



*Introduction to Java using  
IBM Rational Application  
Developer V6*  
(Course Code SW244)

## Student Notebook

ERC 6.0

WebSphere Training and  
Technical Enablement

## Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

Cloudscape™

DB2®

developerWorks®

J™

Perform™

Rational®

Rational Unified Process®

WebSphere®

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.

## May 2005 Edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

**© Copyright International Business Machines Corporation 2003, 2005. All rights reserved.**

**This document may not be reproduced in whole or in part without the prior written permission of IBM.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Trademarks . . . . .</b>	<b>xiii</b>
<b>Course Description . . . . .</b>	<b>xv</b>
<b>Agenda . . . . .</b>	<b>xvii</b>
<b>Unit 1. Introduction to SW244 . . . . .</b>	<b>1-1</b>
Class introductions . . . . .	1-4
Course overview . . . . .	1-5
Course objectives . . . . .	1-6
Course prerequisites . . . . .	1-7
Agenda - Day 1 . . . . .	1-8
Agenda - Day 2 . . . . .	1-9
Agenda - Day 3 . . . . .	1-10
Agenda - Day 4 . . . . .	1-11
Agenda - Day 5 . . . . .	1-12
<b>Unit 2. Introduction to the Java Programming Language. . . . .</b>	<b>2-1</b>
Unit objectives . . . . .	2-4
What Is Java? . . . . .	2-5
Goals of Java . . . . .	2-6
The Java programming language (1 of 4) . . . . .	2-8
The Java programming language (2 of 4) . . . . .	2-9
The Java programming language (3 of 4) . . . . .	2-10
The Java programming language (4 of 4) . . . . .	2-11
Java evolution . . . . .	2-12
The Java platform . . . . .	2-13
The Java execution model . . . . .	2-14
Uses of Java . . . . .	2-15
Programs . . . . .	2-16
Components (1 of 2) . . . . .	2-17
Components (2 of 2) . . . . .	2-19
Checkpoint . . . . .	2-20
Web resources . . . . .	2-21
Recommended reading . . . . .	2-22
Unit summary . . . . .	2-23
<b>Unit 3. Introduction to IBM Rational Application Developer. . . . .</b>	<b>3-1</b>
Unit objectives . . . . .	3-4
What is a workspace? . . . . .	3-5
Switching workspaces . . . . .	3-6
What is the workbench? . . . . .	3-7
Parts of the workbench . . . . .	3-8
Perspectives, views and editors . . . . .	3-9

Perspectives . . . . .	3-10
Customizing perspectives . . . . .	3-11
Saving perspectives . . . . .	3-12
Views . . . . .	3-13
Displaying views . . . . .	3-14
Fast views . . . . .	3-15
Editors . . . . .	3-16
Capabilities . . . . .	3-17
Resources . . . . .	3-18
Projects in Application Developer . . . . .	3-19
Create a new project . . . . .	3-20
Local history . . . . .	3-21
Replace with local history . . . . .	3-22
Comparing files . . . . .	3-23
Importing and exporting resources . . . . .	3-24
Search . . . . .	3-25
Content assist . . . . .	3-26
Tasks . . . . .	3-27
Problems . . . . .	3-28
Help topics: cheat sheets . . . . .	3-29
Help topics: samples gallery . . . . .	3-30
Help topics: tutorials gallery . . . . .	3-31
Help topics: help search and help contents . . . . .	3-32
Checkpoint questions . . . . .	3-33
Unit summary . . . . .	3-34
 <b>Unit 4. Object-Oriented Programming . . . . .</b>	 <b>4-1</b>
Unit objectives . . . . .	4-4
Procedural programming . . . . .	4-5
Introduction to objects . . . . .	4-6
What are objects? . . . . .	4-7
Objects – another example . . . . .	4-8
A person object . . . . .	4-9
Procedural versus object approach – example . . . . .	4-10
What Is object-oriented programming? . . . . .	4-11
Object communication: messages (1 of 2) . . . . .	4-12
Object communication: messages (2 of 2) . . . . .	4-13
Messages and methods (1 of 2) . . . . .	4-14
Messages and methods (2 of 2) . . . . .	4-15
Accessing data . . . . .	4-16
Objects and instances . . . . .	4-17
Classes . . . . .	4-18
What is necessary to define a class? . . . . .	4-19
Benefits of using objects and classes . . . . .	4-20
Java person class example . . . . .	4-22
Unified Modeling Language (UML) . . . . .	4-23
What can UML model? . . . . .	4-24
Class diagrams . . . . .	4-25

Object-oriented design principles .....	4-26
Modularity .....	4-27
Methods for promoting modularity .....	4-28
Abstraction .....	4-29
Encapsulation .....	4-30
Interfaces .....	4-31
Polymorphism .....	4-32
Interface and polymorphism example .....	4-33
Inheritance .....	4-34
Inheritance example .....	4-35
Checkpoint .....	4-36
Unit summary .....	4-37
 <b>Unit 5. Java Syntax Basics – Part 1 .....</b>	 <b>5-1</b>
Unit objectives .....	5-4
Identifiers .....	5-5
Java is case-sensitive .....	5-6
Reserved words .....	5-7
Java primitives .....	5-8
Primitives: integers .....	5-9
Primitives: floating points .....	5-10
Primitives: characters .....	5-11
Primitives: booleans .....	5-12
Primitive literals .....	5-13
Primitive literals: integers .....	5-14
Primitive literals: floating point .....	5-15
Primitive literals: escape sequences .....	5-16
Casting primitive types .....	5-17
Implicit versus explicit casting .....	5-19
Declarations and initialization .....	5-20
Arrays .....	5-21
Operators and precedence .....	5-22
Comments .....	5-23
Statements .....	5-24
Checkpoint .....	5-25
Unit summary .....	5-26
 <b>Unit 6. Java Syntax Basics – Part 2 .....</b>	 <b>6-1</b>
Unit objectives .....	6-4
Objects and messages .....	6-5
Declaring and initializing objects .....	6-6
Identity .....	6-7
Wrapper classes .....	6-8
Using wrapper classes .....	6-9
Strings .....	6-10
Concatenating strings .....	6-11
String messages .....	6-12
Comparing strings .....	6-13

StringBuffer . . . . .	6-14
Conditional statement: if-else . . . . .	6-15
Shortcut for if-else: the ternary operator . . . . .	6-16
Conditional statement: switch . . . . .	6-17
Looping statements: while and do...while . . . . .	6-18
Looping statement: for . . . . .	6-19
for versus while . . . . .	6-20
Branching statements . . . . .	6-21
Sample branching statements . . . . .	6-22
Scope . . . . .	6-23
Checkpoint . . . . .	6-24
Unit summary . . . . .	6-25
<b>Unit 7. Building Classes . . . . .</b>	<b>7-1</b>
Unit objectives . . . . .	7-4
Classes . . . . .	7-5
Implementing classes . . . . .	7-6
Class declaration . . . . .	7-7
Class modifiers . . . . .	7-8
Constructors . . . . .	7-9
Memory management in Java . . . . .	7-10
More about constructors . . . . .	7-11
Default constructors . . . . .	7-12
Overloading constructors . . . . .	7-13
Constructor example . . . . .	7-14
Constructor chaining . . . . .	7-15
Java destructors? . . . . .	7-16
Garbage collector . . . . .	7-17
Working with the garbage collector . . . . .	7-18
Fields . . . . .	7-19
Messages . . . . .	7-20
Methods . . . . .	7-21
Method signatures . . . . .	7-22
Method parameters . . . . .	7-23
Returning from methods . . . . .	7-24
Invoking methods . . . . .	7-25
Overloading methods . . . . .	7-26
Overriding methods . . . . .	7-27
main method . . . . .	7-28
Encapsulation . . . . .	7-29
Static members . . . . .	7-30
Final members . . . . .	7-31
Abstract classes . . . . .	7-32
Packages . . . . .	7-33
Class visibility . . . . .	7-34
import statement . . . . .	7-35
Java 1.4 packages . . . . .	7-36
Core Java packages . . . . .	7-37

Sample package: java.lang .....	7-38
Sample class: String .....	7-39
Sample class: StringBuffer .....	7-40
Checkpoint .....	7-41
Unit summary .....	7-42
<b>Unit 8. Debug Applications .....</b>	<b>8-1</b>
Unit objectives .....	8-4
Debug tooling .....	8-5
Debug perspective .....	8-6
Debug view .....	8-7
Debug view navigation .....	8-8
Stepping through code commands .....	8-9
Breakpoints .....	8-10
Breakpoints view .....	8-11
Adding a breakpoint into a Java application .....	8-12
Where can breakpoints be set? .....	8-13
Configuring breakpoint properties .....	8-14
The Variables view .....	8-15
Viewing and changing variables .....	8-16
Data in the Expressions view .....	8-17
Checkpoint questions .....	8-18
Unit summary .....	8-19
<b>Unit 9. Inheritance.....</b>	<b>9-1</b>
Unit objectives .....	9-4
Class hierarchies .....	9-5
Specialization and generalization .....	9-6
Inheritance relationships .....	9-7
Multiple inheritance .....	9-8
Inheriting fields and methods .....	9-9
Access modifiers .....	9-10
Overriding methods .....	9-11
Restrictions on overriding methods .....	9-12
Example of overriding .....	9-13
Method lookup .....	9-14
Inheritance and static methods .....	9-15
Inheritance and constructors .....	9-16
The superclass in object construction .....	9-17
Default constructors .....	9-18
More on this and super .....	9-19
Polymorphism .....	9-20
Polymorphism through inheritance .....	9-21
Implementing polymorphism .....	9-22
Example .....	9-23
Checkpoint .....	9-25
Unit summary .....	9-26

<b>Unit 10. Design Patterns and Refactoring</b>	<b>10-1</b>
Unit objectives	10-4
A focus on structure	10-5
What is a design pattern?	10-6
Types of patterns	10-7
Key elements of a pattern	10-9
Example - Singleton design pattern	10-10
What is refactoring?	10-12
Why do you need to refactor?	10-13
When do you refactor?	10-14
The refactoring cycle	10-15
To refactor or not to refactor	10-16
Refactoring strategies and techniques	10-17
Refactoring tools	10-18
Using the refactoring tools	10-19
Refactoring example	10-20
Refactored solution	10-21
Checkpoint	10-22
Unit summary	10-23
 <b>Unit 11. Interfaces</b>	 <b>11-1</b>
Unit objectives	11-4
Interfaces	11-5
Protocols	11-6
Implementing interface methods	11-7
Syntax	11-8
Implementing an interface	11-9
Typing and interfaces	11-10
Subinterfaces	11-11
Using interfaces	11-12
Example	11-13
More advantages of using interfaces	11-14
Naming conventions for interfaces	11-15
Checkpoint	11-16
Unit summary	11-17
 <b>Unit 12. Collections</b>	 <b>12-1</b>
Unit objectives	12-4
What Is a collection?	12-5
Collections represent data structures	12-6
The Java Collections Framework	12-7
Benefits of a collections framework	12-8
Specific kinds of collections	12-9
Interfaces in the framework	12-10
The collection interface	12-11
Collections, sets and lists	12-12
The Map interface	12-13
Comparing objects	12-14

Sorted collections . . . . .	12-15
Iterators . . . . .	12-16
The Iterator code pattern . . . . .	12-17
Interfaces and implementations . . . . .	12-18
Example . . . . .	12-19
Implementation choices . . . . .	12-20
Legacy collections . . . . .	12-21
Cloning collections . . . . .	12-22
The Collections class . . . . .	12-23
Collections examples . . . . .	12-24
Checkpoint . . . . .	12-25
Unit summary . . . . .	12-26
<b>Unit 13. Threads and Synchronization . . . . .</b>	<b>13-1</b>
Unit objectives . . . . .	13-4
Processes and threads . . . . .	13-5
Threads in Java . . . . .	13-6
Creating threads in Java . . . . .	13-7
Example: subclassing the Thread class . . . . .	13-8
Example: implementing Runnable . . . . .	13-9
Life cycle of a Thread . . . . .	13-10
Controlling activities . . . . .	13-11
Stopping threads . . . . .	13-12
Daemon threads . . . . .	13-13
Multithreading: need for synchronization . . . . .	13-14
Synchronization (1 of 2) . . . . .	13-15
Synchronization (2 of 2) . . . . .	13-16
Synchronization issues . . . . .	13-17
Checkpoint . . . . .	13-19
Unit summary . . . . .	13-20
<b>Unit 14. Utility Classes . . . . .</b>	<b>14-1</b>
Unit objectives . . . . .	14-4
Conversion functions of wrapper classes . . . . .	14-5
Numeric constants . . . . .	14-6
Examples . . . . .	14-7
The Math class . . . . .	14-8
The System class . . . . .	14-9
The Class class . . . . .	14-10
Pattern matching . . . . .	14-11
Regular expressions . . . . .	14-12
Some useful constructs . . . . .	14-13
Using regular expressions with Strings . . . . .	14-14
Example . . . . .	14-15
More extensive operations . . . . .	14-16
Example . . . . .	14-17
Checkpoint . . . . .	14-18
Unit summary . . . . .	14-19

<b>Unit 15. Exceptions and Exception Handling.....</b>	<b>15-1</b>
Unit objectives .....	15-4
Exceptions .....	15-5
Exception handling .....	15-6
Exception sources .....	15-7
The exception hierarchy .....	15-8
Handling exceptions .....	15-10
Keywords .....	15-11
try/catch blocks .....	15-12
The catch clause .....	15-14
Example .....	15-15
The finally clause .....	15-16
Example .....	15-17
Nested exception handling .....	15-18
The throw keyword .....	15-19
Handling runtime exceptions .....	15-21
Assertions .....	15-22
Using assertions .....	15-23
When to use assertions .....	15-24
Checkpoint .....	15-25
Unit summary .....	15-26
 <b>Unit 16. I/O and Serialization.....</b>	 <b>16-1</b>
Unit objectives .....	16-4
Streams .....	16-5
Operations on streams .....	16-6
Class hierarchy: byte stream input .....	16-7
Class hierarchy: byte stream output .....	16-9
Class hierarchy: character stream input .....	16-11
Class hierarchy: character stream output .....	16-13
Creating streams .....	16-15
Filter streams .....	16-16
Filter stream example: BufferedReader .....	16-17
Filter stream example: BufferedWriter .....	16-18
Dealing with files .....	16-19
Midway checkpoint 1 .....	16-20
Why new I/O? .....	16-21
Key classes .....	16-22
Buffers .....	16-23
The Channel interface .....	16-24
Three steps to read from a channel .....	16-25
Writing to a file .....	16-26
Midway checkpoint 2 .....	16-27
Representing objects outside the JVM .....	16-28
Serialization and externalization .....	16-29
Serializable interface .....	16-30
Transient and readObject() .....	16-31
Serialization object graph .....	16-32

Stream unique identifier - SUID . . . . .	16-33
Example (1 of 2) . . . . .	16-34
Example (2 of 2) . . . . .	16-35
Externalizable . . . . .	16-36
Checkpoint . . . . .	16-37
Unit summary . . . . .	16-38
<b>Unit 17. JavaBeans . . . . .</b>	<b>17-1</b>
Unit objectives . . . . .	17-4
What is a Java bean? . . . . .	17-5
JavaBeans as software components . . . . .	17-6
JavaBeans API . . . . .	17-7
JavaBeans terminology (1 of 2) . . . . .	17-8
JavaBeans terminology (2 of 2) . . . . .	17-9
Three levels of JavaBeans usage . . . . .	17-10
Bean basics . . . . .	17-11
Exported bean features . . . . .	17-13
Bean properties . . . . .	17-14
Bean methods and events . . . . .	17-15
Bean example - Counter . . . . .	17-16
Bean information . . . . .	17-17
Checkpoint . . . . .	17-18
Unit summary . . . . .	17-19
<b>Unit 18. JDBC Overview . . . . .</b>	<b>18-1</b>
Unit objectives . . . . .	18-4
Java and databases . . . . .	18-5
JDBC 3.0 packaging . . . . .	18-6
JDBC architecture . . . . .	18-7
Using JDBC to execute an SQL query . . . . .	18-9
Query example . . . . .	18-10
ResultSet . . . . .	18-11
ResultSet methods . . . . .	18-12
ResultSet Example . . . . .	18-13
SQL exceptions . . . . .	18-14
PreparedStatements (1 of 2) . . . . .	18-15
PreparedStatements (2 of 2) . . . . .	18-16
PreparedStatement methods . . . . .	18-17
CallableStatement . . . . .	18-18
CallableStatement input/output (1 of 2) . . . . .	18-19
CallableStatement input/output (2 of 2) . . . . .	18-20
CallableStatement example (1 of 2) . . . . .	18-21
CallableStatement example (2 of 2) . . . . .	18-22
JDBC efficiency . . . . .	18-23
Checkpoint . . . . .	18-24
Unit summary . . . . .	18-25
Appendix: mapping SQL types . . . . .	18-26

<b>Unit 19. Java Naming and Directory Interface . . . . .</b>	<b>19-1</b>
Unit objectives . . . . .	19-4
The problem: finding resources . . . . .	19-5
The solution: naming and directory services . . . . .	19-6
Uses of naming and directory services . . . . .	19-7
What is a naming service? . . . . .	19-8
Some naming service terms . . . . .	19-9
Naming subcontexts . . . . .	19-10
How does a naming service work? . . . . .	19-11
What is a directory service? . . . . .	19-12
An issue with directory services . . . . .	19-13
JNDI . . . . .	19-14
Why do you need JNDI? (1 of 2) . . . . .	19-15
Why do you need JNDI? (2 of 2) . . . . .	19-16
System component: JNDI . . . . .	19-17
JNDI API: main packages . . . . .	19-18
JNDI naming subcontexts . . . . .	19-20
Using JNDI: naming operations (1 of 5) . . . . .	19-21
Using JNDI: naming operations (2 of 5) . . . . .	19-22
Using JNDI: naming operations (3 of 5) . . . . .	19-24
Using JNDI: naming operations (4 of 5) . . . . .	19-25
Using JNDI: naming operations (5 of 5) . . . . .	19-27
Checkpoint . . . . .	19-28
Unit summary . . . . .	19-29
<b>Unit 20. DataSources . . . . .</b>	<b>20-1</b>
Unit objectives . . . . .	20-4
Rationale for connection pooling . . . . .	20-5
What is connection pooling? . . . . .	20-6
Connection pooling in JDBC 3.0 . . . . .	20-7
DataSource implementation . . . . .	20-8
Using the DriverManager . . . . .	20-9
Using a DataSource . . . . .	20-10
How does a DataSource work? (1 of 2) . . . . .	20-11
How does a DataSource work? (2 of 2) . . . . .	20-12
Example: coding for DataSources . . . . .	20-13
Obtaining a DataSource connection . . . . .	20-14
DataSource example . . . . .	20-15
DataSource exceptions . . . . .	20-16
Checkpoint . . . . .	20-17
Unit summary . . . . .	20-18
Appendix: data sources support . . . . .	20-19
Appendix: creating a new data source . . . . .	20-20
Appendix: configuring a data source . . . . .	20-21
Appendix: configuring a connection pool . . . . .	20-22
Configuring project resources . . . . .	20-23

<b>Unit 21. Wrap-Up.....</b>	<b>21-1</b>
What you have seen .....	21-4
Resources .....	21-5
Advanced reading .....	21-6
More advanced reading .....	21-7
Goodbye .....	21-8



# Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

Cloudscape™

DB2®

developerWorks®

J™

Perform™

Rational®

Rational Unified Process®

WebSphere®

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.



# Course Description

## Introduction to Java using IBM Rational ApplicationDeveloper V6

**Duration:** 5 days

### Purpose

SW244 introduces programmers to the Java language using lectures, demonstrations, and extensive labs. It covers object-oriented programming, Java syntax, key Java classes and interfaces such as collections, exceptions, and I/O. The course also introduces JDBC, JNDI, and DataSources as ways of accessing relational databases.

### Audience

Appropriate audiences for this course would include:

- Entry-level Java programmers
- Programmers skilled in other languages such as COBOL or C

### Prerequisites

Ideally, students should have some experience in structured programming, and should have some knowledge of object-oriented concepts. Experience using relational databases would also be an asset.

### Objectives

After completing this course, you should be able to:

- Use Java syntax to develop simple applications in Java
- Describe the Java object model and its implementation
- Outline some of the important classes and interfaces available in Java, and be able to use them, including:
  - Primitive wrapper classes
  - Classes in the Collections Framework
  - Utility classes
  - I/O classes
  - Threads
  - Exceptions
- Develop simple Java bean software components

- Use JDBC to access and manipulate data in a relational database, and know how to code for a DataSource using JNDI
- Use IBM Rational Application Developer to develop and debug Java programs

## Contents

This course includes the following units:

1. Introduction to SW244
2. Introduction to the Java Programming Language
3. Introduction to IBM Rational Application Developer
4. Object-Oriented Programming
5. Java Syntax Basics - Part 1
6. Java Syntax Basics - Part 2
7. Building Classes
8. Debug Applications
9. Inheritance
10. Design Patterns and Refactoring
11. Interfaces
12. Collections
13. Threads and Synchronization
14. Utility Classes
15. Exceptions and Exception Handling
16. I/O and Serialization
17. JavaBeans
18. JDBC Overview
19. Java Naming and Directory Interface
20. DataSources
21. Wrap-up

## Curriculum relationship

SW244 may be followed by either of the following courses:

- SW284 - Servlet and JSP Development with IBM Rational Application Developer V6
- SW285 - Developing EJBs with IBM Rational Application Developer V6

# Agenda

## Day 1

Introduction to SW244  
Introduction to the Java Programming Language  
Introduction to IBM Rational Application Developer  
Exercise: Introduction to IBM Rational Application Developer  
Object-Oriented Programming  
Java Syntax Basics - Part 1  
Java Syntax Basics - Part 2

## Day 2

Exercise: Writing Simple Java Code in Application Developer  
Building Classes  
Exercise: Building Classes  
Debug Applications  
Exercise: Debugging Java Code

## Day 3

Inheritance  
Design Patterns and Refactoring  
Exercise: Inheritance and Refactoring  
Interfaces  
Collections  
Exercise: Collections and Sorting

## Day 4

Threads and Synchronization  
Exercise: Threads  
Utility Classes  
Exceptions and Exception Handling  
Exercise: Exception Handling  
I/O and Serialization

## Day 5

JavaBeans  
Exercise: JavaBeans  
Exercise: Serialization  
JDBC Overview

Java Naming and Directory Interface

DataSources

Exercise: Using JDBC

Wrap-up

# **Unit 1. Introduction to SW244**

## **What this unit is about**

This unit introduces the topics presented in this course and presents the course agenda. It also provides an opportunity for students to become acquainted with each other and with the instructor.

## **What you should be able to do**

After completing this unit, you should be able to:

- Outline the course objectives
- Give an overview of the course
- List the topics covered by the course

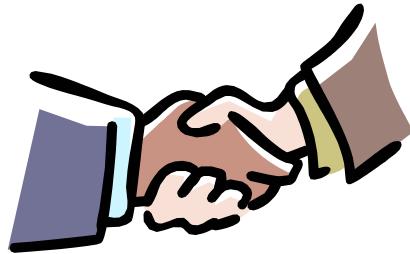


## 1.1 Introduction to SW244

## ***Class introductions***

---

- Instructor
  - Introduction
- Students
  - Please introduce yourself
    - Name and organization
    - Java knowledge
    - Goals you hope to achieve in this course



---

Figure 1-1. Class introductions

SW2446.0

### **Notes:**

## **Course overview**

---

- A five-day course for developers who wish to learn the basic concepts, syntax, and APIs used in developing object-oriented Java applications, and how to create, debug, and run Java applications in IBM Rational Application Developer for WebSphere Software V6
  
- Features covered will include:
  - Overview of object-oriented software development
  - Description of Java syntax for basic programming constructs, classes, objects and methods
  - Outline of how to use threads, collections, streams and serialization in Java, as well as how to use built-in Java classes
  - Overview of how to access relational databases in Java
  - Demonstration of how to use Java development tools in IBM Rational Application Developer for WebSphere Software V6

---

Figure 1-2. Course overview

SW2446.0

### **Notes:**

## **Course objectives**

---

- After completing this course, students will be able to:
  - Outline the principles of object-oriented software development
  - Use Java syntax to create objects, classes and interfaces in Java
  - Use built-in Java classes and methods when writing Java programs
  - Describe and use Java concepts and APIs such as threads, collections, streams and serialization
  - Access relational databases in Java programs using JDBC and DataSources
  - Use the Java development tools in Application Developer to develop and debug Java applications

---

Figure 1-3. Course objectives

SW2446.0

### **Notes:**

## **Course prerequisites**

---

- Students should meet the following prerequisites in order to understand the material covered in this class
  - No previous experience with the Java programming language
  - Experience programming in another language
  - No previous experience using IBM Rational Application Developer
  - Some familiarity with object-oriented programming concepts and design
  - Some experience using relational databases

---

Figure 1-4. Course prerequisites

SW2446.0

### **Notes:**

## Agenda - Day 1

- Module 1 Introduction to SW244
- Module 2 Introduction to the Java Programming Language
- Module 3 Introduction to IBM Rational Application Developer
  - Exercise 1 Introduction to IBM Rational Application Developer*
- Module 4 Object-Oriented Programming
- Module 5 Java Syntax Basics - Part 1
- Module 6 Java Syntax Basics - Part 2

---

Figure 1-5. Agenda - Day 1

SW2446.0

### **Notes:**

## Agenda - Day 2

- Exercise 2 Writing Simple Java Code in Application Developer
- Module 7 Building Classes
  - Exercise 3 Building Classes
- Module 8 Debug Applications
  - Exercise 4 Debugging Java Code

---

Figure 1-6. Agenda - Day 2

SW2446.0

### **Notes:**

## Agenda - Day 3

- Module 9 Inheritance
- Module 10 Design Patterns and Refactoring
  - Exercise 5 Inheritance and Refactoring*
- Module 11 Interfaces
- Module 12 Collections
  - Exercise 6 Collections and Sorting*

---

Figure 1-7. Agenda - Day 3

SW2446.0

### **Notes:**

## Agenda - Day 4

- Module 13 Threads and Synchronization
  - Exercise 7 Threads*
- Module 14 Utility Classes
- Module 15 Exceptions and Exception Handling
  - Exercise 8 Exception Handling*
- Module 16 I/O and Serialization

---

Figure 1-8. Agenda - Day 4

SW2446.0

### **Notes:**

## Agenda - Day 5

---

- Module 17 JavaBeans
  - Exercise 9 JavaBeans*
  - Exercise 10 Serialization*
- Module 18 JDBC Overview
- Module 19 Java Naming and Directory Interface
- Module 20 DataSources
  - Exercise 11 Using JDBC*
- Module 21 Wrap-up

---

Figure 1-9. Agenda - Day 5

SW2446.0

### **Notes:**

# Unit 2. Introduction to the Java Programming Language

## What this unit is about

This unit introduces the various aspects and benefits of the Java programming language, and describes how Java code may be used to build many types of applications and components.

## What you should be able to do

After completing this unit, you should be able to:

- Describe the history and properties of the Java programming language
- Explain the Java execution model, including the use of bytecode and the Java virtual machine
- Outline the types of programs and components that can be built using Java

## How you will check your progress

Accountability:

- Checkpoint questions



## **2.1 Introduction to the Java Programming Language**

## Unit objectives

---

- After completing this unit, you should be able to:
  - Describe the history and properties of the Java programming language
  - Explain the Java execution model, including the use of bytecode and the Java virtual machine
  - Outline the types of programs and components that can be built using Java

---

Figure 2-1. Unit objectives

SW2446.0

### **Notes:**

## **What is Java?**

---

- Java is an object-oriented programming language developed by Sun Microsystems
- Java has a set of standardized class libraries that support predefined reusable functionality
- Java has a runtime environment that can be embedded in Web browsers and operating systems

---

Figure 2-2. What Is Java?

SW2446.0

### **Notes:**

Java shares characteristics of other object-oriented languages such as C++ and Smalltalk, and has the following advantages:

- Thread and synchronization support
- Platform independence
- Built-in security mechanisms
- Built-in networking capabilities
- Facilities for internationalization

## Goals of Java

---

- Java was developed to solve several problems with existing languages
- These issues had an important influence on its development:
  - Object-orientation
  - Portability
  - Performance
  - Security
  - Internationalization
  - Distributed computing
  - Dynamic systems

---

Figure 2-3. Goals of Java

SW2446.0

### **Notes:**

#### **Object Orientation:**

Object orientation has long been considered to be the best paradigm for software development. Java is object-oriented.

#### **Portability:**

A big problem with any compiled language is that different executables are needed for different platforms. The Java virtual machine runtime environment allows source-level and binary-level portability.

#### **Performance:**

Portability has been achieved by interpreted languages such as BASIC. The compromise that has been made is that interpreted languages are dramatically slower than compiled languages because they must be interpreted as they run. The Java solution was the JVM and bytecode, which will be discussed in this unit.

**Security:**

Another problem with portable code is its safety. Java has built-in security mechanisms to help prevent malicious programming. For example, a certain type of Java program called an applet has a much more restricted environment in which to run than does an application.

**Internationalization:**

With portability comes the issue of local preferences, such as language, date and time formats, currencies, and so forth. Java provides facilities for internationalization of programs which may allow interaction between the program and the user's local language.

