objects. When we write programs, you'll see how we can use JNDI.

JMS Administered Objects

Administered objects are nothing but pre-configured objects that encapsulate all the information about the underlying messaging system. The information is usually the hostname, port number etc, used for connecting with the system. These objects are usually configured by the application administrators. As a developer, we just need to use the admin objects and connect with messaging systems. There are only four administered objects in JMS. These are:

For Pub/Sub model

- ✓ TopicConnectionFactory
- ✓ Topic

For Point-to-Point model

- ✓ QueueConnectionFactory
- ✓ Queue

We will see what these are and how to use them in later sections.

Using JMS API

The usage of JMS API is pretty simple. There are few important classes we need to know, and we use them with a simple and standard process. JMS API has classes for both Pub/Sub model and Point-to-Point model. All the classes in the Pub/Sub model start with the word *Topic*, and classes that support point-to-point messaging start with the word *Queue*. The class names are very easy to remember as you can see from the following table.

Table 17.1 JMS API

Pub/Sub API	Point-to-Point API
TopicConnectionFactory	QueueConnectionFactory
Topic	Queue
TopicConnection	QueueConnection
TopicSession	QueueSession
TopicPublisher	QueueSender
TopicSubscriber	QueueReceiver

Following is the standard process to work with JMS.

1. Use JNDI to search and retrieve Topic/QueueConnectionFactory objects.
2. Create a Topic/QueueConnection from the factory class
3. Start the connection
4. Create a Topic/QueueSession from the connection
5. Use JNDI to search and retrieve Topic/Queue
6. Create either TopicPublisher/QueueSender from session
7. Publish/Send the messages
8. Subscribe/Receive messages
9. Close the connection

For point-to-point messaging we use Queue related classes and for pub/sub messaging we use Topic related classes.

JNDI lookup for Administered objects

All the admin objects namely TopicConnectionFactory and Topic for Pub/Sub model and QueueConnectionFactory and Queue for point-to-point model are bound

to the JNDI namespace in the server. As an application developer, we need to use JNDI API and look up (search) for these objects.

The first thing we need to do before looking up for admin objects is to obtain a JNDI context by supplying in the MoM information. This information is usually the context factory class of the messaging system, and the URL with the hostname and port number as shown below:

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,"org.apache.activemq.jndi.
                                         ActiveMQInitialContextFactory");
props.setProperty(Context.PROVIDER_URL,"tcp://localhost:61616");
Context context = new InitialContext(props);
```

The factory class is vendor specific just like a driver class for database. The above factory class is an ActiveMQ proprietary class. These two properties will change based on the underlying messaging system. Once the context is obtained, we do a lookup (search) for administered objects namely, `QueueConnectionFactory` or `TopicConnectionFactory`. For ActiveMQ, these objects are bound with the names "ConnectionFactory". The lookup for objects will therefore be as shown below,

```
TopicConnectionFactory topicFactory =
        (TopicConnectionFactory)context.lookup("ConnectionFactory");
QueueConnectionFactory queueFactory =
        (QueueConnectionFactory)context.lookup("ConnectionFactory");
```

Publish/Subscribe Messaging using JMS

For this messaging model, we use the `TopicConnectionFactory` object obtained from the above look up. Using this factory object, we use other topic related classes in JMS API to publish and subscribe messages.

Obtaining the connection and starting it (Steps 2 & 3)

```
TopicConnection topicConn = topicFactory.createTopicConnection();
topicConn.start();
```

Creating a TopicSession (Step 4)

The JMS method signature to create a topic session is:

```
public TopicSession createTopicSession(boolean transacted, int acknowledge)
```

Method Arguments:

transacted – If true, the session will be transacted.

acknowledge – An integer that specifies the acknowledgement mode by the recipient of the message. The usual values are

AUTO_ACKNOWLEDGE – The message will be automatically acknowledged by the recipient session.

CLIENT_ACKNOWLEDGE – The recipient will explicitly acknowledge the message by calling the acknowledge() method

Using the above method, we create the session as shown below:

```
TopicSession topicSession = topicConn.createTopicSession(false, Session.  
AUTO_ACKNOWLEDGE);
```

Obtain the Topic from JNDI (Step 5)

```
Topic topic = (Topic) context.lookup("dynamicTopics/Sports");
```

Steps 1 though 5 are common for both Publisher and Subscriber code. You can copy paste them.

Following two steps are used in Publisher code.

Creating a Publisher

Create a publisher for the topic using the session as shown below (Step 6)

```
TopicPublisher publisher = topicSession.createPublisher(topic);
```

Creating Messages

Create and publish messages using the above publisher object (Step 7)

There are different types of messages we can create based on the type of data to be sent to the recipient. Following are some of the important JMS messages:

1. TextMessage (Widely used for sending text messages)
2. ByteMessage (Used for sending bytes of data)
3. ObjectMessage (Used for sending objects)
4. MapMessage (Used for send data as name-value pairs)

To send a text message, we use the `TextMessage` class and specify the text as shown below:

```
TextMessage message = topicSession.createTextMessage();
message.setText("A simple message");
```

To publish the above message to the topic, we invoke the `publish()` method as shown below:

```
publisher.publish(message);
```

The overall consolidated code for the publisher will be as shown below:

```
public void sendMessage() throws JMSEException {
    TopicConnection topicConn = topicFactory.createTopicConnection();
    topicConn.start();

    TopicSession topicSession = topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

    Topic topic = (Topic) context.lookup("dynamicTopics/Sports");

    TopicPublisher publisher = topicSession.createPublisher(topic);

    TextMessage message = topicSession.createTextMessage();
    message.setText("A simple message");

    publisher.publish(message);
}
```

Coming to the subscriber part, there are two types of subscriptions we can make as shown below:

1. Non-Durable subscriptions
2. Durable subscriptions

Let's see what these are.

Non-Durable subscriptions

With non-durable subscriptions, the subscriber of the topic will receive the published messages only when the subscriber connection is *active*, i.e, the subscriber application should be up and running and continuously listen for messages. This type of subscription will consume the messages as and when they are published. The subscriber code should register a listener object that should continuously listen and read the published messages.

In order for the subscriber not to lose any of the published messages, the subscriber code must be run before the publisher code. Following are the steps for creating non-durable subscriptions

Creating a Subscriber

The subscriber for a particular topic should be created as shown below:

```
TopicSubscriber subscriber = session.createSubscriber(topic);
```

Create and set the Listener

To create a listener object, we need to write a class that implements `MessageListener` interface and implement the `onMessage()` method as shown below:

```
class MyListener implements MessageListener{  
    public void onMessage(Message msg){  
        // Cast to what ever message published  
        TextMessage tm = (TextMessage)msg;  
        String message = tm.getText();  
        System.out.println(message);  
    }  
}
```

Once the listener class is created, it should be registered with the subscriber as shown below:

```
subscriber.setListener( new MyListener() );
```

This completes the subscriber code. It is now ready to listen for published messages.

Durable Subscriptions

With durable subscriptions, the subscriber of a topic can receive messages even if the subscriber connection is *inactive*. If any messages are published during this inactive time, the messaging system will persist them internally, so that when the subscriber comes alive, he will receive the earlier messages and also the messages from that point onwards. With durable subscriptions, there is one important thing called `clientID` that should be set on subscriber side. The code for durable subscriber will look as shown below:

On Subscriber side:

```
conn.setClientID("Bond");
```

```
TopicSubscriber subscriber = session.createDurableSubscriber(topic, "Bond");
```

All the remaining code will be the same as durable subscriptions. Without wasting any further time, let's see examples with both durable and non-durable subscriptions.

In this example, we will create a publisher that publishes three messages. We will create two subscribers to consume the messages durably and non-durably. See the code in listing 17.1

Listing 17.1a (`Publisher.java`) Publisher code

```
package jms;

//JNDI imports
import java.util.Properties;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Publisher {

    public Publisher() {
        try {
            Properties props = new Properties();
            props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
            props.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");

            Context context = new InitialContext(props);

            TopicConnectionFactory tcf = (TopicConnectionFactory) context
                .lookup("ConnectionFactory");

            TopicConnection conn = tcf.createTopicConnection();
            // Start the connection
            conn.start();

            // and should use automatic message acknowledgement.
            final TopicSession session = conn.createTopicSession(false,
                Session.CLIENT_ACKNOWLEDGE);

            Topic topic = (Topic) context.lookup("dynamicTopics/Sports");
            TopicPublisher publisher = session.createPublisher(topic);

            // Send messages to 3 users.

            TextMessage msg1 = session.createTextMessage();
            msg1.setText("Hi John, How are you doing");

            TextMessage msg2 = session.createTextMessage();
```

```
msg2.setText("Hi Smith, How are you doing");

TextMessage msg3 = session.createTextMessage();
msg3.setText("Hi James, How are you doing");

publisher.publish(msg1);

publisher.publish(msg2);

publisher.publish(msg3);

System.out.println("Messages published");

// Close the connection
conn.close();

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

}

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

Listing 17.1b (Subscriber.java) Subscriber code

```
package jms;

import java.util.Properties;
import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Subscriber {

    public Subscriber() {
        try {
            Properties props = new Properties();

            props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
            props.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");

            Context context = new InitialContext(props);

            TopicConnectionFactory tcf = (TopicConnectionFactory) context
                .lookup("ConnectionFactory");

            TopicConnection conn = tcf.createTopicConnection();

            TopicSession session = conn.createTopicSession(false,
                Session.AUTO_ACKNOWLEDGE);

            Topic topic = (Topic) context.lookup("dynamicTopics/Sports");
        }
    }
}
```

```
TopicSubscriber subscriber = session.createSubscriber(topic);

TextListener topiclistener = new TextListener();
subscriber.setMessageListener(topiclistener);
conn.start();

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

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

}

class TextListener implements MessageListener {

    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Reading message: " + msg.getText());
            } else {
                System.out.println("Message of wrong type: "
                    + message.getClass().getName());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

As you can see from the above Publisher and Subscriber code, they follow the same process that we outlined before. Save the above two files in jms package and compile both the programs as shown below:

```
C:/>JavaTraining>chapter17>javac jms\*.java
```

Now start the ActiveMQ server using the following command.

```
C:/>ActiveMQ>bin>activemq
```

Run the subscriber code using the following command. This will wait indefinitely until the messages are consumed.

```
C:/>JavaTraining>chapter17>java jms.Subscriber
```

Now, open another command window and run the Publisher code using the following command.

```
C:/>JavaTraining>chapter17>java jms.Publisher  
Messages Published
```

The moment you see the above message, you'll also see the actual three text messages in the Subscriber window as shown below:

```
Hi John, How are you doing  
Hi Smith, How are you doing  
Hi James, How are you doing
```

Now, change the main method to add one more subscriber as shown below

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

Save the code and run the Subscriber and Publisher programs again. This time in the Subscriber window, you'll see the messages twice. This is a proof of one-many messaging in which any number of subscribers can subscribe to the same topic and every subscriber will receive the published messages. This is an example of non-durable subscriptions. The subscriber must be executed before the publisher. Try running the publisher code first and then the subscriber code, and you'll never see the published messages. This is because the subscriber is inactive when the messages are published.

Now, let's modify the subscriber code to use durable subscription. First you need to change is set the `clientID` on the connection object as shown below:

```
TopicConnection conn = tcf.createTopicConnection();  
conn.setClientID("Bond");
```

We then have to create a durable subscriber instead of regular subscriber as shown below:

```
TopicSubscriber subscriber = session.createDurableSubscriber(topic, "Bond");
```

That's it. Save the file and run the Publisher first and then the Subscriber. The subscriber will receive the messages. This is because the messaging system retains the messages internally so that when the subscriber comes active, it will deliver the messages. This completes the Pub/Sub messaging model. Let's see how Point-to-Point model works.

Point-to-Point Messaging using JMS

The JMS API for this model looks much the same as Pub/Sub model except that the class names will begin with the word *Queue* instead of *Topic*. The administered objects in this model are *QueueConnectionFactory* and *Queue*.

Obtaining the Queue factory

The factory class for this model is obtained from JNDI lookup as shown below:

```
QueueConnectionFactory queueFactory =  
    (QueueConnectionFactory) context.lookup("ConnectionFactory");
```

Obtaining the connection and starting it (Steps 2 & 3)

```
QueueConnection queueConn = queueFactory.createQueueConnection();  
queueFactory.start();
```

Creating a QueueSession (Step 4)

The JMS method signature to create a queue session is:

```
public QueueSession createQueueSession(boolean transacted, int acknowledge)
```

Method Arguments:

transacted – If true, the session will be transacted.

acknowledge – An integer that specifies the acknowledgement mode by the recipient of the message. The usual values are

AUTO_ACKNOWLEDGE – The message will be automatically acknowledged by the recipient session.

CLIENT_ACKNOWLEDGE – The recipient will explicitly acknowledge the message by calling the acknowledge() method

Using the above method, we create the session as shown below:

```
QueueSession queueSession = queueConn.createQueueSession(false, Session.  
    AUTO_ACKNOWLEDGE);
```

Obtain the Queue from JNDI (Step 5)

```
Queue queue = (Queue) context.lookup("dynamicQueues/Sports");
```

Steps 1 through 5 are common for both producer and consumer code. You can copy paste them.

Following two steps is used in Producer code.

Creating a Sender

Create a sender for the queue using the session as shown below (step 6)

```
QueueSender sender = queueSession.createSender(queue);
```

Sending Messages

Create and send messages using the above queue sender as shown below (Step 7)

```
TextMessage message = topicSession.createTextMessage();
message.setText("A simple message");
```

To send the message to the consumer, we invoke the `send()` method as shown below:

```
sender.send(message);
```

The overall consolidated code for the producer will be as shown below:

```
public void sendMessage() throws JMSEException {
    QueueConnection queueConn = QueueFactory.createQueueConnection();
    queueConn.start();

    QueueSession queueSession = QueueConn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

    Queue queue = (Queue) context.lookup("dynamicQueues/Sports");

    QueueSender sender = QueueSession.createSender(queue);

    TextMessage message = QueueSession.createTextMessage();
    message.setText("A simple message");

    sender.send(message);
}
```

With point-to-point messaging, there will be only one consumer for the messages unlike in pub/sub model where we have multiple subscribers for messages. When the producer send message to the queue, the message will be persisted in the queue until someone consumes the message. The messaging system ensures that the message is never lost.

This is referred to as quality of service. On the consumer side, we again have to register a listener object to read the messages asynchronously.

Following is how the JMS code looks like on the consumer side.

```
public void recieveMessage() throws JMSException {  
    QueueConnection queueConn = QueueFactory.createQueueConnection();  
    queueConn.start();  
  
    QueueSession queueSession = QueueConn.createQueueSession(false, Session.  
        AUTO_ACKNOWLEDGE);  
  
    Queue queue = (Queue) context.lookup("dynamicQueues/Sports");  
  
    QueueReciever receiver = queueSession.createReciever(queue);  
  
    receiver.setMessageLister(new MyListener());  
}
```

Let's now write an example using point-to-point model. In this example we will have a producer put a message in the queue named "movies" and then have the consumer consume it asynchronously. See the code in listing 17.2.

Listing 17.2a (Producer.java) Producer code

```
package jms;  
  
import java.util.Properties;  
import javax.jms.*;  
import javax.naming.Context;  
import javax.naming.InitialContext;  
  
public class Producer {  
  
    public Producer() {  
  
        try {  
  
            Properties props = new Properties();  
            props.setProperty(Context.INITIAL_CONTEXT_FACTORY,  
                "org.apache.activemq.jndi.ActiveMQInitialContextFactory");  
            props.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");  
  
            Context context = new InitialContext(props);  
  
            QueueConnectionFactory tcf = (QueueConnectionFactory) context  
                .lookup("ConnectionFactory");  
  
            QueueConnection conn = tcf.createQueueConnection();  
  
            conn.start();
```

```
QueueSession session = conn.createQueueSession(false,
    Session.CLIENT_ACKNOWLEDGE);

Queue queue = (Queue) context.lookup("dynamicQueues/Movies");

QueueSender sender = session.createSender(queue);

// Send messages to 3 users.

TextMessage msg1 = session.createTextMessage();
msg1.setText("Hi John, How are you doing");

sender.send(msg1);

System.out.println("Message Sent");

conn.close();

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

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

Listing 17.2b (Consumer.java) Consumer code

```
package jms;

import java.util.Properties;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;

public class Consumer {

    public Consumer() {

        try {

            Properties props = new Properties();
            props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
            props.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");

            Context context = new InitialContext(props);

            QueueConnectionFactory tcf = (QueueConnectionFactory) context
                .lookup("ConnectionFactory");

            QueueConnection conn = tcf.createQueueConnection();

            conn.start();
        }
    }
}
```

```
QueueSession session = conn.createQueueSession(false,
    Session.CLIENT_ACKNOWLEDGE);

Queue queue = (Queue) context.lookup("dynamicQueues/Movies");

QueueReceiver receiver = session.createReceiver(queue);

receiver.setMessageListener(new MyListener());

conn.close();

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

}

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

class MyListener implements MessageListener {

    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Reading message: " + msg.getText());
            } else {
                System.out.println("Message of wrong type: "
                    + message.getClass().getName());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

If you look at the above code, the producer sends a message to a queue named “Movies” and the consumer reads the message from the same queue. The only difference in the two programs is that one uses `QueueSender` object to send the message and the other use the `QueueReceiver` object to receive the message. Since this is point-to-point messaging, the messages are always persisted in the `Queue` when the producer sends the messages. The consumer can run at any time and receive the messages. This is a one-to-one messaging model. You cannot have multiple consumers consume the same message from the same queue. Once the message is consumed, the messaging system will permanently delete the message. Compile and execute the Producer code using the following commands.

```
C:/>JavaTraining>chapter17>javac jms\*.java  
C:/>JavaTraining>chapter17>java jms.Producer
```

You'll see the message "Message Sent" in the console. Now run the consumer code using the following command.

```
C:/>JavaTraining>chapter17>java jms.Consumer
```

The consumer code will display the following result in the console.

```
Hello John. How are you doing?
```

In this messaging model, a single queue can only have one consumer. If you need to send messages to several consumers, you need to create a separate queue for each and every consumer.

This completes all the important concepts of messaging. I am sure you now understood how asynchronous messaging works. Most of the real world applications use one of two messaging models. Just try to understand how the Pub/Sub and Point-to-Point messaging works and you'll be fine.

Summary

- ✓ Enterprise applications typically use Message Oriented Middleware (MoM) for both synchronous and asynchronous messaging with the backend systems.
- ✓ JMS technology provides universal way for interacting with messaging systems.
- ✓ Synchronous messaging is a 2-way communication in which every request will be responded immediately.
- ✓ Asynchronous messaging is 1-way communication in which sender sends the message and doesn't wait for any response from the recipient.
- ✓ MoM system acts a middleware between senders and recipients.
- ✓ Publish/Subscribe and Point-to-Point are the two messaging models that every MoM system supports.
- ✓ In pub/sub model, publisher will publish messages to several topics. Subscribers subscribe with the topics to receive messages. This is a one-to-many messaging model.
- ✓ Subscriptions can be durable or non-durable in nature. With durable subscriptions, the recipient can receive messages even while being inactive. With non-durable subscriptions the subscriber application must be up and running to receive the messages.

- ✓ In point-to-point model, sender send a message to queue. This message will be consumed by the consumer. The MoM system retains the message in the queue until it is consumed. This is a one-to-one messaging model.

Time to play 50-50

1. Which of the following is an example for MoM system?
 - a) MySQL
 - b) MQSeries
2. Which J2EE technology is used for accesing MoM systems?
 - a) JNDI
 - b) JMS
3. Which of the following is a 1-way communication?
 - a) Asynchronous
 - b) Synchronous
4. Which of the following is an administered object?
 - a) ConnectionFactory
 - b) TopicConnectionFactory
5. Which of the following messaging model uses topics for messages?
 - a) Point-to-Point model
 - b) Pub/Sub model
6. Which messaging model uses queues for sending and receiving messages?
 - a) Pub/Sub
 - b) Point-to-point

Interview Questions

Question: What are the different messaging models supported by JMS?

Answer: Publish/Subscribe and Point-to-Point models

Question: Explain how Pub/Sub model works?

Answer: This is a one-to-many messaging model in which a publisher publishes messages to topics and subscribers will register with one or more topics to receive messages. A single publisher can publish messages to more than one topic and a single subscriber can subscribe with any number of topics.

Question: How does point-to-point messaging work?

Answer: Point-to-point messaging is a one-to-one messaging in which one party sends a message to the queue and the other party reads the message from the queue. In this model, the message broker will persist the message until it is consumed by some party.

Chapter 18

Spring Framework

This chapter introduces you to yet another open source framework for building enterprise applications, the Spring framework. This chapter explains the important spring modules.

Chapter Goals

- ✓ Understand Spring architecture
- ✓ Inversion of Control
- ✓ Aspect Oriented Programming
- ✓ Understanding spring core module
- ✓ Understanding DAO module.

Environment Setup

1. Download the Spring framework at the following URL. After clicking the download button, make sure you download the distribution with dependencies.

<http://www.springframework.org/download>

2. Copy the following Jar files in the Spring installation directory to the "j2eeLib" directory.

`spring-1.2.2.jar, commons-beanutils.jar, commons-dbcp.jar, commons-logging.jar, commons-collections.jar`

3. Set the CLASSPATH in the env.bat file with the above Jar files just like we did in the previous chapters.

Introduction

This chapter introduces you to one of the most popular and widely used open source framework, the **Spring** framework. Spring is a light weight container framework for building Java applications inside and outside the container. One of the key challenges today's J2EE developers face is to build applications faster and that are easy to maintain with minimum effort. Although the standard J2EE technologies offer solutions for rapid application development, there are some technologies in it that consume whole lot of time in terms of development and testing. One such technology is the EJB technology. Some enterprise applications use EJB technology to build business and persistence components. The one and only one reason for choosing EJB is the support that EJB *container* provides to the EJB components such as transactional, persistent services etc. So, enterprise applications to leverage these features from EJB container use EJB components to build business and persistent components.

Though EJB technology offers several benefits, the usage of EJB container to host the EJB's causes additional overhead in terms of consuming resources and most importantly effecting the length of testing cycle. The EJB container is like a server that needs to be restarted for every change in the business component. The testing cycle will therefore be *code, start server, test, code, start server, test* etc which should ideally be *code, test, code, test* etc. Most of the real world applications are not complex enough to really use the full scale services of the EJB container. Instead they only require using one or two of the important container services such as transactional, persistence services. The question that comes to our mind at this point is, why the heck do we need to use this heavy weight EJB container just to leverage few services and more so at the expense of resources, testing time and probably even performance. This is where we start wondering if there is any alternative solution/technology that completely eliminates the EJB container, yet provides most of the frequently needed services to the business components. This is where spring framework steps in.

Spring is a lightweight framework that provides most of the important services like transactions, persistence that can be used out of the box. Isn't this what we are looking for? EJB containers usually come along with application servers which are mostly commercial and are expensive. Spring framework not only eliminates the necessity of EJB container but also comes at absolutely free of cost. So, who can stop anyone from choosing spring framework to build enterprise applications? This is why we are interested in knowing something about spring framework and the features it has to offer to the developer community.

Let me tell you one thing here. Though we demonstrated our hatred towards EJB, we cannot completely rule out the usage of EJB container just because of the above few reasons. EJB technology is built to meet the highly complex business requirements such as distributed transactions, remoting, security, interoperability etc which are beyond the reach of the most of the real world enterprise applications. If you are starting a business, you want to build the infrastructure at minimum cost, right? This is when you can use spring framework. As your business starts growing and the business requirements are getting complex and complex that are tough to handle by spring alone, then you can think of EJB container. Until then let's use spring framework.

In this chapter, we'll look at the basic architecture of spring framework and also utilize its cool development tools to build better applications that are easy to maintain and easy to test.

Spring Framework

Spring framework unlike other popular open source frameworks like Struts is not a specialized framework. It's more of a generalized framework for building end-to-end J2EE applications. What does this mean? If we take Struts for instance, it can only be used for building web application components that only reside in the presentation tier. With spring framework, we can build components that can be used across all the tiers in a typical n-tier application. In presentation tier, it can be used for building web components, in the business tier it can be used for building transactional and persistent components etc. You can imagine this framework as another Java alternative solution for J2EE.

Goals of Spring Framework

1. Keep the life of J2EE developer simple

Spring doesn't use any complex business components like EJB's. Instead, it uses the simple Java Beans which we also call as POJO's to build applications by making them almost as powerful as EJB's. Developing POJO's hardly takes few minutes and also simplifies the testing of business components.

2. Improve the loose coupling of application components

When developing business components it's very important that they must be as loosely coupled as possibly can. This allows testing the components individually

without any dependencies. Spring uses the notion of *Inversion of Control* to achieve loose coupling. We'll see IoC in detail in later sections.

3. Reduce the application maintenance burden

With the applications no longer using heavy weight components like EJB, code maintenance gets very easy from the point of packaging the code to deploying the code on the server. It also reduces the complexity of configuring the application components.

4. Improve the performance by consuming fewer resources

Being a lightweight framework, spring doesn't consume system resources heavily for providing services to application components. This has a direct effect in improving the performance of the application.

Knowing all the above good things about spring framework, let's begin our journey into the world of spring framework.

Architecture of Spring framework

Following figure shows the high level view of the spring framework.

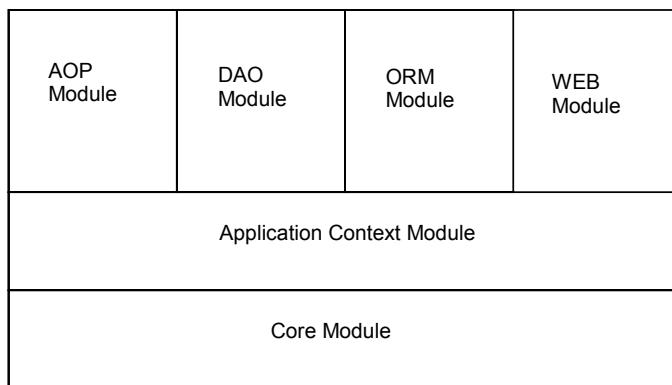


Fig 18.1 Architecture of Spring framework

As you can see from the above figure, the entire spring framework is divided into several modules as listed below:

1. Core Module
2. Application Context Module
3. AOP Module
4. DAO Module
5. ORM Module and
6. Web Module

Let's see the purpose of each individual module.

Core Module

The core module is the heart of spring framework. It includes all the core infrastructure components that support the framework and also factory classes for bean processing. One such factory class is the `BeanFactory` class which is used for working with POJO's or simply JavaBeans. This factory class allows wiring the beans on the fly and also used to populate the bean properties.

Application Context Module

This module is an extension to the core module in the sense it offers services such as access to JNDI components, and more importantly support for internationalization through the use of property files and resource bundles. Using this module, bean properties can be loaded from property files.

AOP Module

This module provides support for Aspect Oriented programming. AOP capabilities are not yet completely utilized in the world of J2EE, but it is very powerful feature that declaratively injects utility services to business components. Using AOP we can cleanly eliminate duplicate code (ex., security logic or logging logic) that is usually scattered across various business methods.

DAO Module

The DAO module is what made spring very popular. Though spring has several other modules, most of the real world applications uses spring framework just for leveraging the services of this module. The reason is simple. This module offers highly simplified solution for persisting data to databases. It does this through something called templates which we'll see later. This module abstracts all the redundant JDBC code like loading the drivers, managing the connections etc and allows users to only work with the important things like executing the SQL queries and processing the results. This way it

keeps the persistence code lot simpler and cleaner. Don't be surprised if you only get DAO module related questions in interviews.

Web Module

This module as the name suggests can be used for building web applications. This module like struts framework is built upon MVC design pattern. The good and bad news is that this module didn't gain as much attention as DAO module. Most real world applications even today still use Struts framework for building web applications. This also means that all the time we spent learning struts didn't go in vain.

ORM Module

This module is used for working with Object Relational Mapping tools. Spring doesn't come with a full implementation of ORM solution, but can integrate third party ORM solutions like Hibernate, JDO etc,. Once integrated, this module provides a consistent way of persisting data by hiding the underlying ORM solution.

Let me tell you one thing here. To cover all the above modules is beyond the scope of this book. This chapter only covers the most important and frequently used modules listed below:

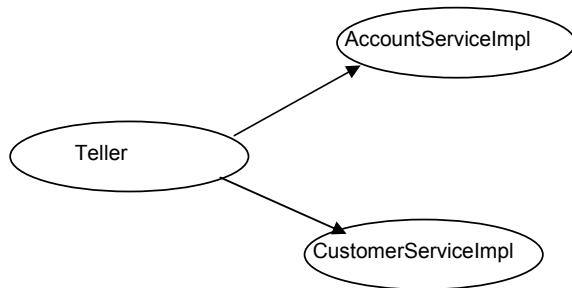
1. Core and Application Module
2. DAO Module

Besides the above modules, we will also learn the concepts of *Inversion of Control* and *Aspect Oriented Programming*. Now that we know spring architecture and the modules it has, let's understand the first concept, Inversion of Control.

Inversion of Control (IoC)

Inversion of Control is not a technology or anything even close. It's just a kind of design practice that promotes loose coupling of application components thereby improving the quality of testing the code. To better understand IoC, let's first look at an object design without IoC and see how it impacts the testing of application components.

Let's say we have a bank teller whose purpose is to provide two types of services to customers namely account services and customer services. Consider the following object design for the three objects `Teller`, `AccountServiceImpl` and `CustomerServiceImpl`.



The Teller object references AccountServiceImpl and CustomerServiceImpl objects to invoke the appropriate services. This is the simplest and straightforward design we can ever get. Let's write the code for the above three classes.

Listing 18.1a (AccountServiceImpl.java) Simple class

```
package spring;

public class AccountServiceImpl {
    public void transferFunds() throws Exception {
        boolean accountNotFound = true;
        if (accountNotFound)
            throw new Exception("Unable to transfer funds");
        System.out.println("Funds Successfully Transferred");
    }
}
```

Listing 18.1b (CustomerServiceImpl.java) Simple class

```
package spring;

public class CustomerServiceImpl {
    public void changeAddress() {
        System.out.println("Address has been updated");
    }
}
```

Listing 18.1c (Teller.java) Simple class

```
package spring;

public class Teller {

    private CustomerServiceImpl cImpl = new CustomerServiceImpl();
    private AccountServiceImpl aImpl = new AccountServiceImpl();

    public void transferFunds() throws Exception {
        aImpl.transferFunds();
    }

    public void updateAddress() {
        cImpl.changeAddress();
    }
}
```

As you can see from the above code, the `AccountServiceImpl` class defines a single method namely `transferFunds()`. This method throws an exception if the account is not found. Similarly `CustomerServiceImpl` class defines method to change customer address. The `Teller` class references the above two classes to invoke the appropriate methods. With all these three classes, let's write the test class for `Teller`.