**Distributed Computing:**

To achieve a Web application environment, the language of choice has to have excellent networking capabilities built in, and Java provides these facilities.

**Dynamic Systems:**

Java uses a large number of separate components which are dynamically interchangeable.

## ***The Java programming language (1 of 4)***

---

- Object-oriented
  - Java supports software development using the notion of objects
  - Software developed using Java is composed of classes and objects
- Network capable
  - Java supports the development of distributed applications
  - Some types of Java applications are designed to be accessed through a Web browser

---

Figure 2-4. The Java programming language (1 of 4)

SW2446.0

### **Notes:**

An object is a unit combining both data and function. This is discussed in the unit on object-oriented programming.

Java has facilities to allow programmers to easily deal with the creation of network connections, and work with network protocols such as HTTP. Applets and Web applications (composed of servlets and JSPs) are two types of Java application that are designed to be accessed through the Web.

## The Java programming language (2 of 4)

- Robust
  - Many aspects of Java promote the development of reliable software
    - Java uses a pointer model which does not allow direct access to memory; memory cannot be overwritten
- Secure
  - Java authentication is based on public-key encryption methods
  - Java's pointer model protects private data in objects and prevents unauthorized applications from accessing data structures

---

Figure 2-5. The Java programming language (2 of 4)

SW2446.0

### **Notes:**

Java does not use pointers, and uses true arrays as opposed to pointer arithmetic. Java's implementation of arrays allows subscript checking to be done. Also, because Java does not use pointers, the end of memory buffers cannot be overwritten, and unauthorized access to memory is prevented.

## The Java programming language (3 of 4)

- Multi-threaded
  - Allows your program to run more than one task at the same time
- Compiled and interpreted
  - Source code is compiled into machine code for the Java virtual machine (JVM) by the Java compiler
  - Machine code for the JVM is also known as bytecode
  - Interpreter of the Java virtual machine interprets and executes instructions

---

Figure 2-6. The Java programming language (3 of 4)

SW2446.0

### **Notes:**

Threads are also described as lightweight processes; they allow for concurrent paths of execution through code.

C and C++ are single-threaded, and allowing applications to have multiple threads can increase the difficulty of writing a program, as such programs must handle concurrent paths of execution through code. Java includes facilities to deal with synchronization.

The Java virtual machine is a piece of software that acts as a virtual CPU, interpreting bytecode instructions and then executing these instructions. Bytecode is platform independent, while the Java virtual machine is specific to a platform.

## The Java programming language (4 of 4)

- Architecture neutral
  - Bytecode instructions are architecture neutral because they run on the JVM, and are not specific to an architecture
  - The same application runs on all platforms, provided the Java virtual machine is installed on that platform
- Portable at source and binary level
  - One piece of source code gets compiled into one set of bytecode instructions for the JVM, and can be run on any platform and architecture without recompiling the code

---

Figure 2-7. The Java programming language (4 of 4)

SW2446.0

### **Notes:**

The Java language is not specific to any computer platform, which allows Java programs to be portable. For example, primitive data types, such as integers and floats, have an identical definition over all platforms.

Though source and binary code are portable between operating system platforms, the JVM is specific to the operating system where the program runs.

## **Java evolution**

---

- Java is a relatively young language
  - It has been in use since 1995
  - It was originally designed for consumer electronic devices
- Java has a huge developer base
  - There is a vast collection of libraries (from Sun and other sources)
- IBM Rational Application Developer V6 supports Java 2, Standard Edition, version 1.4.2 (J2SE 1.4.2)
- Application Developer V6 also supports Java 2 Enterprise Edition (J2EE)
  - J2EE includes a wider range of functions required for enterprise programming

---

Figure 2-8. Java evolution

SW2446.0

### **Notes:**

The most recent version of the JDK is Java 1.5, also known as Java 5. JDK 5.0 introduces new advanced features to the Java language.

## The Java platform

- A platform is a development or deployment environment
- The Java platform runs on any operating system
  - Other platforms are hardware and vendor specific
- The Java platform provides:
  - The Java virtual machine (JVM)
  - Application Programming Interface (API)

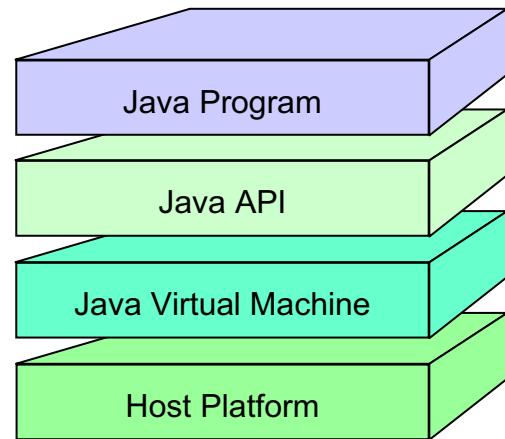


Figure 2-9. The Java platform

SW2446.0

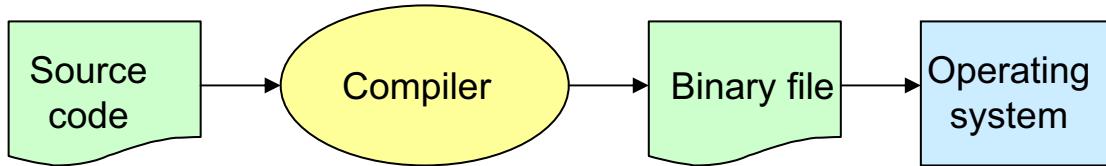
### **Notes:**

Java programs make use of the Java API; when completed, programs are compiled into bytecode. This bytecode is interpreted by the Java virtual machine, and that same bytecode can be interpreted by Java virtual machines residing on other computers of different platforms or architectures.

A Java program runs on the Java virtual machine; the JVM runs on a real machine, on the host platform.

## The Java execution model

- Compiled language execution model
  - Source code is compiled into instructions understood by the host machine; host machine executes this binary file



- Java execution model
  - Source code is compiled into instructions understood by the JVM; bytecode is then interpreted by the JVM

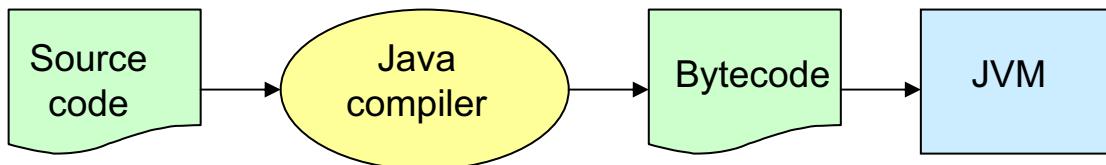


Figure 2-10. The Java execution model

SW2446.0

### Notes:

In a compiled language, a compiler translates the source code of a program into a binary file of machine code. The machine code is understood and executed by the hardware and the operating system.

Java, however, is compiled into bytecode; bytecode is understood and executed by the Java virtual machine. The Java virtual machine runs on an operating system, and the bytecode is interpreted by the JVM. Java source code is written once, and can be run in any environment which has the Java virtual machine installed. The source code and bytecode remain the same no matter where the program is to be run.

## **Uses of Java**

---

- Java can be used to build programs and software components
- Programs are stand-alone entities that can run on the Java virtual machine
  - Applications
  - Applets
- Components are building blocks used to create programs
  - Servlets
  - JavaServer Pages (JSPs)
  - JavaBeans
  - Enterprise JavaBeans (EJBs)

---

Figure 2-11. Uses of Java

SW2446.0

### **Notes:**

A component is a building block of larger programs. Ideally, programs are built from components, and components are designed to be reusable by many programs.

## **Programs**

---

### **•Application**

- A stand-alone program that can access system resources such as files
- Does not need to run in a Web browser
- Is explicitly invoked through the command line or menu selection
- The method `main()` is the entry point for an application

### **•Applet**

- A Java program that is embedded within a Web page; almost always graphical
- Security limits access to system resources
- Code executes on the client inside a Web browser

---

Figure 2-12. Programs

SW2446.0

### **Notes:**

An applet is stored on a server, and is transmitted to the client when its HTML <APPLET> tag is accessed. Applets are usually visual programs and have limited access to the client operating system in order to preserve client security. Also, because Java does not use pointers, memory addresses cannot be directly accessed. Bytecode control flows and potential access violations are checked before being executed. Finally, a security manager can be set to implement various security policies, and the security manager is used by the JVM each time the operation being performed may be insecure; the security manager has the authority to disallow an operation that violates its policies.

## **Components (1 of 2)**

---

### **•Servlet**

- Handles requests from the Web browser and returns responses
- Creates dynamic content on the server
- Runs inside an application server

### **•JavaServer Page (JSP)**

- HTML page embedded with Java code
- Creates dynamic content on the server instead of on the browser
- Runs inside an application server

---

Figure 2-13. Components (1 of 2)

SW2446.0

### **Notes:**

An application server is a special server which allows you to run enterprise Java components such as servlets, JSPs, and EJBs.

#### **Servlet:**

Servlets are commonly used to interact with Web sites. Typically, a set of HTML pages interacts with the user, and then sends the collected data to the servlet to be processed. It usually interacts with database and other server-side components. Servlets are capable of writing HTML dynamically.

#### **JavaServer Page:**

JSPs are a way of creating HTML dynamically as well. They are in essence HTML pages with special tags that represent Java functions.

Servlets and JSPs are Web components that are a part of the Java 2 Enterprise Edition (J2EE) architecture. Servlets and JSPs require special support, and run on a platform, called a Web container, that provides this support.

Servlets and JSPs are the building blocks of enterprise Java applications. Though they run the same way and can provide the same functionality, the way servlets and JSPs are coded in Java lends these components to different tasks within an application.

## **Components (2 of 2)**

---

### **•JavaBeans**

- Java code that has its properties, methods, and events exposed to promote reuse among developers
- Reusable software component that can be manipulated visually in a builder tool

### **•Enterprise JavaBeans (EJB)**

- Distributed objects that allow communication between Java objects in different JVMs
- Encapsulate the business logic and model of an application
- Run inside an application server

---

Figure 2-14. Components (2 of 2)

SW2446.0

### **Notes:**

JavaBeans are a software component that can be reused throughout the Java language by many other types of Java programs and components: JSPs, EJBs, and so forth. A JavaBean is simply a class that has been written in a manner that exposes its properties, methods and events in a standard way; it is a class that adheres to certain standards to promote reuse.

JavaBeans have many different uses, including being used within enterprise Java applications. Some JavaBeans have a visual representation at run time, while others do not. One use of JavaBeans is to store information that must be passed within an application and allow it to be retrieved by different parts of the application.

EJBs are server-side components that are a part of the Java 2 Enterprise Edition (J2EE) architecture. These components are run on a Java-enabled Web server; they also require special support, and are run on a platform, called an EJB container, that provides this support.

## **Checkpoint**

---

1. What is the JVM and what does it do?
2. What is bytecode?
3. What is the difference between a program and a component?
4. What are the differences between an application, an applet and a servlet?



---

Figure 2-15. Checkpoint

SW2446.0

### **Notes:**

## **Web resources**

---

- IBM Web sites
  - <http://www.ibm.com/WebSphere>
  - <http://www.ibm.com/software/awdtools/developer/application/index.html> (IBM Rational Application Developer for WebSphere Software)
  - <http://www.ibm.com/developerworks> (developerWorks)
  - <http://www.redbooks.ibm.com/>
- Sun Web sites
  - <http://java.sun.com/j2se/> (Standard Edition)
  - <http://java.sun.com/j2ee/> (Enterprise Edition)
  - <http://java.sun.com/j2se/1.4.2/docs/api/> (JavaDoc)
  - <http://java.sun.com/products/javabeans/> (JavaBeans)
- Kyle Brown's public Web site
  - Lots of information on various topics
  - <http://hometown.aol.com/kgb1001001/index.html>
- Java Wiki
  - Community-modifiable Web site
  - <http://www.c2.com/cgi/wiki?TopicJava>
    - Get started by clicking “LikePages” or “FindPage” and then typing “Java” as a search parameter

---

Figure 2-16. Web resources

SW2446.0

### **Notes:**

## Recommended reading

---

- *Java Programming: Advanced Topics*, Joe Wigglesworth and Paula McMillan, Third Edition, Thompson Press



- *Java in a Nutshell*, David Flanagan, O'Reilly Press



Figure 2-17. Recommended reading

SW2446.0

### **Notes:**

## Unit summary

- In this module, you should have learned to:
  - Describe the history and properties of the Java programming language
  - Explain the Java execution model, including the use of bytecode and the Java virtual machine
  - Outline the types of programs and components that can be built using Java

---

Figure 2-18. Unit summary

SW2446.0

### **Notes:**



# Unit 3. Introduction to IBM Rational Application Developer

## What this unit is about

This unit introduces the workbench of IBM Rational Application Developer and discusses the various features available through the perspectives, views and menus of Application Developer.

## What you should be able to do

After completing this unit, you should be able to:

- Explain the use of a workspace
- Identify the parts of the workbench
- Navigate the workbench
- Work with perspectives, views and editors
- Create resources using wizards
- Understand what help systems are available in the workbench

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## **3.1 Introduction to IBM Rational Application Developer**

## **Unit objectives**

---

- After completing this unit, you should be able to:
  - Explain the use of a workspace
  - Identify the parts of the workbench
  - Navigate the workbench
  - Work with perspectives, views and editors
  - Create resources using wizards
  - Understand what help systems are available in the workbench

---

Figure 3-1. Unit objectives

SW2446.0

### **Notes:**

## What is a workspace?

- The workspace is a collection of resources that a developer uses to create applications
  - Projects, folders, files
- Multiple workspaces are used to separate projects
- When Application Developer opens, it will prompt the user for the address of a workspace
- The workspace can only be opened by one user at a time

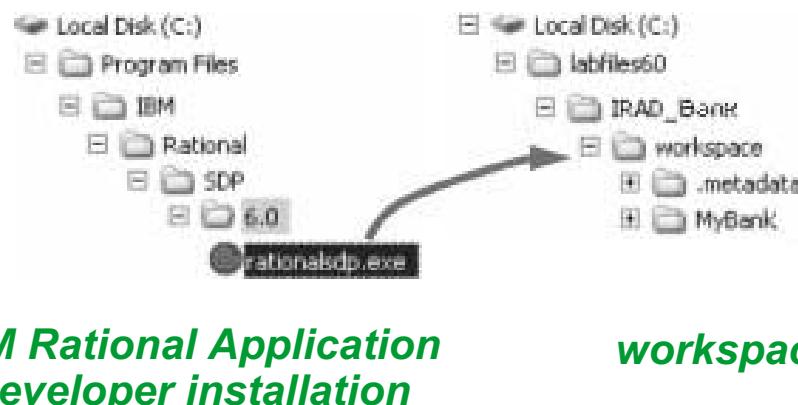


Figure 3-2. What is a workspace?

SW2446.0

### Notes:

Files and folders are just like files and directories in the file system. A folder contains other folders or files. A file contains an arbitrary sequence of bytes. Its content is not interpreted by the platform.

The workspace can be located anywhere, but will typically be separate from the Application Developer installation. The resources associated with a project do not have to be located with the workspace.

Rationalsdp.exe can only link to one workspace at a time. Several instances of Application Developer can be running simultaneously (depending on memory).

When Application Developer is launched, a dialog will ask you to specify the workspace location. If you give the name of a workspace that does not exist, it will be created. Note that this dialog can be disabled by check box, and re-enabled through **Window -> Preferences -> Workbench -> Startup and Shutdown**.

## Switching workspaces

---

- To switch the location of the workspace associated with the workbench:
  - Select **File -> Switch Workspace**
    - The workbench will exit and re-open, even if it is to the same workspace
  - At launch use the `-showlocation` directive
    - <install path>\rationalsdp.exe" -  
`showlocation`
    - This will add the workspace location to the title bar

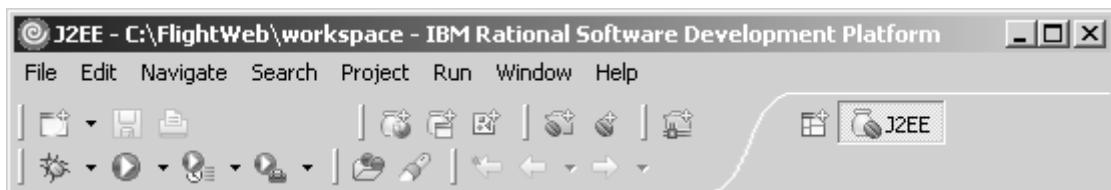


Figure 3-3. Switching workspaces

SW2446.0

### **Notes:**

## **What is the workbench?**

---

- The term *workbench* refers to the desktop development environment used in Application Developer
- Each Application Developer workbench window contains one or more perspectives
  - You can only interact with one perspective at a time in each workbench window
- The workbench gives access to all of the resources of the workspace

---

Figure 3-4. What is the workbench?

SW2446.0

### **Notes:**

The workbench is an integrated development environment. In order to do useful work, it must link to a workspace containing the collection of resources that the developer wants to work with.

## Parts of the workbench

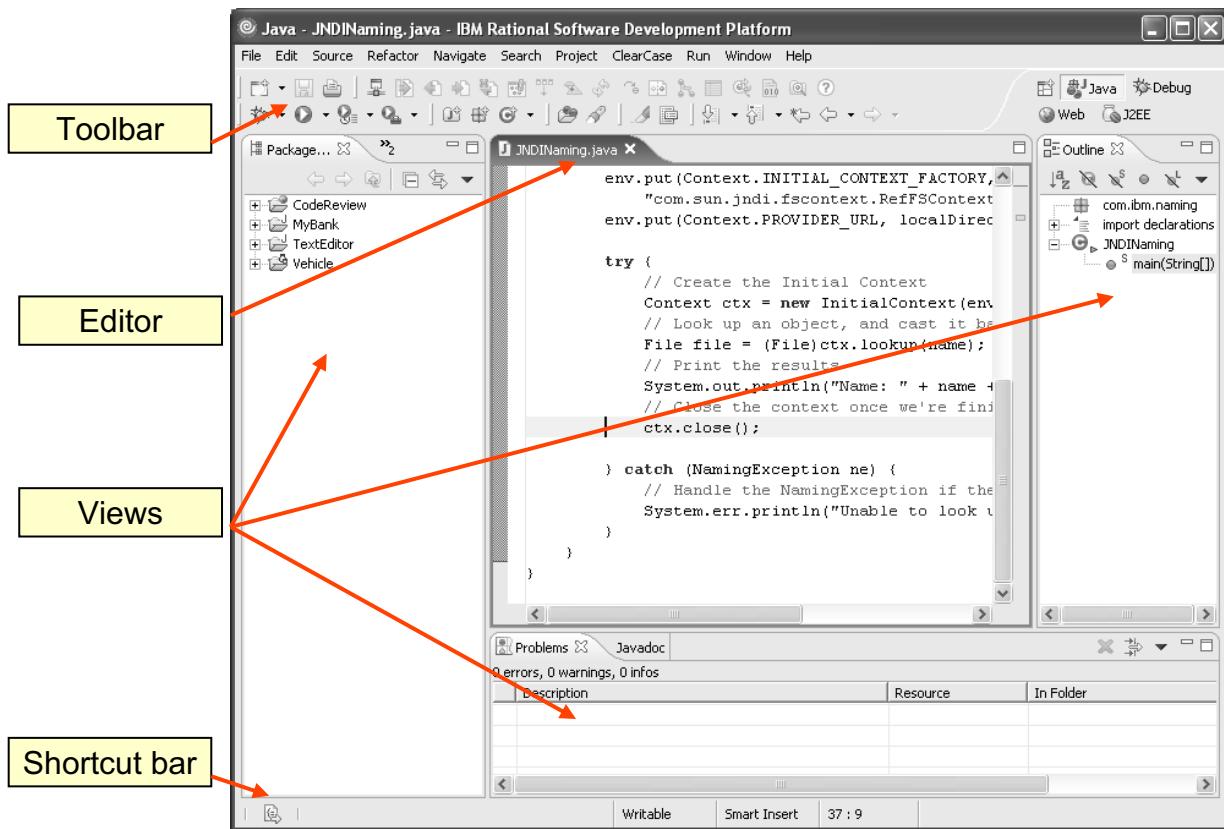


Figure 3-5. Parts of the workbench

SW2446.0

### Notes:

## Perspectives, views and editors

- A perspective is a collection of views, toolbar icons and menus that are shown together in order to accomplish a specific type of work
- A view shows resources in a way that will be useful for a particular task
- Editors allow modification of code

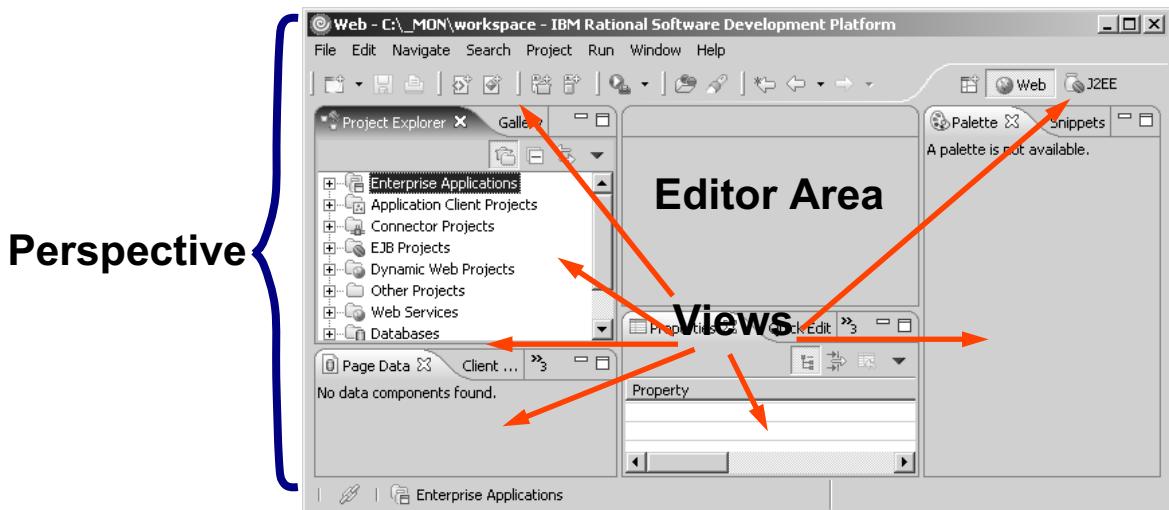


Figure 3-6. Perspectives, views and editors

SW2446.0

### Notes:

Every perspective has a number of views; most have an editor area as well. The views typically provide information that supports the content of the editor.

## Perspectives

- The perspective defines the initial set and layout of views in the workbench window
- Each perspective provides a set of functionality aimed at accomplishing a specific type of task or works with specific types of resources
- Perspectives control what appears in certain menus and toolbars
  - They define visible action sets, which you can change to customize a perspective

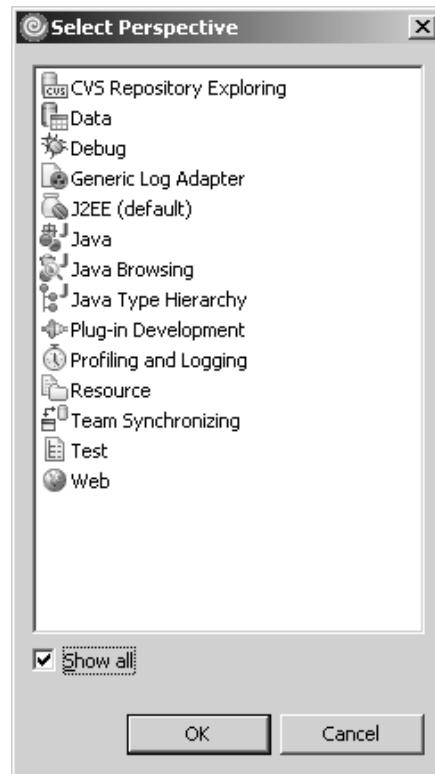


Figure 3-7. Perspectives

SW2446.0

### Notes:

Perspectives provide the specific set of tools needed for a certain development role. For example, when developing a Java project, you would typically use the Java perspective, as it provides a set of views that are geared toward developing Java projects. Likewise, if you were developing an Enterprise Java project (J2EE project), you would likely use the J2EE perspective, as it has a set of views that display information pertinent to this development role. This ensures that you see only information relevant to your development task, and avoid cluttering your screen with useless information.

Perspectives share the same set of editors.

You can set your workbench preferences to open perspectives in the same window or in a new window.

There are 14 predefined perspectives.

## Customizing perspectives

---

- The **Windows -> Customize Perspectives** menu allows you to customize the current perspective
- You can define four aspects of the current perspective:
  - **File -> New**
    - Adds or removes wizards that appear under this menu
  - **Window -> Open Perspective**
    - Adds or removes perspectives that appear under this menu
  - **Window -> Show View**
    - Adds or removes views that appear under this menu
  - **Others**
    - Adds or removes:
      - Wizards that appear in other menus
      - Action sets that appear on the menu bar and toolbar

---

Figure 3-8. Customizing perspectives

SW2446.0

### **Notes:**

Customizing a perspective also involves moving, resizing, opening and closing views.

## Saving perspectives

---

- After creating your own custom arrangement of views, save it as a new perspective
  - Select **Window -> Save Perspective As**

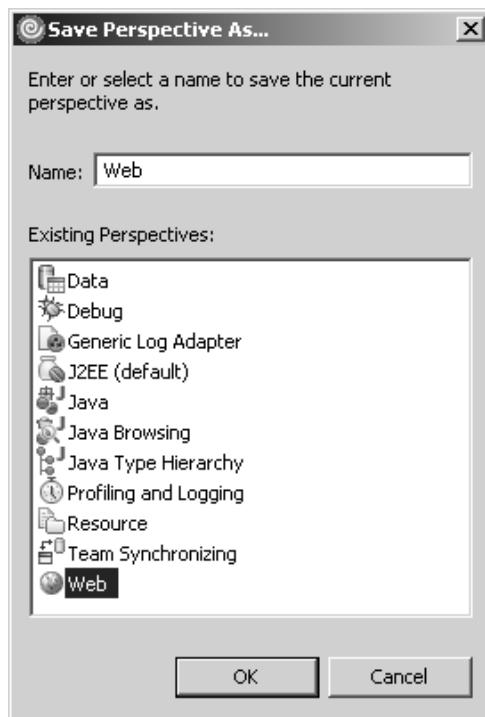


Figure 3-9. Saving perspectives

SW2446.0

### **Notes:**

If you attempt to overwrite an existing perspective, you will be prompted to ensure that you mean to do so. If you do accidentally overwrite a perspective, you can reset the perspective to its default settings by selecting **Window -> Preferences**, expanding **Workbench -> Perspectives**, and then selecting the perspective you wish to restore and clicking the **Reset** button.

## Views

- Views display information and resources relevant to a specific task; they provide information that supports editors
- There are two states for views:
  - Single
    - There are no other views at that position in the perspective
  - Stacked
    - There are other views available represented by tabs at the top of the view pane

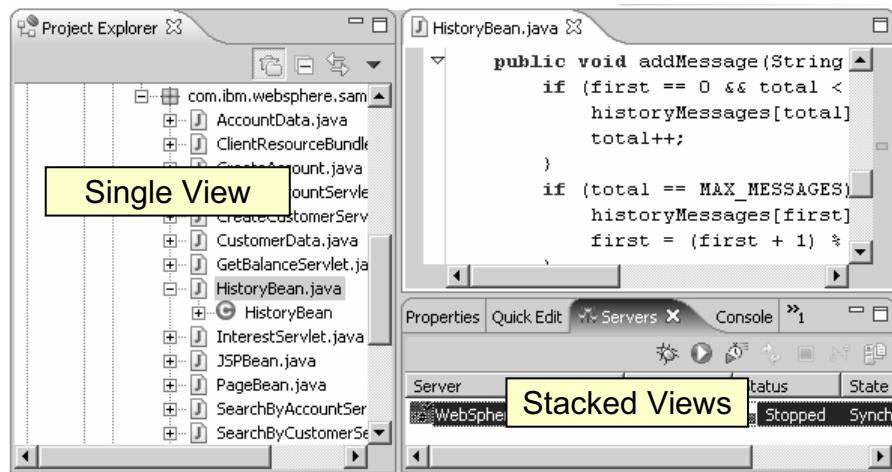


Figure 3-10. Views

SW2446.0

### Notes:

The **>>1** to the right of the stacked views shows that there is one more view open, whose tab is not visible. Clicking the chevrons will list the views that are not being displayed, and allows you to search for a view that you would like to open.

A view may have its own menu items. The Outline view, for example, has items to sort members, hide fields, and others.

Double-clicking a resource in a view will open it in the editor area, if there is an editor available for that resource.