```
package spring;

public class TellerTest {

    public static void main(String args[]) throws Exception {
        Teller t = new Teller();
        t.transferFunds();
        t.updateAddress();
    }
}
```

The above test class simply invokes both the operations on the teller class. If we execute the test class, it throws an exception with the message "Unable to Transfer Funds". Therefore, we can conclude that the teller class as a failure since the exception occurred while invoking the teller class. But if you carefully look at the code, the exception is actually thrown by the `transferFunds()` method in the `AccountServiceImpl` class. But from the testing point of view the culprit seemed to be the `Teller` class. So the question is, does the above code really test the `Teller` class? The answer is big No. A test class should only test the functionality of the corresponding class and not its dependent classes. This is therefore a bad test. The main reason for this is the tight coupling between `Teller` class and its dependent classes.

A good design must allow testing the classes independently irrespective of the dependent classes. This is possible only if we eliminate the dependencies. Can you think

of a solution for this? Very simple. We simply need to reverse the design by injecting the dependent classes to the Teller class rather than having the teller class reach for its dependencies.

This decoupling is achieved by using the so called interfaces. Let's see how we can use interfaces and eliminate the dependencies. Let's have the `CustomerServiceImpl` class implement the `CustomerService` interface as shown below:

```
public interface CustomerService{  
    public void changeAddress();  
}  
  
public class CustomerServiceImpl implements CustomerService{  
    public void changeAddress(){  
        System.out.println("Address has been updated");  
    }  
}
```

The code for `AccountService` will also follow the same lines. With these two updated classes, let's now look at the Teller code.

Listing 18.1d (`Teller.java`) Updated class

```
public class Teller {  
  
    private CustomerService cService;  
    private AccountService aService;  
  
    public void transferFunds() {  
        aService.transferFunds();  
    }  
    public void updateAddress() {  
        cService.changeAddress();  
    }  
    public void setAccountService(AccountService aService) {  
        this.aService = aService;  
    }  
    public void setCustomerService(CustomerService cService) {  
        this.cService = cService;  
    }  
}
```

As you can see from the above Teller class code, we introduced two setter methods to *inject* the implementation classes using the interface references. The class is now completely independent of CustomerServiceImpl and AccountServiceImpl classes. The Teller class can be viewed as a light weight class since it cannot do anything until the actual implementation objects are injected into it. Therefore the test class should first invoke the setters to send the dependent objects, and then invoke the actual business methods as shown below:

```
public class TellerTest{  
  
    public static void main(String args[]){  
  
        Teller t = new Teller();  
  
        t.setAccountService ( new AccountServiceImpl() );  
        t.setCustomerService ( new CustomerServiceImpl() );  
  
        t.transferFunds();  
        t.updateAddress();  
  
    }  
}
```

As you can see from the above code, the dependent objects are passed to the teller class prior to invoking the business methods. If the two impl classes are tested successfully prior to testing Teller class, any failure in the test code from can be attributed solely to the Teller class itself. All this is possible only by injecting the dependencies which is what is what we call as inversion of control. In simple words, if we have three classes namely A, B, C where B,C are dependents of A, then according to IoC, B and C must be injected in to A rather than A reaching for B and C.

Inversion of Control is a design technique that spring uses behind the scenes. With this basic understanding of IoC, let's start looking at the spring modules one by one.

Core Module

The spring's core module essentially comprises of factory classes for working with Java beans. Just like in object oriented programming where we have independent objects that are associated with each other to implement a solution, in spring we use the notion of beans to implement functionality.

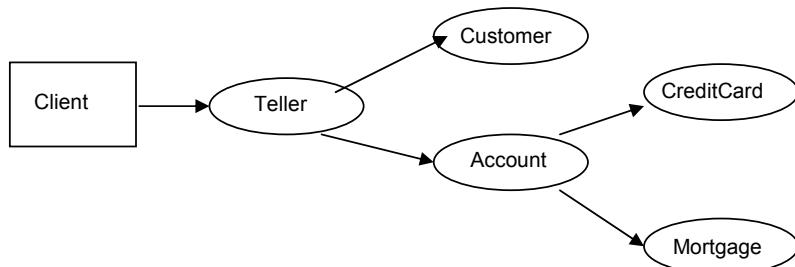
A typical spring application is comprised of several independent beans which we also call them as POJO's (Plain Old Java Objects). In order to get the functionality of out of these beans, it's very important that they be associated with each other in some way or the other. We call this as *wiring* the beans. For instance, certain business requirement

may require the services of Account bean and Customer bean. Before we process them we need to populate the beans properties and associate them with each other. The Spring's core factory classes provide services for wiring the beans. Let's see how this is done.

Spring uses two container classes namely BeanFactory and ApplicationContext for managing the beans. The BeansFactory class is a simple container that provides the basic services like injecting dependencies through Inversion of Control. The ApplicationContext class however is much more sophisticated and is an extension to BeanFactory class. Besides the basic support, it also provides internationalization features though the use of property files and resource bundles. It also allows the bean properties to load from property files. This is the most widely used factory class in real world applications.

Why wiring beans?

In typical applications, most of the functionality will be collectively implemented by several beans. For instance look at the following bean structure:



The above figure shows the bank teller functionality. The teller application needs to use Customer and Account bean for certain business functions. The account bean in turn uses some other beans to fulfill the teller's requests. Although from client's point of view the only bean that needs to be loaded is the Teller bean, it's very important that we also load the behind the scenes beans for the teller bean to work as desired. This means the account bean must first wire the CreditCard and Mortgage beans, the teller should then wire the customer and account beans etc. Wiring the beans usually starts from the bottom node and wiring up to the root node.

Now that we know the importance of wiring beans, let's see the two important core components namely `BeanFactory` and `ApplicationContext` classes that aid in wiring the beans.

BeanFactory

The `BeanFactory` class as the name suggests is a factory of bean classes. Its main purpose is to instantiate and set the properties of different types of bean classes. The `BeanFactory` is an interface that has multiple implementation classes. One such implementation class is the `XMLBeanFactory` which is one of the widely used. This factory uses the XML file that contains the definition of all the beans for creating and loading the bean properties. The `XMLBeanFactory` class is instantiated as shown below:

```
BeanFactory factory = new XMLBeanFactory( new FileInputStream("MyBeans.xml"));
```

Once the reference to factory object is obtained, we can start creating beans using the `getBean()` method. Every bean in the XML is identified with a unique identifier which is used to retrieve the bean as shown below:

```
MyBean bean = (MyBean) factory.getBean("customer");
```

In the above, `customer` is the id of the bean defined in the XML. The definition for the bean in the XML file will look as shown below:

```
<bean name="customer" class="beans.MyBean"/>
```

As we all know the fact that a typical bean stores the data in the form of properties whose values are set and retrieved using the setter and getter methods, the `getBean()` method before returning the bean instance, does two important things:

1. Instantiates the bean
2. Loads the bean properties with the appropriate values.

Before we understand the complete intricacies of beans, let's do a simple example to get a feel. Ok. This example defines a customer bean and loads the properties `firstName`, `lastName` etc with some arbitrary values. Take a look at the code in listing 18.2.

Listing 18.2a (Customer.java) Simple Java Bean

```
package spring.example1;
public class Customer {
```

```
String lastName;
String ssn;
public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
public String getSSN() {
    return ssn;
}
public void setSSN(String ssn) {
    this.ssn = ssn;
}
```

Listing 18.2b (`CustomerTest.java`) Simple test class

```
package spring.example1;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class CustomerTest {
    public static void main(String args[]) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "mybeans.xml"));

        // Get the customer bean

        Customer cus = (Customer) factory.getBean("customer");
        String details = cus.getLastName() + ":" + cus.getSSN();
        System.out.println(details);
    }
}
```

Listing 18.2c (`mybeans.xml`) Beans definition file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="customer" class="spring.example1.Customer">
        <property name="lastName">
            <value>Bond</value>
        </property>
        <property name="ssn">
            <value>999999</value>
        </property>
    </bean>
</beans>
```

The Customer bean doesn't deserve any explanation. Look at the test class. This class loaded the xml bean definitions file, and then retrieved the bean by its id. The xml file uses the bean element to define the bean. The bean element does the following:

1. Specifies the id and fully qualified class name of the bean
2. Set the bean properties using the nested property elements.

When the bean is returned by the factory class, spring will load the bean properties with the values in the bean definition.

Steps to run the program

1. Save the files as

```
C:\JavaTraining\chapter18\spring\example1\Customer.java  
C:\JavaTraining\chapter18\spring\example1\CustomerTest.java  
C:\JavaTraining\chapter18\mybeans.xml
```

2. Compile the two classes as

```
C:/>JavaTraining>chapter18>javac spring\example1\*.java
```

3. Execute the test class as

```
C:/>JavaTraining >chapter18>javac spring.example1.CustomerTest
```

The result of the above program will be

```
Bond:99999
```

Whenever we write a bean class, we also need to define the bean in bean definitions xml file. Now let's look at the second factory class for wiring beans.

ApplicationContext

As said before, this container class is also used for working with the beans. The most significant improvement with this class is that it also loads the properties from the property files. There are multiple implementations available to this interface; however following are two most important ones.

FileSystemXmlApplicationContext: This class is used to load the bean definitions XML file from the file system.

`ClassPathXmlApplicationContext`: This class is used to load the xml file from the class path. You are free to use which ever you want.

The application context using the `FileSystemXmlApplicationContext` is typically created as shown below:

```
ApplicationContext ctx = new  
                    FileSystemXmlApplicationContext("c:/.../mybeans.xml");
```

Similarly, the application context using `ClassPathXmlApplicationContext` is created as shown below:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("mybeans.xml");
```

Once the context is obtained, its usage is again same as `BeanFactory`. Using this context, beans will be retrieved as,

```
MyBean bean = (MyBean) ctx.getBean("customer");
```

Now, let's see the how the application context uses the property files to load the bean properties. Consider the following property file

```
db.properties  
driverName = com.mysql.jdbc.Driver  
username=test  
password=test1
```

In order that the application context load the above property file, we need to define a bean of type `PropertyPlaceholderConfigurer` which is a built-in spring class and specify the property file location as shown below:

```
<bean id="propertyConfigurer"  
      class="org.springframework.beans.factory.  
          config.PropertyPlaceholderConfigurer">  
  
    <property name="location">  
      <value>db.properties</value>  
    </property>  
  
</bean>
```

Once the above bean is defined with all the information, we can read the properties using the `${ }` syntax. For instance, the `driverName` property can be read as `${driverName}` and so on. The following bean definition defines a `DataSource` bean whose properties are loaded from the above property file.

```
<bean id="dataSource" class="beans.DataSource">
    <property name="driver">
        <value>${driverName}</value>
    </property>
    <property name="username">
        <value>${username}</value>
    </property>
</bean>
```

Let's now see the some of the basic scenarios for defining the bean element in the XML file. This element defines several child elements that can be used to make the bean very powerful. Following are the important cases:

Case 1: Setting the bean properties.

To set the properties of the bean with we use the nested `property` element. Following definition defines a bean with a property named `driver` with value "TestDriver". It also defines another property named `userid` whose value is read from the property file.

```
<bean id="customer" class="beans.Customer">
    <property name="driver">
        <value>TestDriver</value>
    </property>
    <property name="userid">
        <value>${username}</value>
    </property>
</bean>
```

Case 2: Initialization and destruction methods.

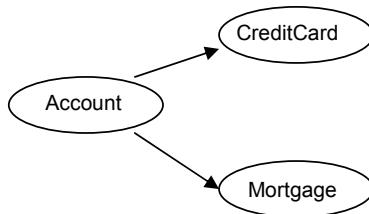
Sometimes before invoking the business logic methods on the bean, we require to invoke the initialization method. Similarly, before the bean is destroyed, we need to invoke certain method for clean up. Spring allows us to declare these methods in the XML file as shown below:

```
<bean id="ds" class="beans.DataSource" init-method="init"
      destroy-method="close" />
```

The `init-method` attribute specifies the initialization method, and the `destroy-method` attribute specifies the method before destruction.

Case 3: Referencing other beans.

One of the most important aspects of wiring the beans is its ability to reference other beans. Let's say we want to wire the beans as shown below:



Referencing the beans should always be done backwards from right to left or bottom to top. We will first define the `CreditCard` and `Mortgage` beans and then wire them to `Account` bean. Following bean declaration demonstrates this.

```

<bean id="cc" class="beans.CreditCard"/>
<bean id="mort" class="beans.Mortgage"/>
<bean id="account" class="beans.Account" >
    <property name="creditCard">
        <value>
            <ref bean="cc"/>
        </value>
    </property>
    <property name="mortgage">
        <value>
            <ref bean="mort"/>
        </value>
    </property>
</bean>
  
```

As you can see from the above definition, we used the `ref` element to reference other beans. The `creditCard` and `mortgage` properties of `account` bean now references the `CreditCard` and `Mortgage` beans.

Case 4: Collection properties.

More often than not, some of the bean properties will be represented as collection of objects. The factory classes also support the setting of collection properties. For instance, the `Account` bean can have a list of account types namely `Checkings`, `Savings`, `Credit Card` etc. Following definition shows how to wire the collection type properties.

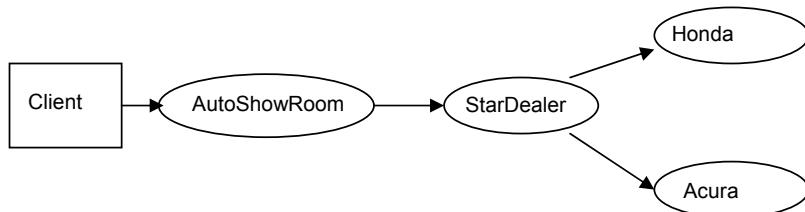
```

<bean id="account" class="beans.Account">
    <property name="accountTypes">
        <list>
            <value>Checkings</value>
            <value>Savings</value>
            <value>CreditCard</value>
        </list>
    </property>
</bean>
  
```

The above definition will populate the `accountTypes` collection with the list of specified values in the same order. Likewise, to store other beans in a collection object, we use the following definition:

```
<bean id="account" class="beans.Account">
<property name="addresses">
    <list>
        <ref bean = "home"/>
        <ref bean = "office" />
    </list>
</property>
</bean>
```

With this knowledge of bean definitions, let's write an example to make things more clear. The following example is an auto application for test driving vehicles. Following figure shows the bean structure:



Listing 18.3a (`Car.java`) Simple interface

```
package spring.example2;

public interface Car {
    public void testDrive();
}
```

Listing 18.3b (`Honda.java`) Simple class

```
package spring.example2;

public class Honda implements Car {

    String power;
    String transmission;

    public String getPower() {
        return power;
    }

    public void setPower(String power) {
        this.power = power;
    }
}
```

```
public String getTransmission() {
    return transmission;
}

public void setTransmission(String transmission) {
    this.transmission = transmission;
}

public void testDrive() {
    System.out.println("Test Driving Honda with " + transmission
        + " transmission whose HP is " + power);
}

}
```

Listing 18.3c (Acura.java) Simple class

```
package spring.example2;

public class Acura implements Car {

    String power;
    String transmission;

    public void testDrive() {
        System.out.println("Test Driving Acura with " + transmission
            + " transmission whose HP is " + power);
    }
    public String getPower() {
        return power;
    }
    public void setPower(String power) {
        this.power = power;
    }
    public String getTransmission() {
        return transmission;
    }
    public void setTransmission(String transmission) {
        this.transmission = transmission;
    }
}
```

Listing 18.3d (Dealer.java) Simple interface

```
package spring.example2;

public interface Dealer {

    public Honda getHonda();
    public Acura getAcura();
}
```

Listing 18.3e (`StarDealer.java`) Simple class

```
package spring.example2;

public class StarDealer implements Dealer {

    private Honda honda;
    private Acura acura;

    public Honda getHonda() {
        return honda;
    }

    public Acura getAcura() {
        return acura;
    }

    public void setHonda(Honda h) {
        honda = h;
    }

    public void setAcura(Acura a) {
        acura = a;
    }
}
```

Listing 18.3f (`ColumbusAuto.java`) Simple class

```
package spring.example2;

public class ColumbusAuto {

    private Dealer dealer;

    public void setDealer(Dealer d) {
        dealer = d;
    }
    public Dealer getDealer() {
        return dealer;
    }
    public void testDriveHonda() {
        dealer.getHonda().testDrive();
    }
    public void testDriveAcura() {
        dealer.getAcura().testDrive();
    }
}
```

Listing 18.3g (`TestClient.java`) Simple test class

```
package spring.example2;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
```

```
public class TestClient {  
    public static void main(String args[]) {  
  
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(  
            "mybeans.xml"));  
  
        ColumbusAuto auto = (ColumbusAuto) factory.getBean("columbusAuto");  
        auto.testDriveHonda();  
        auto.testDriveAcura();  
    }  
}
```

Listing 18.3h (mybeans.xml) Beans definitions

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
      "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <bean id="honda" class="spring.example2.Honda"  
          init-method="start" destroy-method="stop">  
        <property name="power">  
          <value>200</value>  
        </property>  
        <property name="transmission">  
          <value>Manual</value>  
        </property>  
    </bean>  
    <bean id="acura" class="spring.example2.Acura"  
          init-method="start" destroy-method="stop">  
        <property name="power">  
          <value>250</value>  
        </property>  
        <property name="transmission">  
          <value>Automatic</value>  
        </property>  
    </bean>  
    <bean id="stardealer" class="spring.example2.StarDealer">  
        <property name="honda">  
          <ref bean="honda" />  
        </property>  
        <property name="acura">  
          <ref bean="acura" />  
        </property>  
    </bean>  
    <bean id="columbusAuto" class="spring.example2.ColumbusAuto">  
        <property name="dealer">  
          <ref bean="stardealer" />  
        </property>  
    </bean>  
</beans>
```

Steps to run the program

1. Save all the Java files in the following directory

```
c:\JavaTraining\chapter18\spring\example2  
c:\JavaTraining\chapter18\mybeans.xml
```

2. Compile the classes as

```
C:/>....>JavaTraining>chapter18>javac spring\example2\*.java
```

3. Execute the test class as

```
C:/>....>chapter18>javac spring.example2.TestClient
```

Go through bean definitions, and you'll understand the entire program. The test class will produce the following result

```
Test Driving Honda with Manual transmission whose HP is 200  
Test Driving Acura with Automatic transmission whose HP is 250
```

This completes all the basics you need to know about wiring. Let's now look at another important concept of spring framework, *AOP programming*.

Aspect Oriented Programming (AOP)

Often in applications there will be certain operations or services that are required to be used by all the business components across all the tiers. Such services include logging, authentication etc. In such cases we normally end up duplicating the code. One simple solution for this problem is to encapsulate all the redundant code in utility methods and invoke the method wherever needed. This approach solves the problem to some extent, but not completely in the sense that you'd still end up with duplicate method calls, right? This again becomes a maintenance burden. For instance, when we are forced to change the signature of the method, we need to go ahead and update all the references. This is very much like a tight-coupling issue. At the same time we cannot leverage the services without invoking them. This is where AOP comes into the picture and says, "Hey, you can use AOP features and completely eliminate even the duplicate method calls, yet leveraging the services". Let's see how it achieves this.

AOP as the name suggests revolves around the central entity called *Aspect*. An aspect is nothing but a reusable service that can be plugged in declaratively. Aspects one written can be declaratively plugged into the code and this is exactly what spring does. In order to plug-in the service declaratively, spring uses the application context. To better understand, look at the following figure:

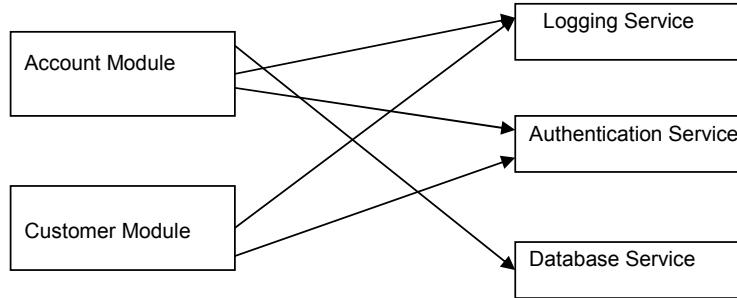


Fig 18.2 Business modules and services

As you can see from the above figure, logging, authentication and database services are independent services that can be used by any application module. For instance, the account module uses all three services while the customer module uses just two services. There is nothing wrong here because every module is free to choose whatever services it needs, right? With Spring AOP features, business modules can use the application context to specify declaratively what services it needs. Based on this information, at runtime, spring will make the services intercept the business method calls and execute the service code. Tomorrow if you think you don't want to use a particular service, without a single line of code change you can get rid of it. This is the true beauty of AOP with spring. So, without wasting any further time, let's see how this works and what is needed to be done from development point of view.

Before we learn something about AOP, let's understand some of its terminology.

Aspect

An aspect is the definition of the service we want to implement. This is like an interface to a class.

Advice

An advice is the actual implementation of aspect. This contains the service code that is executed at runtime.

PointCut

A pointcut defines where the advices should be applied in the code. For instance, should the advice be applied to all the business methods of a class, or only to some methods of the class? There is no use in writing an advice if we don't define a pointcut as to where the advice should be applied.

Now that we know what an advice is and what a pointcut is, let's learn about advices in detail.

Types of Advices

In spring, there are four types of advices namely,

1. Before Advice
2. After Advice
3. Around Advice
4. Throws Advice

Before Advice

I am sure you already guessed what this advice might do. Advices of this type are always invoked *before* the execution of the business method. Once the execution of the advice (service) is completed spring will execute the actual business method. One thing to be noted with this advice is that the target method is *always* executed.

The spring component that supports this advice is the `MethodBeforeAdvice` interface. This interface defines the following method that needs to be implemented:

```
void before(Method method, Object[] params, Object target)
```

Method Arguments

`method` – contains all the information about the target method

`params` – The parameters values of the target method

`target` – The target object itself

To better understand this type of advice, let's write a logging advice.

Listing 18.4a (`LoggingAdvice.java`) A simple logging advice

```
package spring.example3;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingAdvice implements MethodBeforeAdvice {

    public void before(Method method, Object args[], Object target)
        throws Exception {

        System.out.println("Entered into " + method.getName() + "in "
            + target.getClass().getName());
    }
}
```

As you can see from the above code, we implemented the `before()` method in the advice. This method simply logs the name of the method along with the class name that the method belongs to. You can even write JDBC logic in this method and log it to database. We will use this code later. For now, let's learn the next advice.

After Advice

This is the reverse of the before advice. An advice of this type will be executed *after* the completion of the target method. The spring component that supports this advice is `AfterReturningAdvice`.

This interface defines the following method that needs to be implemented.

```
void afterReturning( Object returnVal, Method method, Object[] params, Object
target) throws Throwable
```

This method takes one additional argument which is the return value of the method. We can write another logging advice that also logs the return value. Look at the following code:

Listing 18.4b (`LoggingAfterAdvice.java`) A simple after logging advice

```
package spring.example3;

import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;

public class LoggingAfterAdvice implements AfterReturningAdvice {

    public void afterReturning(Object retVal, Method method, Object args[],
        Object target) throws Exception {

```

```
        System.out.println("Exiting method " + method.getName()
                           + "with a return value " + retVal);
    }
}
```

Around Advice

This advice is can be viewed as a combination of both the before and after advices. The unique feature of this advice is that it allows the user to take the decision whether or not to execute the target method. If you take the before advice, you are always guaranteed the execution of target method unless an exception is thrown. However with this advice, there two things we can do:

1. Take the decision whether or not to execute the target method, and also
2. Change the return value of the target method.

This is the most popular advice used in real world applications. The spring component that supports this advice is `MethodInterceptor`. This interface defines the following method that needs to be implemented.

```
void invoke(MethodInvocation invocation) throws Throwable
```

The best example for this advice is the authentication. Let's say we pass a username and password as method parameters that should be verified before the method is executed. Following is how we implement this scenario using around advice.

Listing 18.4c (`AuthenticationAdvice.java`) A simple around advice

```
package spring.example3;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class AuthenticationAdvice implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {

        Object ret = null;

        String userName = (String) invocation.getArguments()[0];
        String password = (String) invocation.getArguments()[1];

        if ("bond".equals(userName) && "test".equals(password)) {
            ret = invocation.proceed();
        } else {
            throw new Exception("Invalid Credentials");
        }
    }
}
```

```
        return null;  
    }  
}
```

Let's now see how the above code works. The parameter to the `invoke()` method, i.e., the `invocation` parameter is the important one here. Using this parameter we can get all the information about the target method. In our case, we used the `getArguments()` method to read the first and second arguments which happen to be the `username` and `password`. Based on the validation, we are either proceeding to the actual method invocation or throwing an exception. Throwing the exception will force the spring container to bypass the execution of the method.

Throws Advice

One thing common with the previous three advices is that if any of the advice throws an exception, the target method execution is bypassed. With these advices we don't have the ability to perform some cleanup operations in the event of exception like closing connections to database and all that good stuff. This is where the `throws` advice offers support. It allows us to handle exceptions thrown by the target method. The interface we use to create this advice is the `ThrowsAdvice`. This is marker interface that will not have any methods to implement. If this the case, which method do we need to implement? Don't worry. We need to implement one of the following two methods:

```
void afterThrowing(Throwable exception)    or  
void afterThrowing(Method method, Object[] params, Object target, Throwable  
exception)
```

The second method is the widely used one as it also allows us to get the method information. This advice is used whenever you want to handle certain exceptions in a different way. Following is a simple implementation of this advice.

Listing 18.4d (`AccountNotFoundAdvice.java`) A simple exception advice

```
package spring.example3;  
  
import java.lang.reflect.Method;  
import org.springframework.aop.ThrowsAdvice;  
  
public class AccountNotFoundAdvice implements ThrowsAdvice {  
  
    public void afterThrowing(Method method, Object[] params, Object target,  
    Throwable exception) {  
  
        System.out.println("Exception Handled. No need to panic");  
    }  
}
```

```
}
```

Now that we know different types of advices we need to create based on the context, we need to see how we can inject these advices into the real code. There is absolutely no use in writing the above advices if we cannot inject them in the code. This is where pointcuts come to our rescue. Once the advices are created, it's the pointcut that specifies where the advices needed to be applied.

Static Pointcuts

Spring supports both static and dynamic pointcuts. This chapter only covers static point cuts. The most popular static point cut is the `NameMatchMethodPointCut`. As the name suggests, this pointcut simply looks for a matched target method name based on the naming pattern specified in the bean definition. This class has the following two important properties that we normally use to specify the method name patterns.

1. `methodName`
2. `methodNames`

The first property is used to specify either a *single method or a pattern of methods*. For instance, if we need to apply an advice to all the methods whose name starts with "test", we set the property as,

```
<property name="methodName">
    <value>test*</value>
</property>
```

If we only need to apply the advice to just one method, we need to specify the complete method name as shown below:

```
<property name="methodName">
    <value>getCustomerProfile</value>
</property>
```

The second property is used when the method names don't follow a particular pattern. In such cases we can list all the method names as shown below

```
<property name="methodNames">
    <list>
        <value>testProfile</value>
        <value>getAccount</value>
        <value>calculateBalance</value>
    </list>
</property>
```

As always, the best way to understand the anything is by writing an example. So let's write it. In this example, we will create a business component with three methods and apply all the above four advices we learned before. See the code in listing 18.4.

Listing 18.4e (`AccountService.java`) A simple interface

```
package spring.example3;

public interface AccountService {

    public void transferFunds(String userName, String password, int account1,
        int account2, double funds);

    public String createAccount();

    public void changeAccount(String from, String to) throws Exception;
}
```

Listing 18.4f (`AccountServiceImpl.java`) A simple implementation class

```
package spring.example3;

public class AccountServiceImpl implements AccountService {

    public void transferFunds(String userName, String password, int account1,
        int account2, double funds) {

        System.out.println("Transferred " + funds + " from " + account1 + " to"
            + account2);
    }

    public String createAccount() {

        System.out.println("Created Account");
        return "12345678";
    }

    public void changeAccount(String from, String to) throws Exception {

        if (!from.equals("Checking"))
            throw new Exception("Unable to change");

        System.out.println("Updated Accounts Successfully");
    }
}
```

Listing 18.4g (`AccountServiceTest.java`) A simple test class

```
package spring.example3;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class AccountServiceTest {
```

```
public static void main(String args[]) throws Exception {  
    BeanFactory factory = new XmlBeanFactory(new FileSystemResource(  
        "mybeans.xml"));  
  
    AccountService service = (AccountService)  
        factory.getBean("accountService");  
  
    service.transferFunds("bond", "test", 1234, 5678, 200.5);  
    service.createAccount();  
    service.changeAccount("Checking", "Savings");  
}  
}
```

Listing 18.4h (*mybeans.xml*) Beans definitions

```
<beans>  
  
    <bean id="logging" class="spring.example3.LoggingAdvice" />  
    <bean id="loggingafter"  
          class="spring.example3.LoggingAfterAdvice" />  
    <bean id="auth" class="spring.example3.AuthenticationAdvice" />  
    <bean id="error" class="spring.example3.AccountNotFoundAdvice" />  
    <bean id="accountImpl"  
          class="spring.example3.AccountServiceImpl" />  
  
    <bean id="beforeInterceptor"  
          class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  
  
        <property name="mappedName">  
            <value>*Account</value>  
        </property>  
        <property name="advice">  
            <ref bean="logging" />  
        </property>  
  
    </bean>  
  
    <bean id="afterInterceptor"  
          class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  
  
        <property name="mappedName">  
            <value>changeAccount</value>  
        </property>  
        <property name="advice">  
            <ref bean="loggingafter" />  
        </property>  
  
    </bean>  
  
    <bean id="authInterceptor"  
          class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  
  
        <property name="mappedName">  
            <value>transferFunds</value>  
        </property>  
        <property name="advice">
```

```

        <ref bean="auth" />
    </property>

</bean>

<bean id="exceptionInterceptor"
      class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">

    <property name="mappedName">
        <value>changeAccount</value>
    </property>
    <property name="advice">
        <ref bean="error" />
    </property>

</bean>

<bean id="accountService"
      class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="proxyInterfaces">
        <value>spring.example3.AccountService</value>
    </property>
    <property name="interceptorNames">
        <list>
            <value>beforeInterceptor</value>
            <value>afterInterceptor</value>
            <value>authInterceptor</value>
            <value>exceptionInterceptor</value>
        </list>
    </property>
    <property name="target">
        <ref bean="accountImpl" />
    </property>

</bean>
</beans>
```

In this example the code in all the classes is pretty straight forward. The XML definition is where all the spice is added. Let's see one by one.

1. This application has five beans (4 advices and 1 class). Therefore we defined the five beans as below:

```

<bean id="logging" class="springbeans.example3.LoggingAdvice"/>
<bean id="loggingafter"
      class="springbeans.example3.LoggingAfterAdvice"/>
<bean id="auth" class="springbeans.example3.AuthenticationAdvice"/>
<bean id="error" class="springbeans.example3.AccountNotFoundAdvice"/>
<bean id="accountImpl"
      class="springbeans.example3.AccountServiceImpl"/>
```

2. We then defined four pointcuts for four advices. For instance, look at the following pointcut definition.

```
<bean id="beforeInterceptor"
      class="org.springframework.aop.support.
          NameMatchMethodPointcutAdvisor">

    <property name="mappedName">
        <value>*Account</value>
    </property>
    <property name="advice">
        <ref bean="logging"/>
    </property>

</bean>
```

This pointcut specifies that the logging advice should be applied to methods ending with the word “Account”. This evaluates to `createAccount()` and `changeAccount()` methods in the implementation class.

3. The last one is the `accountService` bean that defines the target object and the list of all the pointcuts. This is the bean that will be retrieved by the test class to invoke the business functions.

Steps to run the program

1. Save all the Java files in the following directory

```
c:\JavaTraining\chapter18\spring\example3
c:\JavaTraining\chapter18\spring\mybeans.xml
```

2. Compile the classes as

```
C:/>....>chapter18>javac spring\example3\*.java
```

3. Execute the test class as

```
C:/>....>chapter18>javac spring.example3.AccountServiceTest
```

The output the above code will be

```
Transferred 200.5 from 1234 to5678
Entered into createAccount in springbeans.example3.AccountServiceImpl
Created Account
Entered into changeAccount in springbeans.example3.AccountServiceImpl
Updated Accounts Successfully
Exiting method changeAccount with a return value null
```

Try changing the username to any other value, and you'll see the following result due to `AuthenticationAdvice`.

```
Invalid Credentials
Entered into createAccountin springbeans.example3.AccountServiceImpl
Created Account
Entered into changeAccountin springbeans.example3.AccountServiceImpl
Updated Accounts Successfully
Exiting method changeAccountwith a return value null
```

Try changing the string "Checking" to "Checkings" and you'll see the following result due to `AccountNotFoundAdvice`

```
Invalid Credentials
Entered into createAccountin springbeans.example3.AccountServiceImpl
Created Account
Entered into changeAccountin springbeans.example3.AccountServiceImpl
Exception Handled. No need to panic
Exception in thread "main" java.lang.Exception: Unable to change
```

This completes all the important things you need to know in the core module. At this point you should be comfortable working with spring's aspects and advices along with the factory classes used for wiring the beans. The next module we will look at is the DAO module which is undoubtedly the reason for the success of spring framework.