## Displaying views

- Any view can be added to any perspective by selecting **Window -> Show View** from the menu
- If a view is not shown in the list, select **Other...**
  - In the window that opens, views are arranged by type (Basic, Java and so on)
- Views can be rearranged
  - Drag the view by the title bar to the new location
  - The position of a view is a property of the perspective

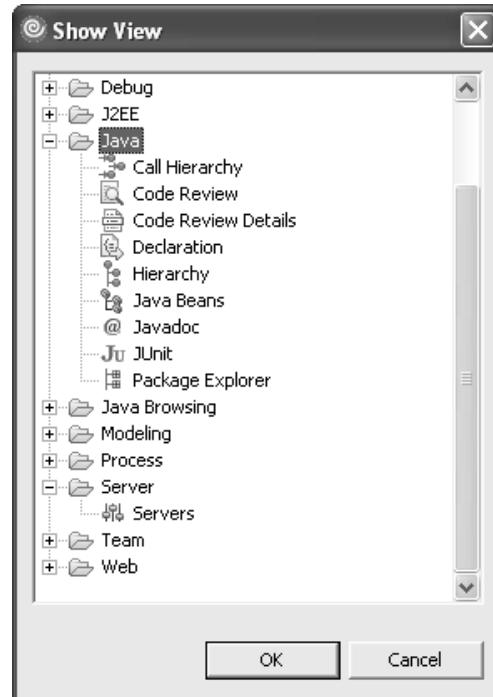


Figure 3-11. Displaying views

SW2446.0

### Notes:

Views can be dragged to be shown on their own, or stacked with other views. The cursor will change to show where a view can be dropped.

## Fast views

- Fast views can quickly be opened and closed
  - Available via the fast view bar (part of the shortcut bar)
- Two ways to create a fast view:
  - Drag and drop a view onto the shortcut bar
  - Right-click the view icon and choose **Fast View**
- To restore a fast view, right-click the icon in the shortcut bar, and uncheck **Fast View**

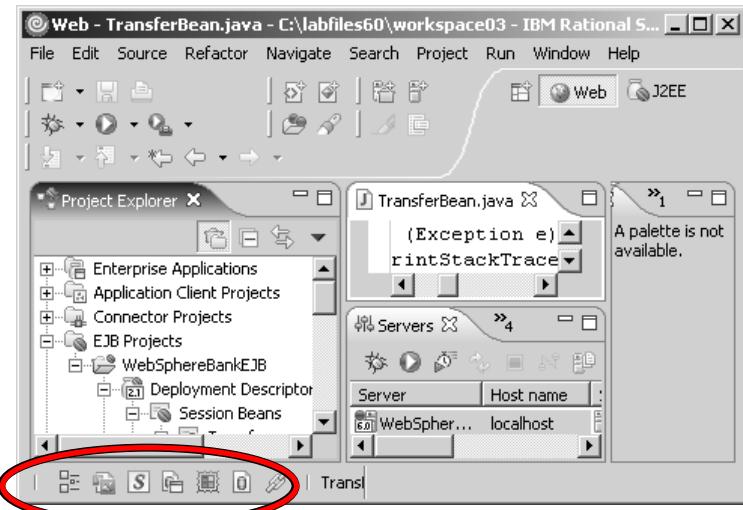


Figure 3-12. Fast views

SW2446.0

### **Notes:**

Fast views work like other views except they do not take up space in your workbench window.

## Editors

- Several editors can be opened at the same time
  - Editors can be **stacked** or **tiled** for simultaneous viewing
- An asterisk (\*) preceding the name of a file indicates that the contents have been modified but not saved
- When an editor is active, the workbench menu bar and tool bar contain operations applicable to that editor

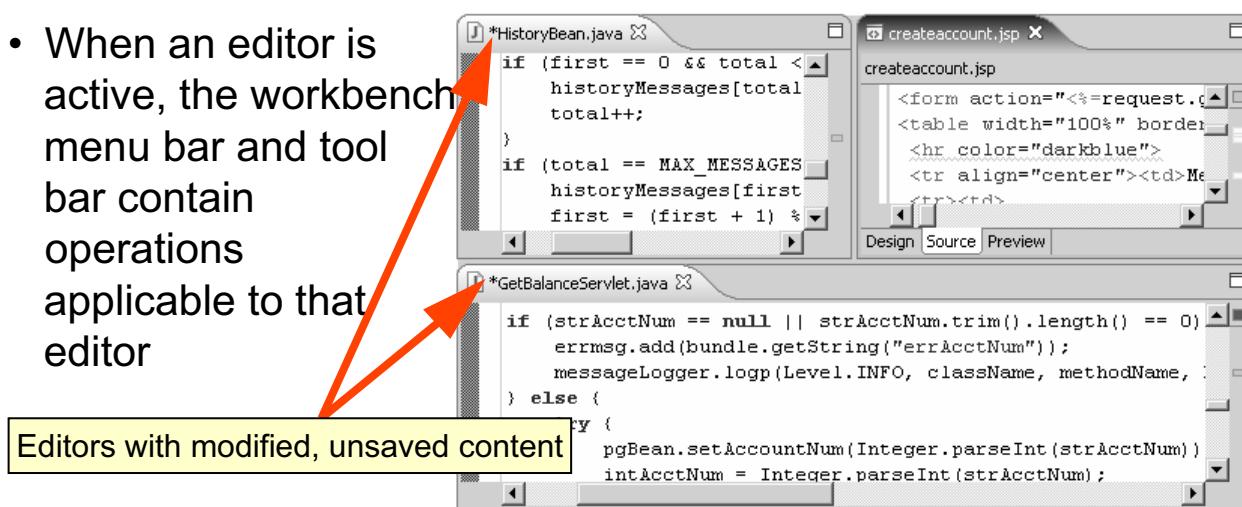


Figure 3-13. Editors

SW2446.0

### Notes:

In any given perspective there is one editor area. The area can contain multiple editors. The vertical ruler appears to the left of the editor, and displays items such as breakpoints and tasks. The overview ruler appears to the right of the editor, and displays items such as errors.

The rectangles in the overview ruler to the right of each editor have the following significance:

**Blue:** There is an auto-generated TODO message.

**Yellow:** There is a problem with the location of some attribute, such as an invalid tag location.

**Red:** There is a syntax error. This will be reflected in the Problems view.

Hovering over each type of indicator will show a pop-up with the nature of the problem.

## Capabilities

- The **Capabilities** preference page allows you to enable or disable various product components
  - For example, Java development, or XML development
- Some capability selections have dependencies on others

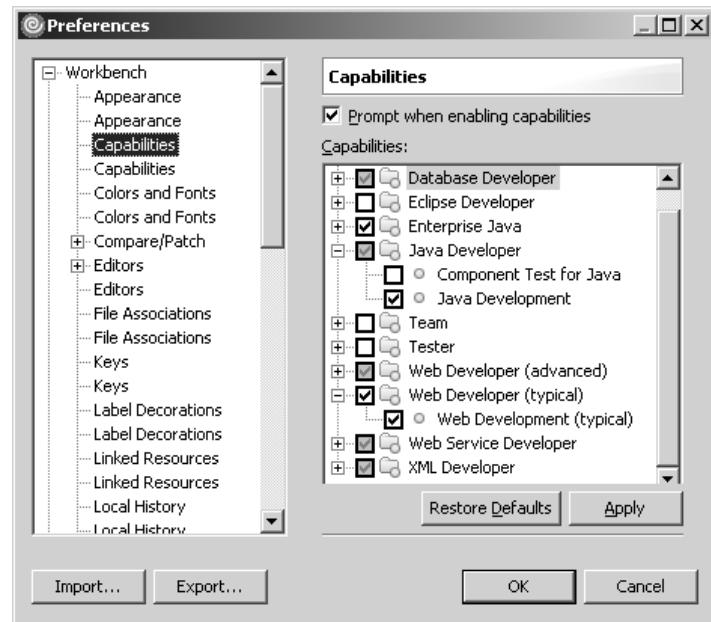


Figure 3-14. Capabilities

SW2446.0

### Notes:

Proper use of capabilities can prevent your workspace from being cluttered with tools unrelated to your development tasks. They are built around the idea of roles in a development team (business analyst, architect, developer, tester, deployment manager, and others).

You do not need to manually enable capabilities. They can be enabled as you use the tools by selecting the **Show all** check boxes that display in various wizards, and then selecting an option that was previously hidden, at which time you will be prompted if you want to enable the related capability. However, it may be more convenient to enable all of the required capabilities before you begin working so that all of the relevant tools and menu items are readily available when you need them. In this course, you will not need to enable additional capabilities.

## Resources

---

- There are three types of resources in Application Developer:
  - Projects
    - Projects can hold folders and files
  - Folders
    - Folders in the workspace represent directories in the file system
  - Files
    - Files in the workspace represent files in the file system
- You can open a file resource in the workbench or open the file on your file system with any available editor

---

Figure 3-15. Resources

SW2446.0

### Notes:

## **Projects in Application Developer**

---

- All resources are held in projects
- You can create projects with different structures, according to the area of development (Java, Web, EJB, and others)
  - Note that there is no XML or database project; these resources would not stand alone, but would be part of some other project
- Projects can be created using the wizards provided by Application Developer
- Projects can be deleted from the workbench or the workspace
  - The default is to delete from the workbench only

---

Figure 3-16. Projects in Application Developer

SW2446.0

### **Notes:**

By default, projects are stored with the workspace to which they are related. A different location can be specified when the project is created.

Deleting a project only from the workbench can improve the performance of Application Developer because it takes up no memory and is not examined during builds.

## Create a new project

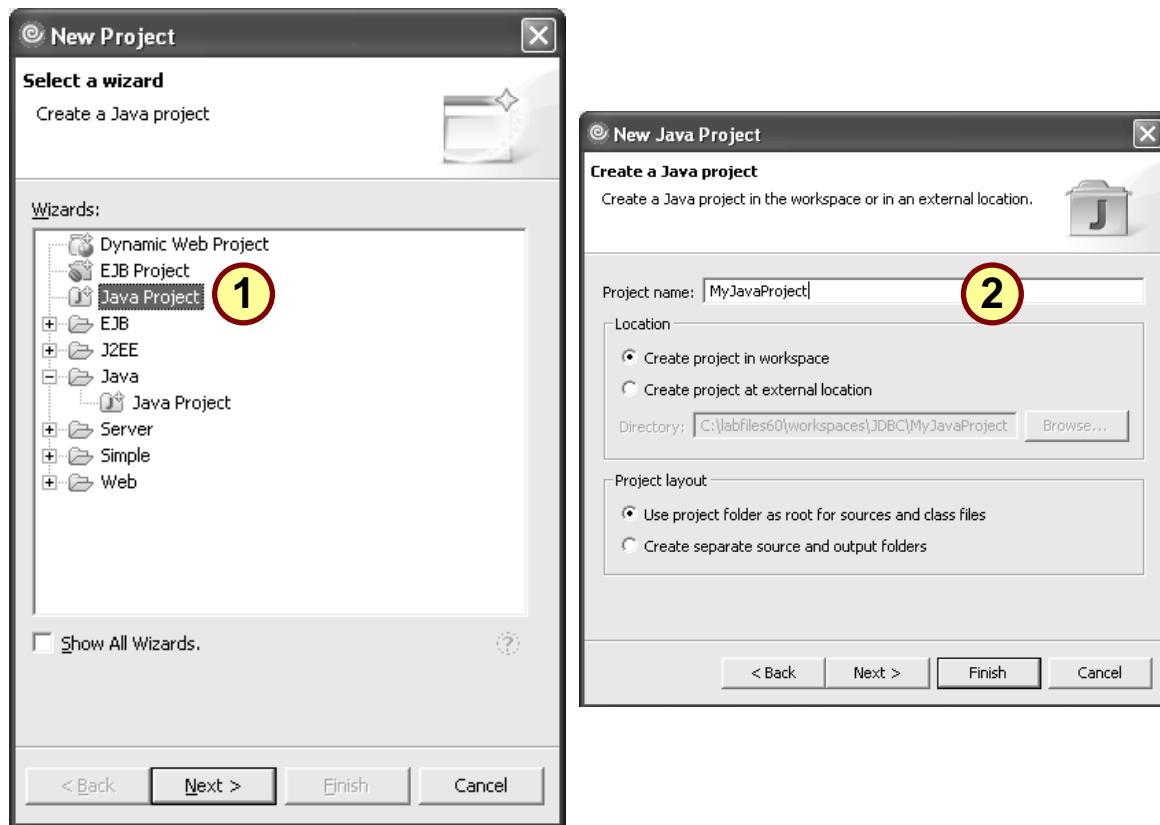


Figure 3-17. Create a new project

SW2446.0

### Notes:

1. Open the New Project wizard from **File -> New -> Project**. Note that the **Show All Wizards** button is available. The list in the left image is not complete because several capabilities have not been set yet. Those wizards are therefore hidden unless the developer explicitly checks the box.
2. Give the project name. The default location is the workspace.
3. You can click **Finish** to create the Java project, or click **Next** to specify additional Java settings, such as the source folders on the build path, libraries that will be used by the project, and so on.

## Local history

- A history of each file is stored locally
- Each **save** operation creates a new edition in the local history
  - No formal repository system is needed
  - You can compare and replace with previous versions
  - History is kept for a limited period of time
    - Configure via the **Window -> Preferences -> Workbench -> Local History**
  - You can set:
    - How long the history files are kept
    - The maximum number of entries in a file
    - The maximum size for a file

Days to keep files:	<input type="text" value="7"/>
Entries per file:	<input type="text" value="50"/>
Maximum file size (MB):	<input type="text" value="1"/>

Figure 3-18. Local history

SW2446.0

### **Notes:**

Local history of a file is maintained when you create or modify a file. This allows you to compare your current file state to a previous state, or replace the file with a previous state. Each state in the local history is identified by the date and time the file was saved.

## Replace with local history

- Local history can be used to return to a previous state in the code
  - You may decide that the code you wrote today is not good, and want to return to yesterday's state
  - A section may have been accidentally deleted

- History is date/time stamped

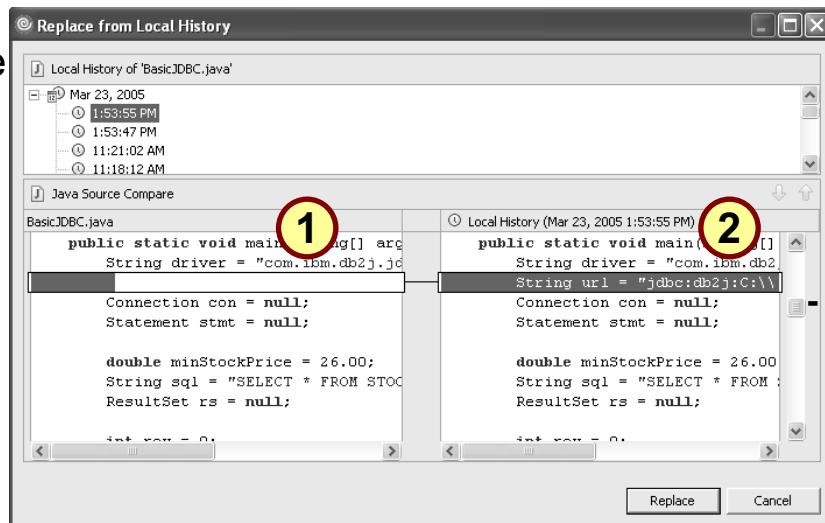


Figure 3-19. Replace with local history

SW2446.0

### Notes:

1. The code as it stands in the workbench at the moment.
2. The code as it is stored in local history.

Differences are shown in two ways:

- a. On a gray background
- b. By a connector between the left and right panes

To replace code with the code from the local history, right-click the file in the Package Explorer view and select **Replace With -> Local History...** from the pop-up menu.

## Comparing files

- To compare two files, highlight any two files, right-click them, and select **Compare With -> Each Other** from the pop-up menu
  - Compare one file to its local history by selecting **Compare With -> Local History...** instead
- You may move information from one file to the other

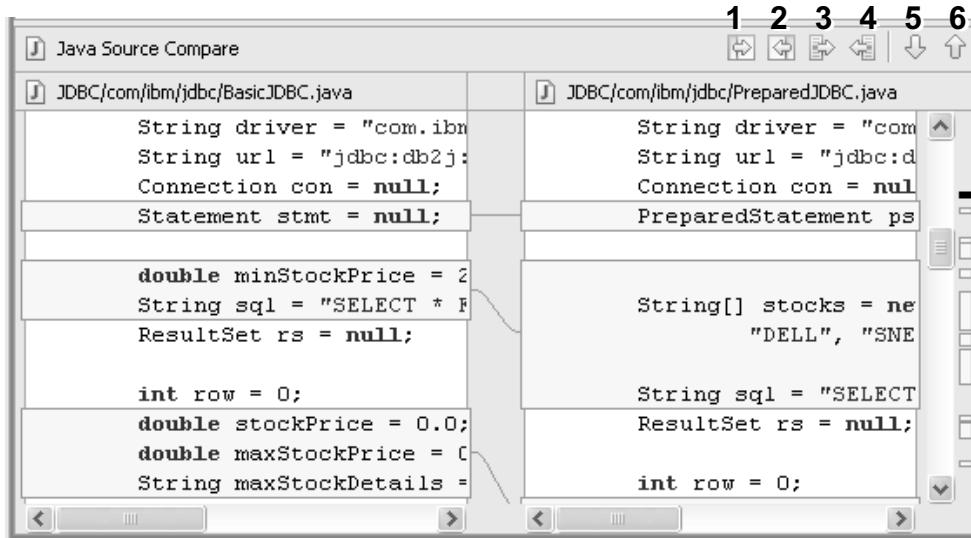


Figure 3-20. Comparing files

SW2446.0

### **Notes:**

Comparing different files only makes sense if the files are very similar. In this image, two files that access a database are compared, and numerous differences are shaded and connected.

It is possible to merge changes from one file to the other. There are four types of merge:

1. Copy all from left to right
2. Copy all from right to left
3. Copy current change from left to right
4. Copy current change from right to left

The last two icons (5 and 6) allow you to navigate between the differences.

## ***Importing and exporting resources***

---

- In Application Developer, you can import and export all major file types
  - JAR, WAR, RAR, ZIP and many others
- Resources from the file system can be dragged and dropped into a directory in the workbench
  - The resource cannot be released onto a blank area of the view; it must be positioned in a folder or a package
- You can also drag and drop from Application Developer to the file system

---

Figure 3-21. Importing and exporting resources

SW2446.0

### **Notes:**

Import and Export wizards can be accessed from both the file menu and the pop-up menu. The advantage of using the pop-up menu item is that the destination folder is automatically filled in, according to where you clicked.

There are two points to note:

1. You can export both a JAR and a ZIP file, but there is no import JAR option. The wizard will detect that a ZIP file contains a manifest file, and will recognize it as a JAR. Import JARs as ZIPs.
2. If you want to place the ZIP or JAR file as-is in your project (for example, you are importing a utility JAR), use the File System import feature. This will place the file in your project at the point you specify. If you use the ZIP import, the file will be opened and the contained files will be placed in your project using their recorded path.

## Search

- Start a search by selecting **Search** from the workbench menu

- The results of a search can be seen in the **Search** view

- Previous search results are stored, and can be retrieved in the **Search** view or **Classic Search** view

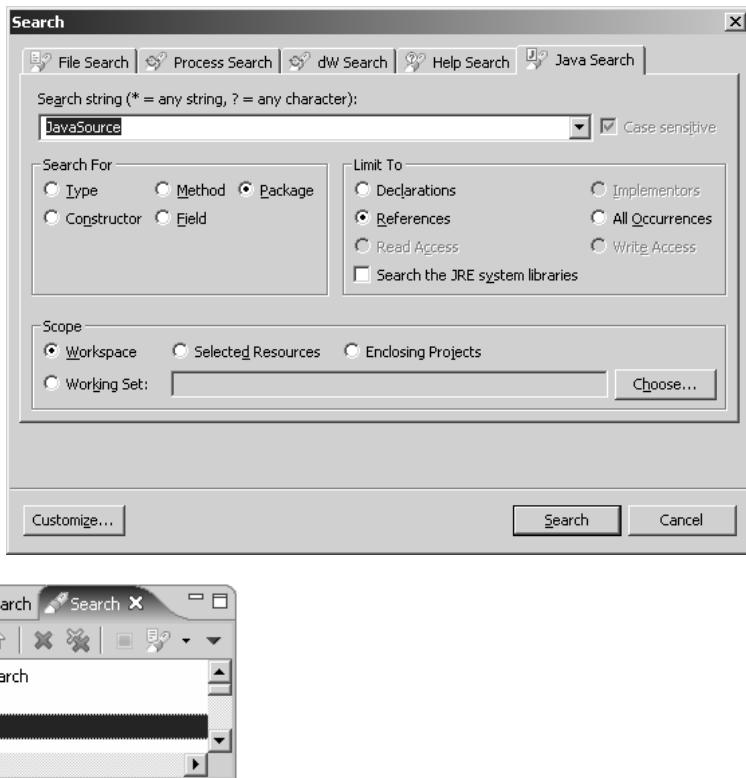


Figure 3-22. Search

SW2446.0

### Notes:

There are five types of search:

- File search* looks for text in files or file names (regular expressions can be used).
- Process search* returns information about the Rational Unified Process. Results appear in the Classic Search view.
- dW search* will look on developerWorks, return the results to the Search view, and open a browser to the Web site article if the user double-clicks an item. Results appear in the Classic Search view.
- Help Search* will find matches in the Application Developer Help.
- Java Search* allows you to quickly find references to, and declarations of, Java elements.

## Content assist

- Content assist is a source editing tool that prompts you with a list of valid alternatives for completing the current line of code
  - This makes programming easier and less error prone
- Content assist can be invoked by pressing **Ctrl+Space**
- In some cases, a second dialog is displayed giving information about the highlighted option

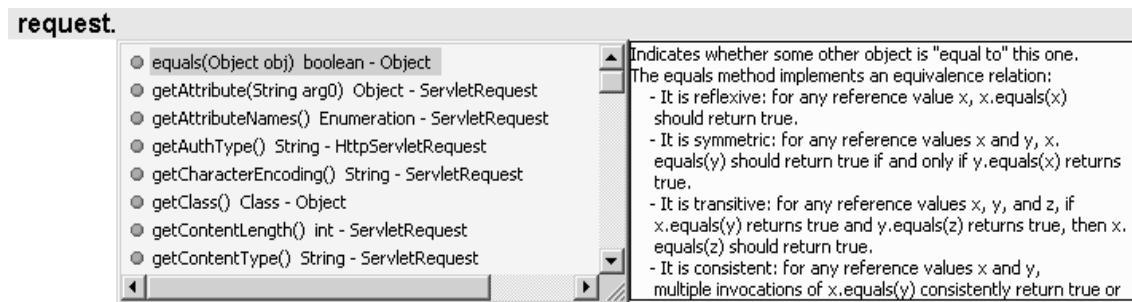


Figure 3-23. Content assist

SW2446.0

### Notes:

Most of the structured text editors have content assist.

## Tasks

- A task is a user-defined statement that can be either associated or unassociated
  - Associated: Referencing a line in a resource
  - Unassociated: Not related to any resource
- A task is associated by opening the resource in an editor, right-clicking in the vertical ruler and selecting **Add Task...**

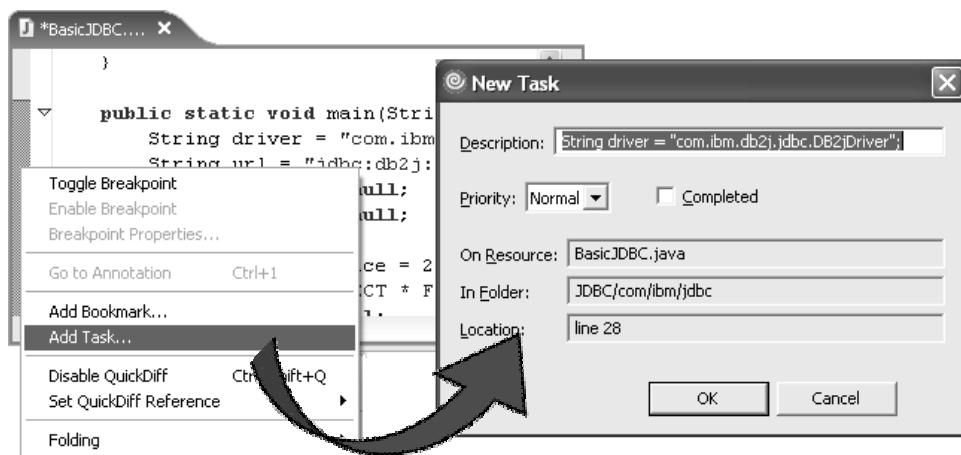


Figure 3-24. Tasks

SW2446.0

### Notes:

The location of the task (the line and file with which it is associated) is filled in automatically. The description is, by default, the code that is on that line (this can be replaced with a message).

Priorities are High, Normal and Low. The **Completed** box can be checked here to indicate that the task has been resolved (for example, a developer may simply want to indicate to the team that the task discussed during a meeting has been addressed).

Both priority and completion are indicated in the leftmost columns of the Tasks view.

## Problems

- If problems are detected, a message is logged and displayed in the **Problems** view
- There are three types of problem:
  - Errors
  - Warnings
  - Information
- These messages cannot be removed manually

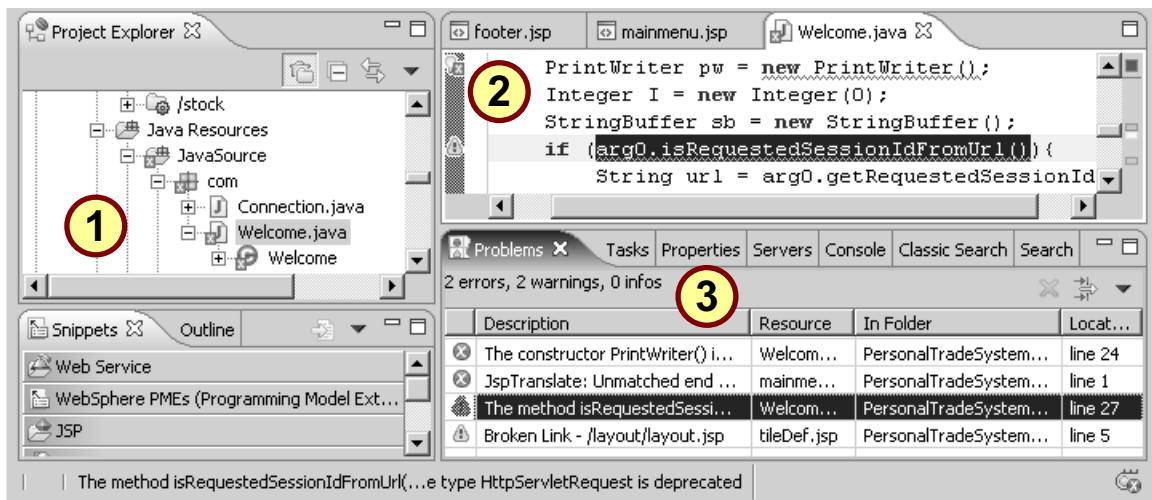


Figure 3-25. Problems

SW2446.0

### Notes:

1. If there is a problem in a class, the package by definition also has a problem, and the Java source folder therefore shows the problem marker as well. There is no indication of how many problems there are in the class.
2. The error is marked by a red circle with a white cross. Hovering over the marker brings up a pane showing the nature of the error.

The warning is marked by a yellow triangle.

The light bulb behind the error marker indicates that Application Developer has a suggested correction (in this case, it suggests adding an argument to match one of the constructors of `PrintWriter`).

3. The Problems view shows the resource which contains the problem and a description. If the description is not completely visible, you can select one row and read the description at the bottom of the window. These messages are deleted only by correcting the problem.

## Help topics: cheat sheets

- A cheat sheet helps you complete some task
  - It lists the sequence of steps required to help you achieve the task
- As you progress from one step to the next, the cheat sheet will automatically launch the required tools for you
- If there is a manual step, you perform the task, and then click a button in the cheat sheet to move on to the next step
- Cheat sheets are accessed from the **Help** menu

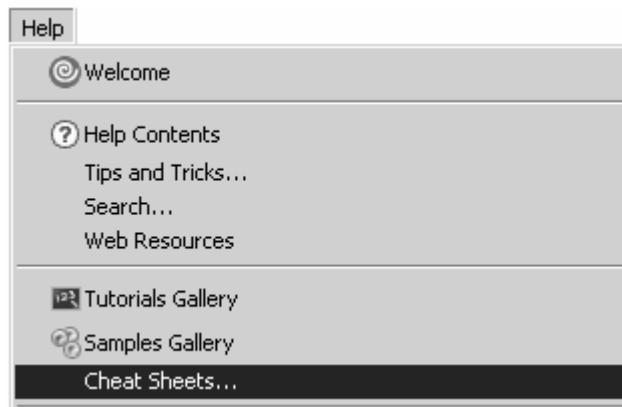


Figure 3-26. Help topics: cheat sheets

SW2446.0

### **Notes:**

Cheat sheets are special views that help guide the user through a series of complex tasks to achieve an overall goal. For example, a cheat sheet could be used to help guide the user through all of the steps needed to create, compile, and run a simple Java program.

The cheat sheet opens as a view. At any time, only one cheat sheet is open and active.

## Help topics: samples gallery

- The samples gallery provides practical code examples for application developers
  - Sample applications that are ready to run in your environment are provided for developers for illustration and learning purposes
- Samples gallery is accessible from the **Help** menu
- Three categories of samples are provided:
  - Showcase samples
  - Application samples
  - Technology samples

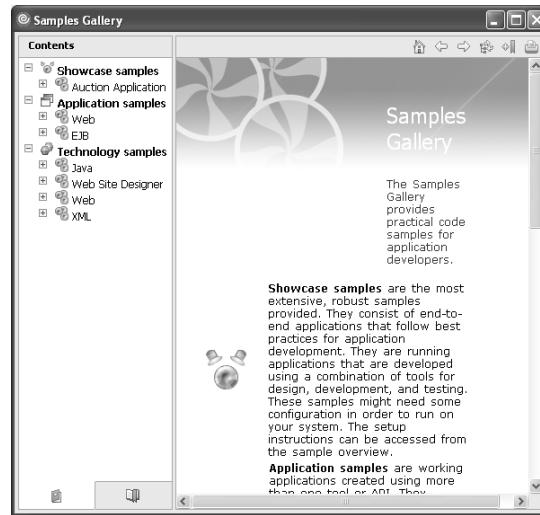


Figure 3-27. Help topics: samples gallery

SW2446.0

### Notes:

*Showcase samples* are the most extensive, robust samples provided. They consist of end-to-end applications that follow best practices for application development. They are running applications that are developed using a combination of tools for design, development, and testing.

*Application samples* are working applications created using more than one tool or API. They illustrate how different tools interact with each other.

*Technology samples* are more granular, code-based samples that focus on a single tool or API.

## Help topics: tutorials gallery

- The tutorials gallery provides educational materials and tutorials to help you learn the concepts and technologies behind the Application Developer tool
- Tutorials gallery is accessible from the **Help** menu
- Three categories of tutorials are provided:
  - Watch and learn
  - Play and learn
  - Do and learn

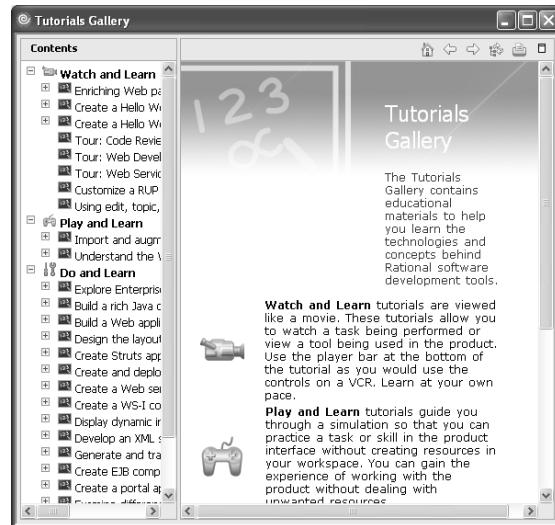


Figure 3-28. Help topics: tutorials gallery

SW2446.0

### **Notes:**

*Watch and learn tutorials* are viewed like a movie. These tutorials allow you to watch a task being performed or view a tool being used in the product.

*Play and learn tutorials* guide you through a simulation so that you can practice a task or skill in the product interface without creating resources in your workspace. You can gain the experience of working with the product without dealing with unwanted resources.

*Do and learn tutorials* are for the more adventurous, hands-on learner; you interact directly with the product and work with real resources. Many of these tutorials also provide aids to help you check your work.

## Help topics: help search and help contents

- The help window (**Help -> Help Contents**) can be used to search for a specific word or phrase
  - Contents by category also available
- A search result page can be put into the context of a category by clicking the icon **Show in table of contents**

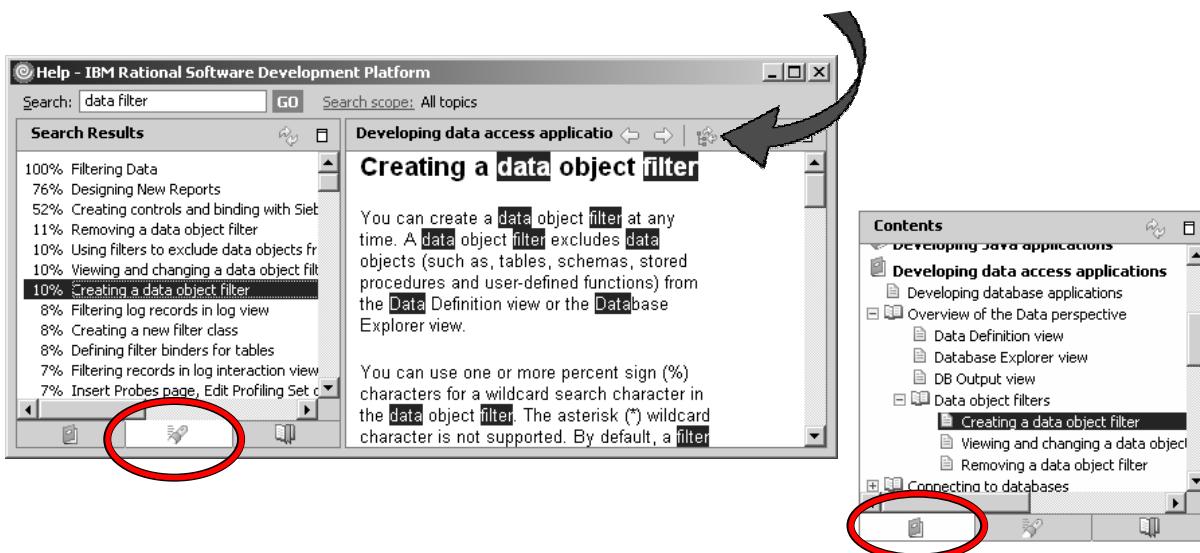


Figure 3-29. Help topics: help search and help contents

SW2446.0

### Notes:

Search results are listed with a percentage value, according to how relevant Application Developer thinks the topic is to the search criterion. Every occurrence of the words entered in the search field is highlighted in the right pane.

By selecting the **Show in table of contents** icon, the tab changes to the Contents view and the category opens to the page selected by the search result.

## Checkpoint questions

1. What is a workspace?
2. What is the workbench?
3. What is a perspective?
4. Using the context menu item **Compare With...**, what can be compared?
5. What are the types of problem and the associated icons in the **Problems** view?



Figure 3-30. Checkpoint questions

SW2446.0

### **Notes:**

## **Unit summary**

---

- In this unit, you should have learned to:
  - Explain the use of a workspace
  - Identify the parts of the workbench
  - Navigate the workbench
  - Work with perspectives, views and editors
  - Create resources using wizards
  - Understand what help systems are available in the workbench

---

Figure 3-31. Unit summary

SW2446.0

### **Notes:**

# Unit 4. Object-Oriented Programming

## What this unit is about

This unit introduces the concept of object-oriented programming, including the principles of object-oriented development. It also introduces the use of UML in relation to object-oriented development, and introduces some simple UML notation that will be used throughout the course.

## What you should be able to do

After completing this unit, you should be able to:

- Define object-oriented programming and contrast it with procedural programming
- Define the terms *class* and *object*
- Explain how objects communicate
- Define UML and explain how UML is used in object-oriented programming
- Describe the support for UML in Application Developer
- Outline several object-oriented design principles

## How you will check your progress

Accountability:

- Checkpoint questions



## 4.1 Object-Oriented Programming

## Unit objectives

---

- After completing this unit, you should be able to:
  - Define object-oriented programming and contrast it with procedural programming
  - Define the terms *class* and *object*
  - Explain how objects communicate
  - Define UML and explain how UML is used in object-oriented programming
  - Describe the support for UML in Application Developer
  - Outline several object-oriented design principles

---

Figure 4-1. Unit objectives

SW2446.0

### **Notes:**

## Procedural programming

- Procedural or structural programming focuses mainly on writing software based on functions
  - Data and functions that manipulate data are kept separate
- Procedural programs cannot easily enforce a common way to access data

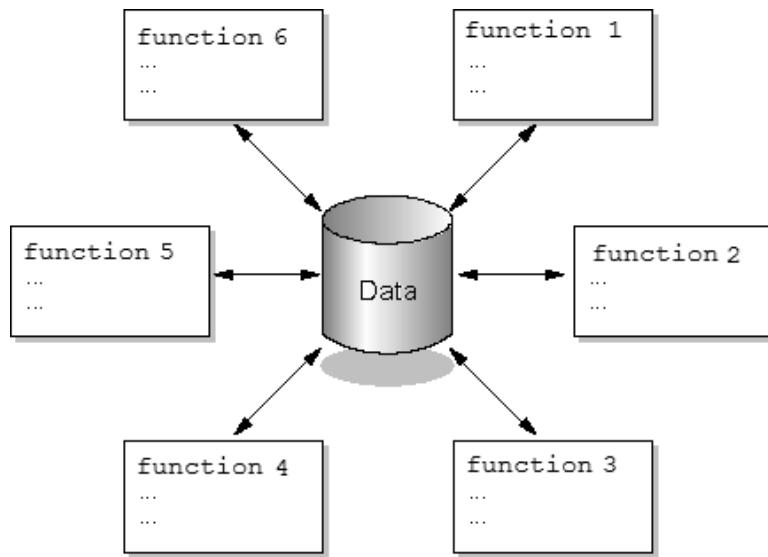


Figure 4-2. Procedural programming

SW2446.0

### **Notes:**

The word *data* in this course refers to information. The word *function* refers to tasks that are completed programmatically.

In procedural programming, data and functions are considered completely separately. The data is stored in a database, and the procedures are written in code. The two only meet if the procedures make a call to the database to retrieve the data.

As depicted by this slide, if data is needed (from the database in the middle), an SQL call is sent to the database from one of the functions. This causes a loss of encapsulation; there is no single place where access to the database is controlled. Implementing a common way of accessing the database requires skilled and careful planning, and is not a trivial task.

## Introduction to objects

- Think of an object in the real world, such as a car
- Associated with a car is a set of:
  - Data that stores information such as the speed of the car, the number of miles the car has been driven, and so on
  - Behaviors the car can do in response to a driver's action, such as accelerating when the gas pedal is pushed, or stopping when the brakes are applied
- Objects of the real world can be modeled in a programming language, by a construct called an *object*



Figure 4-3. Introduction to objects

SW2446.0

### **Notes:**

Objects of the real world have information and behavior that can be modeled in a programming language. For example, a car object maintains information about its state such as how fast it is moving, what gear it is in, and how many miles it has driven. Car objects also have things that they can do in response to a driver's action, such as moving forward, reversing, stopping and so on.

Do you know how an internal combustion engine really works? Probably not. Yet you can drive a car. The idea that you can read information about that car's state (for example, reading the speedometer), and make the car do something (for example, braking), without knowing anything about how the car works inside is also modeled by objects. This concept is called encapsulation, which will be discussed in depth later.

Also, if you get into a different type of car or motorized vehicle from your own, you will notice that it provides almost the same information and functionality; you will still be able to drive, and information such as the car's speed will still be maintained. This is also modeled by objects in a principle called polymorphism, which will also be explained later.

## What are objects?

- Objects are self-contained entities that are made up of both data and functions that operate on the data
  - An object often models the real world
- Data is encapsulated by objects
  - Encapsulation means enclosing, hiding, or containing
  - Implementation details of functions are also encapsulated

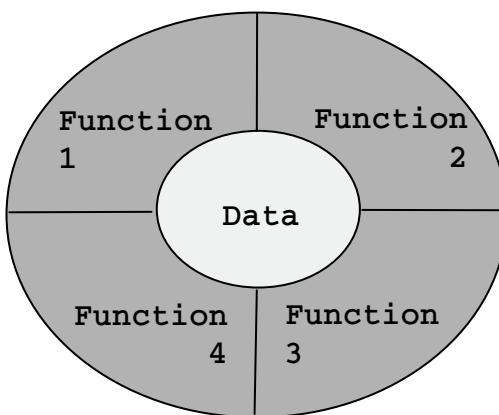


Figure 4-4. What are objects?

SW2446.0

### **Notes:**

Simply put, an object is a combination of data and function. Rather than keeping data and function separate, as was done in procedural programming, data and function are combined in one place: an object.

With objects, the data and function are combined into one concept. To use an object, you only need to know what services it provides, not how they are provided. For example, you can drive a car merely by knowing that pressing the gas pedal will cause the car to accelerate, and pressing the brake will stop the car.

As this diagram illustrates, data is accessed only through the functions provided. In the case of objects, the actual implementation - the code, structure and how things are done - is hidden.

This diagram does not present a standardized notation for an object, but it represents the encapsulation of data and function quite nicely. You will see a standardized notation later in the lecture.

## Objects – another example

- Here is a person, Jim
- Jim has a quantity of money
- How can you determine how much money Jim has?



---

Figure 4-5. Objects – another example

SW2446.0

### **Notes:**

There are basically two solutions:

- Find Jim's money and start counting.
  - You have to know where Jim keeps his money; he may store money in his wallet, sock drawer, and many other locations.
- Ask Jim how much money he has.
  - You do not need to know where his money is kept, and you do not have to count the money, either.

The difference between these two approaches gets at the heart of what object orientation is all about.

## A person object

- Recall that:
  - Data is represented inside the inner circle
  - Functions accessing that data surround the data in the outer circle

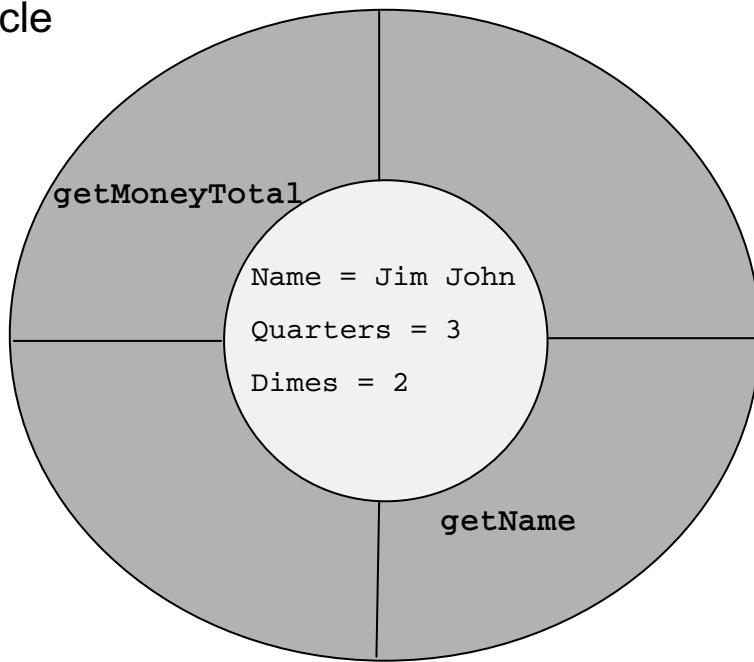


Figure 4-6. A person object

SW2446.0

### Notes:

This is a donut picture of an object. It is not the standard notation used to represent objects, but it captures the idea of encapsulation well. You cannot get to the data without going through the functions.

The function `getMoneyTotal` asks Jim how much money he has; the function `getName` asks Jim what his full name is. In the middle of the donut, the data that is needed to help Jim do these functions is stored.

## **Procedural versus object approach – example**

- System requirement
  - Banking system model withdrawing money from a savings account
- Procedural approach
  - Identify where the data is stored
  - List the algorithmic steps necessary to perform the action
- Object approach
  - Identify what objects are involved; these objects will directly relate to real life objects (Bank, SavingsAccount, Teller and Transaction)
  - Show how these objects interact:
    - To enforce business rules for withdrawals
    - To modify the balance

---

Figure 4-7. Procedural versus object approach – example

SW2446.0

### **Notes:**

A model is an abstraction of the real world, and programming languages describe the computer model of a real world problem; this model determines how the system is designed to meet requirements.

Procedural languages:

- Model only the business processes or the data structure.
- Produce solutions that are collections of functions.

Object-oriented programming languages:

- Model real world objects of the business, encompassing both data structure and related processes.
- Allow users, analysts, designers and programmers to communicate more effectively because they are all speaking of the business.

## **What is object-oriented programming?**

---

- It is a programming implementation technique based on objects
- It incorporates software engineering best practices
  - The use of objects allow systems to become extensible, scalable, maintainable and adaptable
- It allows a computer system model to closely represent objects of the real world, keeping the programmer in touch with the problem

---

Figure 4-8. What Is object-oriented programming?

SW2446.0

### **Notes:**

## Object communication: messages (1 of 2)

- Objects communicate by sending messages
  - `getMoneyTotal` and `getName` are examples of messages that can be sent to the person object, Jim
- Sending messages is the only way that objects can communicate

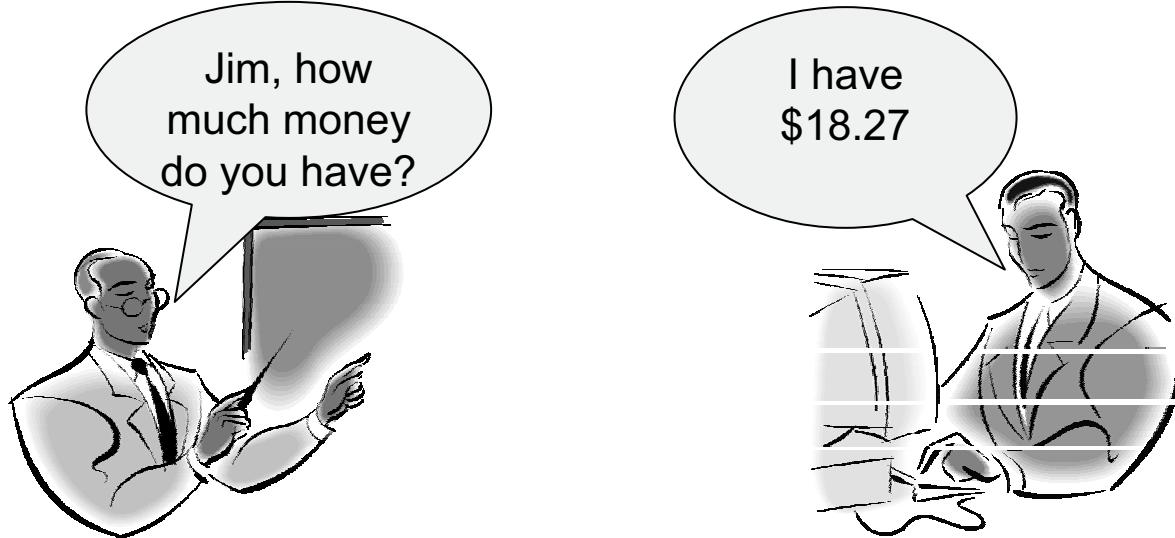


Figure 4-9. Object communication: messages (1 of 2)

SW2446.0

### **Notes:**

Be careful not to confuse the word *message* in its object-oriented sense with other uses of the word. For example, object-oriented messages have nothing to do with e-mail, or TCP/IP, or queuing, or any of the dozens of other uses of this highly overused term.

Here, the instructor asks Jim how much money he has, sending Jim a message. Jim responds to the instructor's message, indicating that he counted the money, and found that it totaled \$18.27.

## Object communication: messages (2 of 2)

- Sending a message is a different concept than calling a function
  - Calling a function indicates that you have identified the actual implementation code that you want to run at the time of the function call
  - Sending a message is just a request for a service from an object; the object determines what to do
  - Different objects may interpret the same message differently

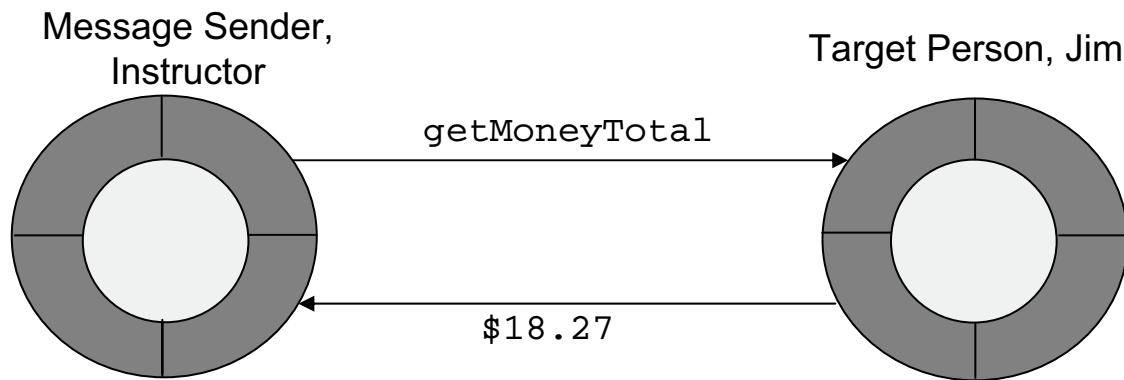


Figure 4-10. Object communication: messages (2 of 2)

SW2446.0

### **Notes:**

## **Messages and methods (1 of 2)**

- Message
  - A message is a request for a service.
- Method
  - A method is the implementation of the service requested by the message
  - In procedural languages, these are known as *procedures* or *functions*

---

Figure 4-11. Messages and methods (1 of 2)

SW2446.0

### **Notes:**

A message is what is sent; a method is what is executed. When sending a message, you do not care how the receiver object actually implements the message, just that it does.

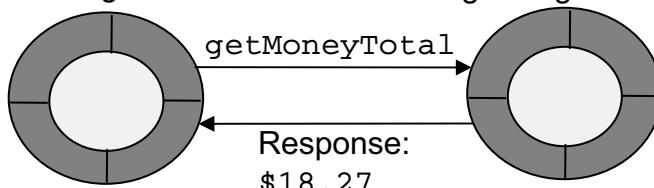
Messages and methods, together, allow objects to access each other's services.

## Messages and methods (2 of 2)

- A message is typically sent from one object to another; it does not imply what actual code will be executed
- A method is the code that will be executed in response to a message that is sent to an object

### Message

Message Sender      Message Target



### Method (code)

```

public double getMoneyTotal(){
    double totalMoney = 0.0;
    totalMoney = totalMoney + (.25*quarters);
    totalMoney = totalMoney + (.10*dimes);
    totalMoney = totalMoney + (.05*nickels);
    totalMoney = totalMoney + (.01*pennies);
    return totalMoney;
}
  
```

Figure 4-12. Messages and methods (2 of 2)

SW2446.0

### Notes:

A method is the implementation of a message by a class. It is the specific set of algorithmic steps and code that will be executed when an instance of that class receives the message.

In this example, the diagram on the left shows that the message sent from the message sender to the message target is `getMoneyTotal`. The method implementation code is displayed on the right.

## ***Accessing data***

---

- Data stored in an object can be accessed directly by other objects or using messages
  - Data is also referred to as *state*
- Using messages is the preferred approach
  - Messages allow the developer to hide the details of the underlying implementation
- Programs use accessor methods to retrieve and modify the data
  - Example: `getMoneyTotal()` may simply access a field called “`moneyTotal`” – or it could hide (or later be replaced by) an addition calculation of money held in different locations (wallet, bank account and so on) to obtain the same value

---

Figure 4-13. Accessing data

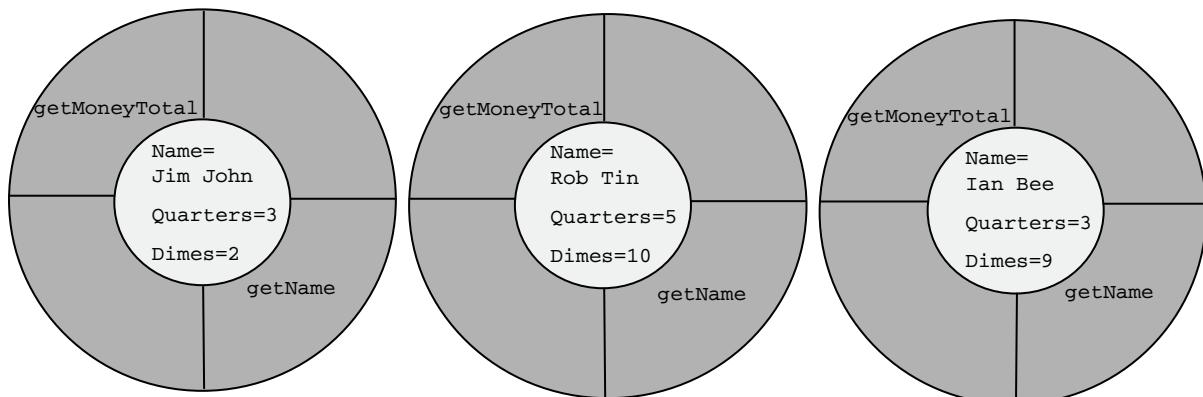
SW2446.0

### **Notes:**

Fields hold the values of a class; the values of all fields represent the current state in that class.

## Objects and instances

- Here are three objects:



- These are all different objects, but each one represents a person
- The actual data is different for each Person object
- The variables needed to store the data are the same in each case; the messages each object responds to are the same
  - These three objects are all *instances* of the same *class* of objects

Figure 4-14. Objects and instances

SW2446.0

### Notes:

*Instance* is another name for an object. Each instance will have its own data, and this data is what will differentiate it from other instances.

Each instance will be able to respond to the same methods. Here each instance can respond to `getMoneyTotal` and `getName` messages.

## Classes

---

- A *class* is an abstraction of a set of objects
- An *object* is one instance of a class
  - *Instance* and *object* are synonymous
- A class defines:
  - Data elements for each instance
  - Messages that each instance can receive
  - An implementation of the messages that each instance can receive
  - These are called *methods*

---

Figure 4-15. Classes

SW2446.0

### Notes:

A class can be thought of as a template. It is what object instances are built from.

A class can also be thought of as a cookie cutter, because every cookie has the same shape and size, but each cookie will be different in its own way (decorations, color, and so forth). While each cookie is different, it came from the same cookie cutter. The memory of the computer can be thought of as the dough. The cookie cutter (class) cuts a cookie (a new object) from dough (memory).

## **What is necessary to define a class?**

---

- Name
  - The class name should describe the real-world object
  - The class name should be singular, concise, and clearly identify the abstraction being modeled
- List of data elements
  - The pieces of data that need to be captured for the abstraction
  - In the Person object, data elements are the person's name, and how many quarters and dimes the person has
- List of messages
  - The messages that the object can receive

---

Figure 4-16. What is necessary to define a class?

SW2446.0

### **Notes:**

## **Benefits of using objects and classes**

---

- Using objects enables:
  - A simple and natural way of thinking about the problem domain, though the processes may still be complex
  - Ability to support changes in:
    - Requirements
    - Technology
- Using objects and classes puts the focus on the combination of methods and data.
  - Allows for a high degree of code reuse
  - Facilitates system models that are representative of the real world

---

Figure 4-17. Benefits of using objects and classes

SW2446.0

### **Notes:**

Object technology is easy to understand at its most basic levels. The world is just a very large example of object orientation.

While code produced in procedural languages may be designed to support change, it takes much more careful consideration, planning and skill to ensure that a system will have these qualities; by programming using objects, these qualities are built into the structure of the system without extra effort. The modularity of the system's objects and the encapsulation of method and data implementation are built into the programming paradigm, whereas only a carefully planned and skilled procedural system design will have these qualities.

In procedural programming, the only type of code reuse that exists is to search through a library of procedures to find one that already exists, and then call it. This is because of the lack of encapsulation - the lowest level (and highest level) denominator is the procedure.

With objects, you really get reuse at three levels:

1. The first and most simple form is through direct reuse of a function.

2. The second form is by reusing an object or set of objects. A system will have many objects to handle most of the requirements of the system. Then, when you add to the system, rather than just looking for procedures, you are actually looking for objects (groupings of data and function) that can be reused to accomplish the new task.
3. The third form is the most powerful form of reuse provided by objects. This form of reuse just reuses the names of the functions and not the actual code within the functions. This will be easier to understand when object-oriented design principles are discussed later.