DAO Module

We now arrived at a point where we begin to start learning the most widely used spring module. For the last one year or so, the DAO module has crept into almost all the J2EE applications that use persistence. You might wonder why we wasted all the time learning JDBC which is also used for building persistence applications. Once again we need to remember that spring is a framework which like any other framework doesn't reinvent the wheel but tries to build a better wheel. Just like with Struts framework in which Servlet and JSP are the underlying technologies, JDBC is the underlying technology for this module.

The DAO module fundamentally simplifies the persistence application development by abstracting all the low level and fixed parts of the program such as loading the drivers, managing the connections and statements etc, and allowing the application developer to concentrate on the important things such as executing the SQL query statements. If you look back at the JDBC applications, every program needs to execute certain statements before arriving at a point where it can execute the SQL queries. Most of the JDBC code will often be cluttered with loading the drivers, creating connections, initializing transactions, creating statements etc. These are the fixed parts of the application. The only variants are query processing statements. For these statements to work, we cannot

ignore the fixed parts of the code. Thankfully with DAO module we no longer have to worry about these anymore. It will take care of the fixed parts of persistence logic behind the scenes and allows the programmer to only concentrate on executing the query statements. Not only this, it will also manage the database transactions if needed by the application. By the end of this section you'll agree with me as to why many companies are moving towards using Spring DAO module for data persistence.

The word DAO stands for Data Access Object. This is the pet name given to those objects that encapsulate persistence logic. Spring's DAO module uses something called *dataSources* and *templates* for working with databases. Let's see what these are.

Data sources are special beans that are typically defined in application context XML file and are used by spring's JDBC templates to work with databases. Before we look at the templates, let's see how we configure a data source in the application context XML file. Following bean definition shows the data source configuration.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

    <property name="driverClassName">
        <value>${driver}</value>
    </property>
    <property name="url">
        <value>${url}</value>
    </property>
    <property name="username">
        <value>${userID}</value>
    </property>
    <property name="password">
        <value>${password}</value>
    </property>
</bean>
```

As you can see from the above bean definition, spring uses `BasicDataSource` to define a data source. This class requires certain properties to be initialized such as driver class, username, password, database schema and all that good stuff. In the above definition, we initialized the properties from the property file. Once the data source bean is defined, we are now ready to work with the Spring's JDBC templates for database operations.

Spring comes with a class called `JDBCTemplate` which is the core DAO class that encapsulates all the crappy code like loading the drivers, creating connections etc.. Not only this, it also defines several utility methods for executing the SQL queries in a flexible manner. Let's understand the important methods this class has before we start writing some examples using it.

According to spring's documentation, following is what it says about `JDBCTemplate` class.

It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving application code to provide SQL and extract results. This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the org.springframework.dao package. Code using this class need only implement callback interfaces, giving them a clearly defined contract. The PreparedStatementCreator callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The RowCallbackHandler interface extracts values from each row of a ResultSet.

Following are the important methods defined by the `JDBCTemplate` class.

```
List query(String sql, Object[] args, RowMapper mapper)
```

This method takes the query and query arguments and returns a list of objects (records). The `RowMapper` interface is a callback interface which will be called for every row returned by the query. This method as you might have guessed is normally used with `SELECT` queries.

```
int update(String sql, Object[] params)
int update(String sql, PreparedStatementSetter pss)
```

These methods are used to execute the non-`SELECT` sql queries like `INSERT`, `UPDATE`, `DELETE` etc. The second form of `update()` method is another convenient form which uses the `PreparedStatementSetter` class to populate the prepared statement.

The above three methods are the most commonly used methods. However there are tons of other methods in the `JDBCTemplate` class and you are free to explore them. Let's start writing examples using the data sources and templates with the above methods. Take a look at the code in listing 18.5.

Listing 18.5a (`Customer.java`) Simple Java bean

```
package spring.example4;

public class Customer {

    String name;
    Integer age;
    Integer ssn;
    String state;
    String zip;
    String country;
```

```
public Integer getAge() {
    return age;
}
public void setAge(Integer age) {
    this.age = age;
}
public String getCountry() {
    return country;
}
public void setCountry(String country) {
    this.country = country;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public Integer getSsn() {
    return ssn;
}
public void setSsn(Integer ssn) {
    this.ssn = ssn;
}
public String getState() {
    return state;
}
public void setState(String state) {
    this.state = state;
}
public String getZip() {
    return zip;
}
public void setZip(String zip) {
    this.zip = zip;
}
```

Listing 18.5b (CustomerDAO.java) Simple DAO

```
package spring.example4;

import org.springframework.jdbc.core.JdbcTemplate;

public class CustomerDAO {

    private JdbcTemplate jdbcTemplate;

    public void addCustomer(Customer customer) {

        String sql = "INSERT INTO CUSTOMERS_1 VALUES(?,?,?,?,?,?)";
        Object args[] = new Object[] { customer.getName(), customer.getAge(),
            customer.getSsn(), customer.getState(), customer.getZip(),
            customer.getCountry() };

        jdbcTemplate.update(sql, args);
    }
}
```

```
        }
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

Listing 18.5c (`InsertCustomerTest.java`) DAO test class.

```
package spring.example4;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class InsertCustomerTest {

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new FileSystemXmlApplicationContext(
            "MDITBeans.xml");

        CustomerDAO dao = (CustomerDAO) ctx.getBean("customerDAO");

        Customer c = new Customer();
        c.setName("Test");
        c.setSsn(new Integer("12345"));
        c.setAge(new Integer("20"));
        c.setState("OH");
        c.setZip("1234");
        c.setCountry("USA");

        dao.addCustomer(c);

        System.out.println("Inserted record");
    }
}
```

Listing 18.5d (`jdbct.properties`) DAO properties file.

```
driver=com.mysql.jdbc.Driver
url = jdbc:mysql://localhost:3306/MyDB
```

Listing 18.5e (`mybeans.xml`) Bean definitions file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
            PropertyPlaceholderConfigurer">
        <property name="location">
            <value>jdbc.properties</value>
        </property>

```

```

</bean>

<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName">
      <value>${driver}</value>
    </property>
    <property name="url">
      <value>${url}</value>
    </property>
  </bean>

<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource">
      <ref bean="dataSource" />
    </property>
  </bean>

<bean id="customerDAO" class="spring.example4.CustomerDAO">
  <property name="jdbcTemplate">
    <ref bean="jdbcTemplate" />
  </property>
</bean>
</beans>
```

If you observe the above code, we defined a Customer bean with properties whose values will be inserted into the database table. The best way to understand the above code is by looking at the beans definitions. The first bean loads the property file `jdbc.properties` that contains the database information. The next bean defines a datasource object by supplying the database information as shown below

```

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName">
    <value>${driver}</value>
  </property>
  <property name="url">
    <value>${url}</value>
  </property>
</bean>
```

The third bean defines the template object. As I said before, this object will simply reference the datasource bean. The important thing in this definition is the class attribute which should be the spring's built in class `BasicDataSource`. Following shows the template definition.

```

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

Once the template bean is defined, we are now ready to use it for database operations. If you look at the DAO class, it defines the following property:

```
private JdbcTemplate jdbcTemplate;
```

Therefore, the DAO bean must set this property before performing the SQL operations. This is exactly what is done by the following DAO bean definition:

```
<bean id="customerDAO" class="example4.CustomerDAO">
    <property name="jdbcTemplate">
        <ref bean="jdbcTemplate"/>
    </property>
</bean>
```

With the above definition, when the DAO bean is retrieved using application context, the template property will be set automatically and we can start invoking the persistence methods on it. The DAO class defines one method `insertCustomer()` that takes the customer bean as the parameter. This method invokes the `update()` method in the `JDBCTemplate` class and passes the SQL query along with the array of parameters as shown below:

```
String sql = "INSERT INTO CUSTOMERS VALUES(?, ?, ?, ?, ?, ?)";
Object args[] = new Object[]{customer.getName(), customer.getAge(),
                           customer.getSsn(), customer.getState(),
                           customer.getZip(), customer.getCountry()};
jdbcTemplate.update(sql, args);
```

See how clean the above code is. No creating connections, no loading drivers and all that crappy stuff. The above code will insert the data into the `CUSTOMERS` table. Without spring, the equivalent JDBC code will look as shown below:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection("blab la ");
String sql = "INSERT INTO CUSTOMERS VALUES(?, ?, ?, ?, ?, ?)";
PreparedStatement stmt = con.prepareStatement(sql);
stmt.setString(...)
stmt.setString(...)
int status = stmt.executeUpdate();
stmt.close();
con.close();
```

See the number of lines we reduced and more than that how we made the code look cleaner. No wonder why people are getting addicted to this module. Now look at the test class. It simply gets the DAO bean and passes the `Customer` bean with all the data which will be inserted into the table.

Steps to run the program

1. Save all the Java files in the following directory

c:\JavaTraining\chapter18\spring\example4

Save the bean definitions file as

c:\JavaTraining\chapter18\mybeans.xml

Save the property file as

c:\JavaTraining\chapter18\jdbc.properties

2. Compile the classes as

C:/>....>chapter18>javac spring\example4*.java

3. Execute the test class as

C:/>....>chapter18>javac spring.example4.CustomerDAOTest

Make sure you put the MySQL driver in the classpath before running the program. Also create the following table in your MyDB database schema:

```
CREATE TABLE CUSTOMERS (
    Name varchar(50),
    Age NUMERIC(2),
    ssn NUMERIC(10),
    state varchar(10),
    zip varchar(10),
    country varchar(20)
)
```

You'll see the record inserted into the table. Sometimes to be safer, people will use the following update method to also specify the column types.

```
String sql = "INSERT INTO CUSTOMERS VALUES(?,?,?,?,?,?)";
Object args[] = new Object[]{customer.getName(),customer.getAge(),
                           customer.getSsn(),customer.getState(),
                           customer.getZip(),customer.getCountry()};

int types[] = new int[]{Types.VARCHAR,Types.INTEGER,Types.INTEGER,
                      Types.VARCHAR,Types.VARCHAR,Types.VARCHAR};

jdbcTemplate.update(sql,args,types);
```

Use the above code in the DAO class and re-run the program. Now that we've seen how to execute the DML queries, let's see how to execute SELECT queries. For executing the SELECT queries, we use the following method:

```
List query(String sql, Object[] args, RowMapper mapper)
```

The important parameter in the above method is the `RowMapper` object. Let's first write the code and then understand the details. Take a look at the code in listing 18.6.

Listing 18.6a (`MyHandler.java`) Simple row mapper class.

```
package spring.example5;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;
import spring.example4.Customer;

class MyHandler implements RowMapper {

    public Object mapRow(ResultSet rs, int arg1) throws SQLException {

        Customer customer = new Customer();

        customer.setName(rs.getString(1));
        customer.setSsn(new Integer(rs.getInt(3)));
        customer.setAge(new Integer(rs.getInt(2)));
        customer.setState(rs.getString(4));
        customer.setZip(rs.getString(5));
        customer.setCountry(rs.getString(6));

        return customer;
    }
}
```

Listing 18.6b (`CustomerDAO.java`) Simple DAO class.

```
package spring.example5;

import java.util.List;
import org.springframework.jdbc.core.JdbcTemplate;
import spring.example4.Customer;

public class CustomerDAO {

    private JdbcTemplate jdbcTemplate;

    public void addCustomer(Customer customer) {

        String sql = "INSERT INTO CUSTOMERS_1 VALUES(?,?,?,?,?,?)";
        Object args[] = new Object[] { customer.getName(), customer.getAge(),
            customer.getSsn(), customer.getState(), customer.getZip(),
```

```

        customer.getCountry() };

    jdbcTemplate.update(sql, args);

}

public List getCustomers() {

    String sql = "SELECT * FROM CUSTOMERS";
    final Customer customer = new Customer();
    List customers = jdbcTemplate.query(sql, new MyHandler());

    return customers;

}

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

    this.jdbcTemplate = jdbcTemplate;
}
}

```

Listing 18.6c (`CustomerDAOTest.java`) Simple DAO test class.

```

package spring.example5;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import spring.example4.Customer;
import java.util.List;

public class CustomerDAOTest {

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

        ApplicationContext ctx = new FileSystemXmlApplicationContext(
            "myBeans.xml");

        CustomerDAO dao = (CustomerDAO) ctx.getBean("customerDAO");

        List customers = dao.getCustomers();
        for (int i = 0; i < customers.size(); i++) {

            Customer customer = (Customer) customers.get(i);

            System.out.println(customer.getName() + "," + customer.getAge() + ","
                + customer.getSsn() + "," + customer.getState() + ","
                + customer.getZip() + "," + customer.getCountry());
        }
    }
}

```

If you recall the `query()` method in the `JDBCTemplate` class, one of the parameter is `RowMapper`. This is an interface which means we need to pass an object of the class that

implements this interface. The `RowMapper` interface defines the following callback method that the spring container will invoke for every record returned by the query.

```
public Object mapRow(ResultSet rs, int arg1) throws SQLException
```

The first thing we need to do is write a class and implement the above method. This method should read the record, populate the necessary JavaBean which in our case is the customer bean and return it as shown below:

```
public Object mapRow(ResultSet rs, int arg1) throws SQLException {  
  
    Customer customer = new Customer();  
  
    customer.setName(rs.getString(1));  
    customer.setSsn(new Integer(rs.getInt(3)));  
    customer.setAge(new Integer(rs.getInt(2)));  
    customer.setState(rs.getString(4));  
    customer.setZip(rs.getString(5));  
    customer.setCountry(rs.getString(6));  
  
    return customer;  
}
```

Now look at the DAO class. We added a method named `getCustomers()` as shown below:

```
public List getCustomers() {  
  
    String sql = "SELECT * FROM CUSTOMERS";  
    List customers = jdbcTemplate.query(sql,null,new MyHandler());  
    return customers;  
}
```

Look how simple the method is. It passes the SQL query along with the object of the handler class that implemented the `RowMapper` interface. What will happen behind the scenes is that spring container will invoke the handler method for every record, gather all the customer objects and returns them as a `List`. We can then iterate over the list and display the results. This is exactly what we did in the test class.

Execute the test class and you'll see all the records displayed. This is all you need to know about DAO module. In real world applications, 99% of the time we write code similar to our examples. Try playing with the above two examples to get a good feel for this module.

This completes all the important things you need to know in spring framework. Though spring has several modules, they are less frequently used. If you are interested in learning other modules like Web, ORM modules, I'd recommend you to read a good

spring book. Trust me, if you understand just the DAO module, you are in pretty good shape.

Summary

- ✓ Spring is an open source framework for building enterprise applications.
- ✓ Spring framework includes several modules like core module, web module, DAO module etc.,
- ✓ Spring uses plain vanilla like objects for encapsulating business logic as opposed to heavy weight components like EJB's.
- ✓ Spring uses inversion of control to promote loose coupling of components.
- ✓ The AOP features of spring allow applications to declaratively specify the services that business modules need to leverage.
- ✓ Spring's core module includes factory classes for wiring the beans. The important factory classes are the BeanFactory and ApplicationContext.
- ✓ Spring's DAO module is the widely used module for building persistent applications.
- ✓ The DAO module uses the notion of data sources and templates for persisting data to databases.
- ✓ The DAO module hides all the fixed parts of the program like loading drivers, creating connections etc and allows programmers to only work in the variant parts of the program like executing the SQL queries.

Time to play 50-50

1. Which of the following class is used for loading the beans?
 - a) BeanContext
 - b) BeanFactory
2. Which of the following class is used to load the bean properties from property files?
 - a) BeanFactory
 - b) ApplicationContext
3. Which of the following element in the beans definition XML file defines a bean?
 - a) bean
 - b) pojo

4. Which of the following advice is used for intercepting the method call and allows taking decision whether or not to execute the method?
 - a) Before advice
 - b) Around advice
5. Which of the following allows to specify which advices should be applied to business methods?
 - a) pointcut
 - b) joinpoint
6. Which of the following spring class is used to define data source?
 - a) DataSource
 - b) BasicDataSource
7. Which of the following spring class is used for executing the SQL queries?
 - a) Template
 - b) JDBCTemplate

Interview Questions

Question: What are the important factory classes used for loading beans?

Answer: BeanFactory and ApplicationContext

Question: What is the difference between BeanFactory and ApplicationContext classes?

Answer: ApplicationContext class can be used to load the bean properties from property files and also support internationalization while BeanFactory cannot.

Question: List the different types of advices spring supports?

Answer: Before advice, After advice, Around advice and Throws advice.

Question: Which class in DAO module is used for executing SQL queries?

Answer: JDBCTemplate

Question: What is inversion of control?

Answer: Refer to page 553.

Chapter 19

Hibernate

This chapter introduces you to one of the most popular open source ORM solution, Hibernate. By the end of this chapter you'll understand how to use hibernate framework for mapping objects to relational database systems and how to retrieve objects from database using hibernate query features.

Chapter Goals

- ✓ Understand the importance of ORM
- ✓ Understanding hibernate components.
- ✓ Persisting JavaBeans
- ✓ Understanding Associations.
- ✓ Understand Polymorphic Associations
- ✓ Understand Hibernate Query Language (HQL)

Environment Setup

1. Download the Hibernate framework at the following URL.

<http://www.hibernate.org/6.html>

2. Copy the following Jar files in the Hibernate installation directory to the "j2eeLib" directory.

hibernate3.jar,log4j-1.2.11.jar,xerces-2.6.2.jar,dom4j-1.6.1.jar,**mysql-connector-java-5.0.3-bin.jar**, ehcache-1.1.jar,cglib-2.1.3.jar,jta.jar,antlr-2.7.6rc1.jar,asm-attrs.jar,asm.jar

3. Add all the above Jar files to the classpath in the batch file. Make sure to include MySQL Jar in the classpath.

Introduction

Data persistence is one of the most critical part of any enterprise application. Applications that don't persist data are practically less useful and cannot service the customers well enough to retain them. Over the years several persistence technologies have emerged to provide a better and flexible way of persisting the data to databases. Every new persistence technology has proven to be somewhat better and flexible than the older ones. It is important to understand that there has never been an end and will never be an end to the emergence of persistence technologies. Every now and then some new persistence solution will pop up and we will be forced to use it. This should give you some idea about how important data persistence is to the success of any application.

There are infinite numbers of free and commercial persistence solutions available in the market. Choosing a solution not only depends on how well it works but also depends on other factors like budget, adaptability and several others. Some applications may go for freeware solutions and some others for commercial ones. It all boils down how much persistence our application needs and how much complexity is involved during persistence. There is no point in using a highly sophisticated technology just to persist data to couple of tables, right?

Since we are in the world of Java and J2EE, we need to find the persistence solutions that can be easily integrated into the Java applications. In the last few chapters we have seen couple of persistence technologies like JDBC and Spring. Every technology has its own strengths and weaknesses. We simply cannot rule them out completely. In some applications where complex SQL processing is involved Spring JDBC is the way to go. All this is to say that it is the application demand that should determine which technology to use. The main challenge with persistence is the "mismatch" that exists between databases and Java application objects.

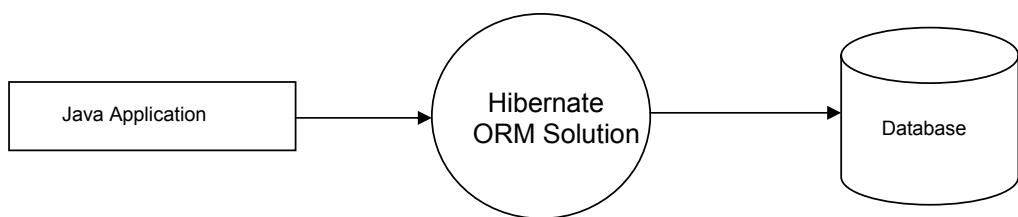
The problem with any relational database is that it stores the data in the form of tables which are two dimensional in nature. However, Java applications represent the data in the form of objects whose structure is hierarchical in nature (i.e., like a tree which is multi-dimensional). Do you see a mismatch here? I am sure you got the point. There is no way you can directly map the Java objects to tables in the database. This is the problem. However if we have an "Object Relational databases" that can store the objects directly, then the problem is solved, and we don't need any technologies, right? Though there are such object databases available in the market, they are not accepted by the industry in the same way relational databases are accepted, whatever may be the reason. Instead of drilling into the details of why they are not accepted, let's work with the problem we have in our hand. The problem is mapping Java objects to tables in

relational database. Though it is not impossible to map the data in Java objects to tables, it involves some level of complexity. The application developer should be responsible to understand the underlying table structure and then write the complex SQL queries to persist the data to the right tables. Historically this is what we were doing.

We as Java developers are more comfortable working with Java objects than working with SQL operations to the databases. As a business logic developer, we should spend more time developing the business logic than working with SQL queries to map the data to database tables. Our goal as a developer is to persist the data to the database at any cost. The question is how much effort we need to put for the same. This eventually boils down to how well the persistence layer is designed, right? If the design is good, using JDBC or any other technology doesn't make any difference. This has now become a designers concern and not the developers concern. Historically designers suggested the idea of using EJB entity beans using CMP as a persistence solution. Though entity beans served the J2EE community for a long time, most real world applications started to get away from them due to the complex O/R mappings. Moreover, an EJB container is a must to run the entity beans which makes the application heavier and also consumes lot of resources thereby affecting the application performance. So, companies started looking for a light weight persistence solution that not only makes life easy but also eliminates the EJB container.

The primary goal of any persistence solution is to handle all the persistence mappings by itself behind the scenes by acting as a middleware between Java application and the database and thereby eliminating the burden of complex manual mapping from the application developer. This will save whole lot of time and the application developer can sleep peacefully only worrying about the Java application objects and not the underlying tables. One such solution that has become very popular these days is the Hibernate framework. This is an open source persistence framework which is built using XML and Java. Within short period of time, Hibernate has proven its capability by providing solutions to complex persistence problems. This is why we are interested in learning this framework.

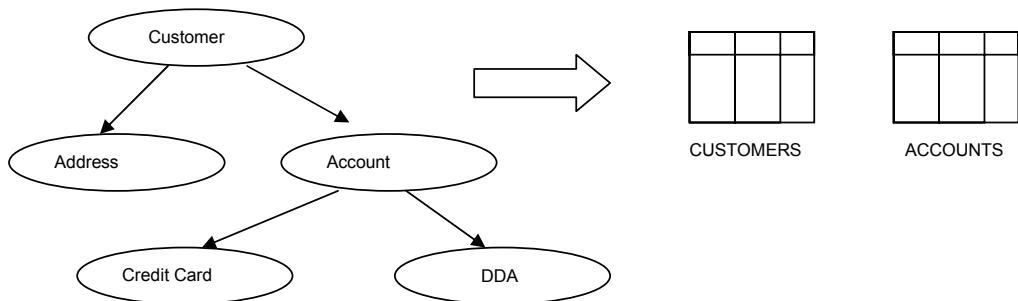
Any persistence solution that maps objects to tables in database is called as ORM (Object Relational Mapping) solution. Hibernate is one such ORM solution that sits in between Java applications and database as shown in the following figure.

**Fig 19.1 Java and Hibernate**

As you can see from the above figure, Java applications use Hibernate to map Java objects to *Tables* in relational database. So, without wasting any further time, let's see how Hibernate simplifies our life.

Why do we need ORM solution?

To better understand the importance of ORM solution, let's take a simple example and analyze some persistence details. Take a look at the following figure.

**Fig 19.2 Mapping mismatch**

As you can see from the above figure, we have five objects to represent the customer's data in two tables namely CUSTOMERS and ACCOUNTS. This is a 5:2 mismatch. Let's say we have the Customer and Address object represent the data in CUSTOMERS table, and the remaining objects represent the data in ACCOUNTS table. Both the above tables should be related with each other through a foreign key to associate the account information with the customer information. For every operation that I do with the objects, I need to manually execute the equivalent SQL query. For instance, to delete the address, I can simply set the address object to null, but at the same time I need to execute the following query:

```
UPDATE CUSTOMERS SET ADDRESS=null WHERE SSN=213213
```

So, deleting an object in this case is not a SQL DELETE because address is part of the customer record and we can only update it. This may lead to some confusion to developers. Like wise, if I want to delete the customer, then I need to delete the record in ACCOUNTS table first, followed by the record in the CUSTOMERS table due to the foreign key relationship. In real world applications, object structure and database structure will be completely apart and persisting and loading data to and from the tables get whacky and error prone. This is where we need a reliable ORM solution like Hibernate which can take care of creating and executing the SQL queries on the fly and behind the scenes.

Features of Hibernate ORM Solution

1. Hibernate ORM solution provides us with a clean API for persisting data objects.
2. Provides a dialect (Hibernate Query Language aka HQL) to specify the SQL operations.
3. Provides an XML based mapping tool to map the Java bean properties to the columns in relational tables.

Advantages of Hibernate ORM Solution

1. Allows application developers to manage the Java objects and abstracts all the SQL operations from the developer.
2. It's a light weight ORM solution and doesn't even need a application server or container to run the applications.
3. Offer loose coupling of Javabeans with the underlying tables though the use of XML as the medium of communication.
4. Consumes less resources there by improves the overall performance of the application.
5. Offers significant amount of flexibility in designing persistence applications.

Typical Hibernate Components

Hibernate commonly uses two XML files for persisting data in Java objects.

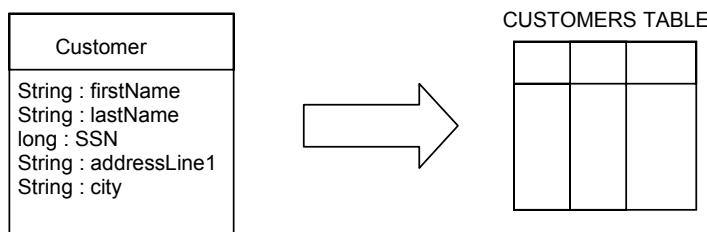
1. One is the user defined XML file that contains all the mapping information.
2. A standard **hibernate.cfg.xml** file that contains all the database configuration information along with the locations of user defined mapping XML in step 1.

Let me tell you one thing here. Understanding Hibernate is somewhat tough than the previous ones since all the concepts are tied with the ORM concepts. What I will do in

this chapter is cover only the important and most frequently used parts of Hibernate in real world applications. So, let's start with a simple example.

Mapping Beans to Tables

In our first example, we will map the JavaBean properties to the columns in a table. Following figure shows the mapping diagram.



Take a look at the code in listing 19.1.

Listing 19.1a (Customer.java) Simple Java Bean

```
package hibernate.example1;

public class Customer {

    private String firstName;
    private String lastName;
    private int ssn;
    private String addressLine1;
    private String city;
    private String state;
    private String country;

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLasttName() {
        return lastName;
    }
    public void setLasttName(String lastNamme) {
        this.lastName = lastNamme;
    }
    public int getSSN() {
        return ssn;
    }
    public void setSSN(int ssn) {
        this.ssn = ssn;
    }
}
```

```

    }
    public String getAddressLine1() {
        return addressLine1;
    }
    public void setAddressLine1(String addressLine1) {
        this.addressLine1 = addressLine1;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
}

```

Listing 19.1b (example1.hbm.xml) Mapping document

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>

    <class name="hibernate.example1.Customer" table="CUSTOMERS1">
        <id name="ssn">
            <generator class="assigned" />
        </id>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
        <property name="addressLine1" column="ADDRESS_LINE"/>
        <property name="city" column="CITY"/>
        <property name="state" column="STATE"/>
        <property name="country" column="COUNTRY"/>
    </class>
</hibernate-mapping>

```

Look at the above mapping definition for the Customer bean. We have a Customer bean with several properties whose values should be inserted into the database. The heart of the program is the “example1.hbm.xml” file which we call it as the mapping document. This mapping file defines a **class** element that does the following:

1. Tells Hibernate to map Customer bean to CUSTOMERS1 table using the following line

```
<class name="hibernate.example1.Customer" table="CUSTOMERS1">
```

2. Specifies the bean properties to the column names. The only tricky part is the **id** element. Every table that we map using the **class** element **must** have a primary key column and **id** element must be used to map a bean property to the primary key. In our case, since we want to use SSN as the primary key in the table, we map the bean property **ssn**.
3. The **id** element should also specify a **generator** child element with a **class** attribute that tells the Hibernate as to who will pass the value for the primary key. See below:

class="assigned" means Hibernate should use the bean property value as the primary key

class="increment" means Hibernate should use a unique number as the primary key and increment it for every record.

class="native" means Hibernate should generate the primary key.

In our case we used the **assigned** class since we know that **ssn** should be used as primary key.

If you notice, every property element also specifies the column name. If the property names and column names match each other (case insensitive) then we don't even have to specify the column attribute as shown below. Hibernate will automatically figure out that the column name is same as property name.

```
<property name="firstName" />
```

The **example1.hbm.xml** is the mapping file just for this example. Every example we write will have a new mapping XML file. Before we run the application, we need to tell Hibernate about two important things.

1. The details about database like Driver, URL and all that good stuff
2. The location of the example mapping files.

Though there are several ways to specify this information, we will use the XML style. The name of the XML configuration file should be **hibernate.cfg.xml**. Listing 19.1c shows this file.

Listing 19.1c (`hibernate.cfg.xml`) Hibernate configuration file.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/MyDB
        </property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.pool_size">10</property>
        <property name="show_sql">true</property>
        <property name="dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Mapping files -->
        <mapping resource="example1.hbm.xml" />

    </session-factory>
</hibernate-configuration>
```

As you can see from the above configuration file, it configures **session-factory** element by specifying all the database information and finally the location of the mapping documents. For this example, we specified the mapping file as shown below:

```
<mapping resource="example1.hbm.xml"/>
```

Now that we configured the Hibernate **SessionFactory** we need to write a test class using the above sessionfactory and start persisting the `Customer` bean. Look at the test class shown below:

Listing 19.1d (`CustomerTest`) Test class.

```
package hibernate.example1;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class CustomerTest {
```

```
public static void main(String[] args) {  
    Session session = null;  
  
    try {  
        SessionFactory sessionFactory = new Configuration().configure()  
            .buildSessionFactory();  
        session = sessionFactory.openSession();  
  
        session.beginTransaction();  
  
        Customer customer = new Customer();  
        customer.setFirstName("James");  
        customer.setLastName("Bond");  
        customer.setSsn(999998);  
        customer.setAddressLine1("1111 S St");  
        customer.setCity("London");  
        customer.setState("LDN");  
        customer.setCountry("UK");  
  
        session.save(customer);  
  
        session.getTransaction().commit();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

As you can see from the test class, before persisting the object we need to do couple things.

1. Create a SessionFactory object

```
SessionFactory sessionFactory = new Configuration().configure()  
    .buildSessionFactory();
```

2. Open the Session

```
session = sessionFactory.openSession();
```

3. Begin the transaction

```
session.beginTransaction();
```

Once we completed the above, we simply created the customer bean and populated all the properties with the values to be persisted to the CUSTOMERS table. To **insert** the customer bean, we simply need to call the `save()` method on the session as shown below:

```
session.save(customer);
```

When the above method is called, Hibernate will mark the object as “ready to be persisted” but will still not physically write it to the table. Once we **commit** the transaction, that’s when hibernate will magically insert the bean properties in the table behind the scenes. Do you see any SQL in the program? This is the beauty of the Hibernate. All you need to do is define the mappings and hibernate will take care of the rest. You don’t even have to create the tables. Trust me.