## Java person class example

```
public class Person {  
    private String name = "Jim John";  
    private int quarters;  
    private int dimes;  
    private int nickels;  
    private int pennies;  
  
    public double getMoneyTotal(){  
        double totalMoney = 0.0;  
        totalMoney = totalMoney + (.25*quarters);  
        totalMoney = totalMoney + (.10*dimes);  
        totalMoney = totalMoney + (.05*nickels);  
        totalMoney = totalMoney + (.01*pennies);  
        return totalMoney;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

Person class with data members name, quarters, dimes, nickels and pennies, and with instance methods getMoneyTotal() and getName()

Figure 4-18. Java person class example

SW2446.0

### **Notes:**

In this example, the state information (data) is kept in variables that are declared to be private (the private modifier is before each declaration of a data member). This means that the variable can only be accessed from within this class. If you had another object which required name information regarding the person, you would be forced to access the information using methods that were exposed to you; that is, methods declared to be public would be used to access the name and other private information.

## Unified Modeling Language (UML)

---

- UML is a standard notation for the modeling of real-world objects as a first step in developing an object-oriented design methodology
- UML has been established as the standard notation for distributed object systems
- It aids communication among analysts, designers and programmers

---

Figure 4-19. Unified Modeling Language (UML)

SW2446.0

### **Notes:**

The Unified Modeling Language (UML) is just a graphical notation for capturing object-oriented models. It has been established by the Object Management Group (OMG) standards body as the standard notation for distributed object systems. As a result, using UML aids in communication among developers, clients, and other project stakeholders involved in the software development process provides a standard notation for conveying ideas. This allows everyone involved in a project to speak the same language and use the same artifacts.

## **What can UML model?**

---

- Classes
  - Class diagrams capture the classes, attributes and methods for a class and the relationships between classes
- Use cases
  - Use case diagrams capture the requirements of the system
- System interactions
  - Interaction diagrams show how objects communicate when the system is running

---

Figure 4-20. What can UML model?

SW2446.0

### **Notes:**

There are many more diagrams defined within UML; however, this lecture will focus solely on class diagrams.

The diagrams listed above provide multiple perspectives of the system under development. The two main perspectives that these diagrams provide are:

- Static View -- the structure of the system.
- Dynamic View -- the behavior of the system (how it changes over time).

## Class diagrams

- A class diagram is used to show classes, including their attributes, methods and relationship to other classes

Person class in Java code

```
public class Person {
    private String name = "Jim John";
    private int quarters;
    private int dimes;
    private int nickels;
    private int pennies;

    public double getMoneyTotal(){
        double totalMoney = 0.0;
        totalMoney = totalMoney + (.25*quarters);
        totalMoney = totalMoney + (.10*dimes);
        totalMoney = totalMoney + (.05*nickels);
        totalMoney = totalMoney + (.01*pennies);
        return totalMoney;
    }

    public String getName() {
        return name;
    }
}
```

Person class in class diagram

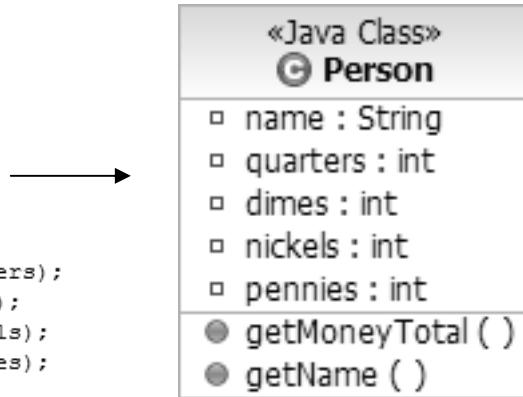


Figure 4-21. Class diagrams

SW2446.0

### Notes:

Here the format for a class in UML notation is illustrated - a three-part box. It includes:

- Class name (top box)
- List of attribute names (middle box)
- List of methods the class implements for its instances (bottom box)

Note: UML calls the data elements of a class *attributes*.

## Object-oriented design principles

- Visual models enable you to design your system with these concepts in mind:
  - Modularity
  - Abstraction
  - Inheritance
- These object-oriented design principles encompass software engineering best practices

---

Figure 4-22. Object-oriented design principles

SW2446.0

### **Notes:**

Some of these principles are not new concepts only found in object-oriented programming. However, object-oriented development is built around these processes, and object-oriented languages such as Java offer built-in support for these concepts. Procedural languages make some of these principles difficult to implement or offer little support for these concepts.

## Modularity

- Modularity aims to limit the complexity of large-scale systems by breaking the problem into units
  - The entire system should consist of a set of modules
  - Each module should be relatively small
- You might divide a car into modules such as:
  - Cooling module
  - Air intake module
  - Ignition module
  - Fuel module



Figure 4-23. Modularity

SW2446.0

### **Notes:**

In procedural development, modules were created at the function level. In object-oriented development, modules are created at the class and package levels (packages will be discussed later).

Modules may be hierarchical in structure; that is, modules may contain other modules.

## ***Methods for promoting modularity***

---

- Reduce the amount of dependency between different units
  - For example, your brakes should still work even if your car runs out of gas
  - This is desirable because it minimizes the effect of changes from one unit to another
- Each unit should be specialized
  - The cooling module of the car should contain only parts relevant to the task of cooling

---

Figure 4-24. Methods for promoting modularity

SW2446.0

### **Notes:**

## Abstraction

- The set of messages that a class can accept should be separate from the implementation
  - This is known as *abstraction*
- Abstraction takes several major forms in object-oriented systems
  - Encapsulation
  - Interfaces
  - Polymorphism

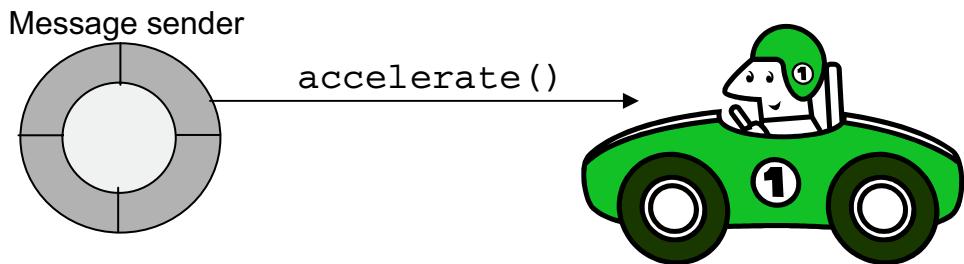


Figure 4-25. Abstraction

SW2446.0

### **Notes:**

## Encapsulation

---

- A message sender does not need to know the method's implementation
  - Information and implementation details of a service provided by a class are hidden
  - This principle is known as *encapsulation*
- For example, in order to drive a car, you should not need to know how the engine works

---

Figure 4-26. Encapsulation

SW2446.0

### **Notes:**

Message senders should not rely on a certain implementation of a method, and encapsulating implementation details allows the implementation to change without affecting the service provided.

## Interfaces

- Many classes of vehicle provide the same services as the car class: trucks, boats, planes and so on
- These vehicles accept the same set of messages that allow them to be *driven*, and thus have the same *interface* to their services
  - Accelerate
  - Decelerate
  - Steer
- Interfaces provide a list of methods separate from the actual implementation of these methods

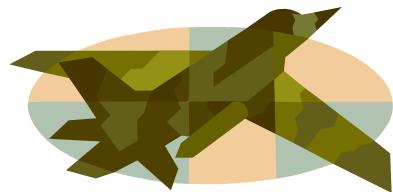


Figure 4-27. Interfaces

SW2446.0

### **Notes:**

The set of messages a class accepts is called an *interface*, and you will recall that a message provides access to a specific service (method) provided by a class. When multiple classes accept the same set of messages, the interface to the services provided by a class may be abstracted and may stand on its own.

Interfaces are a topic that is covered in more detail in later lectures, when the coding implementation of this concept is discussed.

## Polymorphism

---

- If you plan to travel, you may use a car, truck or possibly a boat
  - Whatever vehicle you drive, the result is the same: you will reach your destination
  - The way the service is delivered may be different:
    - You can drive a car across land using roads
    - You can drive a truck over hills, on rough terrain, and on roads
    - You can drive a boat through water
- These various classes accept the same set of messages, and provide the same set of services
  - They may be interchanged without affecting the message sender
- This principle is known as *polymorphism*

---

Figure 4-28. Polymorphism

SW2446.0

### **Notes:**

Interfaces and polymorphism are topics that are covered in more detail in later lectures, when the coding implementation of these concepts is discussed.

## Interface and polymorphism example

- The Drivable interface is a contract that states the behavior of all vehicles
  - The Boat, Car and Truck classes choose to implement the Drivable interface; implementation of the methods may be different
  - They understand the same messages and store the same variables
- Since these classes provide the same service, they may be interchanged without affecting the service that is provided
  - A driver can ask any class that implements the Drivable interface to accelerate, and the result would be the same

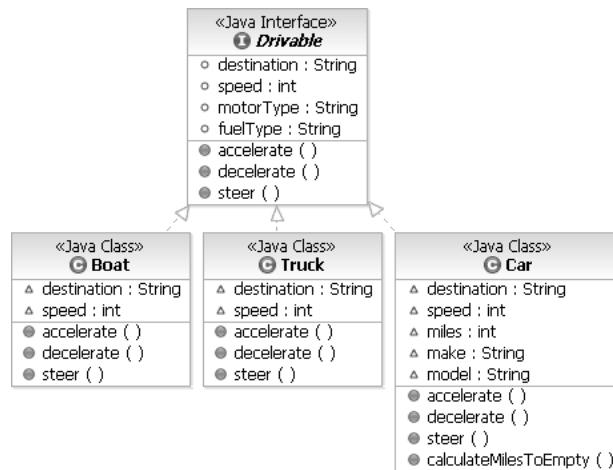


Figure 4-29. Interface and polymorphism example

SW2446.0

### Notes:

It is important to note that the Boat, Car and Truck classes are under obligation to accept the same messages if they implement the Drivable interface, but they may handle the messages completely differently. For example, while a car will use a gas pedal to implement its accelerate() method, a boat may use a throttle lever. It is important to note that the implementation code of methods may be different.

## Inheritance

---

- Inheritance is an *is-a* relationship between classes
  - A car *is* a vehicle
- The Car class models various types of cars
  - Whatever type of car is represented, you know that a car will have wheels and an engine, and will be able to drive
  - These cars have many common features, but they may be represented by different classes because of their differences
- Similarities may be abstracted into a common class
  - This class is called the *superclass*
  - Classes may inherit variables and methods from a superclass; these are called *subclasses*

---

Figure 4-30. Inheritance

SW2446.0

### Notes:

Multiple inheritance, or a class inheriting from more than one superclass, is not permitted in Java.

Subclasses may add variables and behaviors to those that were provided to them by their superclass. Subclasses can also override methods that were inherited from a superclass, providing a different implementation than that provided by the superclass.

The inheritance tree can be as deep as you need; you are not limited to one level of inheritance.

## Inheritance example

- The Car superclass provides the basic variables and methods used by all classes of cars
- CompactCar, LuxuryCar and SportsCar inherit all variables and methods in Car
  - These subclasses also extend the Car class, providing additional variables and methods when needed

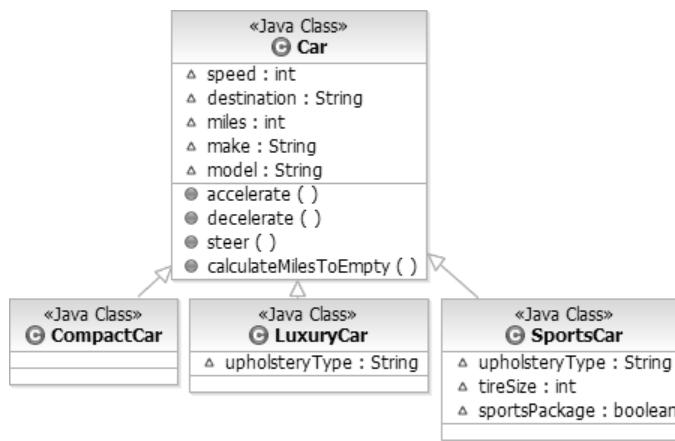


Figure 4-31. Inheritance example

SW2446.0

### Notes:

You saw in the previous example that the Car class implements the Drivable interface. However, the Car class can also have other methods and store other data in order to provide additional services. The Car class is not limited to only providing the services outlined by the Drivable interface.

Here, the CompactCar, LuxuryCar and SportsCar all extend the functionality of the Car class. These classes are subclasses of the Car class, and the Car class is the superclass of the CompactCar, LuxuryCar, and SportsCar classes. The CompactCar, LuxuryCar and SportsCar classes could also choose to override methods in the Car superclass if they need to.

Inheritance allows for classes to be related through their common functionality, and allows for easier maintenance of code. If a change must be made in code that exists in a superclass, this change will propagate down to all subclasses. This prevents updates from having to be made in all classes that use that piece of code.

## ***Checkpoint***

---

1. How are classes and objects related?
2. How do objects communicate?
3. What is UML?
4. What is polymorphism?
5. What is inheritance?



---

Figure 4-32. Checkpoint

SW2446.0

### ***Notes:***

## Unit summary

---

- In this unit, you should have learned to:
  - Define object-oriented programming and contrast it with procedural programming
  - Define the terms *class* and *object*
  - Explain how objects communicate
  - Define UML and explain how UML is used in object-oriented programming
  - Describe the support for UML in Application Developer
  - Outline several object-oriented design principles

---

Figure 4-33. Unit summary

SW2446.0

### **Notes:**



# Unit 5. Java Syntax Basics – Part 1

## What this unit is about

This unit introduces the basic Java syntax for simple programming constructs such as identifiers, primitives, operators, arrays, statements and comments. Reserved words are identified, as is operator precedence.

## What you should be able to do

After completing this unit, you should be able to:

- Outline naming conventions used by Java programs
- Construct a valid identifier
- Describe the Java primitive data types, and explain how and why each one is used
- Declare and initialize Java variables and arrays
- Identify reserved words

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 5.1 Java Syntax Basics - Part 1

## Unit objectives

---

- After completing this unit, you should be able to:
  - Outline naming conventions used by Java programs
  - Construct a valid identifier
  - Describe the Java primitive data types, and explain how and why each one is used
  - Declare and initialize Java variables and arrays
  - Identify reserved words

---

Figure 5-1. Unit objectives

SW2446.0

### **Notes:**

## Identifiers

- Identifiers are:
  - Text strings that represent variables, methods, classes or labels
  - Case-sensitive
- Characters can be digit, letter, '\$' or '\_'
- Identifiers cannot:
  - Begin with digit
  - Be the same as a reserved word

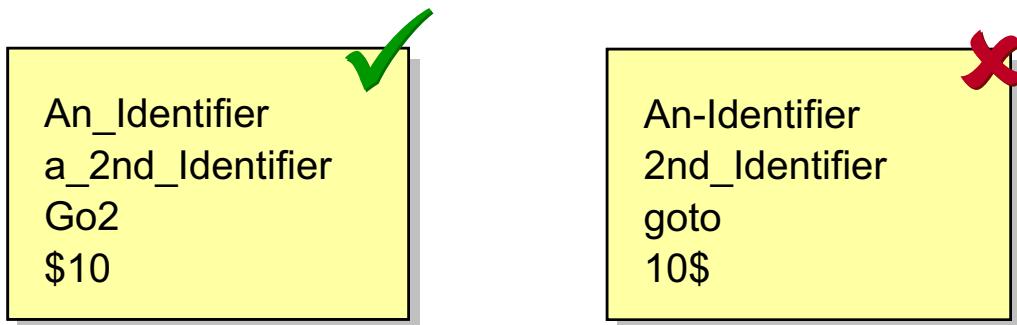


Figure 5-2. Identifiers

SW2446.0

### **Notes:**

## **Java is case-sensitive**

---

- Java is case-sensitive
  - `yourname`, `yourName`, `Yourname`, `YourName` are four different identifiers
- Conventions:
  - Package: all lower case
    - `theexample`
  - Class: initial upper case, composite words with upper case
    - `TheExample`
  - Method/field: initial lower, composite words with upper case
    - `theExample`
  - Constants: all upper case
    - `THE_EXAMPLE`

---

Figure 5-3. Java is case-sensitive

SW2446.0

### **Notes:**

## Reserved words

- Literals
  - null true false

- Keywords
  - abstract assert boolean break byte case catch char class continue default do double else extends final finally float for if implements import instanceof int interface long native new package private protected public return short static strictfp super switch synchronized this throw throws transient try void volatile while

- Reserved for future use
  - byvalue cast const future generic goto inner operator outer rest var volatile

Figure 5-4. Reserved words

SW2446.0

### **Notes:**

Some of the “reserved for future use” words, such as `goto`, are not ever expected to be implemented, and some of them, such as `const`, `inner` and `cast`, express concepts that have already been implemented in other ways. Nevertheless, they are reserved.

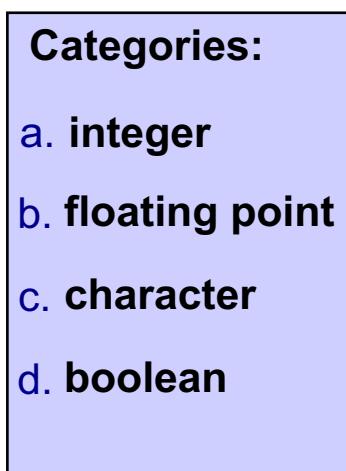
Since Java is case-sensitive, it is possible to use a slightly different form of a reserved word, for example by capitalizing the first letter, or all letters. This will usually lead to confusion, and is not a good practice.

`assert` is a new keyword for J2SE 1.4.

## Java primitives

---

- Every variable must have a data type
  - Primitive data types contain a single value
  - The size and format of a primitive data type are suited to its type
- Java has four categories of primitives



---

Figure 5-5. Java primitives

SW2446.0

### **Notes:**

The sizes of the primitives are defined by the JVM.

They are the same on all platforms.

## Primitives: integers

- Signed whole numbers

- Initialized to zero

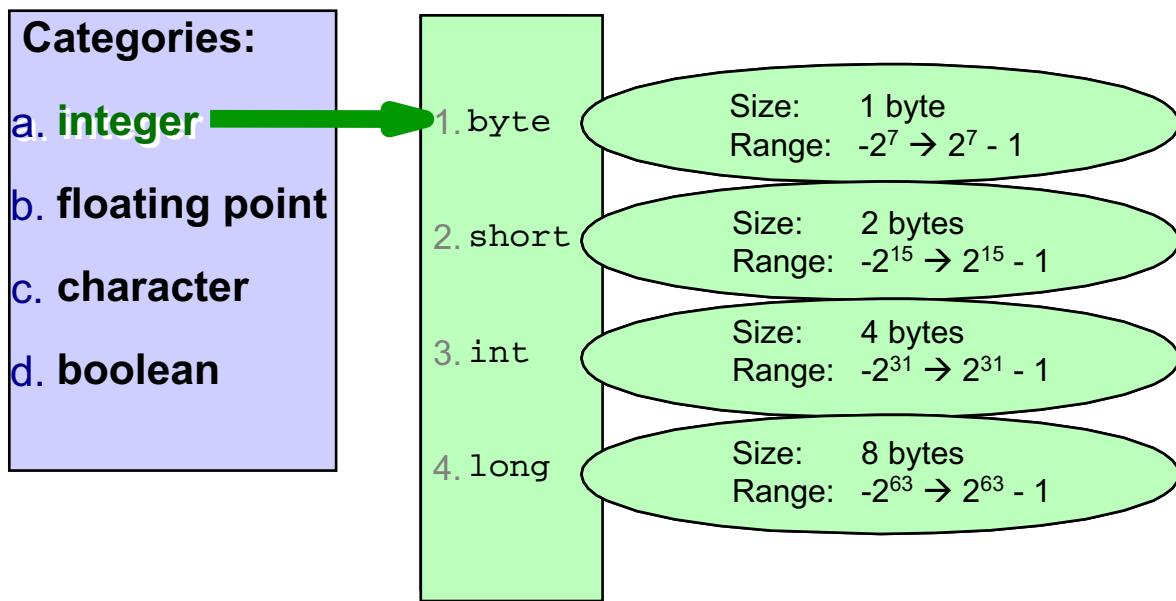


Figure 5-6. Primitives: integers

SW2446.0

### Notes:

## Primitives: floating points

- “General” numbers (can have fractional parts)
- Initialized to zero

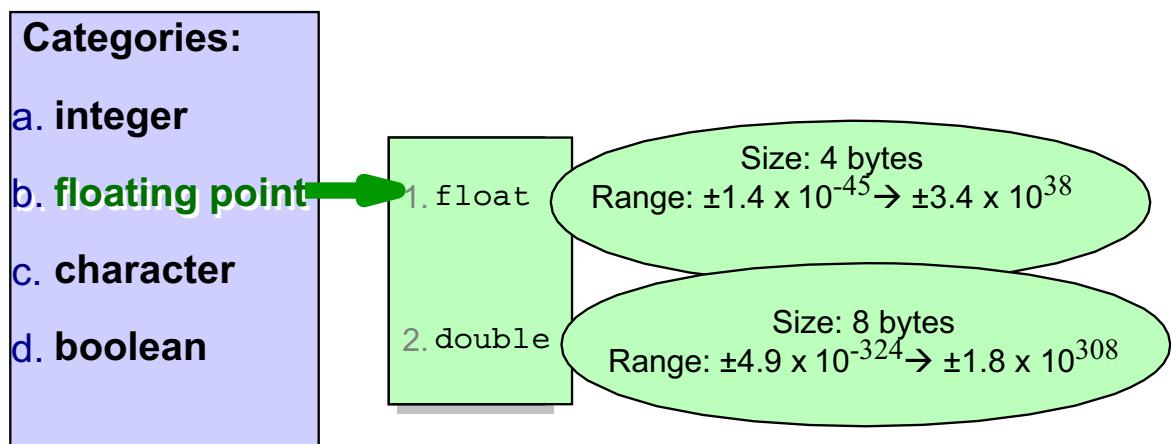


Figure 5-7. Primitives: floating points

SW2446.0

### Notes:

## Primitives: characters

- Any unsigned Unicode character is a **char primitive** data type
- A **character** is a single Unicode character between two single quotes
- Initialized to zero (\u0000)

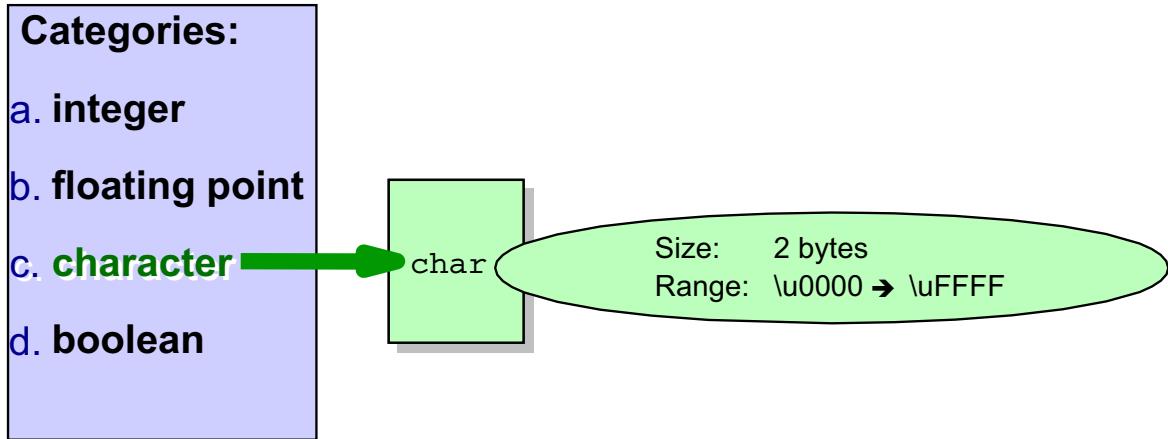


Figure 5-8. Primitives: characters

SW2446.0

### Notes:

A char is an Integral type – a char value can be assigned to an int or a long without loss of precision. When an unsigned short is desired, a char is sometimes used (they are equivalent in size, but the char type is unsigned). However, they do behave differently, so be careful when using the char type in this manner.

You can assign a char a Unicode value using the Unicode escape sequence followed by four hexadecimal digits, for example:

```
char uni = '\u05D0';
```

You can assign a char a character literal:

```
char a = 'A';
```

## Primitives: **booleans**

- **boolean** values are distinct in Java
  - An **int** value can NOT be used in place of a **boolean**
  - A **boolean** can store either true or false
- Initialized to `false`

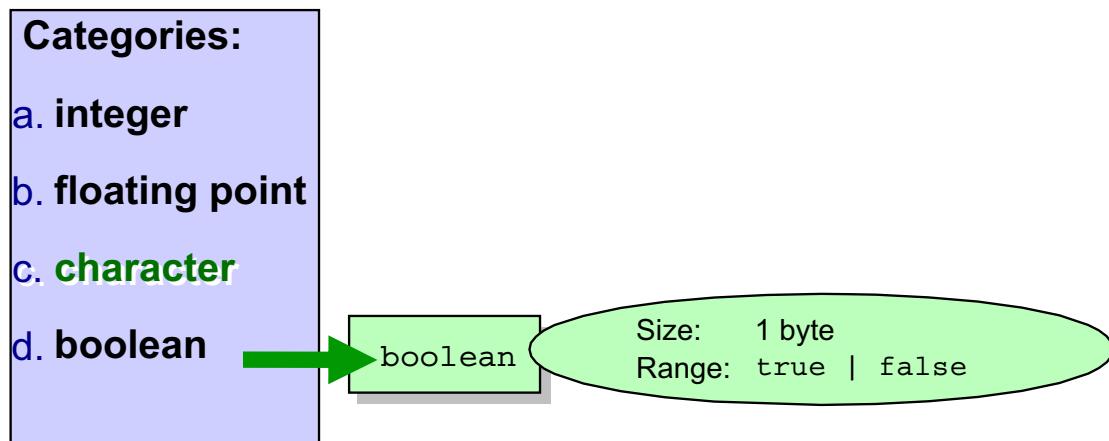


Figure 5-9. Primitives: booleans

SW2446.0

### Notes:

## ***Primitive literals***

- A literal is a value

- Five kinds:

- integer
- floating point
- boolean
- character
- string

Literals

integer.....	7
floating point...	7.0f
boolean.....	true
character.....	'A'
string.....	"A"

Figure 5-10. Primitive literals

SW2446.0

### **Notes:**

7.0 is treated as a double (not a float) by default; 7.0f is treated as a float.

7 is treated as an int (not a short).

## Primitive literals: integers

- Octals are prefixed with a zero
  - 032
- Hexadecimals are prefixed with a zero and an x
  - 0x1A
- Follow a literal with “L” to indicate a long
  - 26L
- Upper and lower case are equivalent**

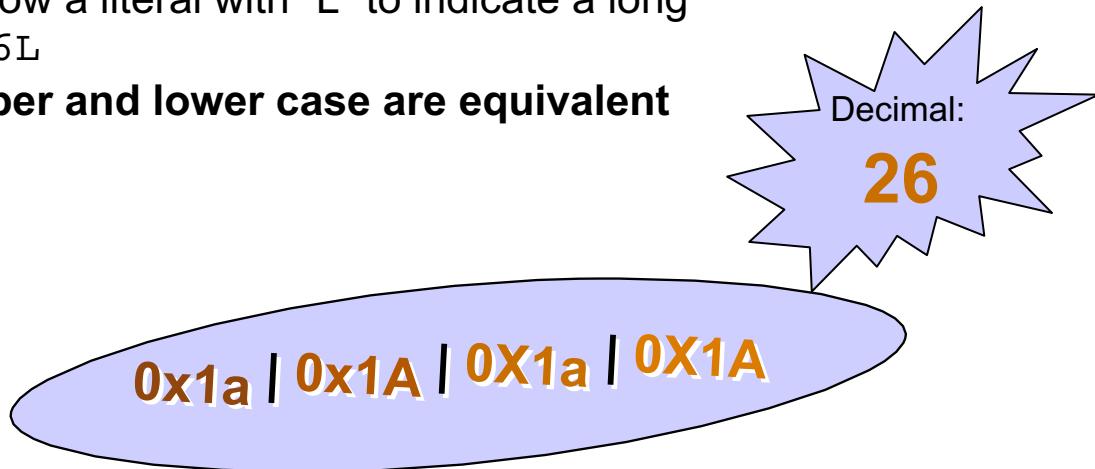


Figure 5-11. Primitive literals: integers

SW2446.0

### Notes:

Literals of type int are written in the usual decimal notation, like 29 or -325, but can also be written in octal or hexadecimal.

Literals of type long are written by adding an L (or lowercase l, but “L” is preferred to “l” to avoid confusion with “1”), for example, `29L` or `-325L`. You can combine these; for example, you may write a long in hexadecimal. Note that you must still append the L to the number.

There is no way to write a literal byte or short; sometimes Java will automatically cast an int literal to the appropriate type.

## ***Primitive literals: floating point***

---

- float literals end with an f (or F)  
-7.1f
- double literals end with a d (or D)  
-7.1D
- e (or E) is used for scientific notation  
-7.1e2
- A floating point number with no final letter is a double  
-7.1 is the same as 7.1d
- Upper and lower case are equivalent**

---

Figure 5-12. Primitive literals: floating point

SW2446.0

### **Notes:**

## ***Primitive literals: escape sequences***

---

- Some keystrokes can be simulated with an escape sequence
  - \b backspace
  - \f form feed
  - \n newline
  - \r return
  - \t tab
- Some characters may need to be escaped when used in string literals
  - \" quotation mark
  - \' apostrophe
  - \\ backslash
- Hexadecimal Unicode values can also be written ‘\uXXXX’

---

Figure 5-13. Primitive literals: escape sequences

SW2446.0

### **Notes:**

Escape sequences are used to print non-printing ASCII characters, and to escape specific punctuation characters that have a special meaning in the Java programming language. These can be used in char literals, and also in string literals.

## Casting primitive types

- Java is a strictly typed language
  - Assigning the wrong type of value to a variable could result in a compile error or a JVM exception
- Casting a value allows it to be treated as another type
- The JVM can implicitly promote from a narrower type to a wider type
- To change to a narrower type, you must cast explicitly

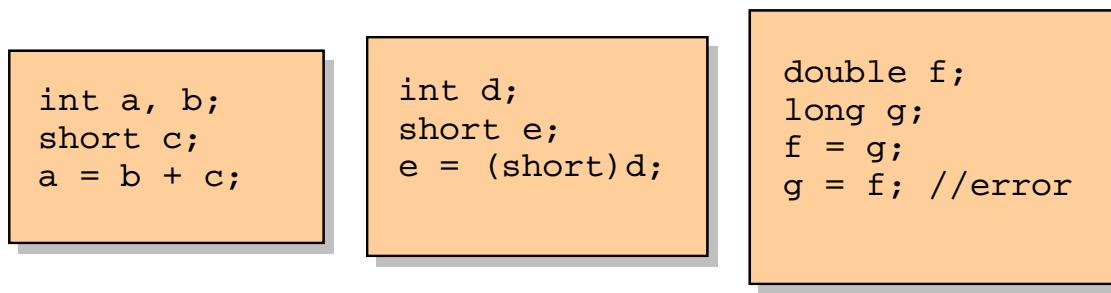


Figure 5-14. Casting primitive types

SW2446.0

### Notes:

The Java language permits conversions between integer values and floating-point values. In addition, because every character corresponds to a number in the Unicode encoding, values of the char type may be converted to and from the integer and floating-point types. The boolean type is the only primitive type that cannot be converted to or from another primitive type in Java.

There are two types of conversions: widening and narrowing conversions.

*Widening conversions* occur when a value of one type is converted to a type that is represented with more bits, therefore having a larger range. Widening conversions are performed by Java automatically when you assign an int literal value to a double variable or a char literal to an int variable.

*Narrowing conversions* occur when a value is converted to a type that is represented with fewer bits. Narrowing conversions are more risky; since you are converting from a type of a wider range to a smaller range, you risk losing data. For example, you may convert an int literal value of 21 to a byte, but you could not properly convert an int literal value of 1099 to a byte, since the byte type has a possible range of values from -128 to 127. However, if you

must do a narrowing conversion, you may use a cast to do so by putting the name of the desired type (the result type) in parentheses before the value to be converted. These examples cast an int literal value to a byte:

```
int i = 25;  
byte b = (byte) i;
```

This example casts a double literal value to an int:

```
int i = (int) 25.123; // The resulting value of i will be 25
```

## Implicit versus explicit casting

- Casting is automatically done when no loss of information is possible
  - byte → short → int → long → float → double
- An explicit cast is required when there is a "potential" loss of accuracy

```
long p = (long) 12345.56;    // p == 12345
int g = p;      // illegal even though an int
                 // can hold 12345
char c = 't';
int j = c;     // automatic promotion
short k = c;   // why is this an error?
short k = (short) c; // explicit cast
float f = 12.35; // what's wrong with this?
```

Figure 5-15. Implicit versus explicit casting

SW2446.0

### **Notes:**

## ***Declarations and initialization***

---

- Variables must be declared before they can be used
- Single value variables (variables that are not arrays) must be initialized before their first use in an expression
  - Declarations and initializations can be combined
  - Use = for assignment (including initialization)
- Examples:

```
int i, j; // Variable declarations
i = 0;
int k=i+1; // Declare and initialize at once
float x=1.0, y=2.0;
System.out.println(i); // Prints 0
System.out.println(k); // Prints 1
System.out.println(j); // Compile error
```

---

Figure 5-16. Declarations and initialization

SW2446.0

### **Notes:**

# Arrays

- Arrays must also be declared before use
  - Have fixed size
    - May be specified by a literal, by an expression, or implicitly
  - May be optionally initialized
  - Have default values depending on their type
  - Are always zero-based (array [0] is the first element)

- Examples:

```

int MAX = 5;
boolean bit[] = new boolean[MAX];
float[] value = new float[2*3];
int[] number = {10, 9, 8, 7, 6};
System.out.println(bit[0]);           // prints "false"
System.out.println(value[3]);         // prints "0.0"
System.out.println(number[1]);         // prints "9"

```



Figure 5-17. Arrays

SW2446.0

## Notes:

While primitive data types hold exactly one value, an array may hold many values. Therefore, an array is a compound, composite data type that may contain multiple values. Java also handles arrays (and classes) differently than primitive data types are handled. So far, you have seen that primitive data types have an identifier (variable name) that is associated with one value. However, an array is treated differently. An array is a *reference* type, rather than a primitive type, which means that its value is actually the address of the value or set of values represented by the variable (that is, the array name is a *reference to* the data values). A reference is called a *pointer* in other languages. However, in Java, this is transparent to the programmer; since the variable name is used to access the array and its values, the underlying referenced address is never seen or manipulated by the programmer.

See <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html> for more details.

## Operators and precedence

---

- Operators are the “glue” of expressions
- Precedence – which operator is evaluated first – is determined explicitly by parentheses or implicitly as follows:

– Postfix operators	[] . (params) x++ x--
– Unary operators	++x --x +x -x ~ !
– Creation or cast	new (type)x
– Multiplicative	* / %
– Additive	+
– Shift	<< >> >>>
– Relational	< > <= >= instanceof
– Equality	== !=
– Bitwise AND	&
– Bitwise exclusive OR	^
– Bitwise inclusive OR	
– Logical AND	&&
– Logical OR	
– Conditional (ternary)	? :
– Assignment	= *= /= %= += -= >>= <<= >>>= &= ^=  = =

Figure 5-18. Operators and precedence

SW2446.0

### **Notes:**

When in doubt, use parentheses to explicitly determine precedence. Use of parentheses also clarifies the expression for other developers.

See <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html> for more details.

## **Comments**

---

- Java supports three kinds of comments:

```
// The rest of the line is a comment  
// No line breaks  
  
/* Everything between  
is a comment */  
  
/** Everything between  
* is a javadoc comment */
```

---

Figure 5-19. Comments

SW2446.0

### **Notes:**

Javadoc tool generates HTML pages of documentation from the javadoc comments.

Javadoc comments must start with `/**` and end with `*/`.

Special tags:

- Classes
  - `@author`, `@version`, `@see` (creates a “see also” reference)
- Methods
  - `@param`, `@return`, `@exception`, `@see`, `@deprecated`
- Fields
  - `@see`

## ***Statements***

---

- Statements are terminated by a semicolon
- Several statements can be written on one line
- A statement can be split over several lines

```
System.out.println(  
    "This is part of the same line");
```

```
a=0; b=1; c=2;
```

---

Figure 5-20. Statements

SW2446.0

### ***Notes:***

## Checkpoint

1. What are the four types of Java integers?
2. What are the two types of Java floating point numbers?
3. What is the difference between a byte and a **char**?
4. When does Java provide implicit casts?
5. What are the three types of comments, and when would a developer use them?



Figure 5-21. Checkpoint

SW2446.0

### **Notes:**

## **Unit summary**

---

- In this unit, you should have learned to:
  - Outline naming conventions used by Java programs
  - Construct a valid identifier
  - Describe the Java primitive data types, and explain how and why each one is used
  - Declare and initialize Java variables and arrays
  - Identify reserved words

---

Figure 5-22. Unit summary

SW2446.0

### **Notes:**

# Unit 6. Java Syntax Basics – Part 2

## What this unit is about

This unit introduces Java syntax for classes, objects, Strings, StringBuffers, and conditional, looping, and branching statements. Variable scope is also discussed.

## What you should be able to do

After completing this unit, you should be able to:

- Create and initialize objects
- Use the identity (==) operator
- Identify and use primitive wrapper classes
- Outline Java's implementation of Strings, and work with Strings
- Explain the difference between the String and StringBuffer classes
- Use conditional statements
- Use looping and branching structures
- Explain variable scope

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 6.1 Java Syntax Basics - Part 2

## Unit objectives

---

- After completing this unit, you should be able to:
  - Create and initialize objects
  - Use the identity (==) operator
  - Identify and use primitive wrapper classes
  - Outline Java's implementation of Strings, and work with Strings
  - Explain the difference between the String and StringBuffer classes
  - Use conditional statements
  - Use looping and branching structures
  - Explain variable scope

---

Figure 6-1. Unit objectives

SW2446.0

### **Notes:**

## Objects and messages

- Objects provide more complex behavior than primitive data types
- Objects respond to messages
  - The dot “.” operator is used to send a message to an object

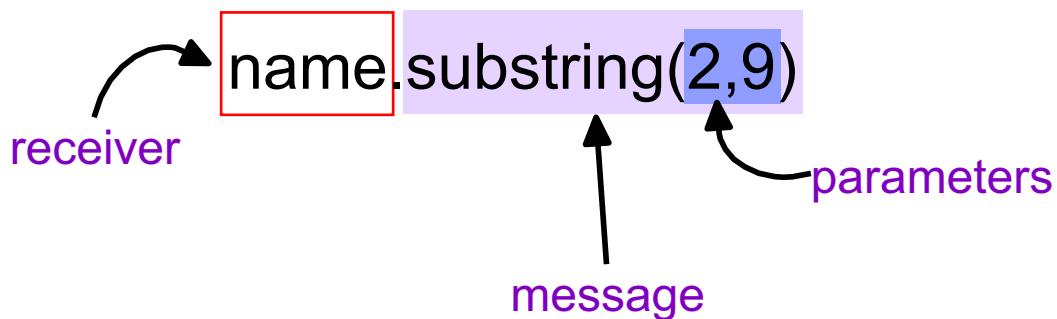


Figure 6-2. Objects and messages

SW2446.0

### Notes:

Recall that objects are built from a template called a class. Classes provide more complex behavior than primitives, because not only do they store data, but they hold methods that are related to accessing and manipulating that data; methods allow messages to be sent to objects. Objects are built from classes, and respond to messages.

## Declaring and initializing objects

- Just like primitives and arrays, objects must be declared before they can be used
  - The declaration requires the *type* of the object
    - The type is the class of the object
  - Use = for assignment (including initialization)
  - Initialization of an object often uses the new operator
    - The new operator is used if you want to create a new object
  - An object can be initialized to null
- Arrays of objects are declared just like arrays of primitives
  - Arrays of objects default to initialization with null

- Examples:

```
Employee emp1 = new Employee(123456);  
Employee emp2;  
emp2 = emp1;  
Department dept[] = new Department[100];  
Test[] t = {new Test(1), new Test(2)};
```

---

Figure 6-3. Declaring and initializing objects

SW2446.0

### Notes:

Classes, like arrays, are different from primitive data types in Java in that they are compound, or composite, data types. Classes are also a reference data type just like arrays, and so the name of an object that is created from a class template acts as a reference to that object's information in memory. While a primitive data type can hold exactly one value, classes (and arrays) are aggregate types, and may contain multiple values.

## Identity

- The == relational operator
  - When this operator is used on objects, it tests for exact object identity
  - Checks whether two variables reference the same object
  - When this operator is used on primitive types, it checks for equal values

```
Employee a = new Employee(1);
Employee b = new Employee(1);
if (a==b) ... // false
```

```
Employee a = new Employee(1);
Employee b = a;
if (a==b) ... // true
```

```
int a = 1;
int b = 1;
if (a==b) ... // true
```

Figure 6-4. Identity

SW2446.0

### Notes:

Recall that objects are made from classes, which are *reference* data types. Therefore, the variable name (identifier) of an object is actually a memory address; the variable is a reference to the object's data in memory. Using the == operator on two objects will test to see if the two identifiers reference the same object in memory.

## Wrapper classes

---

- Primitives have no associated methods; there is no behavior associated with primitive data types
- Each primitive data type has a corresponding class, called a *wrapper*
  - Each wrapper object simply stores a single primitive variable and offers methods with which to process it
  - Wrapper classes are included as part of the base Java API

Primitive Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Figure 6-5. Wrapper classes

SW2446.0

### Notes:

Each of the primitive data types have a corresponding *wrapper* class that allows an object of the related type (Integer, Boolean, Float, and so on) to be created. Wrapper classes “wrap” primitive data, and provide behavior for the primitive data that is relevant to that type. Wrapper classes have the same name as the primitive type *except* that they begin with a capital letter, and may spell out the primitive type name in full.

## Using wrapper classes

```
double number = Double.parseDouble("42.76");
```

```
String hex = Integer.toHexString(42);
```

```
double value = new Integer("1234").doubleValue();
```

```
String input = "test 1-2-3";
int output = 0;
for (int index = 0; index < input.length(); index++) {
    char c = input.charAt(index);
    if (Character.isDigit(c))
        output = output * 10 + Character.digit(c, 10);
}
System.out.println(output) // 123
```

Figure 6-6. Using wrapper classes

SW2446.0

### Notes:

## ***Strings***

---

- The String type is a class, and not a primitive data type
- A String literal is made up of any number of characters between double quotes:

```
String a = "A String";
String b = "";
```

- String objects can be initialized in other ways:

```
String c = new String();
String d = new String("Another String");
String e = String.valueOf(1.23);
String f = null;
```

---

Figure 6-7. Strings

SW2446.0

### **Notes:**

String literals can contain any of the escape sequences that can appear in char literals.

## Concatenating strings

- The + operator concatenates Strings:
  - `String a = "This" + " is a " + "String";`
  - There are more efficient ways to concatenate Strings (this will be discussed later)
  
- Primitive types used in a call to `println` are automatically converted to Strings
  - `System.out.println("answer = " + 1 + 2 + 3);`
  - `System.out.println("answer = " + (1+2+3));`
  - Do you get the same output from the above examples?

---

Figure 6-8. Concatenating strings

SW2446.0

### **Notes:**

You can also use multiple `System.out.print()` statements to avoid concatenation.

For example:

- `System.out.print("This is");`
- `System.out.print("a");`
- `System.out.println("String");`

## String messages

- Strings are objects; objects respond to messages
  - Use the dot (.) operator to send a message
  - String is a class, with methods (more later)

```
String name = "Joe Smith";
name.toLowerCase();    // "joe smith"
name.toUpperCase();   // "JOE SMITH"
" Joe Smith ".trim(); // "Joe Smith"
"Joe Smith".indexOf('e'); // 2
"Joe Smith".length();   // 9
"Joe Smith".charAt(5); // 'm'
"Joe Smith".substring(5); // "mith"
"Joe Smith".substring(2,5); // "e S"
```

Figure 6-9. String messages

SW2446.0

### **Notes:**

## Comparing strings

- Several messages can be sent to a String to test for equivalence with another String
- oneString.equals(anotherString)
  - Tests for equivalence
  - Returns true or false
- oneString.equalsIgnoreCase(anotherString)
  - Case insensitive test for equivalence
  - Returns true or false
- oneString == anotherString is problematic

```
String name = "Joe";
if ("Joe".equals(name))
    name += " Smith";
```

```
boolean same = "Joe".equalsIgnoreCase("joe");
```

Figure 6-10. Comparing strings

SW2446.0

### Notes:

For processing efficiency, Strings are usually only stored once, so each assignment of a particular literal results in a reference to the same copy of that literal. Thus, in the following code:

```
String s1 = "Hello";
String s2 = "Hello";
```

Strings s1 and s2 refer to the same underlying string, and (s1 == s2) will return true. However, in the code:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
```

Strings s1 and s2 are constructed to each be a new string, so (s1 == s2) will return false. For this reason, it is always safer to use .equals to determine whether two strings have the same content.

## **StringBuffer**

- The StringBuffer class provides a more efficient mechanism for building strings
  - String concatenation can get very expensive
  - String concatenation is converted by most compilers - including IBM Rational Application Developer - into a StringBuffer implementation
- If building a simple String, just concatenate; if building a String through a loop, use a StringBuffer

```
StringBuffer buffer = new StringBuffer(15);
buffer.append("This is ");
buffer.append("String");
buffer.insert(7, " a");
buffer.append('.');
System.out.println(buffer.length());      // 17
System.out.println(buffer.capacity());     // 32
String output = buffer.toString();
System.out.println(output);   // "This is a String."
```

Figure 6-11. StringBuffer

SW2446.0

### **Notes:**

Strings are designed not to change. This property is called immutability, and it allows the storage-conserving optimization described in the “Comparing Strings” slide. When strings are concatenated, new objects must be created to hold the new value of the result. It is the creation of new objects that makes string concatenation so expensive.

## ***Conditional statement: if-else***

- Conditional expression must evaluate to a boolean
- The else clause is optional
- Braces are not needed for single statements but are highly recommended for clarity

```
if (x > 10) {  
    if (x != 20) {  
        System.out.println("x is not 20");  
    }  
    else {  
        System.out.println("x = " + x);  
    }  
}  
else {  
    System.out.println("x is less than 11");  
}
```

Figure 6-12. Conditional statement: if-else

SW2446.0

### **Notes:**

Braces around single statements will also make it easier and less error-prone to replace single statements with multiple statements when required.

## ***Shortcut for if-else: the ternary operator***

---

- Shortcut for if-else statement:

(<boolean-expr> ? <true-choice> : <false-choice>)

- Can result in shorter code

- Make sure code is still readable

Code using if-else

```
if (x>LIMIT) {  
    warning = "Too Big";  
} else {  
    warning = null;  
}
```

Code using ternary operator

```
warning = (x>LIMIT) ? "Too Big" : null ;
```

---

Figure 6-13. Shortcut for if-else: the ternary operator

SW2446.0

### **Notes:**

## ***Conditional statement: switch***

- Tests a single variable for several alternative values and executes the corresponding case
- Any case without break will “fall through”
  - Next case will also be executed
- default clause handles values not explicitly handled by a case

```

switch (day) {
    case 0:
    case 1:
        rule = "weekend";
        break;
    case 2:
        ...
    case 6:
        rule = "weekday";
        break;
    default:
        rule = "error";
}

```

The diagram illustrates the conversion of a standard Java switch statement into an equivalent if-else structure. On the left, a standard switch statement is shown with cases for day 0, 1, 2, and 6, and a default case. On the right, this is converted into an if-else structure that checks if day is 0 or 1 (using ||), then sets rule to "weekend". If day is greater than 1 and less than 7, it sets rule to "weekday". Otherwise, it sets rule to "error".

```

if (day == 0 || day == 1) {
    rule = "weekend";
} else if (day > 1 && day < 7) {
    rule = "weekday";
} else {
    rule = error;
}

```

Figure 6-14. Conditional statement: switch

SW2446.0

### **Notes:**

The switch statement evaluates its expression and then executes the corresponding case. A single integer is used to decide which case statement will be executed.

When a break statement is encountered, the switch statement is terminated and the first statement after the switch block is executed.

## Looping statements: while and do...while

- Executes a statement or block as long as the condition remains true
- `while()` executes zero or more times
- `do...while()` executes at least once

```
int x = 2;
while (x < 2) {
    x++;
    System.out.println(x);
}
```

```
int x = 2;
do {
    x++;
    System.out.println(x);
} while (x < 2);
```

---

Figure 6-15. Looping statements: while and do...while

SW2446.0

### **Notes:**

The loop in the first example will not execute (because the value of x is not less than 2 at the beginning of the loop), while the second example will execute once, printing “3”.

## Looping statement: for

- A for loop executes the statement or block {} that follows it
  - Evaluates "start expression" once
  - Continues as long as the "test expression" is true
  - Evaluates "increment expression" after each iteration
  
- A variable can be declared in the for statement
  - Typically used to declare a "counter" variable
  - Typically declared in the "start" expression
  - Its scope is restricted to the loop

```
for (start expr; test expr; increment expr) {
    // code to execute repeatedly
}
```

```
for (int index = 0; index < 10; index++) {
    System.out.println(index);
}
```

Figure 6-16. Looping statement: for

SW2446.0

### Notes:

In the second example, attempts to access `index` outside of the loop will fail, since its scope is restricted to the loop by its declaration inside the `for` statement.

You can leave out some of the expressions in the `for` loop declaration to create different types of loops (for example, an infinite loop, exited from within based on a particular condition). You can also include multiple statements for the `start` and `increment` expressions by separating them with a comma. For example, the following `for` loop is valid:

```
for (int i=0, j=0; i+j<10; i+=2, j--)
```

## for versus while

---

- These statements provide equivalent functionality
  - Each can be implemented in terms of the other
- These looping structures are typically used in different situations
  - `while` tends to be used for open-ended looping
  - `for` tends to be used for looping over a fixed number of iterations

```
int sum = 0;
for (int index = 1; index <= 10; index++) {
    sum += index;
}
```

```
int sum = 0;
int index = 1;
while (index <= 10) {
    sum += index;
    index++;
}
```

Figure 6-17. for versus while

SW2446.0

### Notes:

Any loop implemented as a `for` loop can also be implemented as a `while` loop, and vice versa.

## **Branching statements**

---

- **Break**

- Can be used outside a switch statement
- Terminates a for, while or do...while loop
- Two forms:
  - Labeled: execution continues at next statement after labeled loop
  - Unlabeled: execution continues at next statement outside loop

- **Continue**

- Like break, but merely skips the remainder of this iteration of the loop, then continues by evaluating the boolean expression of the innermost loop
- Labeled and unlabeled forms

- **Return**

- Exits the current method
- May include an expression to be returned
  - Type must match method's return type
  - A void return type means no value can be returned

---

Figure 6-18. Branching statements

SW2446.0

### **Notes:**

The continue branching statement allows you to skip the current iteration of any type of loop. An unlabeled continue statement skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, skipping the remainder of this iteration of the loop.

## Sample branching statements

```
public int myMethod(int x) {
```

```
    int sum = 0;
```

```
    outer: for (int i=0; i<x; i++) {
```

```
        inner: for (int j=i; j<x; j++) {
```

```
            sum++;
```

```
            if (j==1) continue;
```

```
            if (j==2) continue outer;
```

```
            if (i==3) break;
```

```
            if (j==4) break outer;
```

```
        }
```

```
    }
```

```
}
```

```
return sum;
```

Figure 6-19. Sample branching statements

SW2446.0

### **Notes:**

## Scope

- A variable's scope is the region of a program within which the variable can be referenced
  - Variables declared in a method can only be accessed in that method
  - Variables declared in a loop or a block can only be accessed in that loop or block

```

int a = 1;
for (int b = 0; b < 3; b++) {
    int c = 1;
    for (int d = 0; d < 3; d++) {
        if (c < 3) c++;
    }
    System.out.print(c);
    System.out.println(b);
}
abcd
abc
a
✖ a = c; // ERROR! c is out of scope

```

Figure 6-20. Scope

SW2446.0

### Notes:

## Checkpoint

---

1. Provide two ways to create an array of five integers, initialized to zero
2. Explain the difference between the code (`x==y`) and `x.equals(y)`
3. What are the differences between a String and a StringBuffer object, and why would a developer choose to use one over the other?
4. What are Java's three branching statements?



---

Figure 6-21. Checkpoint

SW2446.0

### Notes:

## Unit summary

- In this unit, you should have learned to:
  - Create and initialize objects
  - Use the identity (==) operator
  - Identify and use primitive wrapper classes
  - Outline Java's implementation of Strings, and work with Strings
  - Explain the difference between the String and StringBuffer classes
  - Use conditional statements
  - Use looping and branching structures
  - Explain variable scope

---

Figure 6-22. Unit summary

SW2446.0

### **Notes:**



# Unit 7. Building Classes

## What this unit is about

This unit introduces the Java syntax for classes, and shows how classes may be declared, constructed and used. Class modifiers are discussed, and memory management and garbage collection are explained.

## What you should be able to do

After completing this unit, you should be able to:

- Declare a Java class
- Define constructors
- Create methods and fields, and set the appropriate modifier
- Explain how memory is managed in Java
- Outline the role of packages in Java

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 7.1 Building Classes

## Unit objectives

---

- After completing this unit, you should be able to:
  - Declare a Java class
  - Define constructors
  - Create methods and fields, and set the appropriate modifier
  - Explain how memory is managed in Java
  - Outline the role of packages in Java

---

Figure 7-1. Unit objectives

SW2446.0

### **Notes:**

## **Classes**

---

- Encapsulate attributes (fields) and behavior (methods)
  - Attributes and behavior are *members* of the class
- Members may belong to either of the following:
  - The whole class (class variables and methods, indicated by the keyword `static`)
  - Individual objects (instance variables and methods)
- Classes can be:
  - Independent of each other
  - Related by inheritance (superclass/subclass)
  - Related by type (interface)

---

Figure 7-2. Classes

SW2446.0

### **Notes:**

Recall from earlier discussions that classes act as a template for creating objects. Objects store data, and have methods for manipulating and accessing that data.

## Implementing classes

---

- Classes are grouped into packages
  - A package contains a collection of logically-related classes
- Source code files have the extension .java
  - There is one public class per .java file
- A class is like a blueprint; a class is used to create an object or *instance* of the class typically

---

Figure 7-3. Implementing classes

SW2446.0

### **Notes:**

## Class declaration

- A class declaration specifies a type
  - The identifier in a class declaration specifies the name of the class
  - The optional `extends` clause indicates the superclass
  - The optional `implements` clause lists the names of all of the interfaces that the class implements

```
public class BankAccount extends Account
    implements Serializable, BankStuff {

    // Class Body
}
```

Figure 7-4. Class declaration

SW2446.0

### **Notes:**

Recall from the lectures on Java syntax that there are two data types that exist in Java: primitive data types and reference data types. Classes are reference data types; variables of a type specified by a class may be created. Since classes are reference data types, such a variable would reference the address in memory where the object is stored.

## ***Class modifiers***

---

- The declaration may include class modifiers (`public`, `abstract`, `final`) which affect how the class can be used
- If the class is declared `public`, it may be accessed by any Java code that can access its containing package
  - Otherwise it may be accessed only from within its containing package
- `Abstract` classes can contain anything that a normal class can contain (variables, methods, constructors)
  - Cannot be instantiated, only subclassed
  - Provide common information for subclasses
- A class is declared `final` if it permits no subclasses

---

Figure 7-5. Class modifiers

SW2446.0

### **Notes:**

If the same modifier appears more than once in a class declaration, a compilation error occurs.

An abstract class can (but does not have to) contain abstract methods (methods without an implementation). Only abstract classes may have abstract methods; subclasses can fill in the implementation details for these methods, and typically an abstract class will provide at least a partial or complete implementation of at least one method. If a class that is not abstract contains an abstract method, a compilation error occurs. A class is considered abstract even if only one method is declared abstract.

If the name of a final class appears in the `extends` clause of another class declaration, a compilation error occurs. If a class is declared both final and abstract, a compilation error occurs because the implementation of such a class could never be completed.

## **Constructors**

- The class body contains at least one *constructor*, which is a method that sets up a new instance of a class
  - The method has the same name as the class
- Use the `new` keyword with a constructor to create *instances* of a class

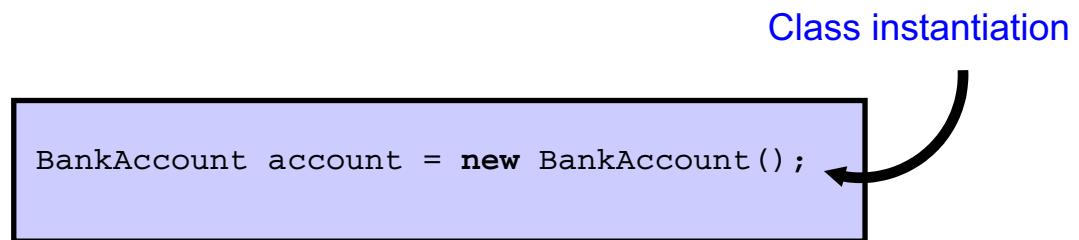


Figure 7-6. Constructors

SW2446.0

### **Notes:**

## **Memory management in Java**

---

- Since Java does not use pointers, memory addresses cannot be accidentally or maliciously overwritten
- The problems inherent in user allocated and deallocated memory are avoided, since the Java Virtual Machine handles all memory management
  - Programmers do not have to keep track of the memory they allocate from the heap and explicitly deallocate it

---

Figure 7-7. Memory management in Java

SW2446.0

### **Notes:**

Reference data types allow a variable to hold a memory address which references an object's location in memory. This is transparent to the programmer because the programmer never accesses or manipulates memory addresses. Instead, an object is referenced by its variable name.

## More about constructors

- Constructors are used to create and initialize objects
  - A constructor always has the same name as the class it constructs (case-sensitive)
- Constructors have no return type
  - Constructors return no value, but when used with new return a reference to the new object

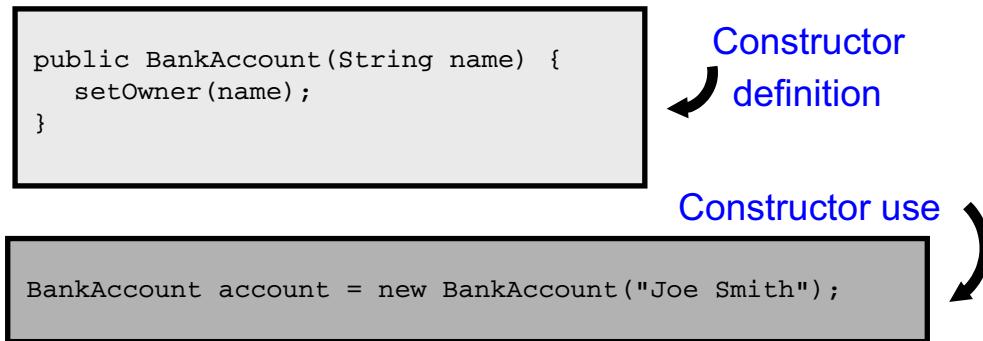


Figure 7-8. More about constructors

SW2446.0

### **Notes:**

The `new` operator is used to create a new object. The JVM handles the allocation of memory for this new object. When no more references exist to that object, the JVM handles the deallocation of memory for that object as well.

## **Default constructors**

---

- The constructor with no arguments is a default constructor
- The Java platform provides a default constructor only if you do not explicitly define any constructor
- When defining a constructor, you should also provide a default constructor

---

Figure 7-9. Default constructors

SW2446.0

### **Notes:**

## Overloading constructors

- There may be any number of constructors with different parameters
  - This is called *overloading*
- Constructors are commonly overloaded to allow for different ways of initializing instances

```
BankAccount new_account =
    new BankAccount();

BankAccount known_account =
    new BankAccount(account_number);

BankAccount named_account =
    new BankAccount("My Checking Account");
```

Figure 7-10. Overloading constructors

SW2446.0

### **Notes:**

## Constructor example

- In a constructor, the keyword `this` is used to refer to other constructors in the same class

```

...
public BankAccount(String name) {
    super();
    owner = name;
}

public BankAccount() {
    this("TestName");
}

public BankAccount(String name, double initialBalance) {
    this(name);
    setBalance(initialBalance);
}
...

```

Figure 7-11. Constructor example

SW2446.0

### Notes:

The keyword `this` can also be used to represent the current object, when used with the dot operator. For example:

`this.x = 0;`

`this.doStuff();`

## Constructor chaining

- Superclass objects are built before the subclass
  - `super(argument list)` initializes superclass members
- The first line of your constructor can be one of:
  - `super(argument list);`
  - `this(argument list);`
- You cannot use both `super()` and `this()` in the same constructor
- The compiler supplies an implicit `super()` constructor for all constructors

---

Figure 7-12. Constructor chaining

SW2446.0

### **Notes:**

One constructor can invoke another constructor within the class, and this is called constructor chaining.

Chained constructor statements are of the form:

- `this (argument list);`

The call is only allowed once per constructor, and it must be the first line of code. This is done to share code among constructors.

**Warning:** an overridden subclass method may be called during construction of superclass.

## Java destructors?

- Java does not have the concept of a destructor for objects that are no longer in use
- Deallocation of memory is done automatically by the JVM
  - A background process called the garbage collector reclaims the memory of unreferenced objects
  - The association between an object and an object reference is severed by assigning another value to the object reference, for example:

```
objectReference = null;
```

- An object with no references is a candidate for deallocation during garbage collection

---

Figure 7-13. Java destructors?

SW2446.0

### **Notes:**

Just as the JVM manages the allocation of memory for objects, it also handles the deallocation of this memory once an object is no longer needed.

## **Garbage collector**

---

- The garbage collector sweeps through the JVM's list of objects periodically and reclaims the resources held by unreferenced objects
- All objects that have no object references are eligible for garbage collection
  - References out of scope, objects to which you have assigned null, and so forth
- The JVM decides when the garbage collector is run
  - Typically, the garbage collector is run when memory is low
  - May not be run at all
  - Unpredictable timing

---

Figure 7-14. Garbage collector

SW2446.0

### **Notes:**

## ***Working with the garbage collector***

---

- You cannot prevent the garbage collector from running, but you can request it to run soon
  - `System.gc();`
  - This is only a request, not a guarantee
- The `finalize()` method of an object will be run immediately before garbage collection occurs
  - This method should only be used for special cases (such as cleaning up memory allocation from native calls) because of the unpredictability of the garbage collector
  - Things like open sockets, files, and so forth should be cleaned up during normal program flow before the object is dereferenced

---

Figure 7-15. Working with the garbage collector

SW2446.0

### **Notes:**

## Fields

- Objects retain state in *fields*
  - Fields are defined as part of the class definition
  - Each instance gets its own copy of the instance variables
- Fields can be initialized (if desired) when declared
  - Default values will be used if fields are not initialized

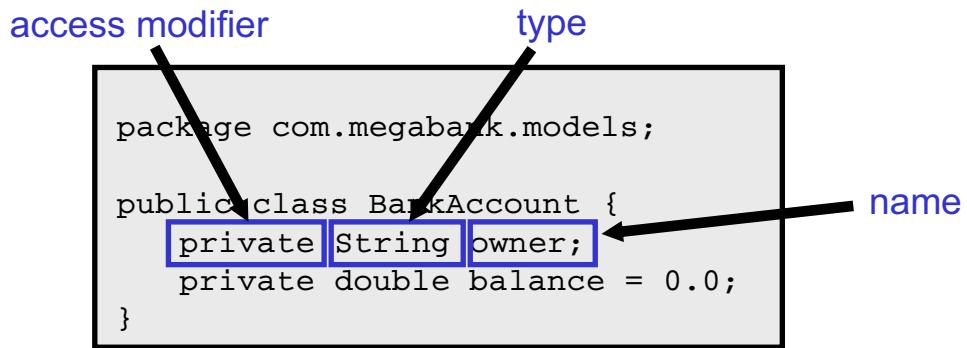


Figure 7-16. Fields

SW2446.0

### Notes:

## Messages

- Use messages to invoke the behavior of an object

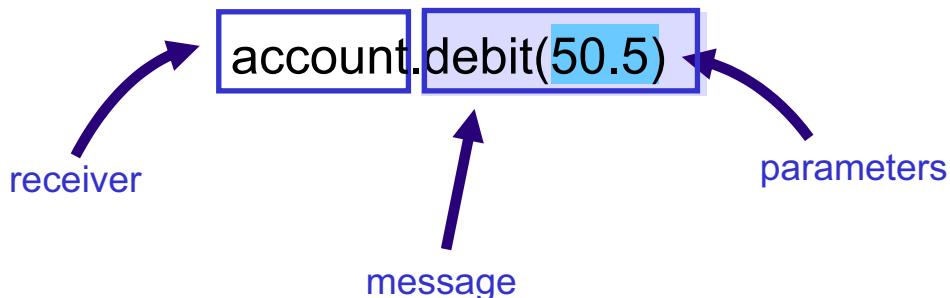
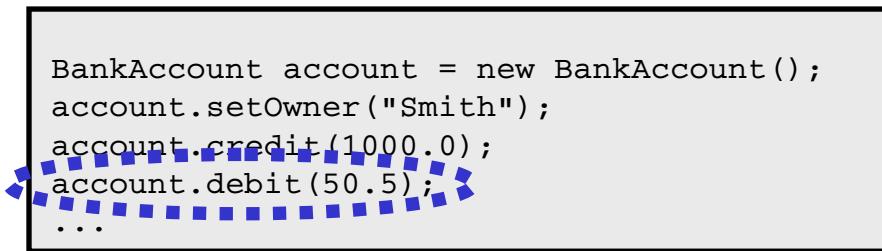


Figure 7-17. Messages

SW2446.0

### Notes:

Messages are simply method calls in Java acting on an object. Sending a message to an object allows you to access the behavior of that object.

A message is sent to an object by using the dot “.” operator on an object. The message sent corresponds to the method name called. Data is sent as part of the message, and is included in parentheses following the method name.

## Methods

- Methods define how an object responds to messages
- Methods define the behavior of the class
  - All methods belong to a class

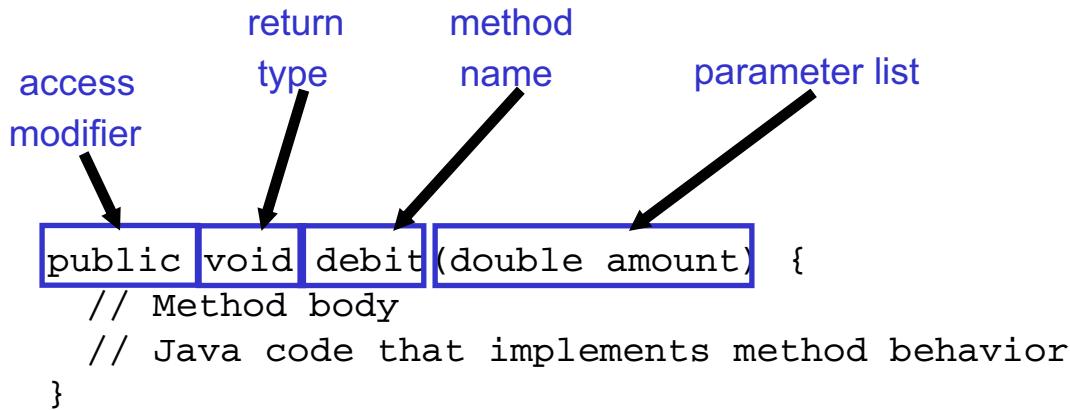


Figure 7-18. Methods

SW2446.0

### Notes:

## Method signatures

- A class can have many methods with the same name
  - Each method must have a different signature
- The method signature consists of:
  - The method name
  - Argument number and types

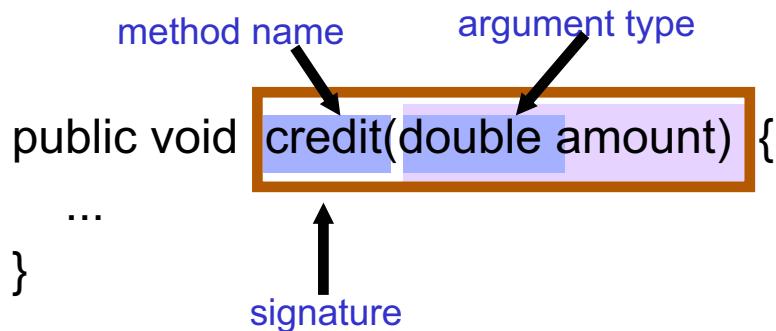


Figure 7-19. Method signatures

SW2446.0

### **Notes:**

Arguments appear in parentheses in a comma-separated list and are used to pass information to a method.

## Method parameters

- Arguments (parameters) are passed:
  - By value for primitive types
  - By object reference for reference types
- Primitive values cannot be modified when passed as an argument

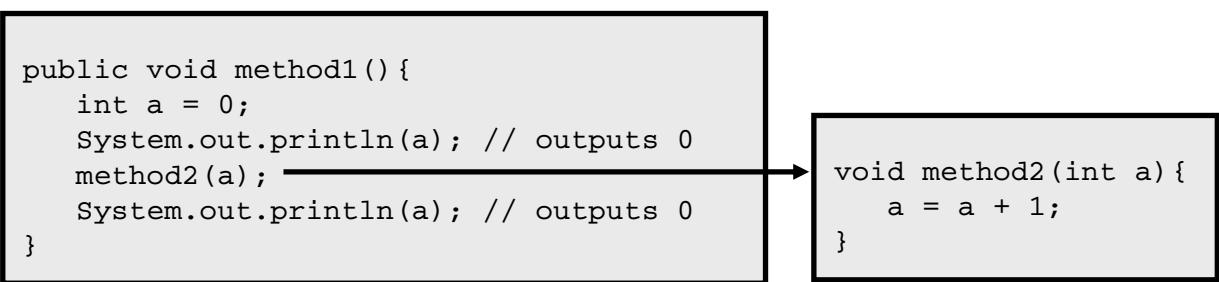


Figure 7-20. Method parameters

SW2446.0

### **Notes:**

## ***Returning from methods***

---

- Methods return at most one value or one object
  - If the return type is void, the return statement is optional
- The return keyword is used to return control to the calling method
  - There may be several return statements in a method; the first one reached will be executed

```
public void debit(double amount) {  
    if (amount > getBalance()) return;  
    setBalance(getBalance() - amount);  
}
```

```
public String getFullName() {  
    return getFirstName() + " " + getLastName();  
}
```

Figure 7-21. Returning from methods

SW2446.0

### **Notes:**

## Invoking methods

- To call a method, use the dot “.” operator
  - The same operator is used to call both class and instance methods
  - If the call is to a method of the same class, the dot operator is not necessary

```
BankAccount account = new BankAccount();  
account.setOwner("Smith");  
account.credit(1000.0);  
System.out.println(account.getBalance());  
...
```

BankAccount method

```
public void credit(double amount) {  
    setBalance(getBalance() + amount);  
}
```

Figure 7-22. Invoking methods

SW2446.0

### Notes:

## **Overloading methods**

---

- The same name may be used for many different methods, as long as they have different signatures
  - This is known as overloading
- The `println()` method of `System.out.println()` has 10 different parameter declarations:
  - `boolean`, `char[]`, `char`, `double`, `float`, `int`, `long`, `Object`, `String`, and one with no parameters
  - You do not need to use different method names (such as `"printString"` or `"printDouble"`) for each data type you may want to print

---

Figure 7-23. Overloading methods

SW2446.0

### **Notes:**

Overloading allows multiple methods in the same class to share a name if they have different sets of parameters.

## Overriding methods

- A method with a signature and return type identical to a method in the superclass overrides the method of the superclass

```
public class BankAccount {  
    private float balance;  
    public int getBalance() {  
        return balance;  
    }  
}  
  
public class InvestmentAccount extends BankAccount {  
    private float cashAmount  
    private float investmentAmount;  
    public int getBalance() {  
        return cashAmount + investmentAmount;  
    }  
}
```

Figure 7-24. Overriding methods

SW2446.0

### **Notes:**

Overriding methods allows a class to inherit a method from a superclass that has a purpose or behavior similar to that which is to be provided by a subclass, but gives the subclass the freedom to implement the behavior as needed.

## ***main method***

---

- An application cannot run unless at least one class has a `main()` method
- The JVM loads a class and starts execution by calling the `main(String[] args)` method
  - `public`: the method can be called by any object
  - `static`: no object need be created first
  - `void`: nothing will be returned from this method

```
public static void main(String[] args) {  
    BankAccount account = new BankAccount();  
    account.setOwner(args[0]);  
    account.credit(Integer.parseInt(args[1]));  
    System.out.println(account.getBalance());  
    System.out.println(account.getOwner());  
}
```

---

Figure 7-25. `main` method

SW2446.0

### **Notes:**

## Encapsulation

- Private state can only be accessed from methods in the class
- Mark fields as private to protect the state
  - Other objects must access private state through public methods

```
package com.megabank.models;

public class BankAccount {
    private String owner;
    private double balance = 0.0;
}
```

```
public String getOwner() {
    return owner;
}
```

Figure 7-26. Encapsulation

SW2446.0

### **Notes:**

Recall from the lecture on object-oriented programming that public getter and setter methods are used to provide outside objects the ability to access and update private data. It is best practice to hide data from direct access using the `private` keyword, and allow access only through public methods.

## ***Static members***

---

- Static fields and methods belong to the class
  - Changing a value in one object of that class changes the value for all of the objects
- Static methods and fields can be accessed without instantiating the class
- Static methods and fields are declared using the static keyword

```
public class MyDate {  
    public static long getMillisSinceEpoch() {  
        ...  
    }  
    ...  
    long millis = MyDate.getMillisSinceEpoch();
```

Figure 7-27. Static members

SW2446.0

### **Notes:**

## Final members

- A final field is a field which cannot be modified
  - This is the Java version of a constant
- Typically, constants associated with a class are declared as static final fields for easy access
  - A common convention is to use only uppercase letters in their names

```
public class MyDate {  
    public static final long SECONDS_PER_YEAR =  
        31536000;  
    ...  
}  
...  
long years = MyDate.getMillisSinceEpoch() /  
(1000*MyDate.SECONDS_PER_YEAR);
```

Figure 7-28. Final members

SW2446.0

### **Notes:**

Static final fields may therefore be regarded as global constants.

## **Abstract classes**

---

- Abstract classes cannot be instantiated; they are intended to be a superclass for other classes

```
abstract class Learner {  
    public abstract String getName();  
    public abstract int getAge();  
    public int getMaxGrade() {  
        return getAge() - 5;  
    }  
}
```

- abstract methods have no implementation
- If a class has one or more abstract methods, it is abstract, and must be declared so
- Concrete classes have full implementations and can be instantiated

---

Figure 7-29. Abstract classes

SW2446.0

### **Notes:**

## Packages

- Classes can be grouped:
  - Logically, according to the model you are building
  - As sets designed to be used together
  - For convenience
- By convention, package names are in lower case
- Different packages can contain classes with the same name

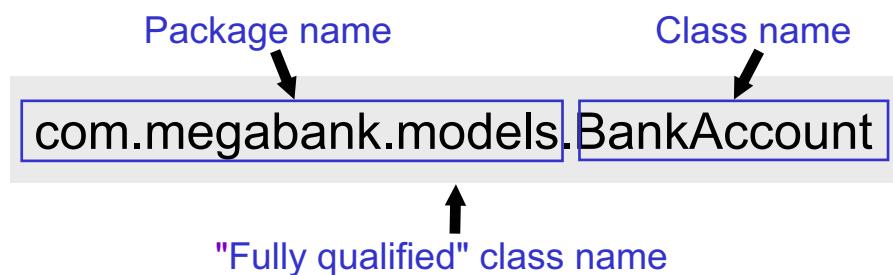


Figure 7-30. Packages

SW2446.0

### Notes:

A package is a group of related classes that provides access protection and namespace management.

## ***Class visibility***

---

- Classes can reference other classes within the same package by class name only
- Classes must provide the fully qualified name (including package) for classes defined in a different package.
  - Below, Tester and BankAccount are defined in different packages

```
package com.megabank.testing;

public class Tester {
    public static void main(String[] args) {
        com.megabank.models.BankAccount account1
            = new com.megabank.models.BankAccount("Smith");
        account1.credit(1000.0);
        System.out.println(account1.getBalance());
    }
}
```

---

Figure 7-31. Class visibility

SW2446.0

### **Notes:**

## import statement

- Use import statements to import packages or classes to make other classes directly visible to your class

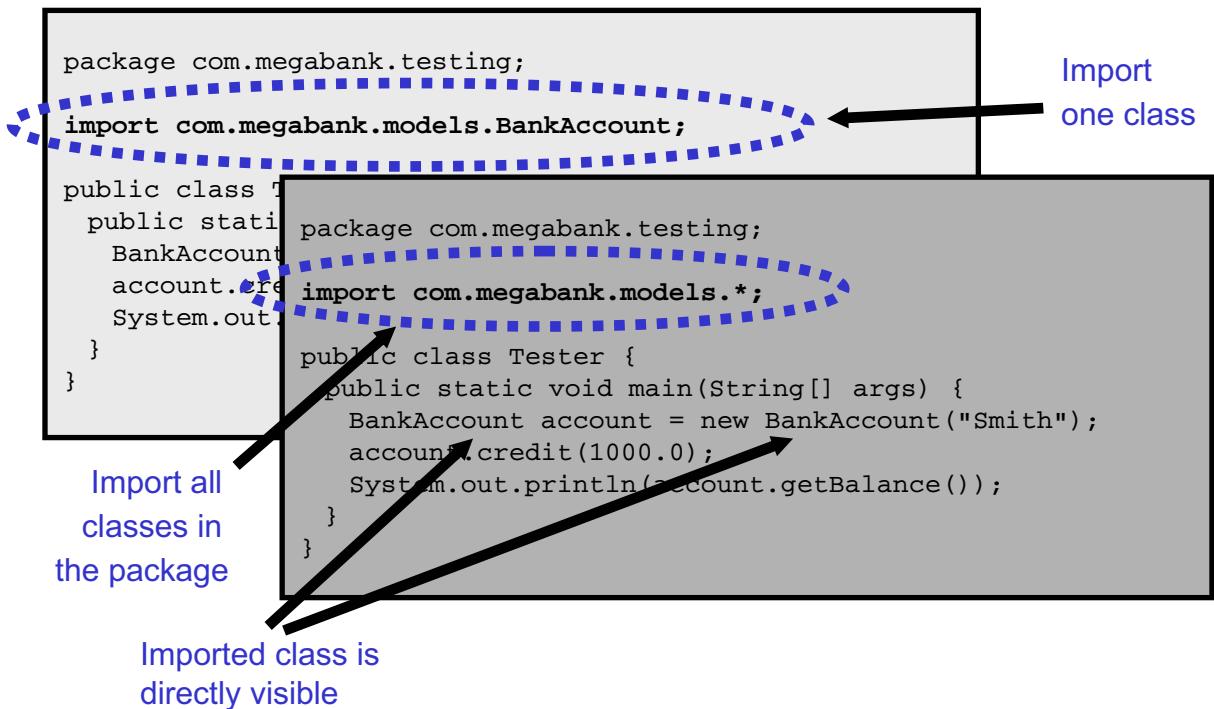


Figure 7-32. import statement

SW2446.0

### Notes:

## Java 1.4 packages

---

- java.applet
- java.awt
- java.beans
- java.io
- java.lang
- java.math
- java.net
- java.nio
- java.rmi
- java.security
- java.sql
- java.text
- java.util
- javax.accessibility
- javax.crypto
- javax.imageio
- javax.naming
- javax.net
- javax.print
- javax.rmi
- javax.security
- javax.sound
- javax.sql
- javax.swing
- javax.transaction
- javax.xml
- org.ietf.jgss
- org.omg.CORBA
- org.omg.CosNaming
- org.omg.Dynamic
- org.omg.IOP
- org.omg.Messaging
- org.omg.PortableInterceptor
- org.omg.PortableServer
- org.omg.SendingContext
- org.omg.stub.java.rmi
- org.w3c.dom
- org.xml

---

Figure 7-33. Java 1.4 packages

SW2446.0

### **Notes:**

The javax packages began life as extensions to Java, but some are now part of J2SE. Note that some packages have both java and javax versions.

# Core Java packages

## •java.lang

- Provides classes that are fundamental to the design of the Java programming language
  - Includes wrapper classes, String and StringBuffer, Object, and so on
  - Imported implicitly into all classes

## •java.util

- Contains the collections framework, event model, date and time facilities, internationalization, and miscellaneous utility classes

## •java.io

- Provides for system input and output through data streams, serialization and the file system

## •java.math

- Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal)

## •java.sql

- Provides the API for accessing and processing data stored in a data source (usually a relational database)

## •java.text

- Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages

Figure 7-34. Core Java packages

SW2446.0

## Notes:

## Sample package: java.lang

---

- Contains the following classes:
  - Basic Entities
    - Class, Object, Package, System
  - Wrappers
    - Number, Boolean, Byte, Character, Double, Float, Integer, Long, Short, Void
  - Character and String Manipulation
    - Character.Subset, Character.UnicodeBlock, String, StringBuffer
  - Math Functions
    - Math, StrictMath
  - Runtime Model
    - Process, Runtime, Thread, ThreadGroup, ThreadLocal, InheritableThreadLocal, RuntimePermission
  - JVM
    - ClassLoader, Compiler, SecurityManager
  - Exception Handling
    - StackTraceElement, Throwable
- Also contains Interfaces, Exceptions and Errors

---

Figure 7-35. Sample package: java.lang

SW2446.0

### **Notes:**

Recall that the java.lang package is imported into all Java classes implicitly.

## Sample class: String

- Sample Constructors:

- String()
- String(byte[] bytes)
- String(byte[] bytes, int offset, int length)
- String(char[] value)
- String(char[] value, int offset, int length)
- String(String original)
- String(StringBuffer buffer)

- Sample Methods:

- char charAt(int index)
- boolean equals(Object anObject)
- int indexOf(String str)
- int length()
- boolean matches(String regex)
- String substring(int beginIndex, int endIndex)
- String toUpperCase()
- String trim()

---

Figure 7-36. Sample class: String

SW2446.0

**Notes:**

## Sample class: StringBuffer

---

- Constructors:

- `StringBuffer()`
- `StringBuffer(int length)`
- `StringBuffer(String str)`

- Sample Methods:

- `StringBuffer append(...)`
- `StringBuffer insert(...)`
- `StringBuffer delete(int start, int end)`
- `int length()`
- `StringBuffer reverse()`
- `String substring(int start, int end)`
- `String toString()`

---

Figure 7-37. Sample class: StringBuffer

SW2446.0

### **Notes:**

## **Checkpoint**

1. What are `super` and `this`?
2. How can the garbage collector be explicitly invoked?
3. What is the difference between overloading and overriding?
4. What kind of class should have a `main` method?
5. How are constants declared in Java?
6. What is an abstract class and what is it used for?
7. Which package contains the collection classes?



Figure 7-38. Checkpoint

SW2446.0

### **Notes:**

## **Unit summary**

---

- In this unit, you should have learned to:
  - Declare a Java class
  - Define constructors
  - Create methods and fields, and set the appropriate modifier
  - Explain how memory is managed in Java
  - Outline the role of packages in Java

---

Figure 7-39. Unit summary

SW2446.0

### **Notes:**

# Unit 8. Debug Applications

## What this unit is about

This unit introduces the Application Developer debugging capabilities. You learn to step through code, add and configure breakpoints, and view and modify variable values during debugging.

## What you should be able to do

After completing this unit, you should be able to:

- Use the **Debug** perspective to debug Java applications
- Step through code using the **Debug** view toolbar
- Add and configure breakpoints in a Java application
- View and change variables during debugging
- Execute and inspect expressions during debugging

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 8.1 Debug Applications

## Unit objectives

---

- After completing this unit, you should be able to:
  - Use the **Debug** perspective to debug Java applications
  - Step through code using the **Debug** view toolbar
  - Add and configure breakpoints in a Java application
  - View and change variables during debugging
  - Execute and inspect expressions during debugging

---

Figure 8-1. Unit objectives

SW2446.0

### **Notes:**

## Debug tooling

- Application Developer provides a visual debugging tool for finding problems in Java code
  - Launch the debug tool using the **Debug** button  on the toolbar
  - The workbench prompts you to open the **Debug** perspective and provides debugging capabilities
- Debug tool capabilities:
  - Set breakpoints in the Java editor
  - Modify data values while stepping through code
  - Modify and apply changes to the application while debugging

---

Figure 8-2. Debug tooling

SW2446.0

### **Notes:**

## Debug perspective

- Debug perspective allows you to inspect, change, and step through code
- Source can be changed and saved, and execution resumed

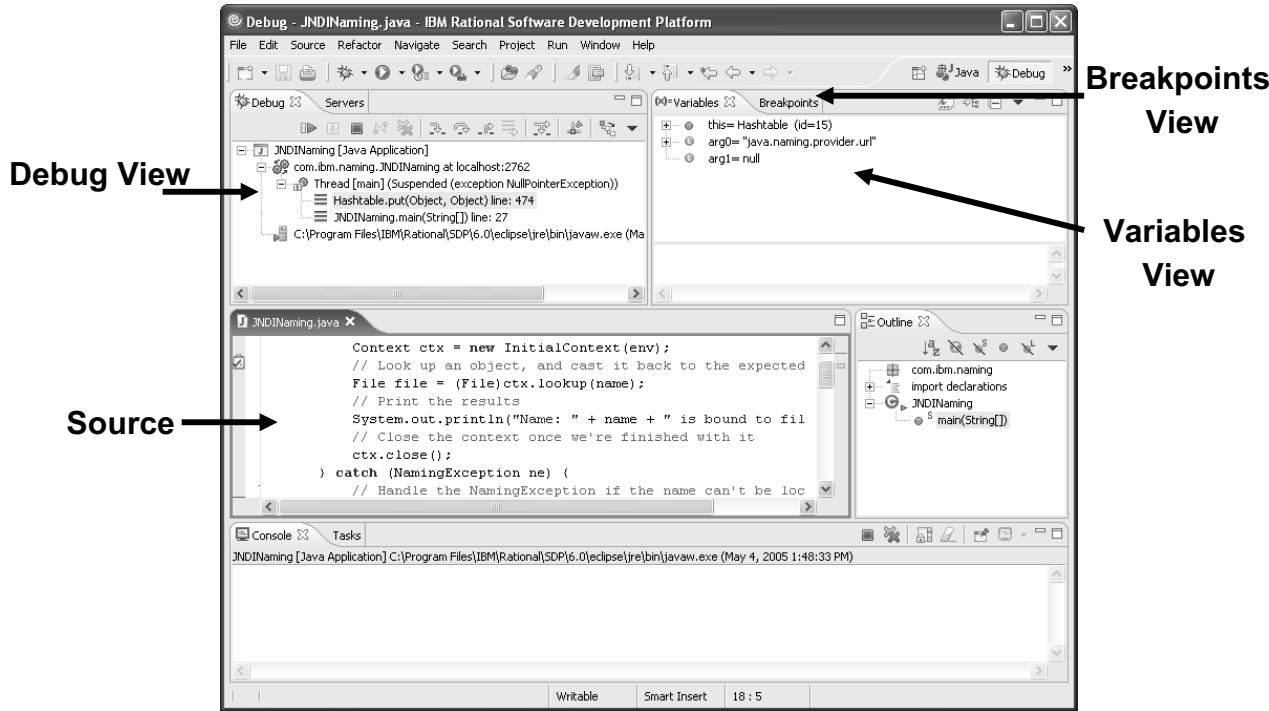


Figure 8-3. Debug perspective

SW2446.0

### Notes:

The **Debug** perspective is designed for debugging your Java program. It includes an editor area and the following views:

- Debug
- Breakpoints
- Expressions
- Variables
- Display
- Outline
- Console

## Debug view

- **Debug** view displays:
  - The process for each target being run
  - A list of stack frames for each target being debugged
    - A stack frame corresponds to a method called before the exception or breakpoint occurred
    - Stack frames are in reverse chronological order (last one was executed first)
  - Terminated launches

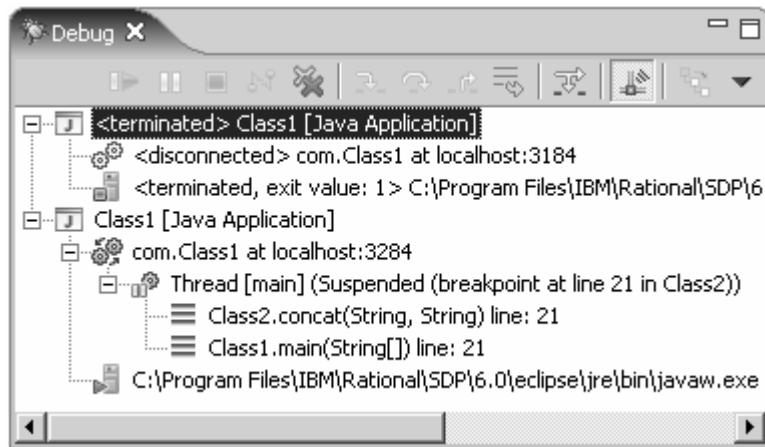


Figure 8-4. Debug view

SW2446.0

### Notes:

## Debug view navigation

- When execution has been suspended at a breakpoint, the method source for the current stack frame can be executed line by line using the debugger's navigation buttons



1. Resume/suspend/terminate execution
2. Step into/over/return
3. Drop to frame
4. Enable step filters/step debug
5. Enable step-by-step mode

Figure 8-5. Debug view navigation

SW2446.0

### **Notes:**

Step filters allow you to avoid debugging internal WebSphere Application Server methods if you are developing an Enterprise Java application, so that you only debug your application. You can also filter out any other external packages that you do not want to debug.