If you want to update the customer record, we simply have to call the **update()** method as,

```
session.update(customer);
```

Try modifying the bean properties and call the **update()** method, and you’ll see the changes in the table. Similarly, you can delete the customer record using the following:

```
session.delete(customer);
```

Once the mappings are specified, you can do what ever with the bean and hibernate will ensure synchronizing the changes in the table.

Steps to run the program

1. Save the two XML files in the following directory as:

```
C:/JavaTraining/chapter19/hibernate.cfg.xml  
C:/JavaTraining/chapter19/example1.hbm.xml
```

2. Save the Java files as

```
C:/JavaTraining/chapter19/hibernate/example1/Customer.java  
C:/JavaTraining/chapter19/hibernate/example1/CustomerTest.java
```

3. Compile and execute the classes as

```
C:/JavaTraining>chapter19>javac hibernate\example1\*.java  
C:/JavaTraining>chapter19>java hibernate.example1.CustomerTest
```

Make sure you have completed the environment setup explained at the beginning of the chapter. If all goes well, you see the CUSTOMERS table with a record in it.

In the above example, what I don't like is the design of the Customer bean class. Let's make the bean more object oriented by introducing Address bean to store the address information. Our Customer and Address beans will now look as shown below:
Take a look at the code in listing 19.2.

Listing 19.2a (Address.java) Simple Java Bean.

```
package hibernate.example2;

public class Address {

    private String addressLine1;
    private String city;
    private String state;
    private String country;

    public String getAddressLine1() {
        return addressLine1;
    }
    public void setAddressLine1(String addressLine1) {
        this.addressLine1 = addressLine1;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
    public String toString() {
        return addressLine1 + "," + city + "," + state + "," + country;
    }
}
```

Listing 19.2b (Customer.java) Simple Java Bean.

```
package hibernate.example2;

public class Customer {

    private String firstName;
    private String lastName;
    private int ssn;
```

```

private Address address;

public Address getAddress() {
    return address;
}
public void setAddress(Address address) {
    this.address = address;
}
public String getFirstName() {
    return firstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    return lastName;
}
public void setLastName(String lastNamme) {
    this.lastName = lastNamme;
}
public int getSsn() {
    return ssn;
}
public void setSsn(int ssn) {
    this.ssn = ssn;
}
public String toString() {
    return firstName + "," + lastName + "," + ssn + "," + address;
}
}

```

All we did is moved the address properties into a different bean and added the Address reference to the Customer bean. The association between Customer and Address bean is a **strong association**. To map such associations hibernate uses **component** mappings. Take a look at the following mapping document.

Listing 19.2c (example2.hbm.xml) Mapping document

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>

<class name="hibernate.example2.Customer" table="CUSTOMERS2">

    <id name="ssn">
        <generator class="assigned" />
    </id>
    <property name="firstName" />
    <property name="lastName" />
    <!-- Mapping for the Address object -->
    <component name="address" class="hibernate.example2.Address">
        <property name="addressLine1" />
        <property name="city" />
    </component>
</class>

```

```
<property name="state" />
<property name="country" />
</component>
</class>
</hibernate-mapping>
```

Look at the above class mapping for Customer bean. It uses the component element with the property name **address** and then maps the address bean properties to the address columns in the table. With this mapping document implement the following steps.

1. Save the above mapping xml file as

C:\JavaTraining\chapter19\example2.hbm.xml

2. Save the Java classes in the following package

C:\JavaTraining\chapter19\hibernate\example2

3. Write a test class CustomerTest.java with the following bean code

```
session.beginTransaction();

Customer customer = new Customer();
customer.setFirstName("James");
customer.setLastName("Bond");
customer.setSsn(999997);

Address address = new Address();

address.setAddressLine1("1112 S St");
address.setCity("London");
address.setState("LDN");
address.setCountry("UK");

customer.setAddress(address);

session.save(customer);

session.getTransaction().commit();
```

4. Add the following mapping location to the hibernate.cfg.xml

```
<mapping resource="example2.hbm.xml"/>
```

Executing the test class will again insert the record. Make sure you change the SSN number to a different value every time you execute the program, since this is a primary key.

By using components and nested components you can map any bean structure to any table. For instance, let's say we now want to create `HomeAddress` and `OfficeAddress` objects as shown below:



The Address bean will now have the following properties

```

class Address{
    private HomeAddress haddress;
    private OfficeAddress oaddress;

    // Get Set methods
}
  
```

The `class` element in the mapping XML will now look as shown below:

```

<class name="hibernate.example3.Customer" table="CUSTOMERS3">
    <id name="ssn">
        <generator class="assigned" />
    </id>
    <property name="firstName" />
    <property name="lastName" />

    <component name="address" class="hibernate.example3.Address">
        <component name="haddress" class="hibernate.example3.HomeAddress">
            <property name="addressLine1" column="HADDRESSLINE1" />
            <property name="city" column="HCITY" />
            <property name="state" column="HSTATE" />
            <property name="country" column="HCOUNTRY" />
        </component>
        <component name="oaddress" class="hibernate.example3.OfficeAddress">
            <property name="addressLine1" column="OADDRESSLINE1" />
            <property name="city" column="OCITY" />
            <property name="state" column="OSTATE" />
            <property name="country" column="OCOUNTRY" />
        </component>
    </component>
</class>
  
```

```
</component>  
</class>
```

As you can see from the above definition, we nested **component** elements. The haddress and oaddress are the child components of address component. By using nested components, you can map any complex bean structure to the underlying table columns.

In the next section, let's see the different ways we can configure Hibernate.

Configuring Hibernate

One of the striking features of Hibernate is that it can be used in diverse variety of environments. Therefore, it is every important that we understand some of the important ways of configuring Hibernate.

Hibernate uses a class called `SessionFactory` for persisting and loading data objects to and from the database. The `SessionFactory` class as the name suggests is a factory of session objects. Before we open the session, we need to **configure** the `SessionFactory` with the underlying database(s). Hibernate uses Configuration class to configure the session factory.

Usually configuring `SessionFactory` depends on application demands. Some applications may use just one database and others might use several databases for persistence. Also configuration is based on the environment in which Hibernate is used. For instance, using Hibernate with EJB requires a different configuration etc. So, let's see some important configuration schemes for the `SessionFactory`.

Case 1: XML based configuration

This is the scheme we demonstrated in our previous examples where all the database information and mapping files are defined in a XML file named **hibernate.cfg.xml**. With this scheme, the XML should be placed in the classpath of the application which is typically the root directory of the application. Following listing shows the configuration XML file:

Listing 19.3a (`hibernate.cfg.xml`) XML configuration file

```
<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration PUBLIC  
"-//Hibernate/Hibernate Configuration DTD//EN"  
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/MDIT
        </property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.pool_size">10</property>
        <property name="show_sql">true</property>
        <property name="dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <!-- Mapping files -->
        <mapping resource="example3.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

With the above configuration, our application can then create the SessionFactory object as shown below:

```
SessionFactory factory = new Configuration().configure().buildSessionFactory();
```

Once the factory instance is obtained, we can open the **session** objects and start persisting objects as shown below:

```
session = factory.openSession();
session.save(customer);
```

The above scheme works for single session factory. What if we want to work with multiple databases? In such cases, we need multiple configuration XML files such as **security.cfg.xml**, **financial.cfg.xml** etc., with each configuration file containing information about a particular database. In such cases, we can obtain the session factory objects as shown below:

```
SessionFactory factory1 = new
    Configuration().configure("security.cfg.xml").buildSessionFactory();
SessionFactory factory2 = new
    Configuration().configure("financial.cfg.xml").buildSessionFactory();
```

You can use this type of configuration if you prefer to use XML style.

Case 2: Using hibernate.properties file.

This configuration scheme is much the same as the previous one, except that it uses **hibernate.properties** file in the **classpath**. The property file will then list the properties as shown below:

```
hibernate.properties
```

```
# JDBC Properties
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost:3306/MyDB
hibernate.connection.username = root
hibernate.connection.password = root
hibernate.dialect = org.hibernate.dialect.MySQLDialect
```

However, if you are using JNDI based datasources, then you need to use the following properties:

```
hibernate.connection.datasource = jdbc/MySQL
hibernate.transaction.factory_class=
org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class=
    org.hibernate.transaction.JbossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.MySQLDialect
```

There are also several other properties that you can define like transaction properties, cache properties, connection properties etc. Once you have this property file in the classpath, you can obtain the SessionFactory object as shown below:

```
SessionFactory factory = new Configuration().buildSessionFactory();
```

Again, if you want to work with multiple databases, you need to create multiple property files and load them as shown below:

```
Properties p = new Properties();
p.load(new FileInputStream("security.properties"));

Configuration cfg1 = new Configuration();
cfg1.setProperties(p);
```

There is one more thing we missed here. We also need to specify the location of the mapping files, right? Following is how we do:

```
cfg1.addResource("example1.hbm.xml");
cfg1.addResource("example2.hbm.xml");
```

Once the mapping resources are added, we can obtain the session factory object as shown below:

```
SessionFactory factory = cfg1.buildSessionFactory();
```

You can choose which ever style you want. If you prefer XML style then use the Case 1 scheme, if you prefer using property files then use the Case 2 scheme.

Case 3: Programmatic configuration

In this scheme, you neither have to use XML file nor have to use the property file. You'll simply specify the configuration in the program itself as shown below:

```
Configuration cfg1 = new Configuration();  
  
cfg1.addResource("example1.hbm.xml");  
cfg1.addResource("example2.hbm.xml");  
  
cfg1.setProperty("hibernate.connection.driver_class","com.mysql.Driver");  
cfg1.setProperty("hibernate.connection.url","jdbc:mysql://localhost:3306/MyDB")  
cfg1.setProperty("hibernate.connection.username","root");  
cfg1.setProperty("hibernate.dialect","org.hibernate.dialect.MySQLDialect");  
  
SessionFactory factory = cfg1.buildSessionFactory();
```

The advantage with programmatic configuration is that you can pass the property values at runtime. This gives your application more flexibility while working with multiple databases.

This completes all the important and widely used schemes of configuring hibernate. In the next section we will see the most important aspect of ORM, the associations.

Associations

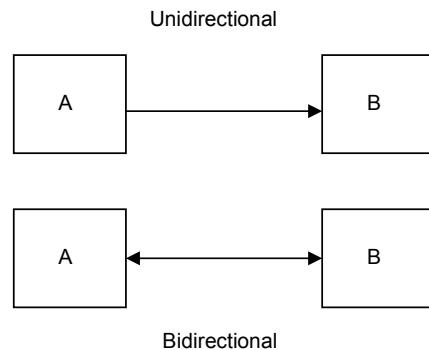
Associations basically indicate that one class retains the relationship to another class over an extended period of time. If you recall our Customer-Address example, an *address* is *associated* with a *customer*. This is a very strong relationship and is normally read as "*has a*" relationship. Following are some of the examples:

Customer *has* Address

Product *has a* Name

A company *has* employees etc..

Moreover associations can be *unidirectional* or *bidirectional*. Consider that we have two parties namely A and B that are associated with each other. If the association is *unidirectional* from A to B, then A can see changes made to B and not vice-versa. (i.e B cannot see the changes in A). With a *bidirectional* association, both A and B can see each others changes. Fig 19.3 shows both unidirectional and bidirectional associations.

**Fig 19.3** Unidirectional and Bi-directional associations

Associations are basically represented in the following four ways:

1. One-to-One Association
2. One-to-Many association
3. Many-to-One association
4. Many-to-Many association

Let's look at each of these associations one by one.

Many-to-One Association

Unidirectional Many-to-One/One-to-many Association

The simple example for this association is multiple books belonging to the same publisher. At the same time if only the Books should be able to see the changes made to the publisher, it becomes *unidirectional*.

BOOKS				PUBLISHERS	
BOOK_ID	PUB_ID	AUTHOR	TITLE	PUB_ID	PUB_NAME
101	123			123	
102	123				

The class definition for Publisher class will be as shown below:

```
<class name="hibernate.example4.Publisher" table="PUBLISHERS4">
    <id name="publisherId" column="PUB_ID">
        <generator class="increment" />
    </id>
    <property name="publisherName" column="PUB_NAME" />
</class>
```

The above definition maps publisherId and publisherName to the respective columns in the PUBLISHERS table. Now look at the class definition for the Book class.

```
<class name="hibernate.example4.Book" table="BOOKS4">
    <id name="bookId" column="BOOK_ID">
        <generator class="increment" />
    </id>
    <many-to-one name="publisher" column="PUB_ID" />
    <property name="author" column="AUTHOR" />
    <property name="title" column="TITLE" />
</class>
```

In the above definition, the **many-to-one** element tells hibernate to use the PUBLISHERS primary key as the foreign key in the BOOKS table. The PUB_ID column in BOOKS table will now have the same value as the PUB_ID column in the PUBLISHERS table.

Take a look at the code in listing 19.4.

Listing 19.4a (Publisher.java) Publisher bean

```
package hibernate.example4;

public class Publisher {

    int publisherId;
    String publisherName;

    public int getPublisherId() {
        return publisherId;
    }

    public void setPublisherId(int publisherId) {
        this.publisherId = publisherId;
    }

    public String getPublisherName() {
        return publisherName;
    }

    public void setPublisherName(String publisherName) {
        this.publisherName = publisherName;
    }
}
```

Listing 19.4b (Book.java) Book bean

```
package hibernate.example4;

public class Book {

    int bookId;
    Publisher publisher;
```

```

String author;
String title;

public String getAuthor() {
    return author;
}
public void setAuthor(String author) {
    this.author = author;
}
public int getBookId() {
    return bookId;
}
public void setBookId(int bookId) {
    this.bookId = bookId;
}
public Publisher getPublisher() {
    return publisher;
}
public void setPublisher(Publisher publisher) {
    this.publisher = publisher;
}
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}
}

```

Listing 19.4c (`Book.java`) Beans definitions

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="hibernate.example4.Publisher" table="PUBLISHERS4">
    <id name="publisherId" column="PUB_ID">
        <generator class="increment" />
    </id>
    <property name="publisherName" column="PUB_NAME" />
</class>
<class name="hibernate.example4.Book" table="BOOKS4">
    <id name="bookId" column="BOOK_ID">
        <generator class="increment" />
    </id>
    <many-to-one name="publisher" column="PUB_ID" />
    <property name="author" column="AUTHOR" />
    <property name="title" column="TITLE" />
</class>
</hibernate-mapping>

```

Listing 19.4d (Test.java) Unidirectional many-to-one test.

```
package hibernate.example4;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class Test {

    public static void main(String[] args) {

        Session session = null;
        try {
            SessionFactory sessionFactory = new Configuration().configure()
                .buildSessionFactory();
            session = sessionFactory.openSession();

            session.beginTransaction();

            // Create a publisher
            Publisher pub = new Publisher();
            pub.setPublisherName("Wrox");

            // Create Books and set the same publisher
            Book book1 = new Book();
            book1.setPublisher(pub);
            book1.setAuthor("James");
            book1.setTitle("J2EE1");

            Book book2 = new Book();
            book2.setPublisher(pub);
            book2.setAuthor("John");
            book2.setTitle("J2EE2");

            session.save(pub);
            session.save(book1);
            session.save(book2);

            session.getTransaction().commit();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

As you can see from the test class, we created one publisher and associated with many books as shown below:

```
Publisher pub = new Publisher();
pub.setPublisherName("Wrox");

Book book1 = new Book();
book1.setPublisher(pub);
book1.setAuthor("Phani1");
```

```
book1.setTitle("J2EE1");

Book book2 = new Book();
book2.setPublisher(pub);
book2.setAuthor("Phani2");
book2.setTitle("J2EE2");
```

Execute the test class and you'll see 1 record inserted in the PUBLISHERS4 table, and 2 records in the BOOKS4 table.

Note: Hibernate will also create the tables for you with all the proper table constraints. In this example it creates PUBLISHERS4 and BOOKS1 tables that we specified in the xml file as shown below. These tables will be created only once.

```
<class name="hibernate.example4.Publisher" table="PUBLISHERS4">
<class name="hibernate.example4.Book" table="BOOKS4">
```

Test to verify the unidirectional association

Test: Given a Book, display the publisher information

```
SessionFactory sessionFactory = new Configuration().configure()
                                                .buildSessionFactory();
Session session = sessionFactory.openSession();
session.beginTransaction();

Book book = (Book) session.get(Book.class, new Integer(1));
Publisher pub = book.getPublisher();
System.out.println(pub.getPublisherName());

session.getTransaction().commit();
```

The above code will print the publisher name of the Book.

Bidirectional Many-to-one/One-to-Many

In this case, the publisher should have the knowledge of all the books it published. Therefore, the publisher class should have a **collection** property for books as shown below:

```
public class Publisher {

    int publisherId;
    String publisherName;
    Set books;
    // Getter and Setters

    void addBook(Book book) {
        books.add(book);
    }
}
```

```

        book.setPublisher(this);
    }
}

```

Look at the addBook() utility method. If we try to associate a book to a publisher, all we need to do is just update the record in the books table to set the publisher using the following update statement.

update BOOKS set PUB_ID=? where BOOK_ID=?

However hibernate recognizes two changes, one for updating the book and the other for updating the publisher of the book. But logically speaking, both these operations are same. To make Hibernate just execute the above update statement, we need to set the **inverse** attribute to **true** as shown in the following class definition:

```

<class name="hibernate.example4.Publisher" table="PUBLISHERS4">
    <id name="publisherId" column="PUB_ID">
        <generator class="increment" />
    </id>
    <property name="publisherName" column="PUB_NAME" />
    <set name="books" inverse="true">
        <key column="PUB_ID" />
        <one-to-many class="hibernate.example4.Book" />
    </set>
</class>

```

Note that the set element defines that each book should have the same PUB_ID and there will be a one-to-many association between the publisher and the book.

Test to verify the bidirectional association

Test: Given a Publisher, display all the Books

```

SessionFactory sessionFactory = new Configuration().configure()
    .buildSessionFactory();
Session session = sessionFactory.openSession();
session.beginTransaction();

Publisher pub = (Publisher) session.get(Publisher.class, new Integer(1));
Set books = pub.getBooks();
System.out.println(books);

session.getTransaction().commit();

```

As you can see from the above code, we searched for a publisher whose ID is 1 and then displayed the books. The above code will print all the book objects.

Both one-to-many and many-to-one relationships work just the same way. As can you see from the previous example, a bidirectional many-to-one also includes a one-to-many association.

One-to-One Association

As the name suggests, with this association, one object (record in one table) should be associated with one and only one object (record in another table). For example, a book can only be associated with one publisher. Let's again look at both unidirectional and bidirectional one-to-one associations using the Book and Publisher.

This association can be implemented in two ways.

1. Using Foreign Key
2. Using Primary Key

Unidirectional Foreign Key Association

This is the simplest of two. In this case, a one-to-one association is derived from a many-to-one relationship by setting the **unique** property to **true** for the **PUB_ID** column in the **BOOKS** table. The **PUB_ID** then becomes the foreign key to the **PUBLISHERS** table primary key. For instance let's say we have the following tables:

BOOKS				PUBLISHERS	
BOOK_ID	PUB_ID	AUTHOR	TITLE	PUB_ID	PUB_NAME
101	123				
102	123				

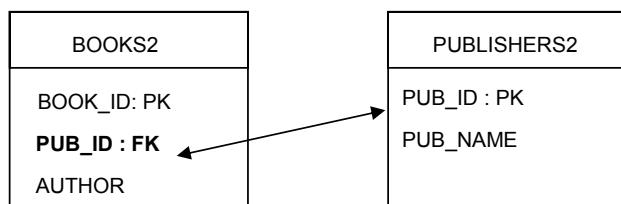
As you can see from the above table, a one-to-one relationship can be obtained from a many-to-one relationship by eliminating the duplicate foreign key (**PUB_ID**) in the books table. We tell this to hibernate by simply setting the **unique** attribute to **true** for the **PUB_ID** in the class definition for book as shown below:

```
<class name="hibernate.example5.Book" table="BOOKS5">
    <id name="bookId" column="BOOK_ID">
        <generator class="increment" />
    </id>
    <many-to-one name="publisher" column="PUB_ID" unique="true" />
    <property name="author" column="AUTHOR" />
    <property name="title" column="TITLE" />
</class>
```

The class definition for Publisher will remain the same as shown below:

```
<class name="hibernate.example5.Publisher" table="PUBLISHERS5">
    <id name="publisherId" column="PUB_ID">
        <generator class="increment" />
    </id>
    <property name="publisherName" column="PUB_NAME" />
</class>
```

Notice that we are using different table names for this example, BOOKS5 and PUBLISHERS5. When Hibernate generates the DDL for the tables, it adds a foreign key constraint to the PUB_ID in the Books table as shown below:



With the above relationship, if we try to add more than one book to the same publisher it will add the first book and then throws an exception. Following is the test code to verify this fact.

```
Publisher pub = new Publisher();
pub.setPublisherName("Wrox");

Book book1 = new Book();
book1.setPublisher(pub);
book1.setAuthor("Phani1");
book1.setTitle("J2EE1");

Book book2 = new Book();
book2.setPublisher(pub);
book2.setAuthor("Phani2");
book2.setTitle("J2EE2");

session.save(pub);
session.save(book1);
session.save(book2);
```

When you run the above code, you'll see one record inserted in the PUBLISHERS2 table, and one record in the BOOKS2 table. You'll then see an exception something like shown below:

```
Caused by: java.sql.BatchUpdateException: Duplicate entry '1' for key 2
      at
com.mysql.jdbc.ServerPreparedStatement.executeBatch(ServerPreparedStatement.java:657)
```

The test to verify the unidirectional association is same as unidirectional one-to-many/many-to-one association we discussed before.

Bidirectional Foreign Key Association

Making this association bidirectional is nothing but having the publisher access the book information. The only difference is that in this case, it has to be just one book. Firstly, we need to add the book reference in the **Publisher** class as shown below:

```
public class Publisher {  
  
    int publisherId;  
    String publisherName;  
  
    Book book;  
  
    // Getter and Setters for book  
  
}
```

Secondly, the class definition for the **Publisher** will look as shown below:

```
<class name="hibernate.example5.Publisher" table="PUBLISHERS5">  
    <id name="publisherId" column="PUB_ID">  
        <generator class="increment" />  
    </id>  
    <property name="publisherName" column="PUB_NAME" />  
    <one-to-one name="book" class="hibernate.example5.Book"  
        property-ref="publisher" />  
</class>
```

As you can see from the above definition, we added a one-to-one mapping of the book to the publisher using the **property-ref** attribute. This tells Hibernate that the **book** association in **Publisher** is the reverse of **publisher** association in **Book**.

Test to verify the bidirectional association

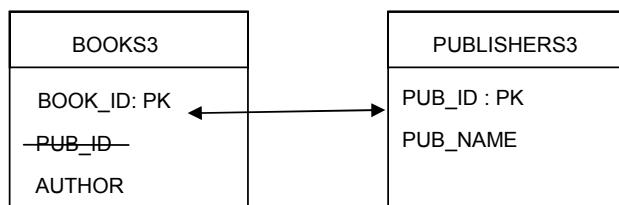
Test: Given a **Publisher**, display the associated **Book**

```
SessionFactory sessionFactory = new Configuration().configure()  
    .buildSessionFactory();  
session = sessionFactory.openSession();  
session.beginTransaction();  
  
Publisher pub = (Publisher)session.get(Publisher.class,new Integer(1));  
Book book = pub.getBook();  
System.out.println(book);  
  
session.getTransaction().commit();
```

The above test code will display the book information for the publisher whose id is 1. This completes both unidirectional and bidirectional one-to-one association using foreign keys. Let's now look at the same using primary key.

Unidirectional Primary Key Association

With this type of association, both the books and the publishers table will have the same primary key as shown in the following figure.



As you can see from the above figure, both the BOOK_ID and PUB_ID must always be the same. This way we can completely delete the foreign key PUB_ID from the books table. The challenge we have here is that, we somehow need to tell hibernate to use PUB_ID in the publisher table as both **primary key** and also the **foreign key** of the BOOK_ID in the books table. Only then, hibernate will insert same value in both the tables. This is done using the **constrained=true** attribute as shown in the following publisher class definition:

```

<class name="hibernate.example5.Publisher" table="PUBLISHERS6">
    <id name="publisherId" column="PUB_ID">
        <generator class="foreign">
            <param name="property">book</param>
        </generator>
    </id>
    <property name="publisherName" column="PUB_NAME" />
    <one-to-one name="book" constrained="true" />
</class>

```

There are three important things we need to understand with the above definition.

1. The direction of the association is from Publisher to Book. This is because, we first need to generate the primary key of the Book, and then use its value as the primary key in the publisher table. This means publisher is dependent on the book. This is shown by the "book" property in the above definition.

2. The **constraint=true** attribute tells hibernate that there is a foreign key constraint on the primary key of the publisher table.
3. A special generator named **foreign** must be used which tells hibernate to use the value of the primary key on the books table as the primary key value in the publishers table.

Hmmm.. Confusing, right?. I can understand but learn it twice and you'll get it.

The class definition for the book will be simply be a straight forward mapping as shown below:

```
<class name="hibernate.example5.Book" table="BOOKS6">
    <id name="bookId" column="BOOK_ID">
        <generator class="increment" />
    </id>
    <property name="author" column="AUTHOR" />
    <property name="title" column="TITLE" />
</class>
```

The test code for adding records to the table should be done carefully. Let me ask you a question. Should the publisher object set the value for book, or should the book object set the value for publisher? Since the association is from publisher to book, publisher should set the book. This is very important to understand otherwise you will get an exception. Following is how the test code looks:

```
session.beginTransaction();

Book book1 = new Book();
book1.setAuthor("Phani1");
book1.setTitle("J2EE1");

Publisher pub = new Publisher();
pub.setPublisherName("Wrox");
pub.setBook(book1);

session.save(book1);
session.save(pub);

session.getTransaction().commit();
```

As you can see from the above code, we first created the book object and then passed the book to the publisher object. Here is a simple trick to remember:

Trick: If we have a method invocation as **a.setTest(b)**, where a and b are two objects, then the direction of the association is from a to b.

Bidirectional Primary Key Association

Making the previous association as bidirectional is very simple. This time we need to add a publisher property in the book object using a one-to-one association as shown below:

```
<class name="hibernate.example5.Book" table="BOOKS6">
    <id name="bookId" column="BOOK_ID">
        <generator class="increment" />
    </id>
    <property name="author" column="AUTHOR" />
    <property name="title" column="TITLE" />
    <one-to-one name="publisher" class="hibernate.example5.Publisher"/>
</class>
```

The above tells hibernate to also retrieve the publisher while retrieving the book. You can now use either of the following test code to get the details of book and publisher:

```
Publisher pub = (Publisher)session.get(Publisher.class,new Integer(1));
Book b = pub.getBook();

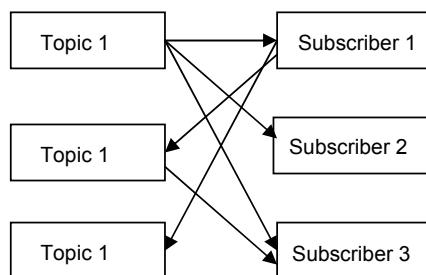
(or)

Book book = (Book)session.get(Book.class,new Integer(1));
Publisher pub = book.getPublisher();
```

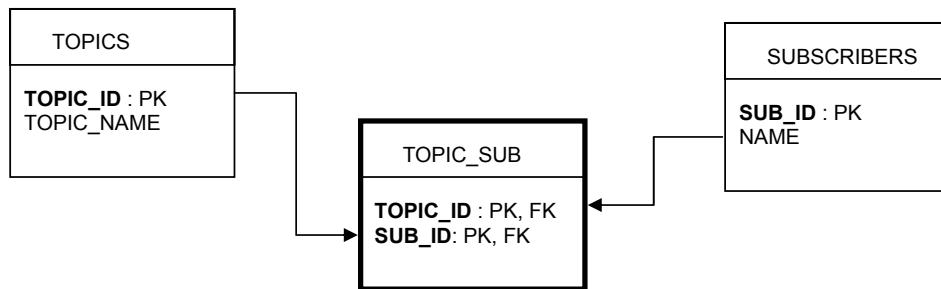
This completes every thing we need to know about one-to-one association. The last one is the many-to-many association. This is one of the rarely seen associations. Let's see how it looks.

Many-to-Many Association

The classical example for this association is the one between Topics and Subscriptions. As you know, a single topic will have more than one subscriber and a single subscriber can subscribe to more than one topic. Do you see a many-to-many relationship here? I am sure you did. Following figure shows this association:



Implementing a many-to-many relationship requires an additional table called the link table. There is no way you can implement this scenario using two tables. The link table basically contains the primary keys of both the tables as shown in the following diagram:



Let's see both unidirectional and bidirectional many-to-many associations.

Unidirectional Many-to-Many association

In this case, the Topic object will define a collection of subscriber objects as shown below:

```

public class Topic{
    int topicid;
    String topicName;
    Set subscribers;
    // Getters and Setters
}
  
```

The subscriber class will define a collection of topics objects as shown below:

```

public class Subscriber{
    int sid;
    String name;
    Set topics;
    // Getters and Setters
}
  
```

With unidirectional association, a topic will have the knowledge of all the subscribers and not vice-versa. In this case, the class definition for Topic class will look as shown below:

```
<class name="hibernate.example6.Topic" table="TOPICS6">
    <id name="topicId" column="TOPIC_ID">
        <generator class="native" />
    </id>
    <property name="topicName" column="TOPIC_NAME" />
    <set name="subscribers" table="TOPIC_SUB6">
        <key column="TOPIC_ID" />
        <many-to-many column="SUB_ID"
                      class="hibernate.example6.Subscriber" />
    </set>
</class>
```

As you can see from the above definition, the set element tells hibernate to insert TOPIC_ID from TOPICS6 table and SUB_ID from SUBSCRIBERS6 table into the link table TOPIC_SUB6 when the association is made.

Following is the class definition for subscriber class. This class will not have the subscribers property defined as we should not allow subscribers to access topics.

```
<class name="hibernate.example6.Subscriber" table="SUBSCRIBERS6">
    <id name="sid" column="SUB_ID">
        <generator class="increment" />
    </id>
    <property name="name" column="NAME" />
</class>
```

Following is the source code with the complete XML mappings and the bean classes.

Listing 19.5a (Topic.java) Simple bean

```
package hibernate.example6;

import java.util.HashSet;
import java.util.Set;

public class Topic {

    int topicId;
    String topicName;
    Set subscribers = new HashSet();

    public Set getSubscribers() {
        return subscribers;
    }

    public void setSubscribers(Set subscribers) {
        this.subscribers = subscribers;
    }

    public int getTopicId() {
        return topicId;
    }
}
```

```
public void setTopicId(int topicId) {
    this.topicId = topicId;
}

public String getTopicName() {
    return topicName;
}

public void setTopicName(String topicName) {
    this.topicName = topicName;
}

public void addSubscriber(Subscriber sub) {
    subscribers.add(sub);
    sub.getTopics().add(this);
}
}
```

Listing 19.5b (Subscriber.java) Simple bean

```
package hibernate.example6;
import java.util.HashSet;
import java.util.Set;
public class Subscriber {

    int sid;
    String name;
    // Bi directional
    Set topics = new HashSet();
    public Set getTopics() {
        return topics;
    }
    public void setTopics(Set topics) {
        this.topics = topics;
    }
    public void addTopic(Topic topic) {
        topics.add(topic);
        topic.getSubscribers().add(this);
    }
    /////////////////////
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSid() {
        return sid;
    }
    public void setSid(int sid) {
        this.sid = sid;
    }
}
```

Listing 19.5c (example5.hbm.xml) Class definitions.

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="hibernate.example6.Topic" table="TOPICS6">
    <id name="topicId" column="TOPIC_ID">
        <generator class="native" />
    </id>
    <property name="topicName" column="TOPIC_NAME" />
    <set name="subscribers" table="TOPIC_SUB6">
        <key column="TOPIC_ID" />
        <many-to-many column="SUB_ID"
                      class="hibernate.example6.Subscriber" />
    </set>
</class>
<class name="hibernate.example6.Subscriber" table="SUBSCRIBERS6">
    <id name="sid" column="SUB_ID">
        <generator class="increment" />
    </id>
    <property name="name" column="NAME" />
</class>
</hibernate-mapping>

```

In both the Topic and Subscriber classes, we added a collection property along with the following utility methods for subscribers to add topics and vice versa.

```

public void addSubscriber(Subscriber sub) {
    subscribers.add(sub);
    sub.getTopics().add(this);
}

public void addTopic(Topic topic) {
    topics.add(topic);
    topic.getSubscribers().add(this);
}

```

Now, let's write the test code to add 3 subscribers to 1 topic as shown below:

Listing 19.5d (Test.java) Test class.

```

package hibernate.example6;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class Test {

    public static void main(String[] args) {

```

```
Session session = null;
try {
    SessionFactory sessionFactory = new Configuration().configure()
        .buildSessionFactory();
    session = sessionFactory.openSession();

    session.beginTransaction();

    Subscriber sub1 = new Subscriber();
    sub1.setName("Phani1");

    Subscriber sub2 = new Subscriber();
    sub2.setName("Phani2");

    Subscriber sub3 = new Subscriber();
    sub3.setName("Phani3");

    Topic topic = new Topic();
    topic.setTopicName("Topic 1");

    topic.addSubscriber(sub1);
    topic.addSubscriber(sub2);
    topic.addSubscriber(sub3);

    session.save(sub1);
    session.save(sub2);
    session.save(sub3);
    session.save(topic);

    session.getTransaction().commit();

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

As you can see from the above code, we created 3 subscribers and added them to one topic. When we execute the above code, following is how the contents of three tables will look like:

TOPICS	
TOPIC_ID	TOPIC_NAME
1	Movies

TOPIC_ID	SUB_ID
1	1
1	2
1	3

SUBSCRIBERS	
SUB_ID	NAME
1	James
2	John
3	Sara

As you can see from the above link table, it inserts the primary keys of the both TOPICS and SUBSCRIBERS table. With the above data, when we do a search for the topic using the topic id, hibernate will return all the three subscribers with the following code:

```
Topic topic = (Topic)session.get(Topic.class,new Integer(1));
```

```
System.out.println(topic.getSubscribers());
```

This is how a unidirectional many-to-many association is implemented. Let's now look at the bidirectional many-to-many association.

Bidirectional Many-to-Many association

In this case, we need to update the class definition of the subscriber to include the topics property as a set element as shown below:

```
<class name="hibernate.example6.Subscriber" table="SUBSCRIBERS6">
    <id name="sid" column="SUB_ID">
        <generator class="increment" />
    </id>
    <property name="name" column="NAME" />
    <set name="topics" table="TOPIC_SUB" inverse="true" >
        <key column="SUB_ID" />
        <many-to-many column="TOPIC_ID" class="hibernate.example6.Subscriber" />
    </set>
</class>
```

As you can see from the above definition, this is simply the reverse of the subscribers property in the Topic class definition. However, we need to add the **inverse="true"** attribute to tell hibernate to synchronize both the ends of the association. The class mapping for the Topic will remain the same as the one with unidirectional association.

With these class definitions, let's have James subscribe to two new topics as shown below:

```
Topic t1 = new Topic();
t1.setTopicName("Java");

Topic t2 = new Topic();
t2.setTopicName("J2EE");

// Get the Subscriber
Subscriber sub = (Subscriber) session.get(Subscriber.class, new Integer(1));

sub.addTopic(t1);
sub.addTopic(t2);

session.save(t1) ;
session.save(t2) ;
session.save(sub) ;
```

With the above code, the tables will now look as shown below :

TOPICS	
TOPIC_ID	TOPIC_NAME
1	Movies
2	Java
3	J2EE

TOPIC_SUB	
TOPIC_ID	SUB_ID
1	1
1	2
1	3
2	1
3	1

SUBSCRIBERS	
SUB_ID	NAME
1	James
2	John
3	Sara

Now, to retrieve the topics for a given subscriber, we can use the following code:

```
Subscriber sub = (Subscriber)session.get(Subscriber.class,new Integer(1));
System.out.println(sub.getTopics());
```

You know what, we have now successfully completed all the associations. Though they may look confusing at the beginning, read them 2-3 times and you'll get the trick.

In the next section, we will learn new type of associations called polymorphic associations. These associations are based on inheritance.

Polymorphic Associations

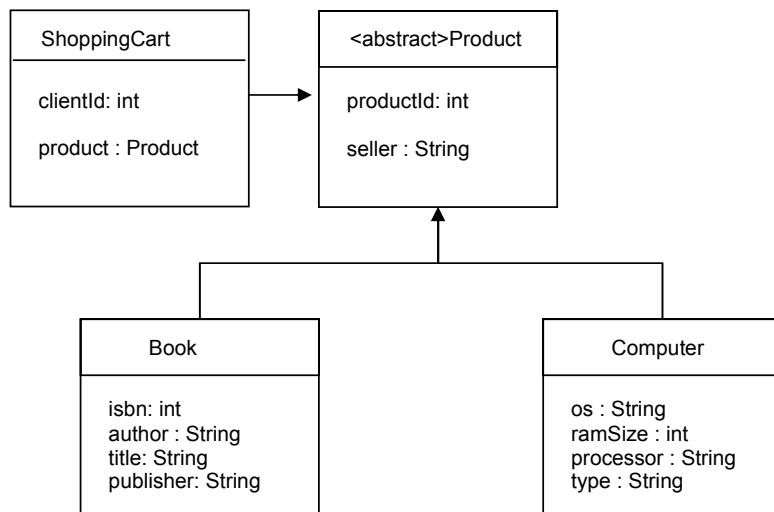
As we all know that Inheritance is one of the key features in object oriented design where specific information is represented by the child classes. This being a common scenario when modeling Java objects, we need to know how Hibernate assists us in mapping inheritance hierarchy to relational tables. Clearly, inheritance hierarchy is the main pointer for the structural mismatch with the tables.

Polymorphic association is an association in which a child object is referenced by a parent class reference. The parent component can be a concrete parent class, or an abstract class or even an interface. With polymorphic associations following are the two cases we need to look at:

1. Mapping a inheritance hierarchy to tables
2. Write queries that return all the child objects that match the parent interface.

Let's take a simple example and then see how we can implement the above two cases.

Take a look at the following inheritance hierarchy.



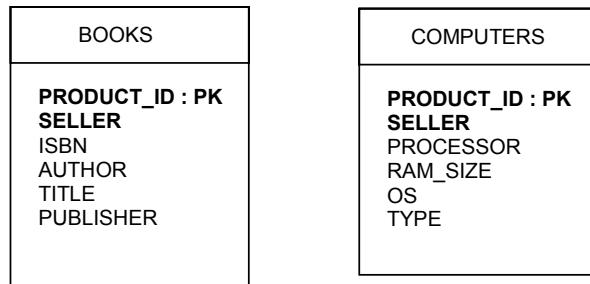
As you can see from the above figure, the shopping cart object references a product object which can be either be object of Book or Computer class. There are three approaches with which we can map the above hierarchy to the database tables.

1. Table per concrete class
2. Table per hierarchy
3. Table per child class

Let's look at each of them and find out the best approach.

Table per Concrete class

In this approach, as the name suggests we use one table per concrete class in the hierarchy. In our case `Book` and `Computer` are two concrete classes of abstract `Product` class. Therefore, Hibernate will create one table for each subclass with columns for all the properties in the sub classes including the inherited properties as shown below:



As you can see from the above two tables, hibernate will create `PRODUCT_ID` and `SELLER` columns for the inherited properties in each of the table. It's important to note that hibernate will not create a table for the `Product` class as it is abstract. Here lies the challenge. `ShoppingCart` object references either `Book` or `Computer` object through the `Product` interface, right? In this approach, since hibernate will not create a table for `product` class which is like a broken link, how does hibernate associate `SHOPPINGCART` table directly with the `BOOK` and `COMPUTER` tables? The solution is the `ShoppingCart` table should add two columns as listed below:

1. A column that defines the **type** of the subclass, for instance `PRODUCT_TYPE` whose value identifies the type of the subclass object (`Book` or `Computer`)
2. A column that stores the primary keys of both the `Book` and `Computer` table some thing like `ITEM_ID`. This is not a foreign key but can be thought of as a composite foreign key since it holds to both the subclass tables primary keys.

The `ShoppingCart` element should use the element **any** to tell hibernate about the two columns as shown below:

```
<any name="product" meta-type="string" id-type="int" cascade="save-update">
  <meta-value value="BOOK" class="hibernate.example7.Book" />
  <meta-value value="COMPUTER" class="hibernate.example7.Computer" />
  <column name="PRODUCT_TYPE" />
  <column name="ITEM_ID" />
</any>
```

As you can see from the above mapping, the **any** element defined two columns as shown below:

```
<column name="PRODUCT_TYPE" />
<column name="ITEM_ID" />
```

The `PRODUCT_TYPE` column contains two values to identify two sub classes. We also need to tell Hibernate what possible values it can have and also the type of values. We use `meta-value` element for the former and `meta-type` attribute for the later. The `id-`

type attribute specifies the column type for ITEM_ID column. Let's now look at the complete code for this example.

Listing 19.6a (`Product.java`) An abstract product class.

```
package hibernate.example7;

public abstract class Product {

    int productId;
    String seller;

    public abstract int getProductId();

    public abstract void setProductId(int productId);

    public abstract String getSeller();

    public abstract void setSeller(String seller);

}
```

Listing 19.6b (`Book.java`) A simple class

```
package hibernate.example7;

public class Book extends Product {

    String isbn;
    String author;
    String title;
    String publisher;

    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getPublisher() {
        return publisher;
    }
    public void setPublisher(String publisher) {
```

```
        this.publisher = publisher;
    }
    public int getProductId() {
        return productId;
    }
    public void setProductId(int productId) {
        this.productId = productId;
    }
    public String getSeller() {
        return seller;
    }
    public void setSeller(String seller) {
        this.seller = seller;
    }
}
```

Listing 19.6c (Computer.java) A simple class

```
package hibernate.example7;

public class Computer extends Product {

    String processor;
    int ramSize;
    String os;
    String type;

    public String getOs() {
        return os;
    }
    public void setOs(String os) {
        this.os = os;
    }
    public String getProcessor() {
        return processor;
    }
    public void setProcessor(String processor) {
        this.processor = processor;
    }
    public int getRamSize() {
        return ramSize;
    }
    public void setRamSize(int ramSize) {
        this.ramSize = ramSize;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public int getProductId() {
        return productId;
    }
    public void setProductId(int productId) {
        this.productId = productId;
    }
    public String getSeller() {
```

```

        return seller;
    }
    public void setSeller(String seller) {
        this.seller = seller;
    }
}

```

Listing 19.6d (`ShoppingCart.java`) A simple class

```

package hibernate.example7;

public class ShoppingCart {

    int clientId;
    Product product;

    public int getClientId() {
        return clientId;
    }
    public void setClientId(int clientId) {
        this.clientId = clientId;
    }
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
}

```

The above code represent the JavaBean classes for `Product`, `Book`, `ShoppingCart` and `Computer`. Now, let's write the mapping file.

Listing 19.6e (`example7.hbm.xml`) Class definitions

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="hibernate.example7.Book" table="BOOK7"
          polymorphism="implicit">
        <id name="productId" column="BOOK_ID">
            <generator class="increment" />
        </id>
        <property name="seller" column="SELLER" />
        <property name="author" column="AUTHOR" />
        <property name="title" column="TITLE" />
        <property name="publisher" column="PUBLISHER" />
    </class>
    <class name="hibernate.example7.Computer" table="COMPUTERS7"
          polymorphism="implicit">
        <id name="productId" column="PRODUCT_ID">

```

```

        <generator class="increment" />
    </id>
    <property name="seller" column="SELLER" />
    <property name="processor" column="PROCESSOR" />
    <property name="ramSize" column="RAM" />
    <property name="os" column="OS" />
    <property name="type" column="TYPE" />
</class>
<class name="hibernate.example7.ShoppingCart" table="SHOPPINGCART7">
    <id name="clientId" column="PRODUCT_ID">
        <generator class="increment" />
    </id>
    <any name="product" meta-type="string" id-type="int"
        cascade="save-update">
        <meta-value value="BOOK" class="hibernate.example7.Book" />
        <meta-value value="COMPUTER"
            class="hibernate.example7.Computer" />
        <column name="PRODUCT_TYPE" />
        <column name="ITEM_ID" />
    </any>
</class>
</hibernate-mapping>

```

Notice the class mappings for Book and Computer. They simply map their respective properties to columns in tables like we did in the very first example. The only difference is that we need to add **polymorphism="true"** attribute to tell hibernate that their associations are polymorphic in nature. Also note that both the classes uses the inherited productId as the primary key. That's it. We now can write a test class to store and retrieve the objects.

The following code creates a Book object and associates with a ShoppingCart polymorphically.

```

session.beginTransaction();

Book b = new Book();
b.setAuthor("James");
b.setTitle("J2EE");
b.setPublisher("Pub Inc");
b.setSeller("JC Inc");

ShoppingCart cart = new ShoppingCart();
cart.setProduct(b);

session.save(cart);
session.getTransaction().commit();

```

The above code will insert one record in the books table and one in the shoppingcart table. Now, we can use the following code to retrieve the shopping cart which also retrieves the book object.

```
ShoppingCart c1 = (ShoppingCart)session.get(ShoppingCart.class,new Integer(1));
Book b = (Book)c1.getProduct();
```

This approach though works well is very seldom used due to its inherent complexity. Use this technique when you don't require polymorphic association.

Table per Hierarchy

In this approach, we just use one table to map the entire hierarchy that includes Product, Book and Computer classes. This one table contains columns for all the properties defined in all the classes as shown below:

PRODUCTS	
PRODUCT_ID : PK	
SELLER	
PRODUCT_TYPE <dis>	
ISBN	
AUTHOR	
TITLE	
PUBLISHER	
PROCESSOR	
RAM_SIZE	
OS	
TYPE	

As you can see, the table now has all the columns. Following is the challenge we have in our hand with this approach.

- ✓ How does hibernate know which columns are associated with Book class and which columns with Computer class?

The good thing is hibernate gave us the solution by asking us to use **subclass** element along with the **discriminator** column while mapping the class. The discriminator column tells hibernate which columns to populate/read for the sub classes. This is why in the above table, hibernate also created **PRODUCT_TYPE** column to store some unique identifiers that tells which specific object to store or retrieve. The class definition for the product class will be as shown below:

```
<class name="hibernate.example7.Product" table="PRODUCT8"
polymorphism="implicit">
<id name="productId" column="PRODUCT_ID">
    <generator class="increment" />
</id>
<discriminator column="PRODUCT_TYPE" type="string" />
```

```

<subclass name="hibernate.example7.Book" discriminator-value="BOOK">
    <property name="seller" column="SELLER" />
    <property name="author" column="AUTHOR" />
    <property name="title" column="TITLE" />
    <property name="publisher" column="PUBLISHER" />
</subclass>
<subclass name="hibernate.example7.Computer"
        discriminator-value="COMPUTER">
    <property name="seller" column="SELLER" />
    <property name="processor" column="PROCESSOR" />
    <property name="ramSize" column="RAM" />
    <property name="os" column="OS" />
    <property name="type" column="TYPE" />
</subclass>
</class>

```

As you can see from the above definition for the product class, it defined a discriminator column along with two subclass elements, one for book and other for computer class. Also note that the subclass elements specified a discriminator values namely BOOK and COMPUTER for hibernate to distinguish between the two.

Now, the mapping from ShoppingCart to Product can be implemented as a one-to-one association as shown below:

```

<class name="example7.ShoppingCart" table="SHOPPINGCART3">
    <id name="clientId" column="CLIENT_ID">
        <generator class="increment" />
    </id>
    <many-to-one name="product" column="PRODUCT_ID" unique="true"
                 cascade="save-update" />
</class>

```

You might wonder why I used many-to-one association here. Recall that a one-to-one association can be implemented as many-to-one association with **unique="true"** attribute. This is exactly what we did here. Run the previous example with the above class mappings, and you'll see the same result. The only difference is that you'll see just one table created in the database as shown below:

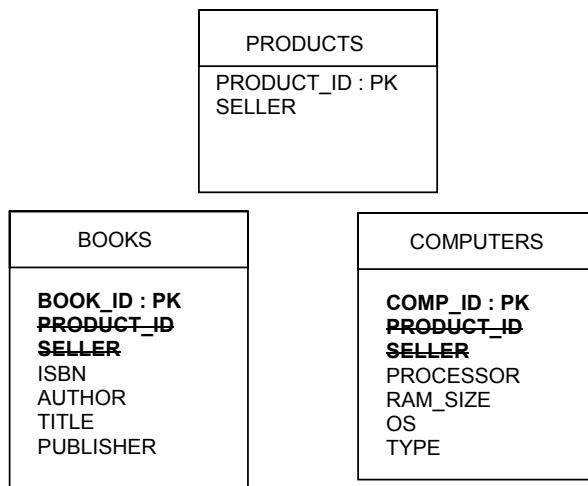
PRODUCT_ID	PRODUCT_TYPE	SELLER	AUTHOR	TITLE	PUBLISHER	PROCESSOR	RAM	OS	Type
1	BOOK	ABC, INC	James	J2EE	Apex				

As you can see from the above, hibernate added BOOK as the discriminator value and populated columns corresponding to Book. Try adding a computer to the shopping cart and you'll see the other columns filled with book columns with blank values.

This is the simplest type of polymorphic association and also easy to understand without any confusions. Moreover, this is the most efficient way of persisting objects. So, keep this approach in mind. Most real word applications use this one.

Table per Sub Class

This approach is same as table per concrete class except that hibernate will not create columns for inherited properties as shown in the following figure.



As you can see from the above, we observe the following two important things:

1. Hibernate will eliminate the columns for the inherited properties in the Books and Computers table, and adds a primary key column.
2. Because of the elimination of the columns in step1, it will create a table for the abstract class Product with columns for inherited properties.

When we persist on of the computer or book class, hibernate will insert the book or computer related data in the books or computers table and the inherited properties from the product class in the products table. So, every time we add a new product, one record will be inserted in one of the specific tables, and one in the products table.

Now that there are three tables each with their own primary keys, hibernate will simply use the primary keys of the sub classes to map with the super class. With this approach,

we need to tell hibernate using the `joined-subclass` element as shown below. The mapping is just the same as we did with `subclass` element.

```
<class name="hibernate.example7.Product" table="PRODUCT9"
    polymorphism="implicit">
    <id name="productId" column="PRODUCT_ID">
        <generator class="increment" />
    </id>
    <property name="seller" column="SELLER" />
    <joined-subclass name="hibernate.example7.Book" table="BOOKS9">
        <key column="BOOK_ID" />
        <property name="seller" column="SELLER" />
        <property name="author" column="AUTHOR" />
        <property name="title" column="TITLE" />
        <property name="publisher" column="PUBLISHER" />
    </joined-subclass>
    <joined-subclass name="hibernate.example7.Computer" table="COMPUTERS9">
        <key column="COMP_ID" />
        <property name="seller" column="SELLER" />
        <property name="processor" column="PROCESSOR" />
        <property name="ramSize" column="RAM" />
        <property name="os" column="OS" />
        <property name="type" column="TYPE" />
    </joined-subclass>
</class>
```

As you can see from the above class mapping for `Product`, it defined its own properties like `productId` and `seller` and then defined the mappings for its subclasses using the `joined-subclass` element. The only difference you'd notice with this element compared to the `subclass` element in the previous example is that we specified the table names highlighted in bold and defined the key column in both the subclass tables.

With the above mapping for product class, run the previous test class and you'll see the same results. One of the main concerns with this approach is the performance. Behind the scenes hibernate will use complex queries that involve outer joins which affect the performance. Use this approach only when the subclasses have fewer properties defined.

This completes all the polymorphic associations that we need to know. Just remember the table per class hierarchy which the simplest of all and you are good.

In the next and the last section, we'll see something about Hibernate Query Language (HQL) and see how we can efficiently retrieve the persisted objects.

Hibernate Query Language (HQL)

HQL is a query language used by Hibernate to query the persisted objects in the tables. Until now we've been using SQL to query the tables and then building the Java objects

from the results. This is what we did using JDBC. With HQL, you don't have to worry about writing complex SQL queries which are more error prone and instead use simple user friendly queries to directly fetch the desired objects from the tables. Hibernate will translate all the HQL queries into SQL queries behind the scenes and fetch the objects. This will really simplify the life of the developer as he/she no longer need to build the complex SQL queries.

Hibernate supports two important classes namely `Query` and `Criteria` to query for objects. One thing you need to understand is that we can use either `Query` or `Criteria` class for retrieving objects. We don't have to use both of them. In some cases using `Query` is more flexible and in some other cases using `Criteria` is more flexible. Let's see some cases using both `Query` and `Criteria` classes.

Case 1: To retrieve all the Book objects from the BOOKS table.

Using Query:

```
Query query = session.createQuery("from Book");
List records = query.list();
```

In this case, we first need to create the `Query` object from the session by specifying the query string. When we say "from Book", hibernate will automatically figure out the table name based on the mappings we defined in the xml and returns all the Book object populating all the properties with the column data. Behind the scenes, hibernate will translate the above into the following SQL:

```
Select bookId, author, publisher from BOOKS;
```

Using Criteria

```
Criteria c = session.createCriteria(Book.class);
List records = c.list();
```

In this case, we just have to specify the class of the objects we want to retrieve. Hibernate will return all the Book objects from the BOOKS table.

Case 2: Binding parameters at runtime.

Binding parameters at runtime allows us to execute dynamic SQL statements, just like we used to do with `PreparedStatement` in JDBC. Let's say we want to retrieve all the Books authored by James.

Using Query

```
String qs= "from Book b where b.author like :author";
Query q = session.createQuery(qs);
q.setString("author","James");
List results = q.list();
```

In the above query, “:author” is the query parameter. With HQL, all the parameters must be prefixed with a colon as shown. Based on the type of data to be passed to the parameter, we can use the set methods defined by the `Query` interface. One such method is the `setString()` method to set the character data. Likewise, there are several other set methods for passing dates, numbers etc,. The above query retrieves all the Book objects that are authored by James.

We can also pass multiple parameters to the query as shown below:

```
String qs = "from Book b where b.author=:author and b.publisher=:publisher";
List results = session.createQuery(qs).
    setString("author","James").
    setString("publisher","Premium Publishing").
list();
```

HQL also permits using indices for parameters instead of names as shown below:

```
String qs= "from Book b where b.author like ?";
Query q = session.createQuery(qs);
q.setString(0,"James");
List results = q.list();

String qs = "from Book b where b.author=? and b.publisher=?";
List results = session.createQuery(qs).
    setString(0,"James").
    setString(1,"Premium Publishing").
list();
```

These are all the basic HQL operations you need to know. Using both `Query` and `Criteria` interfaces you can build any complex queries for retrieving the objects.

This completes all the concepts you need to know to start building applications using Hibernate. Try reading this chapter twice, since some of the concepts like associations might be confusing at the first place. Let's now summarize all we learned in this chapter.

Summary

- ✓ Hibernate is an open source framework for mapping Java objects to tables in relational database.

- ✓ Hibernate is an alternate solution for EJB entity beans and eliminates the need to a container.
- ✓ Hibernate uses XML to define the mappings for various objects and the underlying table columns.
- ✓ SessionFactory is the class that hibernate uses to persist objects to databases.
- ✓ SessionFactory can be configured using XML, Property files and even programmatically.
- ✓ The XML mapping document defines mappings for class properties to columns in tables.
- ✓ Hibernate supports one-to-one, one-to-many, many-to-one and many-to-many associations using special elements in the mapping file.
- ✓ Hibernate also supports polymorphic associations in three different ways.
- ✓ Table per concrete class, Table per Hierarchy and Table per sub class are the three different ways to implement polymorphic associations.
- ✓ Hibernate comes with a query language called HQL for retrieving objects from the tables.
- ✓ The two important components used for defining HQL queries are the Query and Criteria classes.

Time to Play 50-50

1. Which of the following class is used for persisting objects to database?
 - a) SessionFactory
 - b) BeanFactory
2. Which of the following class is used for configuring hibernate?
 - a) Configurer
 - b) Configuration
3. A one-to-one association can be realized from which of the following associations?
 - a) Many-to-Many
 - b) One-to-Many
4. Which of the following method in the session object is used to persist the objects?
 - a) persist()
 - b) save()

Interview Questions

Question: What is the name of the XML file used to configure hibernate?

Answer: hibernate.cfg.xml

Question: What are the different ways of configuring hibernate?

Answer: Using XML file, Properties file and Programmatically.

Question: List the different types of associations?

Answer: One-to-One, One-to-Many, Many-to-One and Many-to-Many.

Question: What is the name of the attribute to use to realize a one-to-one association from a many-to-one association?

Answer: unique=true

Question: How should the Java bean property be represented for a one-to-many association?

Answer: Collection

Question: What are polymorphic associations?

Answer: Associations that map the child class objects referenced by parent class references in the inheritance hierarchy to tables in the database are called as polymorphic associations.

Question: List the different approaches for polymorphic associations.

Answer: Table per concrete class, Table per hierarchy and Table per sub class.

Question: Explain Table per concrete approach.

Answer: In this approach hibernate creates one table non-abstract child class with columns for inherited properties as well as child class properties.

Question: Explain Table per hierarchy approach?

Answer: In this approach hibernate creates just one table for the entire hierarchy with columns for each and every property in all the classes. Hibernate uses discriminator column to identify the child class properties.

This completed all you need to know in Hibernate.

Chapter 20

J2EE Design Patterns

This chapter explains you the important J2EE design patterns that are normally used while designing enterprise applications. Knowing these design patterns help you to design applications that perform well.

Chapter Goals

- ✓ Understand the importance of design patterns
- ✓ Types of J2EE design patterns
- ✓ Creational patterns
- ✓ Behavior patterns
- ✓ Presentation tier patterns
- ✓ Business tier patterns
- ✓ Integration tier patterns

Introduction

The success of any application can undoubtedly be attributed to its design. A sound design leads to a resounding application. When we say “success”, what kind of success are we talking about? Are we talking about success in terms money it generated or something else? You guessed it right. The success we are talking is the success in terms of its ability to serve the customers in a better way, such as the ability to respond fast, the ability to be easily scalable, the flexibility it offers and all that good stuff. This level of success is only achieved by using a good design to build the application. A good design makes the application more stable and makes the application more profitable and popular.

One important thing we must all know is the fact that we cannot see design flaws in an application until we get the application up and running. What I mean to say is that we can build an application at the first place based on some preliminary design which should not be assumed to meet all the requirements in terms of performance, scalability, flexibility, security and all that stuff under all conditions. For instance, the performance of the application may be very good during the initial stages which could probably be attributed to lesser load on the application. As the application gains more popularity the load on the application would certainly increase by many times, and its performance may degrade. This is normal behavior and we should accept it. The reason for this is the fact that it is impossible to come up with perfect solution at the first place itself that meets all the requirements. It is time that determines whether an application is properly designed or not. This doesn't mean we have the luxury to poorly design the application. A good design should anticipate as many possible problems as possibly can upfront and incorporate solutions to address them. One thing we as designers should strive for is to come up with a best solution rather than an ideal solution. This is where the notion of design patterns come into picture.

What is a design Pattern?

In simple words, a design pattern *documents* a proven solution to a *recurring* problem in a particular *context*.

Though the definition looks very simple, there is lot to understand. Consider for instance, we built an application incorporating a solution that addresses a performance issue at the database level. Tomorrow when we build another similar application which throws the same performance problem at the database, we don't have to scratch our minds to reinvent the solution to the same problem, right? Put in simple words, we don't want to reinvent the wheel when we already have one. This is possible only if we

“document” the solution at the first place.

Now, we need to understand why we used the word “context”. If we take an n-tier application, we will have several tiers like presentation tier, business tier, integration tier etc. Let’s say we are having a performance problem in the presentation tier and we only have a solution to a performance issue in business tier. Can we apply the same solution in the presentation tier? Good guess. No we can’t. This is simply because the “context” in which the problem occurred is different. This leads to the following conclusion:

Every solution to a problem has a boundary within which it works as desired. The moment the pattern or the solution crosses the boundary it becomes an, “anti-pattern”.

Therefore a design pattern,

- ✓ Documents a proven solution.
- ✓ Is associated with a context or environment.
- ✓ Has a boundary that should never be crossed.

Now that we know what a design pattern is, defining what a J2EE design pattern is simple.

A J2EE design pattern documents a proven solution to common problems encountered in J2EE based enterprise applications. This is why we need to understand the important J2EE design patterns. Let me tell you one thing here. There are tons of J2EE design patterns and this chapter will only cover the most important and widely used ones in real world applications.

Types of J2EE Patterns

All the J2EE design patterns are categorized into several groups as listed below:

1. Creational Patterns
2. Behavior Patterns
3. Structural Patterns
4. Presentation tier Patterns
5. Business tier Patterns
6. Integration tier patterns

In this chapter, we will see the following design patterns from each category

Creational Patterns	Factory, Singleton
Structural Patterns	Decorator, Façade
Presentation Tier	Front Controller, Business Delegate
Business Tier	Transfer Object, Session Facade
Integration Tier	Data Access Object (DAO)

Creational Patterns

As the name suggests, patterns belonging to this category are used for creating objects. You might wonder why we need patterns for creating objects when we already have the keyword `new` for creating objects. The reason is simple. With `new` keyword we can know for sure what type of object we are creating. For instance, following statement clearly tells that we are creating a `Box` object.

```
Box b = new Box();
```

The problem with the above statement is that our application can only work with `Box` objects but not with any other objects, right? Often in applications we need to create objects whose type can only be determined at runtime. In such cases we need to abstract the object creation details from certain application components. For instance, say we have an application that sells different products to customers. Now, a product can be a Book, or a DVD or something else. Therefore, based on the product the customer selects, some times we need to create book objects, and some times DVD objects or even both. This type of creating objects is more complicated than creating objects by simply using `new` keyword. This is where two important creational patterns listed below help us.

- ✓ Factory Pattern
- ✓ Singleton Pattern

Factory Pattern

A Factory Pattern can be thought as a factory that creates objects. In reality, a factory that can create different types of objects is more useful than the one that can only create one type of objects. For instance, a product factory can create different objects like computers, DVD players, Television sets etc,. At the same time, a factory like `SAXParserfactory` can only create `SAXParser` objects. In most of the enterprise applications, the actual object that needs to be created will only be known at runtime. In such situations using a factory pattern that abstracts the object creation details is more useful. Following figure shows the factory pattern

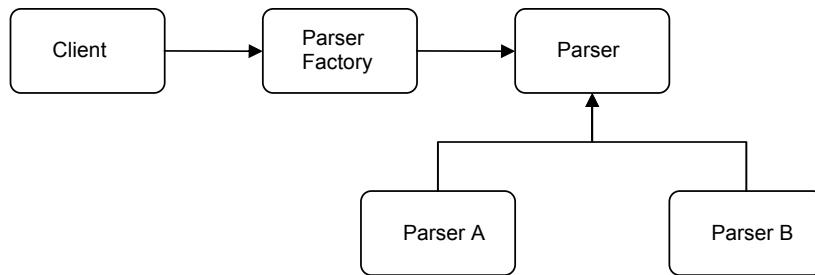


Fig 20.1 Class Diagram

As seen from the above figure, client uses the ParserFactory class to create objects of Parser A and B. Let's write the example that implements the above scenario.

Listing 20.1 Factory Pattern

Parser.java

```

abstract class Parser{
    abstract void parse(String fileName);
}
  
```

IBMPARSER.java

```

public class IBMPARSER extends Parser{
    public void parse(String filename){
        System.out.println(" Using IBM Parser to parse " + fileName);
    }
}
  
```

BEAPARSER.java

```

public class BEAPARSER extends Parser{
    public void parse(String filename){
        System.out.println(" Using BEA Parser to parse " + fileName);
    }
}
  
```

ParserFactory.java

```

public class ParserFactory{
    public Parser createParser ( String parserName){

        if ("IBM".equals(parserName))
            return new IBMPARSER();
        else if ("BEA".equals(parserName))
            return new BEAPARSER();

        return null;
    }
}
  
```

```
}

ParserClient.java

public class ParserClient{
    public static void main(String args[]){
        ParserFactory factory = new ParserFactory();

        Parser parser =factory.createParser("IBM");
        parser.parse();

        parser = factory.createParser("BEA");
        parser.parse();
    }
}
```

Take a look at the `ParserFactory` class. It either returns an `IBMParser` or a `BEAParser` based on which ever is requested to the client. The advantage of using factory pattern is the flexibility it offers in terms of creating objects. Tomorrow if we want to use an updated parser from IBM, we can simply plug it in without affecting the client code.

Singleton Pattern

This is the second type of creational pattern. This pattern ensures that only one object of a class is created per JVM process. For instance, if we want to use the same `Logger` object to log all the entries, we can implement the logger as a singleton. Following code demonstrates this pattern.

```
public class Logger{

    private static FileLogger _newInstance = null;

    public static FileLogger getLogger(){

        if( _newInstance == null){
            _newInstance = new FileLogger();
        }
        return _newInstance;
    }
}
```

As you can see from the above code, we used a conditional statement to always return the same object. Another important thing is that we need to declare the object as `static` so that JVM allocates memory for one object only.

Both Factory and Singleton patterns are generic patterns and can be implemented in any

tier within an n-tier application.

Structural Patterns

Structural Patterns are typically used to build or represent larger objects from smaller objects. This pattern is visible in our day to day life. For instance, a Television object is built from smaller objects like Monitor, Amplifiers, Capacitors etc. So, a television can be called as composite object. There are two important patterns in this category namely *Decorator* and *Façade*. Let's see the details one by one.

Decorator

As the name suggests, this pattern decorates the object. So what does decoration mean? Decoration is nothing but adding or modifying the properties of object. For instance, let's say our application gets some messages from a mainframe system which requires to be formatted before we can display them to the user. This may include adding some header and footer notes etc. What we are doing here is that we are kind of "decorating" the object by giving some additional features. Following figure shows the decorator pattern.

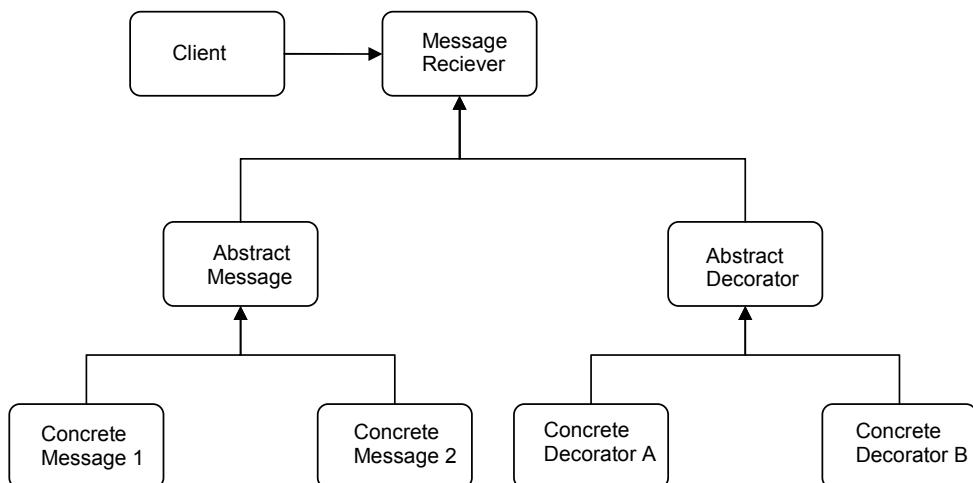


Fig 20.2 Decorator Pattern

If you see from the above figure the `MessageReciever` uses the `Message` and `Decorator` classes to decorate the message. The `Decorator` object takes a `message` object and returns a decorated message. Following code demonstrates the implementation of this pattern.

Listing 20.2 Decorator Pattern

TestClient.java

```
public class TestClient{  
    public static void main(String args[]){  
        MessageReciever receiver = new MessageReciever();  
  
        Message m1 = receiver.getMainFrameMessage();  
        Message m2 = receiver.getDatabaseMessage();  
  
        System.out.println( m1.getMessage());  
        System.out.println( m2.getMessage());  
    }  
}
```

MessageReciever.java

```
public class MessageReciever{  
    public Message getMainFrameMessage(){  
        Decorator d = new MainFrameDecorator();  
        Message m = new MainFrameMessage();  
        m.setMessage("Hello");  
        return d.decorate(m);  
    }  
  
    public Message getDatabaseMessage(){  
        Decorator d = new DatabaseDecorator();  
        Message m = new DatabaseMessage();  
        m.setMessage("Hello");  
        return d.decorate(m);  
    }  
}
```

Message.java

```
public abstract class Message{  
    public getMessage();  
    public setMessage();  
}
```

MainframeMessage.java

```
public class MainFrameMessage extends Message{  
  
    String msg;  
    public String getMessage(){  
        return msg;  
    }  
    public void setMessage(String msg){  
        this.msg = msg;  
    }  
}
```

```
    }  
}
```

DatabaseMessage.java

```
public class DatabaseMessage implements Message{  
  
    String msg;  
    public String getMessage(){  
        return msg;  
    }  
    public void setMessage(String msg){  
        this.msg = msg;  
    }  
}
```

Decorator.java

```
public abstract class Decorator{  
    public Message decorate(Message m);  
}
```

MainframeDecorator.java

```
public class MainFrameDecorator extends Decorator{  
  
    public Message decorate(Message m){  
  
        String str = m.getMessage();  
        m.setMessage("mf://" + str);  
  
        return m;  
    }  
}
```

DatabaseDecorator.java

```
public class DatabaseDecorator extends Decorator{  
  
    public Message decorate(Message m){  
  
        String str = m.getMessage();  
        m.setMessage("db://" + str);  
  
        return m;  
    }  
}
```

Look at the code in the `MessageReciever` class. The methods in this class create the respective messages and decorators and use the decorator to decorate the messages. Run the test class and you'll see the decorated messages.

Façade Pattern

This pattern is used to hide all the underlying complexities of a system by providing a global interface for client interaction. This interface provides a single point of entry for all the client interactions with the underlying subsystems. Following are some of the advantages of this pattern:

- ✓ Client code doesn't have to worry about understanding the nuances of underlying complex systems. The client should only know how to talk with just one global interface.
- ✓ It is easy to add and remove the underlying systems without impacting the client application.
- ✓ Decreases the network traffic and improves the application performance.

Following figure shows various components in this pattern.

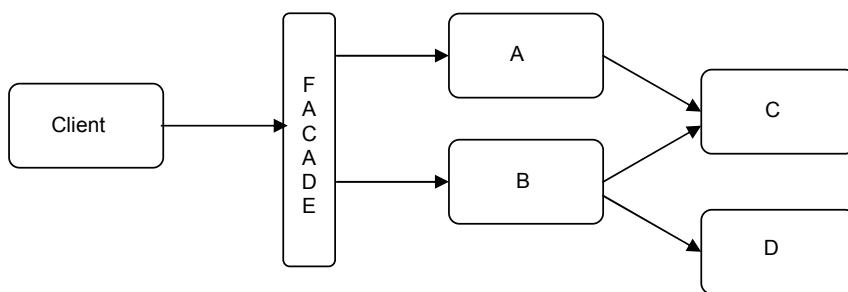


Fig 20.3 Façade Pattern

Following code demonstrates the implementation of this design pattern.

```
public class TestClient{
    public static void main(String args[]){
        AccountServiceFacade f = new AccountServiceFacade();

        f.transferFunds();
        f.applyLoan();
        f.getTransactionHistory();
        f.getAccountOffers();

    }
}
```

As you can see from the above code, client code invokes all the services through the façade class. This class internally connects with the appropriate systems and processes the requests. Now, let's look at the implementation of `AccountServiceFacade`.

```
public class AccountServiceFacade{  
    public void transferFunds () {  
        // Connects to System A  
        systemA.transferFunds ();  
    }  
    public void applyLoan () {  
        // Connects to System B  
        systemB.applyLoan ();  
    }  
    public void getTransactionHistory () {  
        // Connects to System C  
        systemC.getTransactionHistory ();  
    }  
    public void getAccountOffers () {  
        // Connects to System D  
        systemD.getAccountOffers ();  
    }  
}
```

The above code is just a skeleton code. In reality, this class will use several utilities like JNDI, RMI etc to connect with underlying systems. These details are completely hidden from the client application.

Presentation Tier Patterns

These patterns address the common problems that occur in presentation tier. In a typical enterprise internet application, the presentation tier comprises of all the web related components like JSP pages, servlets, struts actions etc,. There are three important design patterns that can be used to solve various problems pertaining to this tier. These are,

- ✓ Front Controller
- ✓ Business Delegate
- ✓ Model-View-Controller

Front Controller

A *Front Controller* pattern as the name suggests stands in front of all the presentation tier components and controls the application flow within the presentation tier. This involves, intercepting the requests, dispatching the requests to request handlers, co-ordinating the actions of various components and finally returning the response to the client. The front controller is like a façade in the presentation tier. The best example of this design pattern is the controller servlet in Struts framework. If you recall from Struts, following is what the controller servlet is responsible for:

- ✓ It's a centralized component that reads the requests from the browser
- ✓ It acts as a single point of entry and exit to/from the application
- ✓ It manages all the interactions in within the application, by coordinating with other components
- ✓ Processes the requests and prepares the models
- ✓ Selects the View to display the models

Following figure shows class and sequence diagrams for the front controller pattern.

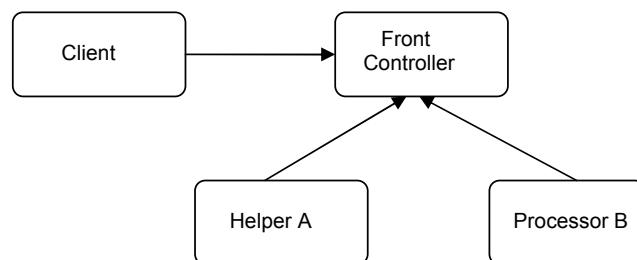


Fig 20.3a Class Diagram

As you can see from the above class diagram, the controller class uses various other classes such as delegators, helpers, views to process the client requests.

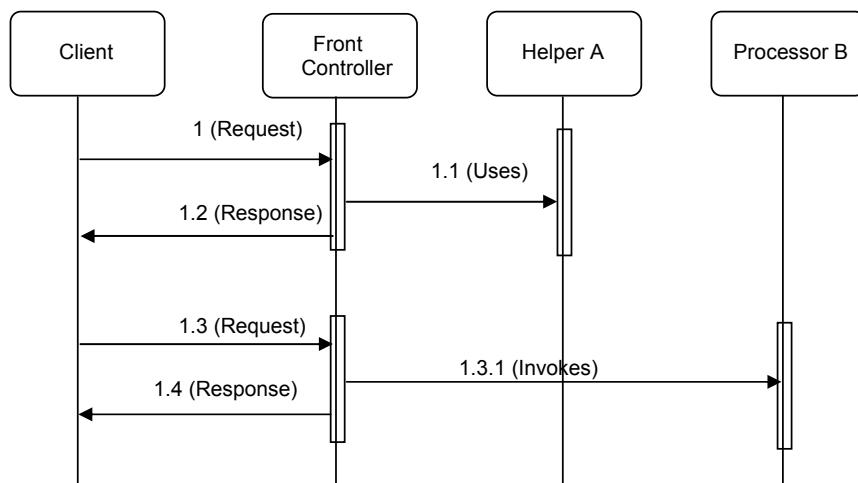


Fig 20.3b Sequence Diagram

Use the front controller pattern when you need a centralized control in the presentation tier.

Business Delegate

This design pattern delegates all the presentation tier calls to business tier components. A business delegate is normally the last component among the presentation tier components after which the request will be delegated to the business tier.

A business delegate typically provides abstraction of business services by hiding them from the presentation tier components and also lends its hand in promoting loose coupling of presentation tier components from business tier services. By using this design pattern, the presentation components like action classes, JSP etc need not worry about nuances of the business services and can concentrate more on the presentation of information to the user.

Following shows the class and sequence diagrams for business delegate pattern.

Class Diagram

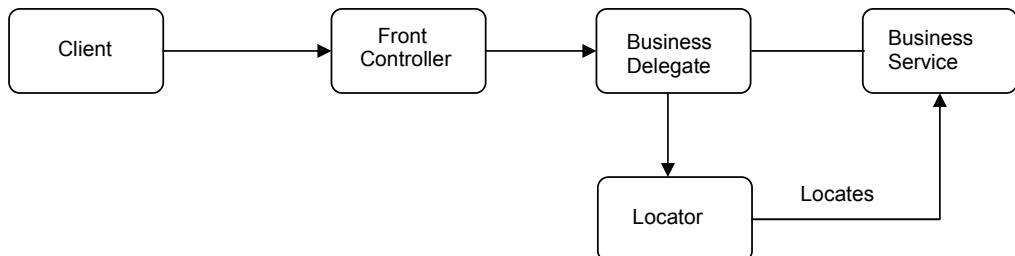


Fig 20.4a Class Diagram

As you can see from the above class diagram, the request from the client is delegated to the business delegate component which uses a lookup tool to search and retrieve the business service. Once the service is retrieved, it uses the service to invoke the business operations.

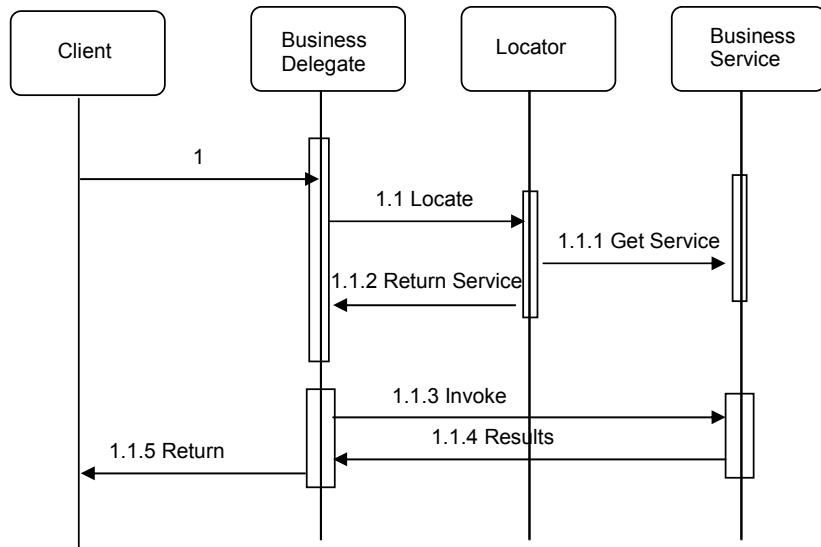


Fig 20.4b Sequence Diagram

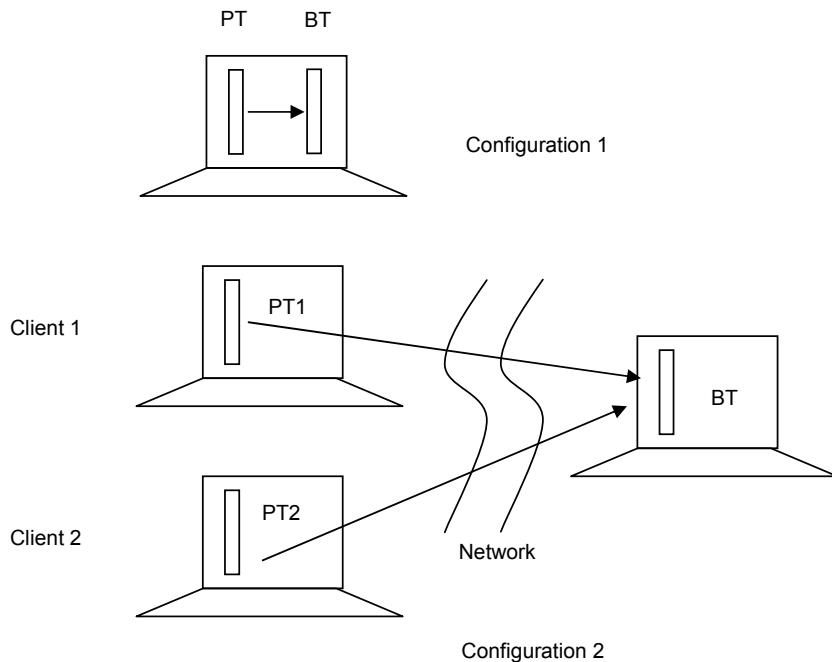
Model View Controller Design

Please refer to the Model View Controller design pattern explained in detail in Struts chapter.

This completes the most important and widely used presentation tier design patterns. Let's now look at the Business Tier design patterns.

Business Tier Patterns

These design patterns provide solutions to problems that arise in the business tier due to business invocations from the presentation tier. Based on the size of the applications and the nature of the clients, we can envision the following two configurations between the presentation tier and business tier.



If you look at configuration 1, we have both the presentation tier and business tier on the client's machine. This is a typical 2-tier architecture which is not suitable for enterprise internet applications. However, this configuration gives better performance since there is no physical network in between the two tiers.

Now look at configuration 2. This is a typical n-tier architecture where the presentation and business tiers are distributed across various systems. This architecture supports diverse application clients which have their own presentation tiers (PT1 and PT2) and the business logic that drives all the clients will still be the same. It is because of this reason, the business tier will be on a separate centralized system which we also call it as "Middleware" that will be accessed by all the presentation tiers. Here, the important thing we need to observe is the physical network involved in between the presentation tiers and business tier. This configuration poses serious performance problems due to heavy network traffic. This is the main problem with the business tier and we need to look at the design patterns that address this problem and improve the performance of the application by reducing the network traffic.

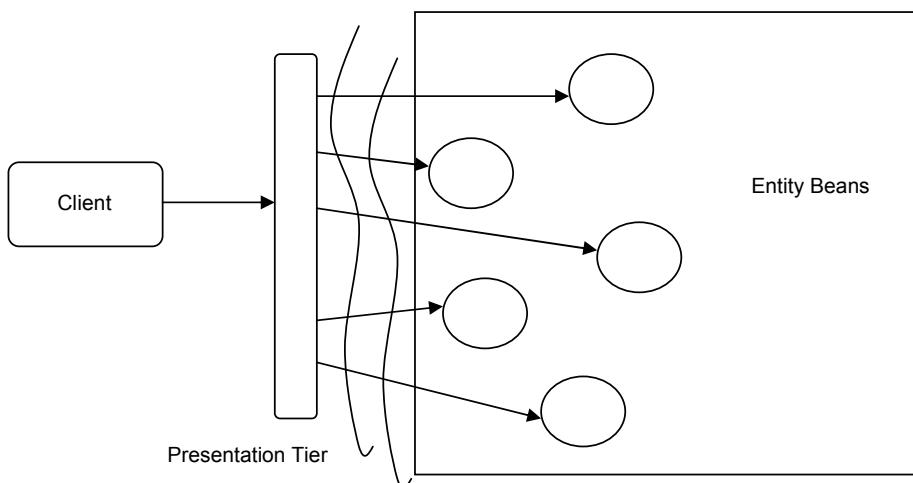
There are two design patterns that come to our rescue. One is *Session Façade* and the

other is the *Transfer Object* or *Value Object* design pattern. Let's look at them in detail.

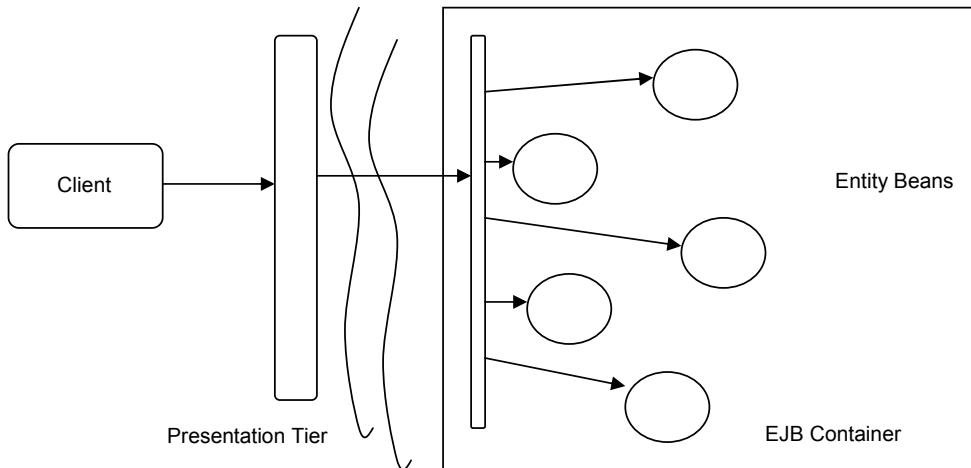
Session Façade

In many J2EE applications we use enterprise Java beans to encapsulate business and persistence logic. Enterprise Java Bean a.k.a EJB technology comes with two important components namely session beans and entity beans. Session beans are used to encapsulate business logic and entity beans are used to encapsulate persistence logic.

Accessing EJB's (especially entity beans) involve several remote calls which leads to increased network traffic. This is because the presentation tier components attempt to read and store application data in the database. Without a good design pattern, the system will look as shown in the following figure:



As you can see from the above figure, the presentation tier makes calls to every entity bean in the EJB container. Assuming that a business transaction requires using all the five entity beans, it should send five separate requests to get hold of five entity beans, right? This gets manifested to $2*5 = 10$ traversals (to and fro) across the network leading to serious performance problems. Strictly speaking the situation will be even worse since accessing a single entity bean takes at least two calls. With this factor plugged into the equation it turns out to be 20 traversals across the network. Now look at the following figure:



In the above figure we introduced a Session bean to act as a gateway for interactions with the entity beans. From the presentation tier point of view, it should only access the one session bean which in turn interacts with all the entity beans. Two important things to be noted here:

1. The presentation tier no longer has to worry about calling the entity beans, which reduces $5 * 2 = 10$ traversals. This cuts down the network traffic by 50%. Good. Client application should only access the session bean which adds $2 * 1 = 2$ traversals. The total traversals will now be $10 + 2 = 12$ which is still better than without a session bean.
2. There is no network between the session bean and the entity beans which will also add up to improving the performance.

Based on the above facts, we can well accept the above design. The Session bean we are taking about is the solution which we call it as *Session Façade*. So, a session façade can be thought of as a single point of entry into the business tier. Now let's look at the class and sequence diagrams. See next page.

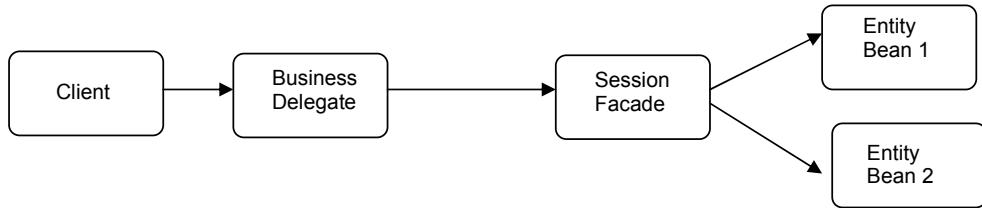


Fig 20.5a Class Diagram

As you can see from the above figure, the business delegate is the last component in the presentation tier and the Session Façade is the first component in the business tier. The business delegate makes all the calls to the façade which uses the behind the scenes components to process the requests. These components however can be anything, doesn't necessarily have to be just entity beans.

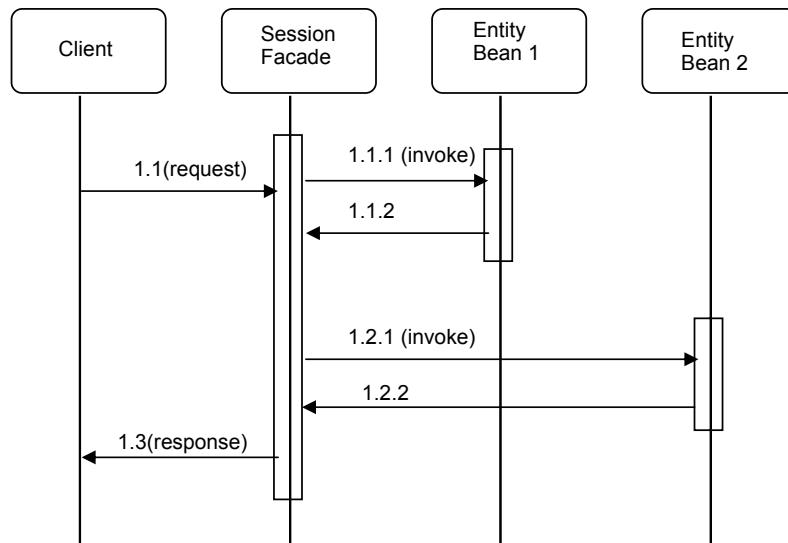


Fig 20.5b Sequence Diagram

The above sequence diagram shows the sequence of events from the point the client making a business request until the client getting response. This is one of the widely used design pattern and also a candidate during interviews. So, keep this in mind.

Transfer Object

When we talked about session façade we talked a lot about network traffic. So, what kind of traffic are we talking about? Is it just the traffic for making the calls to business objects? No. We are talking about two kinds of traffic. One is the traffic due to locating the business objects, and the other is the traffic due to data exchange between the presentation tier components and business tier components. The second traffic is the most significant one which amounts to almost 80% of the overall network traffic. Minimizing this traffic is the key to improving the performance of the application. This is where the *Transfer Object* pattern comes into picture.

The presentation tier components instead of sending pieces of data one by one across the network, it can use a transfer object to encapsulate all the data and then send it once to the business tier. The same applies to business tier components when it comes to returning the response back to the presentation tier components. It uses the transfer object pattern to encapsulate all the response data and send it back to the presentation tier.

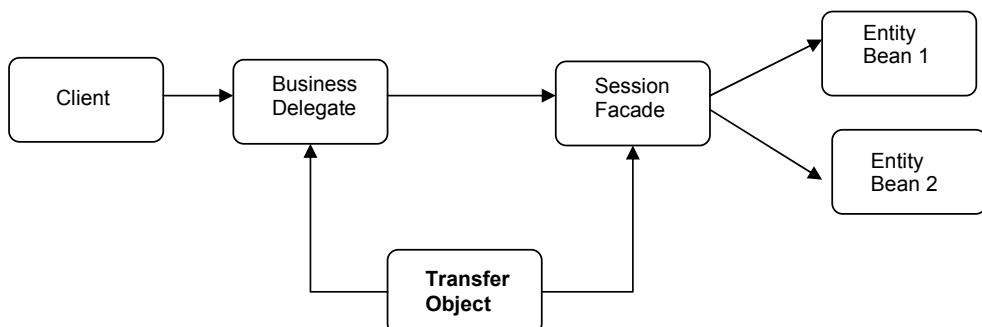


Fig 20.6a Class diagram

As you can see from the above class diagram, both the business delegate and the session façade will use the transfer object to send data across the network. Usually a transfer object can be implemented as a simple Java Bean or a POJO.

Features

- ✓ Reduces network traffic
- ✓ Improves performance

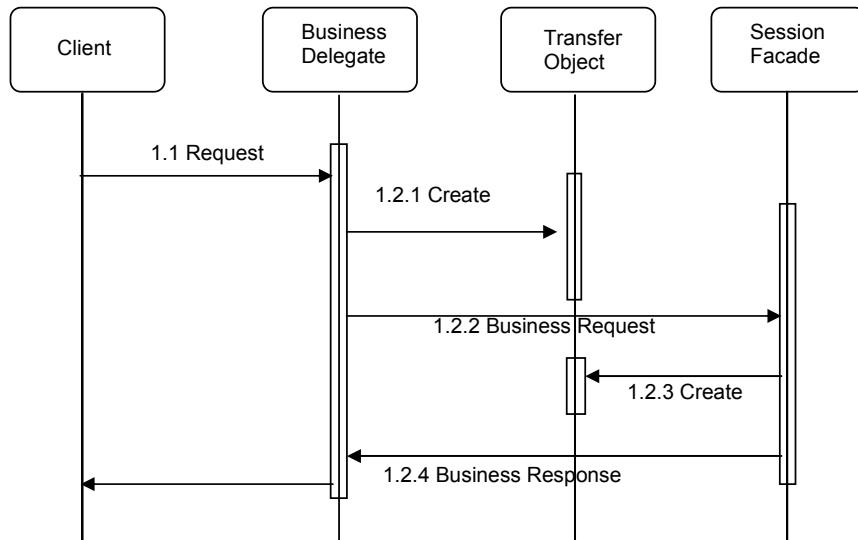


Fig 20.6b Sequence Diagram

Integration Tier patterns

These patterns are used to integrate the application business components with the underlying persistence repositories. In typical enterprise applications, we often use relational databases (RDBMS) to persist the data. However, there can also be several other repositories such as mainframe systems, flat files etc. Every persistent system will have its own API for accessing, storing and retrieving data. Therefore making the application logic loosely coupled with the persistent systems is extremely critical. This is achieved by abstracting all the details of persistent system from the application or business components. There is one design pattern that helps us in this situation. It is popularly referred to as *Data Access Object*. Let's see the details of this pattern.

Data Access Object

As the name suggests, this pattern is an implementation for accessing data from the persistent stores. The DAO implementation hides all the underlying data store access details and provides interfaces (methods) to the application components to universally access the underlying data systems. Following figures shows the class and sequence diagrams for this pattern.

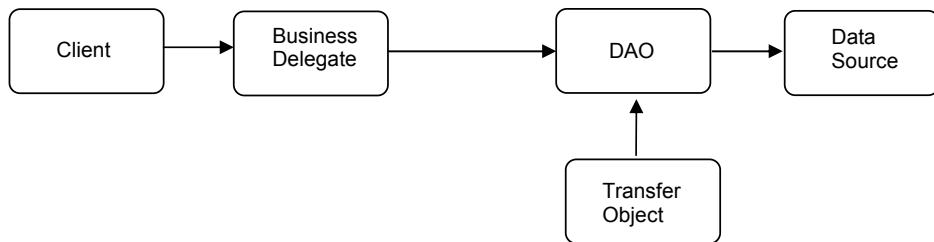


Fig 20.7a Class diagram

As you can see from the above class diagram, business components make calls to the DAO object, which in turn interacts with the persistent systems (data sources) like relational databases. Upon retrieving all the data from the data source, it encapsulates it in a transfer object and returns it back to the client application.

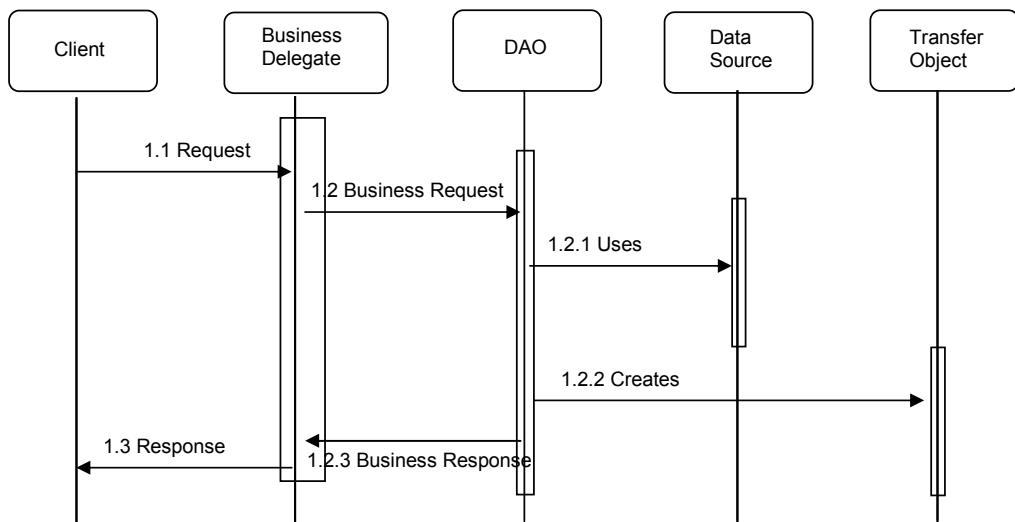


Fig 20.7b Sequence diagram

Following is the implementation of DAO pattern.

```

public class CustomerDAO{

    private JDBCTemplate template;
    private MainframeMessenger messenger;

    public void updateAddress (Address address) {
        // Use the JDBC template to insert the bean to the database
    }
}
  
```

```
template.update( . . . . . );
}

public ProfileVO getCustomerProfile (String ssn) {
    // Connect with mainframe system and get the profile
    String response = mainFrameMessenger.getProfile(ssn);

    // Convert the mainframe response to Transfer Object
    ProfileVO profile = parseMessage( response );

    return profile;
}
}
```

As you can see from the above DAO code, the `updateAddress()` method uses JDBC templates to update the data in the relational database system. At the same time, the `getCustomerProfile()` method uses some mainframe messenger to get the customer profile and return a profile object. From the business component point of view, it just have to invoke the DAO method by passing the SSN and take the response object returned by it. It need not worry about what mechanism is used inside the DAO method to retrieve the customer profile. This way the business component is abstracted from all the underlying persistent systems.

This completes all the important J2EE design patterns that you need to know. Let's summarize all the patterns we learned until now.

Summary

- ✓ A design pattern documents a proven solution to a recurring problem in a particular context.
- ✓ Design Patterns are categorized as Creational, Structural, Behavioral patterns.
- ✓ J2EE design Patterns are categorized as Presentation tier, Business tier and Integration tier patterns.
- ✓ Factory and Singleton patterns are used for creating objects.
- ✓ Decorator pattern is used to change the behavior of the object through decoration.
- ✓ Front Controller pattern is used to build centralized components in the presentation tier.
- ✓ Business Delegate pattern is used in the presentation tier to delegate all the requests to the business tier.
- ✓ Session Façade pattern is used in the business tier as a single point of entry into the business tier. This reduces calls between presentation tier and business tier.

- ✓ Transfer Object is used to encapsulate all the data exchanged between presentation tier and business tier. This improves the performance by decreasing the network traffic.
- ✓ DAO pattern is used in the Integration tier to encapsulate the data sources.

Time to play 50-50

1. Which of the following is a creational patterns?
 - a) Façade
 - b) Factory
2. Which of the following pattern ensures creating just one object per JVM?
 - a) Singleton
 - b) Factory
3. Which of the following is a presentation tier pattern?
 - a) Business Delegate
 - b) Session Facade
4. Which of the following pattern is used to build a centralized component in the presentation tier?
 - a) Transfer Object
 - b) Front Controller
5. Which of the following pattern is used in Struts framework?
 - a) MVC
 - b) Factory
6. Which of the following pattern is used to encapsulate data sources?
 - a) MVC
 - b) DAO Pattern

Interview Questions

Question: What is a Factory pattern?

Answer: A factory pattern allows to model an interface for creating objects which at creation time let's the sub classes to decide which exact object to create.

Question: What is a Singleton pattern?

Answer: A singleton design pattern is a creational patterns that ensures the creation of one and only one object per process.

Question: What is a decorator pattern?

Answer: It's a behavioral pattern which allows to modify the behavior of the objects at runtime by decorating the object with additional characteristics.

Question: What is a Front Controller Pattern?

Answer: A Front Controller pattern is a presentation tier pattern which allows building a centralized component that manages all the interactions from the point of request origination to the point of sending the response to the client. It acts like a manager by co-ordinating all the actions within the presentation tier.

Question: What is a Business Delegate pattern?

Answer: A Business Delegate pattern is implemented in the presentation tier to delegate all the presentation tier requests to the business tier components. It acts as a single point of exit from the presentation tier.

Question: What is a Session Façade pattern?

Answer: A Session Façade is a business tier pattern in which a Session bean is modeled to act a single point of entry into the business tier. This pattern reduces the calls from presentation tier to business tier thereby reducing the network traffic and improving the performance of the application.

Question: What is a Data Access Object?

Answer: Data Access Object is an integration tier pattern which encapsulates all the data sources and persistence logic and providing a thereby making the business components loosely coupled with the underlying repositories

This completes all the basic design patterns that you need to know. You know what, this is also the last technical chapter. Congratulations. You are now a Java and J2EE guru.

Non- Technical

Inside the Company

This chapter is purely non technical and gives you a glimpse of enterprise application development lifecycle. Understanding this chapter is very important and gives you the details about how applications will be developed, how different teams co-ordinate and all that good stuff.

Chapter Goals

- ✓ Understand the Team Structure
- ✓ Understand the Project Life cycle
- ✓ Team responsibilities
- ✓ Version Control Systems
- ✓ Interview Tips and Suggestions.

Introduction

We all know the fact that today's modern enterprises uses e-Commerce applications to run their business with the sole reason of reaching to customers irrespective of geographical location. The advent of internet has completely changed the dimensions of businesses by leaps and bounds. Enterprise software applications not only help the businesses thrive but also help customers to reach for their needs with minimum effort. Many businesses have started taking this advantage of eCommerce applications and started pumping in money for enterprise application development.

There are two key challenges that every business face with eCommerce applications. One is serving the customer base effectively and efficiently and the second one is sustaining immense competition from similar businesses. Out of these two, meeting the first challenge is far more important than the second one. There is no point in competing with other businesses when you can't serve the existing customer base to their satisfaction. Most businesses spend millions of dollars in building sophisticated enterprise applications to capture the market and customers. This is why enterprise application development has become a buzz word in IT industry.

Enterprise applications form the back bone for any enterprise and no wonder why companies continuously invest whole lot of money to make the application stronger, reliable and flexible to retain and add new customers. Any minor hiccup within the enterprise application results in a million dollar loss to a company which could also lead to our favorite phrase, "You are fired". To avoid hearing such phrases, companies take utmost care during enterprise application development.

Enterprise application development is not a one man show. There will be hundreds and even thousands of people working in tandem each contributing to the success of the application. Some people will involve in designing the application, some with development, some other group of people supporting the application day and night and so on.

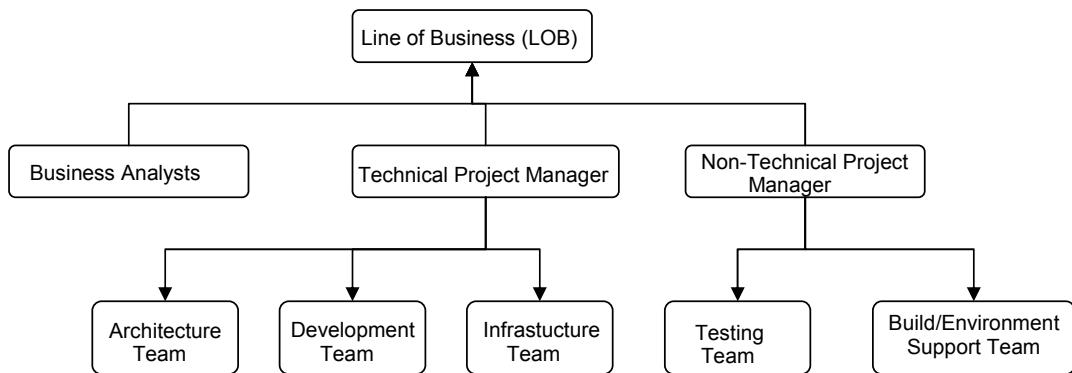
Knowing the importance of enterprise application, every business irrespective of size takes utmost care during the development of application. This involves following standard or proprietary business processes, implementing best practices, using best available tools etc.,

In this chapter we'll see the complete lifecycle of a typical enterprise application development, how the teams will be structured and how different teams co-ordinate

with each other in implementing business requirements and all that good stuff. So, without wasting any further time, let's get into the details.

Enterprise Application Team Structure

Following figure shows a typical team structure involving in an enterprise application development.



As you can see from the above figure, there are several teams involved in the development of an enterprise application. At the top of the hierarchy is the line of business a.k.a LOB folks. These are the people who come up with business ideas. These are also the people who provide funding to the project. This funding can range from several hundred thousand dollars to millions of dollars. These people have strong business ideas, vision who knows the direction in which the business is heading. These people are supported by several intellectuals who do all the research and market analyses of business ideas before the LOB folks approve the funding. The LOB people will then rely on the business analysts, project managers, architects in realizing their business ideas and concepts.

Business Analysts

The duties of business analysts include,

- ✓ Gathering the business requirements from the LOB folks and analyzing the requirements. These people will have the complete knowledge of the business rules and processes.

- ✓ Identify which teams and systems should be involved in realizing the requirements. These folks are also referred to as subject matter experts (SME).
- ✓ Co-ordinate with project managers and team leads to ensure that all the business requirements are properly met as per the needs.
- ✓ Involve in training the team leads with the business processes and standards
- ✓ Set up meetings (JAD Sessions) to bring all the teams on the same page.
- ✓ Clearly outline the responsibilities of each and every team and also discuss the different systems that involve along with the challenges that should be addressed

Technical Project Manager(s)

There can be one or more than one technical project manager based on the size of the application. Technical project managers will have good knowledge of what technologies and development tools to use for implementing the requirements. The main duties of these people are:

- ✓ Managing the resources and time. Should be able to pump in the resources based on the amount of work and the time.
- ✓ Lead different teams like architecture teams, design teams and development teams.
- ✓ Work with the technical infrastructure teams to ensure that proper tools and technologies are used.
- ✓ Constantly monitor the development of application and ensure that all the requirements are met within the time frame.
- ✓ Setup weekly meetings with the development, design and architecture team leads to monitor the progress of work.

Non-Technical Project Manager(s)

These people manage the non technical parts of the application such as testing and building the application. Their duties include:

- ✓ Work with the build teams to build and deploy the application on testing boxes.
- ✓ Co-ordinate the builds with different teams.
- ✓ Ensuring proper testing environment is setup for the various testing teams.
- ✓ Work with various team leads to determine the testing activity.
- ✓ Work with testing teams in getting the application tested in a timely fashion.

Architecture Teams

These are the highly qualified technical folks. These folks should understand the business requirements thoroughly and should be able to drive the development. Their important duties include:

- ✓ Providing the architecture and explaining it to the team members.
- ✓ Suggest the technologies to use to build the application.
- ✓ Should be able to take decisions to increase the development speed.
- ✓ Ensure that all the required features are implemented.
- ✓ Mentor the teams about latest technologies and tools and their usage.
- ✓ Ensure the application meets the performance standards.
- ✓ Making sure that proper standards and best practices are followed.

Development Teams

These are the teams that provide the actual implementation of the business requirements. Every development team will have a team lead who leads the entire team of developers. A team lead takes the business requirements from the business analysts and converts them to functional specifications which are also called as technical designs. These are the low level designs that contain all the coding details like which classes to code, which methods need to be implemented and all that stuff.

Team lead will identify the development tasks and delegates them to the developers. He also ensures the smooth progress of the application development by regularly checking the status of every developer's work through team meetings. These meetings could be once a week or sometimes for every two days based on how critical the deadlines are. Team leads normally maintain a critical path document that contains all the tasks along with the implementation dates. Based on development activities this document will be updated once a week.

Infrastructure Team

This team is responsible for building and supporting the core infrastructure of the application. Folks in this team are like technical leaders who establish the standards and best practices. This team normally builds the core application modules that form the cornerstones of the enterprise application. They also play a key role in doing code reviews with the development teams. Every developer will be required to submit the code to the technical leads who review the code to ensure all the coding standards are strictly followed. Some of these standards include code documentation, proper indentation, Java conventions, proper use of API and all that good stuff.

This team also closely works with architecture teams to constantly improve the architecture and design of the application. Simply saying, this team constantly searches for better ways to implement the business requirements.

Testing Team

This team is responsible for testing the application. As we all know, testing is the key phase of any application development. There is no way a business can launch an application without proper testing no matter how smart the architects and developers are. This team ensures that all the business requirements are met as per the specifications. We'll see testing in detail in the later sections. This team will not give the green signal to the launch of application until all the business requirements are thoroughly tested. So, it's very important that development teams to maintain good harmony and friendship with this team. Don't ever mess with this team. Just kidding.

Build/Environment Support Team

This team is responsible for building and deploying the application on various servers. These teams usually maintain build schedules and accordingly perform the builds. These people work closely with the development teams to ensure that the latest code is deployed on servers for the testing team to test the application. The other responsibility of this team is to maintain the version control systems. In simple words, folks in this team are like application administrators who configure the application based on different environments. For instance, application should point to development databases on development servers, cert databases in UAT or beta environments etc. Sometimes these teams also provide environment support like providing systems to developers, configuring the desktops and all that good stuff.

This is how various teams involve in the application development. Now, lets see the typical lifecycle of enterprise application.

Enterprise Application Lifecycle

The lifecycle of an application begins with capturing an idea and ends with realizing the idea to serve the customers world wide. Between capturing an idea and realizing it, there will be several intermittent steps like proof of the idea, business requirements gathering, business requirement analysis, designing, implementation and what not. Look at Fig 21.1 that demonstrates the complete life cycle.

As you can see from the figure, there are several stages from coming up with a business idea to launching the idea. Let's look at each of these stages in detail.

Framing a Business Idea or Concept

An idea is basically the motivation to start a business. This involves identifying what type of business services you want to provide to the customers, identifying the potential customers who can use your business services, foreseeing the future of business in terms of success, financial growth etc.

Since it is this idea or concept that will drive the business, it's very important that you do case studies, market analysis and all that good stuff. Once the idea is framed, the real fun starts. Let's assume that our business idea for this chapter is to setup a retail financial company.

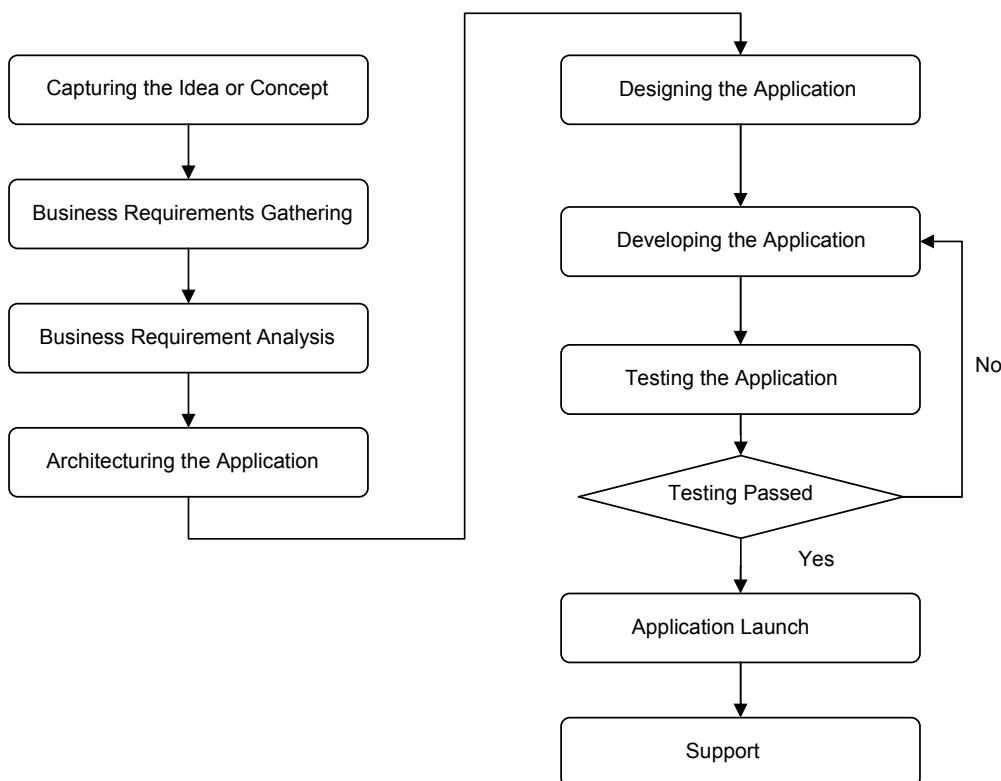


Fig 21.1 Enterprise Application Development Lifecycle

Business Requirements Gathering

This is the second phase in enterprise application development in which business analysts work with LOB people to gather business requirements. These are the high level requirements that the business would like see them implemented. A typical requirement could be to provide instant credit cards offers to customers who open a savings account with \$1000 deposit. Though this requirement looks pretty simple at the first place, it takes several teams to implement coding changes to make it a reality.

Business analyst gathers several of these requirements and prepares a business document with all the details. These details include with teams will be impacted due to the changes, and which systems to use to collect the data and all that good stuff. Every thing is non technical at this point. Business analysts are also required to do the impact analysis like which applications would be impacted due to the changes.

In a typical enterprise application, there can be several smaller applications, though independent of each other from the functionality point of view, they are always tied up with the same underlying systems. For instance, let's say we have two applications with one providing online support for customers and other is a standalone application used in banking centers. If a customer comes and opens an account at the banking center using standalone application, then the online application must also be able to display his information at later time. If impact analysis is not done for a business requirement pertaining to standalone application, then the online customer support application may break. This is the reason why business analysts must be extremely careful while doing the impact analysis. Always remember one thing; breaking one piece of existing functionality is worse than adding five new features.

Business Requirement Analysis

This is the most important phase. In this phase, business analysts and various team leads, architects and designers sit together and analyze the requirements one by one. This involves identifying the responsibilities of every team. Every team lead goes through the business requirement document created by the BA's and tries to identify which things would fall in his plate. Based on the amount of work a particular team should do, team leads come up with an estimation time to fulfill the requirements.

The estimation time is based on the amount of design, development and testing that needs to be done to fulfill the requirement. These meetings would go for several weeks until all the business requirements are finalized. Sometimes based on critical deadlines and resources, certain requirements would be marked as out of scope meaning they cannot be implemented within the given timeframe and will probably be delegated to

later releases. This is however will be decided by the LOB. If LOB needs a particular requirement to be implemented at any cost, then the development teams should burn the midnight oil. No escaping.

Application Architecture

This is the technical part of the application. In order to implement business requirements, application architects need to determine the best technologies and tools to build the application. The application architects should lay all the ground work for the design and development teams. This involves identifying the best technologies, best solutions etc. Application architects should also take the budget into account while deciding which technologies to use. If the budget is small, they should resort to open source technologies that are as reliable as some of the better commercial technologies.

Application architects are the highest technical leaders. They are usually very experienced and have good knowledge technologies and tools. These people also act as decision makers whether or not certain business requirements should be implemented or not. Overall, application architects drive the application development.

Application Design

Once all the technologies and tools are identified by the architects, the designers will get into act. These people are technology specific experts and try do design the application using the best design techniques to simplify the development. The application designers design the core framework of the application like how persistence should be dealt, how messaging with external systems should be dealt and all that good stuff. These people besides designing the application also involve in developing the core components of the application.

The design team also enforces development guidelines such as coding conventions, best practices, API usage etc,. Overall, application designers work closely with the development teams to ensure that application is properly developed.

Application Development

Once the application is designed and the design team established development standards, application development kicks on. In large scale enterprise applications, there will be several people involved during the development of application based on the individual skills. Following are some of the categories of application developers.

Database Developers

These people involve in the development of the persistence components of the application. They are usually specialized in writing complex SQL queries and database stored procedures and all that good stuff. They are also responsible for loading the data in databases and also managing databases. In simple words, these folks besides application development sometimes also don the role of DBA to administer the databases.

Business component developers

These are the folks who develop the critical business components. These people will have excellent core Java skills and utilize the persistence components developed by database developers to build the business components. These developers also form the Middleware team.

Presentation tier developers

These people involve in the development of web components using technologies like Servlet, JSP, HTML etc. These developers are responsible for developing UI components and should come up with good look and feel for the application. This team plays a significant role in the success of the application because the presentation and the flexibility of the application is what bring the customers to the web site. Poorly developed web interfaces though have rich functionality cannot survive longer. No wonder if presentation teams get more credit than the other development teams. Besides application development, every development team is also responsible for unit testing the application components.

Once the developers complete application development and unit testing, there is one important thing that is left to do. They should submit their code to the team leads or technical leads for review. There are two levels of review that is normally done.

1. Level 1 Code review
2. Level 2 Code review

Level 1 code review is normally done by the developers themselves using automated code review tools. These tools will be integrated into the application development software and will report the basic coding violations such as variables usage, keywords usage etc. This code review will make the code look clean and free of basic coding violations. Some of these tools also report violations like method names being longer, method body exceeding the limit etc. These are low priority violations and can be

ignored. The goal of this review to fix as many violations as possibly can and document violations that doesn't make any sense. This completes Level 1 code review. The code should then be submitted to team leads for Level 2 code review.

Level 2 code review is a manual review where the team leads check whether proper coding standards are followed are not, whether are not proper API is used and all that good stuff. Based on the review, team leads come with suggestions to improve the code and convey them to the developers. Developers will then implement the suggestions and resubmit it for review. This is an iterative process which will finally be approved for deployment.

Once the code review process is completed, developers will deliver the code to the build team who will build the application and deploy it on the servers for testing.

Application Testing

This is the critical piece of application development lifecycle. It is this phase that determines whether or not the application should go into production or not. There are three important levels of testing as shown below:

1. Unit Testing
2. IST or Alpha Testing
3. UAT or Beta Testing
4. Performance or Load Testing

Unit Testing

This is the testing that application developers do. This involves writing several test cases to verify whether the actual responses meet with the expected responses. Application developers must write test case for every piece of functionality or simply every method in the class.

One of the popular framework for writing test cases in Java is the JUnit framework. This is a very simple framework in which a test class will be written for every class in the application. For instance, if we have a class named `MessageFormatter`, then we write a test class named `MessageFormatterTest` defining several test cases. A test case is nothing but a method that tests a piece of functionally in the actual class. This involves creating the object of the class, invoking the business methods by passing the valid arguments and verifying the results.

Every test class usually defines several test cases out of which some are positive test cases which test for correct functionality of the class and some are negative test cases that test whether the class responds normally under abnormal situations. One example for negative testing is to pass invalid arguments such as nulls and verify whether proper error messages are returned back or not.

Unit testing is very important task in the application development cycle and team leads ensure that all the developers unit tested their code thoroughly before delivering the code to the server for next level of testing.

IST Testing

IST stands for Integration System Testing. Some people also call this as alpha testing. In typical enterprise applications, there will be several developers working on different pieces of the application. For instance, middleware teams develop the business components; presentation teams develop the UI components etc. Once all the teams deliver their code, it's important to test whether all these individual pieces are properly integrated with each other or not such as the coupling of presentation components with the business components, coupling of business components with the persistence components etc,. This is why we call this as integration system testing.

Usually there will be dedicated IST testing team whose sole responsibility is to test whether the application has met all the business requirements or not. This level of testing can be both automated as well as manual. This phase of testing should ensure the following important things:

1. Make sure that all the business requirements are implemented.
2. Make sure that the existing functionality is not broken.
3. Create and manage defects with the development teams.
4. Ensure that all the defects are fixed.

The IST testing and the application development is iterative in nature. Development teams deliver the code for IST testing. IST testers based on the testing send the defects back to the development teams who fix the defect and reinstall the code for testing. This cycle continues until all the defects are fixed. Finally the IST team will certify the application as meeting all the business requirements.

UAT Testing

UAT stands for User Acceptance Testing. Some people also call it as beta testing. As the name suggests, this testing is done by the business people to ensure that all the

requirements are working as needed. During this phase, business users try to get a feel of the application. This is a pre production and final testing where the business users work with the application and certify it to go to production. This phase usually doesn't involve any serious defects unless they are missed by the IST testing team. Once the application is in UAT, it is just once step away from making it to production.

Performance or Load Testing

This testing is normally done somewhere in between IST and UAT testing. This testing will try to simulate the production environment by subjecting the application with thousands of simultaneous requests to ensure whether or not the application is meeting the performance limits. There will be a dedicated performance testing team who use sophisticated tools for loading the application and analyzing the performance reports.

Once all the above testing is completed, every testing team must certify the application by physically signing the documents. Once all the teams sign of the documents, the application is good to go into production.

Production Support

After the application is launched to production, there will be a dedicated team to support the application during production. Though the application is tested thoroughly, there will always be some hiccups during production while working with real time data. Production support teams identify the issues and opens what we calls as tickets to track the issues. Once a ticket is created for a particular issue, it will be assigned to the appropriate line of business people. This ticket will pass though several hands and finally reaches the team who can solve the issue.

Every ticket will be set with a priority to expedite the fix for the issue. If the priority is set to high, then it means the issue is seriously affecting the business and which ever team is responsible for the issue should immediately leave the current work and fix the issue right away.

Production support is a 24/7 support and every team member will be provided with a dedicated cell phone or pager that he or she should hang around the neck even while sleeping at home. When the pager rings, he/she should identify the issue and bring the required people on board to provide a fix for the issue. This is the reason why most people hate production support jobs. Take it easy.

This completes enterprise application lifecycle. Let's now jump into the last topic of this book, which is the version control. Trust me, this is a simple concept.

Version Control

With any application, the most critical part of it is nothing but the code that we write. Maintaining the code is extremely important for the business to survive. So, what do we mean by the term "maintaining"? Good guess. Maintaining the code means securing the code without getting lost. Historically, we used to back up the code by simply storing multiple copies of the code in storage devices. This storage is usually version specific. So, what we used to do is, every time the code changes, we dump the entire code base into a backup directory. This is a crude way of doing things, since we don't have a dedicated systems for doing the same.

The other challenge while developing the application is code sharing. Usually multiple teams will be involved in application development, where one team of developers need to use the code developed by other teams. In such situations, the simplest way is for one team say A, send a copy of the code through email or some other means to another team B, saying the code is ready to use. What if team A changes the code when team B is still using the old code? This new version of the code should be resent, right?

With the technology advancement, we can now get away from this crude way of backing the code and manual communication of code updates by using the so called version control systems. These systems now take the responsibility of maintaining and versioning the code. Most software applications are now using these version control systems as a de facto standard for versioning the code as they offer several convenient features and flexibility. So, let's take few minutes to understand the basics of version control systems.

What is a Version Control System?

A version control system is a software device that maintains different versions of various software components. A typical version control system involves the following actions.

1. Creating a project
2. Joining the project
3. Checkin the files
4. Checkout the files
5. Synchronizing the files.

Creating a Project

Any version control systems will be first initialized by creating an empty project as shown in Fig 21.2a.

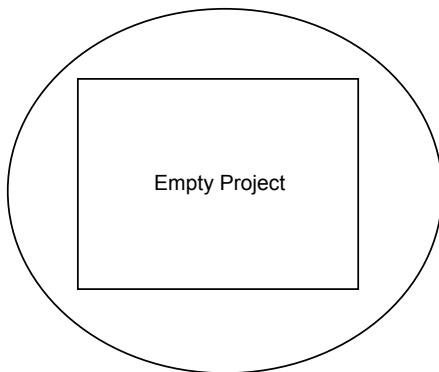


Fig 21.2a Version Control System

Every application will have a dedicated team for administering the version control system. Usually the build team takes this responsibility. When the project begins, this team will create an empty project as shown above in the version control system. You can imagine the version control system as a server program.

Joining the Project

Once the project is created, the build team will notify the development teams that the project is created and ready for development. This is when all the development teams connect to the version control system and join the project. At this point, every developer will have his own local copy of the empty project as shown in Fig 21.2b. (Next Page). Once every developer has his own copy of the project, he/she can start adding files to the project.

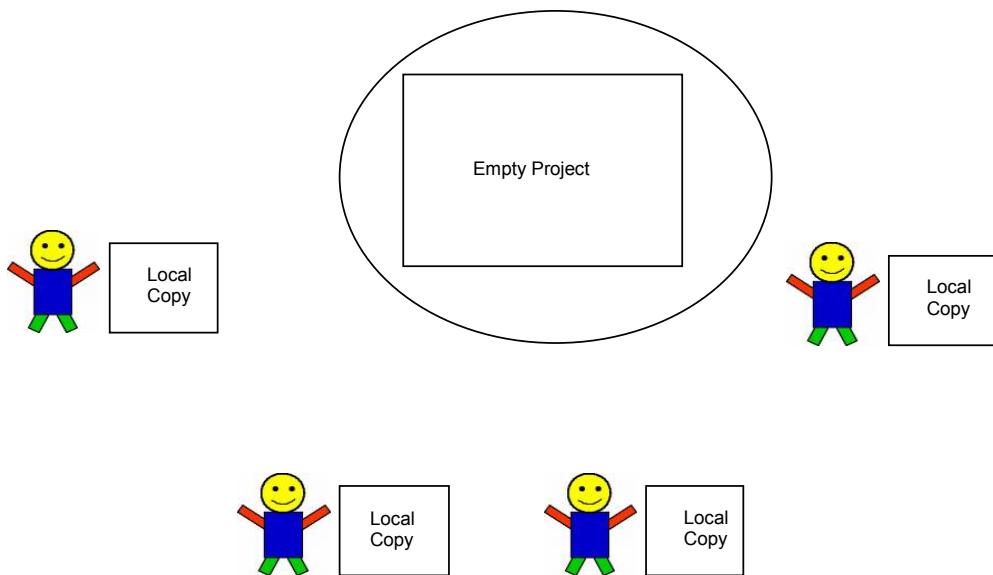


Fig 21.2b Joining the project in version control system

Checkin Operation

This is the most important action. Once all the files are modified by the developer, he/she checks in the file. This means a new version of the file will be created on the version control system. Once the file is checked in, any developer can obtain the latest version by synchronizing the local copy of the project with the server copy. A checked in file will always be in read-only mode.

Checkout Operation

When a developer needs to make any changes to an existing file, he/she must check out the file. This makes the file writable. Upon completing all the changes, the developer will checkin the file again. Usually a checkout operation will be followed by a checkin operation.

Delivering the work

After every developer completes his/her changes to the project, he/she will deliver his work to the server copy of the project in the version control system. The server copy will then have all the latest and greatest code from all the developers. Usually this action is done when the development tasks are completed and ready for testing. The build team

will then take the copy of the project from the version control system and builds the application.

Synchronizing the local project

In this action, every developer will connect to the version control system and synchronize his local copy of the project with the server copy. This will merge all the developers work into the local copy. This way, developers need not email the code. All they have to do is synchronize the project, and all the latest and greatest version of the code will be available in his own local project.

Using version control systems completely eliminates the crude way of backing up the code and also improves the development style. Most version control systems also allow comparisons with previous versions of the file to identify the changes. This helps in identifying the person who made the changes to the code.

This completes all the important things that you need to know to understand how things work within an enterprise. In the next section, I'd like to give you some interview tips and suggestions.

Interview Tips

- ✓ Since 90% of them will be telephonic interviews, don't panic while talking. Speak loud and clear. Don't rush things.
- ✓ Avoid using negative words like "don't know, I am not sure" etc.. Always be positive.
- ✓ Never try to deviate the attention using unrelated answers. This will put you in big trouble. Keep answers simple and to the point.
- ✓ Never lose your heart at any stage. Be confident.
- ✓ During face to face interviews, always maintain a gentle smile on your face. This will boost your confidence.

Always remember one thing, there are plenty of opportunities out there. Don't lose your heart if you miss one. Missing an opportunity means a better one is waiting for you. Always think positively and be brave. No one can stop your success. Make sure that you always move forward.

Software Installation

J2EE, MYSQL and DBVisualizer

Software Installation

J2EE, MYSQL and DBVisualizer

Installing J2EE

1. Go to the following link and download the J2EE 1.4 SDK.

<http://java.sun.com/j2ee/1.4/download.html>

2. Accept all the licences and download the J2EE Setup file.
3. Double click and install the J2EE 1.4 SDK.
4. Accept the defaults and keep clicking OK until finished.
5. You'll see the following directory in C drive.

C:\Sun

This completes installing J2EE.

Installing MYSQL Database

1. Go to <http://dev.mysql.com/downloads/mysql/5.1.html>

Click on the “Download” link shown below and save the zip file to C:\MySQL

Windows downloads ([platform notes](#))

Windows Essentials (x86)	5.1.14-beta	19.9M	Download Pick a mirror
		MD5: d40c3837dde3ecaa47a05ad3fb1c797b	
Windows (x86) ZIP/Setup.EXE	5.1.14-beta	34.9M	Download Pick a mirror
		MD5: 575b1e51a06da60f8a84797915b9f5d2 Signature	
Without installer (unzip in C:\)	5.1.14-beta	37.6M	Download Pick a mirror
		MD5: e54ea728ad20549cd35759e11866fa25 Signature	

Extract the ZIP file into “C:\MySQL” directory. After it extracts, you will see the directory “mysql-5.0.26-win32” in “C:\MySQL”. This completes the MYSQL installation

2. Installing the Drivers

Go to <http://dev.mysql.com/downloads/connector/j/5.0.html> and click on the “DownLoad” link and save the file to the “C:\MySQL” directory.

Software Installation

J2EE, MYSQL and DBVisualizer

We suggest that you [use the MD5 checksums and GPG signatures to verify the integrity of the packages you download.](#)

Source and Binaries (tar.gz)	5.0.0-beta	8.0M	Download Pick a mirror
Source and Binaries (zip)	5.0.0-beta	8.1M	MD5: 8b673dde79ba5520f4018242e410b1fb Signature Download Pick a mirror MD5: d2eae7d6fc2b859ad5a5011ac00fd Signature

You will see “mysql-connector-java-5.0.0-beta” directory in “C:\ MySql” directory.
This directory will have the driver files.

Installing DBVisualizer

- 1 Downloading the DBVisualizer tool.

Go to <http://www.minq.se/products/dbvis/install-436/install.jsp>

Click on “Download” link shown below and save it.



Double Click on the downloaded file and follow the steps below.

- a) Click “Next”



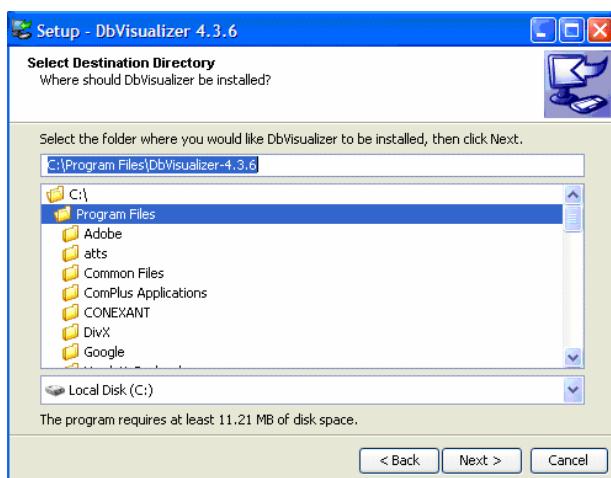
Software Installation

J2EE, MYSQL and DBVisualizer

- b) Accept the license and click Next



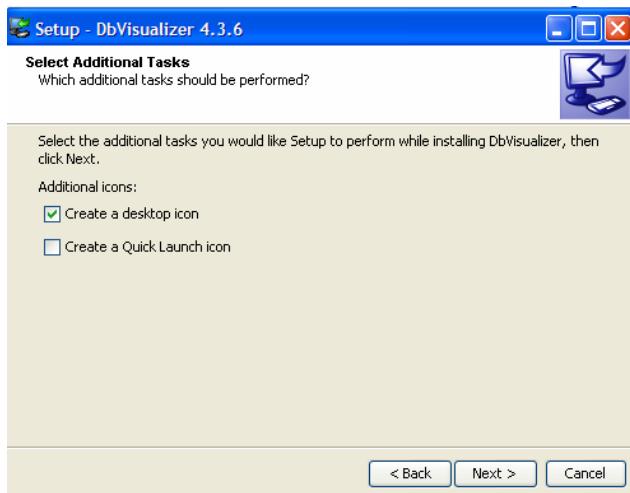
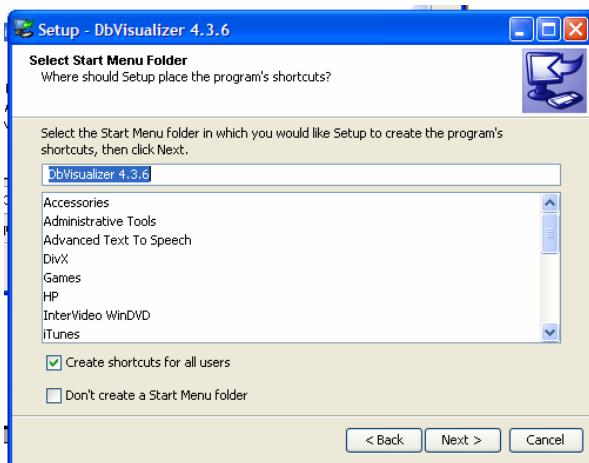
- c) Accept Defaults and Click Next



- d) Keep Clicking Next.

Software Installation

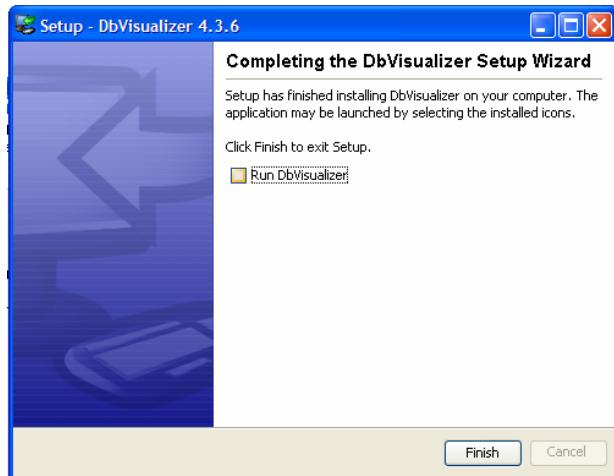
J2EE, MYSQL and DBVisualizer



- e) Uncheck the box, and Click Finish

Software Installation

J2EE, MYSQL and DBVisualizer



This completes the installation

Creating Database in MySQL

- a) Start the Mysql server as follows. Leave the server running

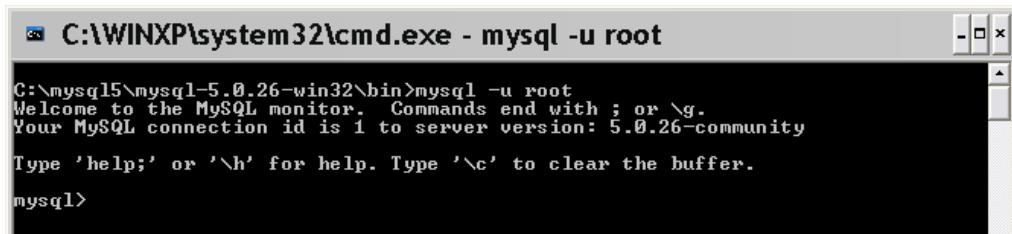


```
C:\WINXP\system32\cmd.exe - mysqld
C:\mysql5\mysql-5.0.26-win32\bin>mysqld
```

A screenshot of a Windows command prompt window titled "C:\WINXP\system32\cmd.exe - mysqld". The command "mysqld" is typed into the window.

- b) Creating a database named "MyDB"

Open a new command window, move to the "bin" directory and run the command as shown below. You will see the "Mysql" prompt.



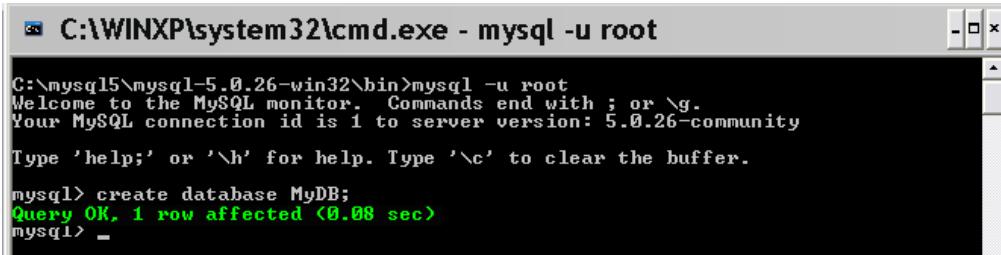
```
C:\WINXP\system32\cmd.exe - mysql -u root
C:\mysql5\mysql-5.0.26-win32\bin>mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.26-community
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

A screenshot of a Windows command prompt window titled "C:\WINXP\system32\cmd.exe - mysql -u root". The command "mysql -u root" is typed into the window. The MySQL monitor starts with the message "Welcome to the MySQL monitor. Commands end with ; or \g.". The prompt "mysql>" is visible at the bottom.

Software Installation

J2EE, MYSQL and DBVisualizer

- c) At the “mysql” prompt, create the database as shown below.



The screenshot shows a Windows Command Prompt window titled "C:\WINXP\system32\cmd.exe - mysql -u root". The window contains the following text:

```
C:\mysql15\mysql-5.0.26-win32\bin>mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.26-community
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database MyDB;
Query OK, 1 row affected (0.08 sec)

mysql> -
```

- d) Type “quit” at the prompt to terminate MySQL.

Configuring DBVisualizer with MySQL

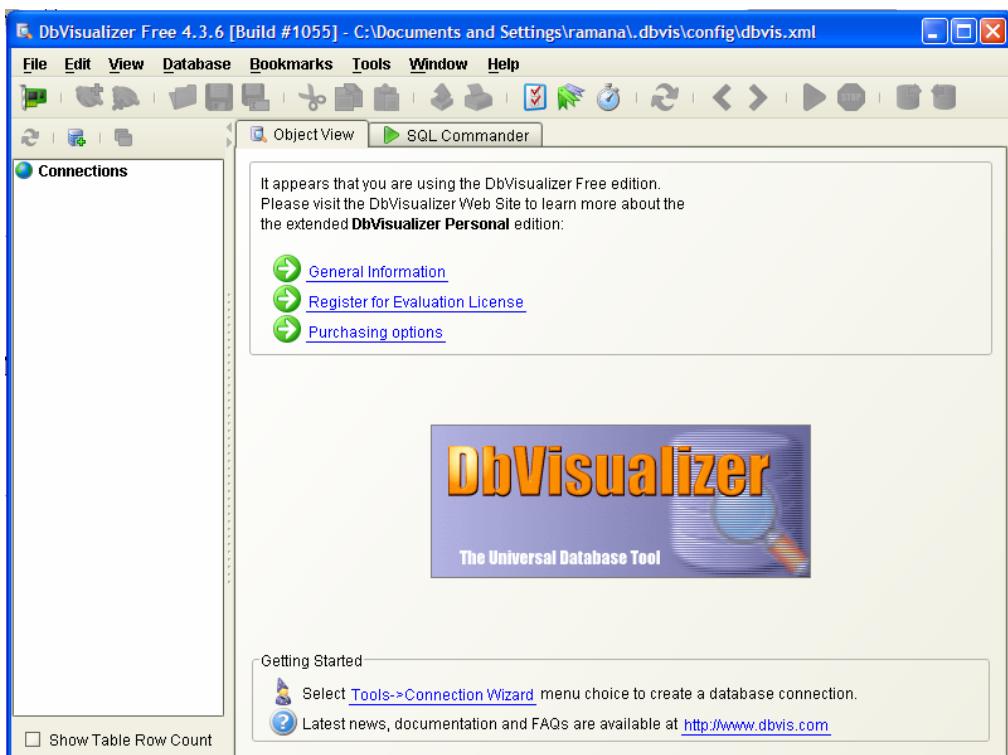
Now, lets open the DBVisualizer tool and look at the database we created. We can do all the DB operations from the command prompt like above. But it would be easy if we have a nice GUI from where we can connect to the database, and execute the SQL's. This is what DBVisualizer is about.

Goto “Start->All Programs->DBVisualizer 4.3.6->DBVisualizer4.3.6

This will display the following window

Software Installation

J2EE, MYSQL and DBVisualizer

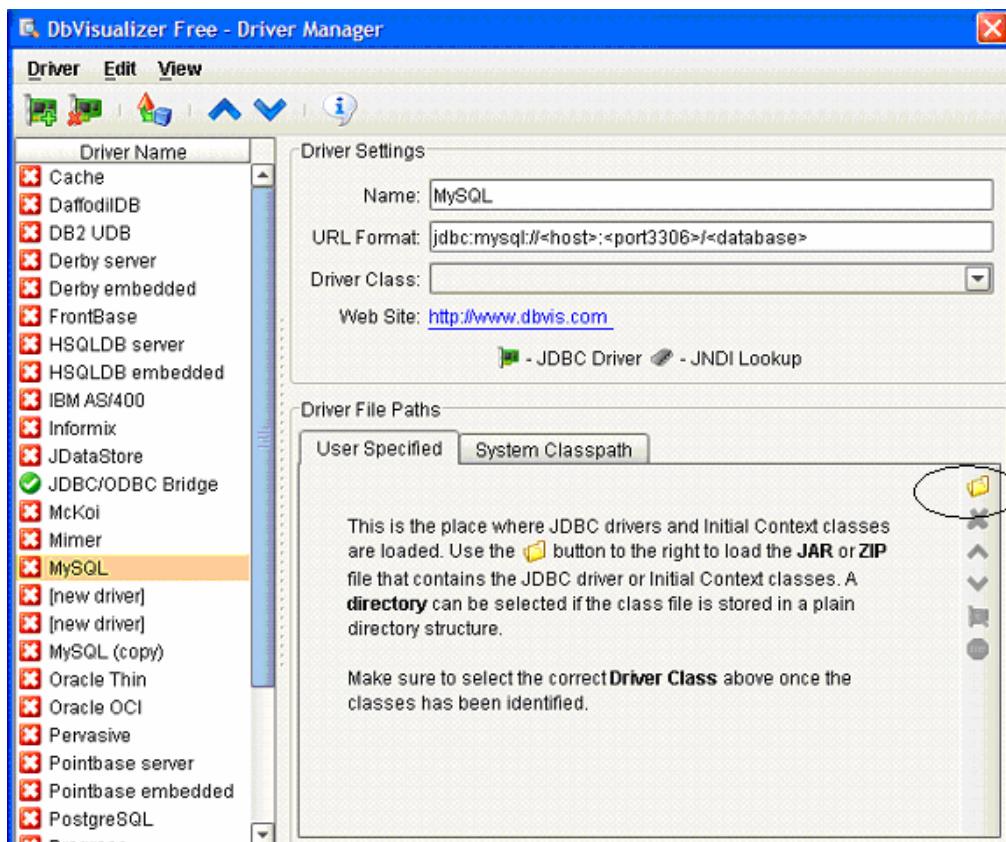


First, we need to configure the tool by loading the MYSQL drivers.

Go to "Tools -> Driver Manager". This will display the following window

Software Installation

J2EE, MYSQL and DBVisualizer



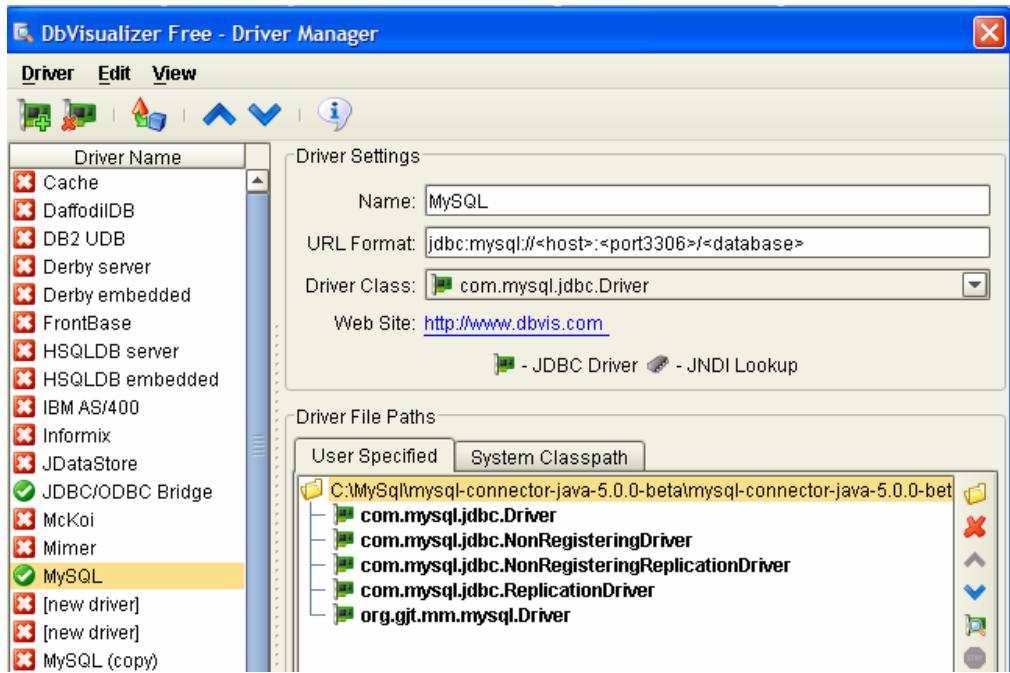
Click on “MySQL” on the left hand side, and then click on the “folder” button on the right hand side as shown above.

Browse to the following Jar file in the “C:\MySQL\mysql-connector-java-5.0.0-beta” and open it. This will load all the drivers as shown in the next window.

mysql-connector-java-5.0.3-bin.jar

Software Installation

J2EE, MYSQL and DBVisualizer



Close the above window.

Go to *Database->Create Database Connection*. Then click “No” on the pop up window. This will display the next window.

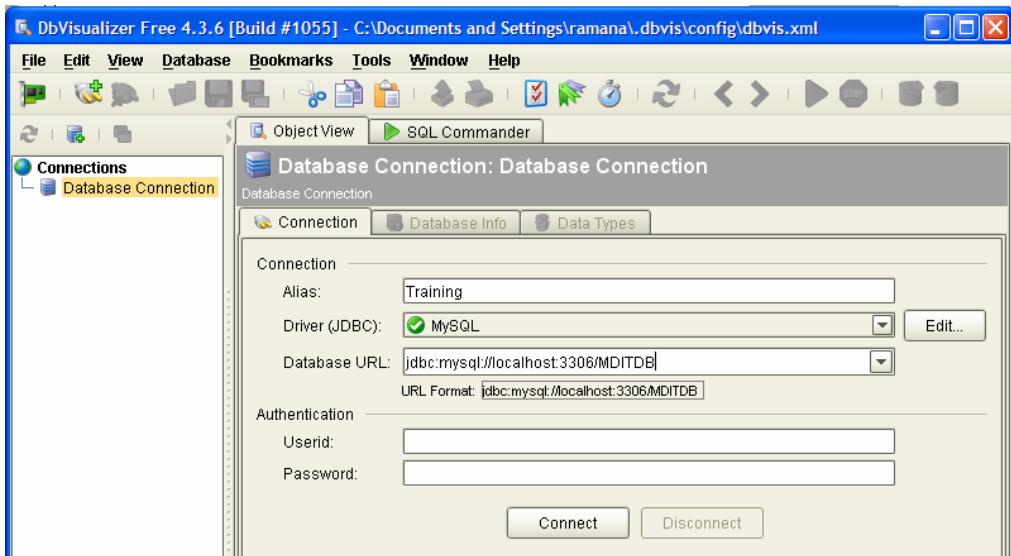
1. Type “*Training*” in alias field,
2. Select “MySQL” from the dropdown
3. Type the following URL in the *Database URL* field.

jdbc:mysql://localhost:3306/MyDB

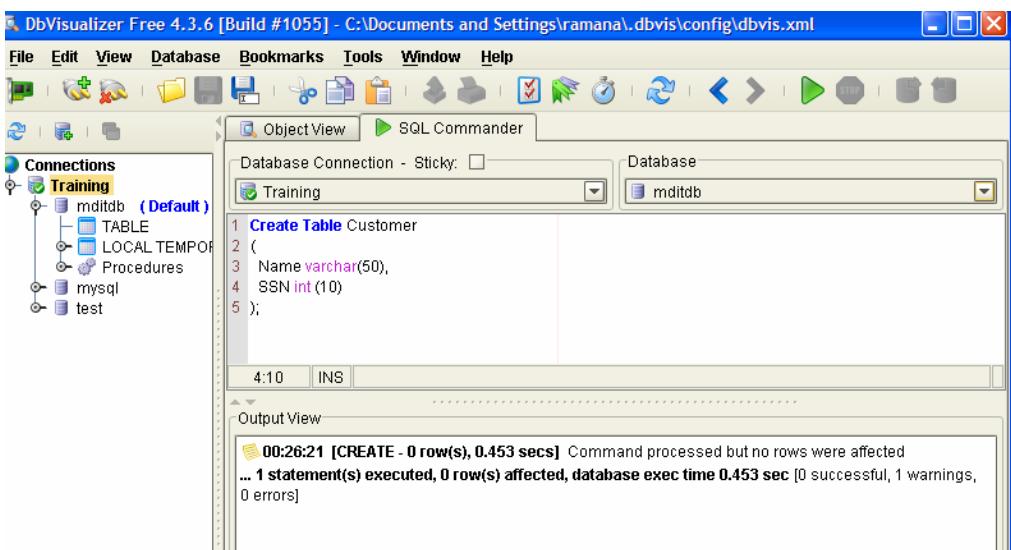
4. Click *Connect*.

Software Installation

J2EE, MYSQL and DBVisualizer



You will get connected to the database as shown below.



In the above window, all the tables will be listed under the TABLE node. Right now, it will be empty since we didn't create any tables.

Conclusion

Conclusion

Firstly, let me congratulate you for successfully completing this mindblowing journey through Java and J2EE. I am sure you enjoyed reading it as much as I enjoyed in explaining the things. The whole idea behind writing this book is to help you in landing in a good job by giving you the essentials that industry is demanding. I suggest the beginners to read the last chapter that gives an idea of how things are done within the enterprise. These non technical details are as important as all the technical details that we covered in this book. I also strongly recommend you to keep reading the summary sections and interview questions when ever time permits and especially when you are getting ready for an interview. The good thing with Java and J2EE is that the concepts are extremely simple. The problem is that there are plenty of these concepts. It is easy to remember one or two tough concepts than remembering 10 very easy concepts. This is why I suggest you to always keep in touch with Java and J2EE concepts.

I tried my best to explain Java and J2EE as simple as I can, sometimes using simple stories. If you find some concepts confusing, leave them temporarily and try visiting them later. If they are still confusing simply ignore them. You'll still be fine. Trust me.

Finally, thank you very much for considering my book. I wish you all the best and good luck. For now, its Good Bye, Au Revoir, Namaste and Khudaa hafiz.

Phani Kosuri

Index

A

abnormal, 157
abstract, 109, 114, 135
Abstract, 108
Action, 459
ActionErrors, 491
ActionForm, 459
ActionMapping, 459
ActionServlet, 449
Advice, 570
After Advice, 572
AfterReturningAdvice, 572
Analysts, 671
AOP, 552, 569
API, 251
application server, 265
ApplicationContext, 558, 561
Architecture, 677
arguments, 31
ArithmaticException, 160
Around Advice, 573
ArrayList, 194
Aspect, 570
association, 605
asynchronous messages, 525
attributes, 316
AUTO_ACKNOWLEDGE, 533
Availability, 257

B

BasicDataSource, 581
batch processing, 298
Bean Tags, 468
bean:message, 469
bean:write, 469
BeanFactory, 552, 558
Beans, 400
Before Advice, 571
broker, 529
built in classes, 183
Business Delegate, 657
bytecode, 2

C

catch, 158
catch block, 167
charAt, 188
Checked Exceptions, 161
Checkin, 684
Checkout, 684
child class, 89, 133
class, 10
Class, 55
CLASSPATH, 248, 250
ClassPathXmlApplicationContext, 562
CLIENT_ACKNOWLEDGE, 533
clientID, 535
code duplication, 98
Code review, 678
Collection, 194
comments, 31
commit, 301, 603
component, 605
composite JSP, 511
concurrent, 223
Configuration, 608
connection pooling, 303
constrained, 621
constructor, 82
constructors, 103, 105
Container, 344
context, 646
Conventions, 211
core module, 557
Core Module, 552
Core Tags, 432
Coupling, 131
Criteria, 641
custom exception, 171
Custom Tags, 420

D

DAO, 552, 580
Data Access Object, 664
Database, 272
DataSource, 562

Index

dataSources, 581
Date, 207
Decorator, 651
default, 51, 106
default constructor, 85
definition, 515
Deployment descriptor, 344
design pattern, 647
Development Teams, 673
Directives, 387
Dirty Read, 305
discriminator, 637
Dispatching, 363
Document, 314, 331
Documentation, 212
DocumentBuilderFactory, 331
doEndTag(), 421
doGet(), 350
DOM, 325, 336
DOM Parsing, 331
doPost(), 350
doStartTag(), 421
DriverManager, 283, 285
Drivers, 280
DTD, 318
Durable, 534
DynaActionForm, 492

E

e-commerce, 256
EJB, 549
Encapsulation, 73
Enterprise Architecture, 265
errors, 163
EVAL_BODY_INCLUDE, 423
exception, 157
executeQuery, 292
executeUpdate, 289
Expressions, 393
extends, 89, 120

F

Façade, 654
factory, 564
Factory Pattern, 648
fat clients, 342
FileSystemXmlApplicationContext, 561
final, 112, 114
finally, 177
for, 21
forEach, 434

Foreign Key, 618
FormBeans, 483
forward(), 364
Front Controller, 655
fully qualified name, 47

G

GET, 350
GET Request, 343
getParameter(), 359

H

HandlerBase, 330
HashMap, 205
HashSet, 194, 197
Hashtable, 206
hasMoreTokens, 210
Hibernate, 595
hibernate.cfg.xml, 597
hibernate.properties, 610
HQL, 640
HTML tags, 481
HTTP, 343

I

immutable, 191
implements, 115
Implicit Objects, 396
include, 390
include(), 364
indexOf, 188
Infrastructure Team, 673
inheritance, 89, 129
init(), 350
Initialization, 353
initialization, 81
interface, 114
Interface, 145
interfaces, 139
Internationalization, 517
interpreter, 2
Interview, 685
IoC, 553
Isolation, 305
IST Testing, 680

J

J2EE, 256
J2EE technologies, 258
JAD Sessions, 672

Index

JAR, 244
JAR file, 245, 246
java.lang, 183
java.util, 183

JavaBean, 76
javadoc, 213

Javascript, 508
JAXP, 326
JDBC, 278
JDBC-ODBC Bridge, 280
JDBCTemplate, 581
JMS, 529
JSP, 384
jsp:getProperty, 402
jsp:setProperty, 401
jsp:useBean, 401
JSTL, 427, 432

L

length(), 188
List, 194
Load Testing, 681
LOB, 671
Logic Tags, 477
logic:iterate, 479
logic:notEmpty, 478

M

main, 25, 27
Main Thread, 229
Many-to-Many, 612, 623
many-to-one, 613
Many-to-One, 612
master layout, 511
MAX_PRIORITY, 232
MessageListener, 535
MethodBeforeAdvice, 571
MethodInterceptor, 573
methods, 146
MIN_PRIORITY, 232
MoM, 526
MoM system, 526
multiple catch, 167
multiple inheritance, 96
multi-processing, 224
multi-tasking, 224
MVC, 446

N

NameMatchMethodPointCut, 575
new, 59

Non-Durable, 534
N-Tier, 264
NullPointerException, 162, 168

O

O/R mappings, 595
Object, 55, 120
object references, 68
One-to-Many, 612
One-to-One, 612, 618
OOP, 55
ORM, 596
overloaded, 85
overloading, 80
Overloading, 78, 102
Overwriting, 97, 102
overwritten, 135

P

package, 44, 45
page, 387
Parsing, 325
pattern, 646
performance, 295
Performance, 257
persistence, 594
Phantom Reads, 306
platform, 1
platform independent, 1
PointCut, 571
Pointcuts, 575
Point-to-Point, 528
POJO, 76
POJO's, 550
Polymorphic, 630
polymorphism, 636
portable, 1
POST Request, 344
PreparedStatement, 283, 295
Primary Key, 618
Priorities, 232
private, 49, 72, 129
Production, 681
prolog, 316
Properties, 206
protected, 49, 51
public, 49, 72
Publish/Subscribe, 527
Pure Java Driver, 282

Index

Q

quality of service, 526
Query, 641
Queue, 528, 530
QueueConnectionFactory, 530

R

references, 128
relational databases, 272
Reliability, 257
Repeatable Read, 305
RequestProcessor, 450
Requirement Analysis, 676
ResultSet, 292
return, 30
rollback, 301
run(), 230
Runnable, 229, 230
Runtime Exceptions, 161

S

SAX, 325, 326, 336
SAXParser, 327
SAXParserFactory, 327
Scheduler, 226
Scriptlets, 394
Servers, 265
Servlet, 345, 347
servletConfig, 354
Session, 361, 363
Session Façade, 660, 662
SessionFactory, 601
SimpleDateFormat, 208
Singleton, 650
specifications, 266
Spring, 549
SQL, 273, 580
SQL Tags, 437
Standalone, 2
start(), 231
stateless, 343
static, 36, 37, 56
String, 75, 187
StringBuffer, 191
 StringTokenizer, 210
Structural Patterns, 651
Struts, 427, 449
Struts Validator, 483, 498
struts-bean, 467
struts-config.xml, 450, 451
struts-html, 467

struts-logic, 467
struts-tiles, 467
subclass, 637
subscriber, 535
substring, 188
super, 100
Synchronization, 239
synchronized, 239
Synchronizing, 685
synchronous messaging, 525

T

Table per Concrete class, 631
Table per Hierarchy, 637
Table per Sub Class, 639
taglib, 391
TagSupport, 421
Team Structure, 671
templates, 552, 581
Testing, 674
thin clients, 342
this, 78, 88
Thread Lifecycle, 227
Thread Queue, 226
throw, 169
Throwing, 171
throws, 174
ThrowsAdvice, 574
Tiles, 510
tiles:put, 515
time-slicing, 226
Topic, 528, 530
TopicConnection, 532
TopicConnectionFactory, 530
TopicSession, 532
toString, 121
Transactions, 300
Transfer Object, 663
TreeSet, 194, 200
try, 158
try block, 166
Type-2, 281
Type-3, 281
Type-4, 282

U

UAT Testing, 680
unique, 638
Unit Testing, 679
usability, 244

Index

V

`validate()`, 491
Validation, 483
`validation.xml`, 499
`validation-rules`, 499
`VARCHAR`, 293
Vector, 196
version control, 682

W

web container, 344
web.xml, 348
`WEB-INF`, 346
well formed, 315

while, 21
wiring, 558
Wrapper, 184

X

Xerces, 326
XML, 314
XML parser, 325
XML Schema, 321
XML Tags, 441
`XMLBeanFactory`, 559

Y

Yielding, 234