## Stepping through code commands

1. **Resume** - Continues a suspended thread's execution until a breakpoint is encountered or the thread ends
2. **Suspend** - Suspends a running thread so that you can step through code, inspect, and so on
3. **Terminate** - Ends the execution of the selected debug target
4. **Remove All Terminated Launches** - Clears all terminated debug targets from the view
5. **Step Into** – Steps into a method call and executes the first line of code in that method; steps into the highlighted statement
6. **Step Over** - Executes the next line of code in the current method; steps over the highlighted statement
7. **Step Return** - Continues execution until the end of the current method (until a return)
8. **Drop to Frame** – Allows you to drop back and reenter a specified stack frame
9. **Step with Filters** – Continues execution until the next line of code which is not filtered out



Figure 8-6. Stepping through code commands

SW2446.0

### Notes:

The **Step Over** feature steps over the highlighted statement. Execution will continue at the next line in the same method, or, if you are at the end of a method, it will continue in the method from which the current method was called. The cursor jumps to the declaration of the method and selects this line.

The **Drop to Frame** feature is similar to “running backwards” and restarting your program part-way through.

Step filtering is a preference, controlled via **Window -> Preferences -> Java -> Debug -> Step Filtering**.

## Breakpoints

- Breakpoints are temporary markers you place in your program to tell the debugger to suspend execution of your program at that point
- Reaching a breakpoint in a method causes the **Debug** perspective to open
  - The method at the breakpoint appears on the stack frame
  - Source can then be stepped through and objects inspected
- Breakpoints are set until they are explicitly removed from a method
  - They can be temporarily disabled
- The breakpoint where execution has paused is marked with an arrow



Figure 8-7. Breakpoints

SW2446.0

### Notes:

You can choose which perspective opens when the debugger encounters a breakpoint (preference setting). The most convenient perspective for this purpose is the **Debug** perspective.

When a breakpoint is enabled, it will cause a thread to suspend whenever it is hit.

1. Enabled breakpoints are indicated by a blue circle.
2. When a breakpoint is disabled, it will not cause threads to suspend. Disabled breakpoints are indicated by a white circle.
3. Enabled breakpoints that are successfully installed in a class in a VM at run time are indicated by a checkmark overlay.
4. Conditional breakpoints have a question mark overlay.
5. The breakpoint at which execution is currently suspended is indicated by an arrow.

## Breakpoints view

- Lists all the breakpoints you have set in the workbench as well as Java Exception breakpoints.
  - Double-click a breakpoint to display its location
  - Add, delete, enable, or disable breakpoints

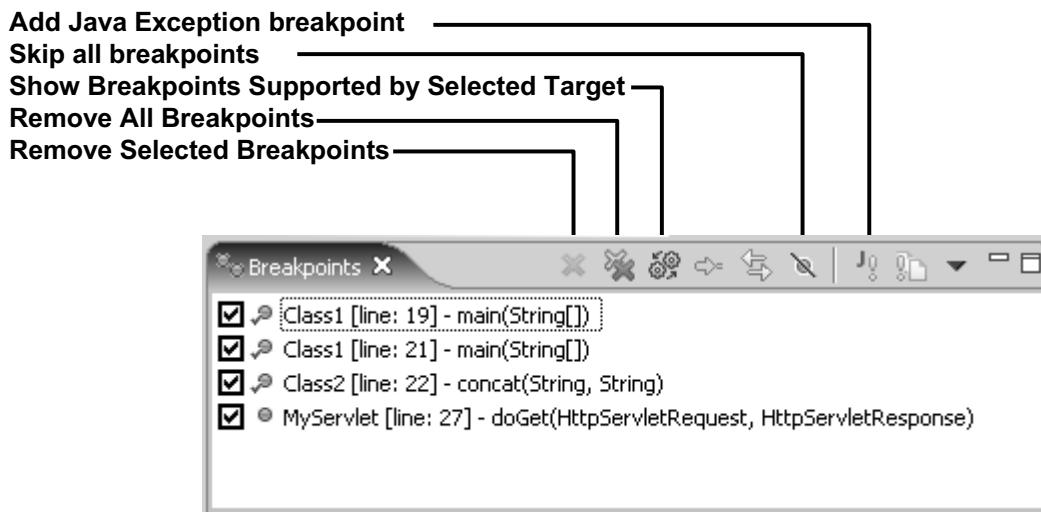


Figure 8-8. Breakpoints view

SW2446.0

### Notes:

Java exception breakpoints suspend execution at the point where the exception is thrown. Execution can be suspended when exception is uncaught, caught or both.

Most of the toolbar icons on the **Breakpoints** view are self-explanatory. The **Show Breakpoints Supported by Selected Target** button deserves an explanation:

Normally a list of all breakpoints (for all debug sessions) appears in the **Breakpoints** view, unless you use the *filter by debug target* action.

To filter out breakpoints that are not related to the current debug session, click the **Breakpoints** view **Show Breakpoints Supported by Selected Target** push button, or right-click in the view and enable the **Show Supported Breakpoints** pop-up menu item (this menu item is enabled when there is a check mark to its left).

## Adding a breakpoint into a Java application

- In the editor area, open the file where you want to add the breakpoint
- In the editor, directly to the left of the line where you want to add the breakpoint, right-click the marker bar (vertical ruler) and select **Toggle Breakpoint** from the pop-up menu.



Figure 8-9. Adding a breakpoint into a Java application

SW2446.0

### **Notes:**

You can also double-click the marker bar, which performs a toggle action.

The debugger will stop immediately before executing the line with the breakpoint.

Breakpoints can be disabled from the **Breakpoints** view or from the pop-up menu.

Java exception breakpoints can only be set from the **Breakpoints** view.

## Where can breakpoints be set?

- Breakpoints are set on an executable line of a program
- If you try to add a breakpoint on a line where there is no executable code, Application Developer will either warn you with a dialog or status message (shown on the right) or add a breakpoint at the next possible location (in Java code)

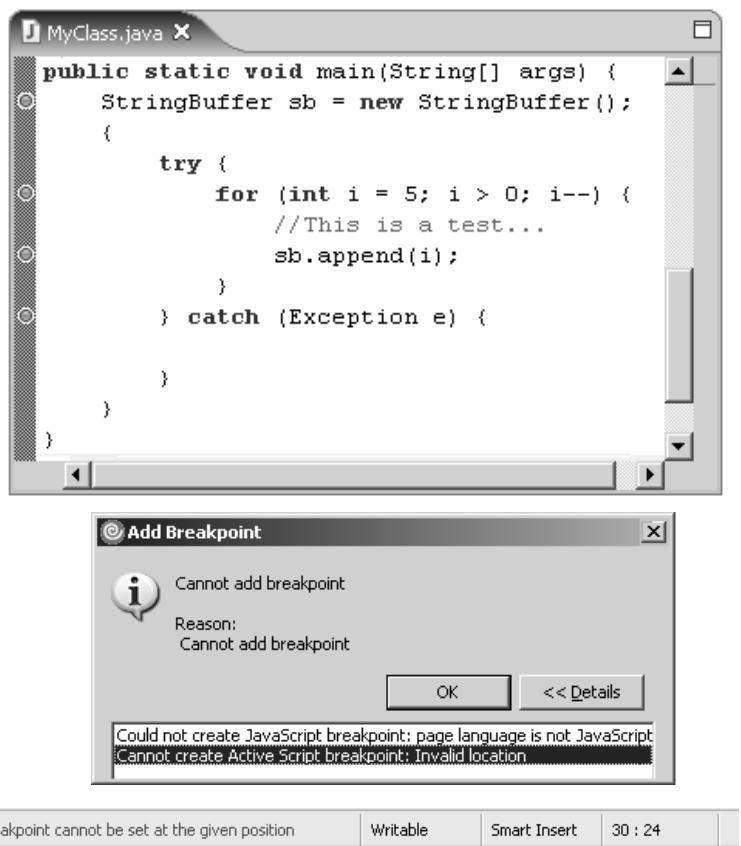


Figure 8-10. Where can breakpoints be set?

SW2446.0

### Notes:

The upper image shows every line that can have a breakpoint. If you try to set one on, for example, the brace after the `StringBuffer` declaration, Application Developer will add one on the `for` loop line. If you try to set one after all executable code (on one of the three closing braces, for example) the attempt will be ignored, and the status message "Breakpoint cannot be set at the given position" will be displayed at the bottom of the Application Developer window as shown in the bottom image.

## Configuring breakpoint properties

- Breakpoint properties can be set in their **Properties** dialog
  - Right-click a breakpoint in the **Breakpoints** view or the vertical ruler to access the **Properties** dialog.
- Breakpoints can be enabled or disabled
  - Newly added breakpoints are enabled by default
- Hit count breakpoints
  - The thread execution suspends when the breakpoint is hit for the  $n^{\text{th}}$  time
  - The breakpoint is disabled until either it is re-enabled or its hit count is changed
- Conditional breakpoints
  - Must evaluate to a boolean
  - Only invoked if condition evaluates to true

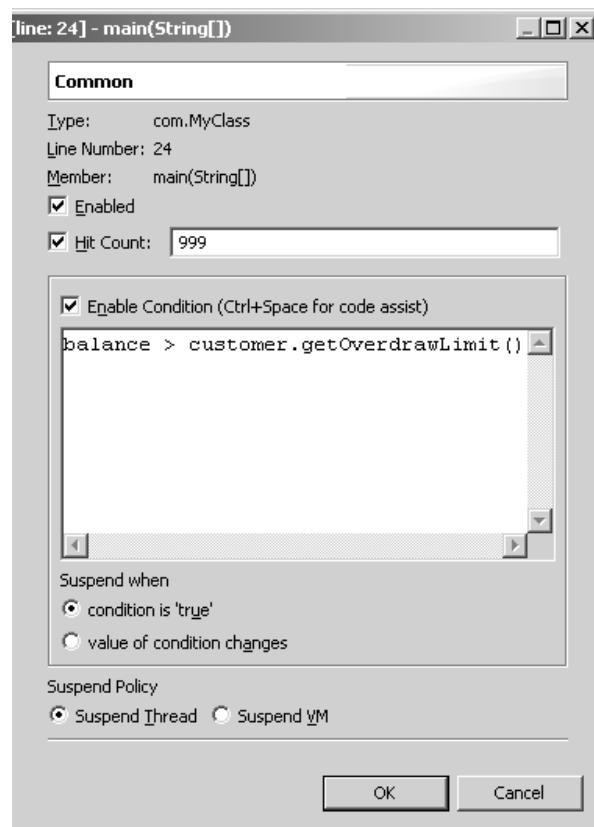


Figure 8-11. Configuring breakpoint properties

SW2446.0

### Notes:

To access the **Properties** dialog for a breakpoint, right-click a breakpoint in the **Breakpoints** view or the vertical ruler. If you are in the **Breakpoints** view, select **Properties...** from the pop-up menu, and if you right-clicked a breakpoint in the vertical ruler of a Java editor, select **Breakpoint Properties...** from the pop-up menu.

The hit count must be a single positive integer.

The condition can be complex, and might suspend execution several times. Any expression is valid as long as it evaluates to a boolean:

- `Name.equals(employee.getName())`
- `Double.compare(Math.random(), 0.5) == 0`
- `thisTemperature != previousTemperature`
- Execution suspends either when the condition is true or when the value changes.

## The Variables view

- Examine the contents of variables when a thread suspends
  - Top stack frame of the thread is automatically selected
  - Visible variables of the selected stack frame are displayed
- The '+' next to a variable name indicates that it is an object
  - The object can be expanded to reveal the values of its attributes
  - Nested objects can also be expanded

```
public String concat(String a, String b) {
    String resultString = a + " " + b;
    return resultString;
}
```

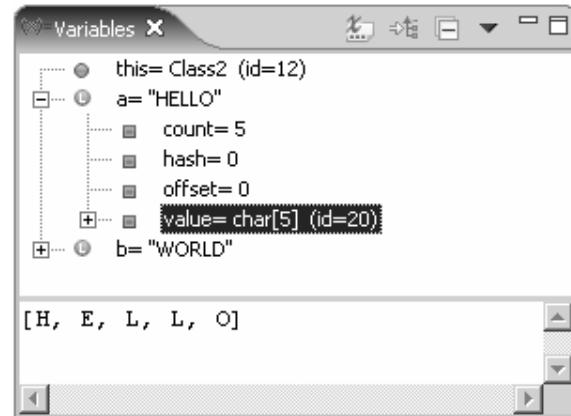


Figure 8-12. The Variables view

SW2446.0

### Notes:

A visible variable is one that has been initialized. Notice that in the above code, execution is suspended on the line where `resultString` is created. The variable `resultString` is not yet visible, so it is not in the **Variables** view. Stepping one line further will add it.

## Viewing and changing variables

- Use the **Variables** view to change the value of a variable
  - Right-click on a variable and select **Change Variable Value** from the pop-up menu
- **Show type names** adds the type before variable identifiers
- Fully qualified names can be displayed

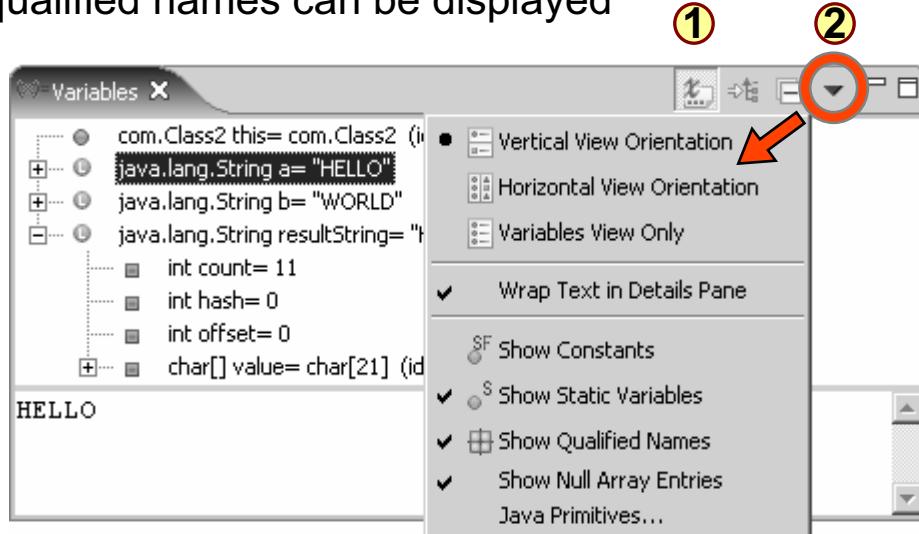


Figure 8-13. Viewing and changing variables

SW2446.0

### Notes:

Note three differences between this image and the one on the previous slide:

1. **Show Type Names** is selected, so 'b' is shown as `String b`.
2. **Show Qualified Names** is checked, so `String` is shown as `java.lang.String`.

The debugger has executed one more line of code, so the variable `resultString` is visible.

## Data in the Expressions view

- Data from a scrapbook page or a stack frame of a suspended thread can be inspected in the **Expressions** view
- The view is not initially open in the **Debug** perspective
  - It opens automatically when an item is added to the view
- To inspect data in this view, select a variable in the code and press **Ctrl-Shift-i** twice
  - The first time opens a window with the expression
  - The second time moves the contents of the window to the **Expressions** view

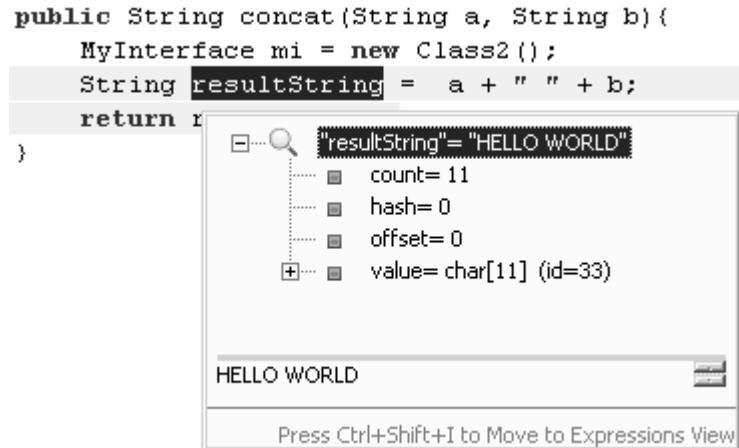


Figure 8-14. Data in the Expressions view

SW2446.0

### Notes:

The **Expressions** view is also opened if you right-click an item and choose **Inspect** from the pop-up menu, and then press **Ctrl-Shift-I** to move to the **Expressions** view.

## **Checkpoint questions**

---

1. In the middle of debugging a method, you decide you want to skip ahead to the point where control is handed back to the caller. What can you do?
2. What information is shown in the **Debug** view?
3. How do you know that a breakpoint has some condition set?
4. On what lines of code are you able to enter a breakpoint?



---

Figure 8-15. Checkpoint questions

SW2446.0

### **Notes:**

## Unit summary

---

- In this unit, you should have learned to:
  - Use the **Debug** perspective to debug Java applications
  - Step through code using the **Debug** view toolbar
  - Add and configure breakpoints in a Java application
  - View and change variables during debugging
  - Execute and inspect expressions during debugging

---

Figure 8-16. Unit summary

SW2446.0

### **Notes:**



# Unit 9. Inheritance

## What this unit is about

This unit introduces the concept and implementation of inheritance in Java. Related concepts such as overriding methods, default constructors and polymorphism are also discussed.

## What you should be able to do

After completing this unit, you should be able to:

- Describe the inheritance of fields and methods
- Explain the concept of a class hierarchy
- Outline how subclasses specialize superclasses
- Explain how method lookup works
- Create and use subclasses
- Describe how polymorphism is implemented

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 9.1 Inheritance

## **Unit objectives**

---

- After completing this unit, you should be able to:
  - Describe the inheritance of fields and methods
  - Explain the concept of a class hierarchy
  - Outline how subclasses specialize superclasses
  - Explain how method lookup works
  - Create and use subclasses
  - Describe how polymorphism is implemented

---

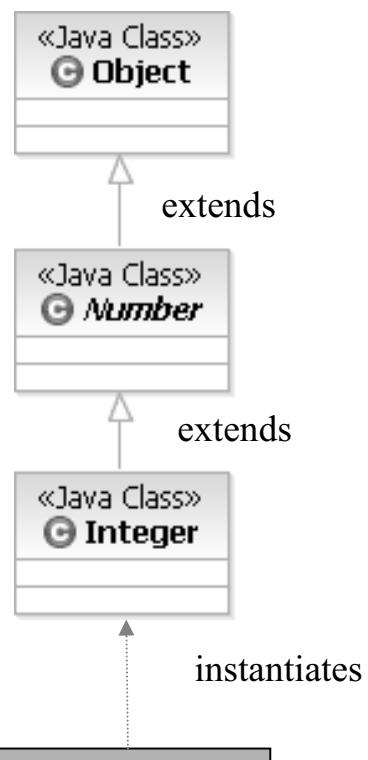
Figure 9-1. Unit objectives

SW2446.0

### **Notes:**

## Class hierarchies

- Every object belongs to a class
  - The objects that are members of a class are its instances
- Every class (except Object) has a superclass
  - In Java, Object is the root of the entire class hierarchy
- When defining new classes, the developer must decide which class is the appropriate superclass



```
Integer zero = new Integer(0);
```

Figure 9-2. Class hierarchies

SW2446.0

### Notes:

## Specialization and generalization

- A subclass is a *specialization* of its superclass
  - Specialize the state and behavior in the subclass by adding fields and extending or changing methods
  
- A superclass is a *generalization* of its subclasses
  - Common state and behavior can be moved to the superclass where it is available to all subclasses
  - Code is written once, and maintained in one place

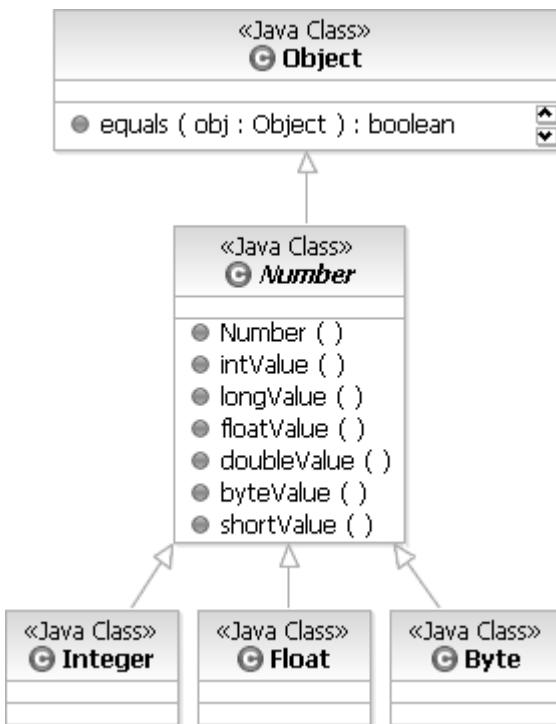


Figure 9-3. Specialization and generalization

SW2446.0

### Notes:

## Inheritance relationships

- Inheritance describes relationships where an instance of the subclass is a special case of the superclass
  - Inheritance relationships are called "**is a**" relationships because the subclass "is a" special case (specialization) of the superclass
    - Examples: A floating point number *is a* number; a number *is an* object
- Inheritance cannot describe "*has a*" relationships
  - This is usually done using the class' fields and their associated methods
    - Example: A floating point number *has a* sign, a radix and a mantissa
  - Collections provide generalized ways to handle "has a" relationships

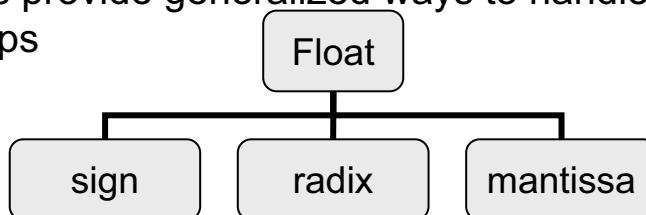


Figure 9-4. Inheritance relationships

SW2446.0

### Notes:

## Multiple inheritance

- Java does not support multiple inheritance
  - Every Java class except Object has exactly one immediate superclass (Object does not have a superclass)
- You can force classes that are not related by inheritance to implement a common set of methods using *interfaces* (this will be discussed in a later section)

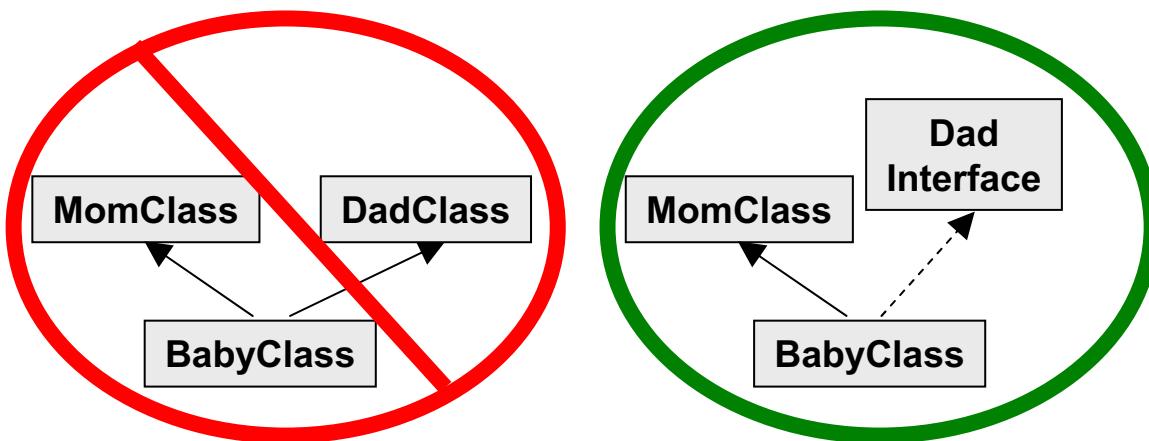


Figure 9-5. Multiple inheritance

SW2446.0

### Notes:

## Inheriting fields and methods

- Each subclass inherits the fields of its superclass
  - These fields in the superclass may have been inherited from classes even further up in the class hierarchy
  
- Each subclass inherits the methods of its superclass
  - An object will understand all messages which its class has implemented or its superclass has either inherited or implemented

```
Integer zero = new Integer(0);
if (zero.equals(x)) {
    byte b = zero.byteValue();
    ...
}
```

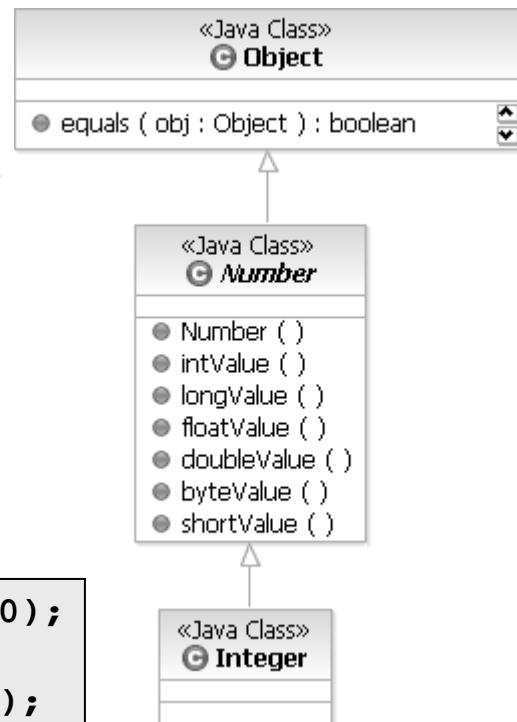


Figure 9-6. Inheriting fields and methods

SW2446.0

### Notes:

Notice how **equals()** and **byteValue()** are not explicitly declared for **Integer**, but are nevertheless considered its methods.

## ***Access modifiers***

---

- Variables and methods in Java have access restrictions, described by the following access modifiers:

- private**

- Access is limited to the class in which the member is declared
  - Example: `private int x;`

- default (this means no modifier is used)**

- Access is limited to the package in which the member is declared
  - Example: `int x;`

- protected**

- Access is limited to the package in which the member is declared, as well as all subclasses of its class
  - Example: `protected void setName() { . . . }`

- public**

- The member is accessible to all classes in all packages
  - Example: `public String getName() { . . . }`

---

Figure 9-7. Access modifiers

SW2446.0

### **Notes:**

## Overriding methods

- You can extend or change superclass behavior by overriding the inherited method in the subclass
- To override a superclass's method, create a new method in the subclass with the same signature (name and parameter list)
  - Java uses the new method in place of the inherited one
  - This new method replaces or refines the method of the same name in the superclass
  - If the signatures are different, the Java interpreter will consider the superclass and the subclass method to be distinct

```
public class MyClass extends Object {
    public boolean equals(Object o) {
        if (o==null)
            ...
    }
}
```

Figure 9-8. Overriding methods

SW2446.0

### **Notes:**

It is never necessary to explicitly declare that a class “extends Object”, but in this case you wanted to underline the fact that MyClass inherited `equals()` from the Object class.

In Java, methods are considered to be the same if they have the same name and argument list, regardless of return type. If the return types are different, the compiler will complain since it interprets the subclass method as overriding the inherited method from the superclass.

## ***Restrictions on overriding methods***

---

- Restrictions on the new method:
  - The parameter list must match the inherited method exactly
  - The return type must be the same as that of the inherited method
  - The access modifier must not be more restrictive than that of the inherited method
    - For example, if overriding a protected method, the new method can be protected or public, but not private

---

Figure 9-9. Restrictions on overriding methods

SW2446.0

### **Notes:**

## Example of overriding

---

```
public class BankAccount {  
    private float balance;  
    public float getBalance() {  
        return balance;  
    }  
}  
  
public class InvestmentAccount  
    extends BankAccount {  
    private float cashAmount;  
    private float investmentAmount;  
    public float getBalance() {  
        return cashAmount + investmentAmount;  
    }  
}
```

---

Figure 9-10. Example of overriding

SW2446.0

### **Notes:**

## **Method lookup**

---

- The compiler looks up the implementation for a method call beginning in the object's class definition
- If the method is not found in the object's own class, the search continues in the superclass and up the hierarchy until it is found
- When the method is found, it is invoked on the object to which the message was passed
- If the method was never implemented in any of the classes in the hierarchy, an error is issued at compile time

---

Figure 9-11. Method lookup

SW2446.0

### **Notes:**

## Inheritance and static methods

- A class can call all static methods defined in its superclass as though they were defined in the class itself
- Static methods can be hidden by static methods in the subclass

```
superclass
static String t = "test";
public static String superTest(String s) {
    s += " was the arg.";
    return s;
}
```

```
subclass
public static void main(String[] args){
    System.out.println(superTest(t));
}
```

Figure 9-12. Inheritance and static methods

SW2446.0

### **Notes:**

See the Java Tutorial for more information on hiding static methods, available at:  
<http://java.sun.com/docs/books/tutorial/java/javaOO/override.html>

## **Inheritance and constructors**

---

- Only constructors within the class being instantiated and within the immediate superclass can be invoked
- A constructor can call another constructor in its superclass using the keyword `super` and the parameter list
  - The parameter list must match that of an existing constructor in the superclass
- Constructors in the same class are invoked with the keyword `this` and the parameter list
- The first line of your constructor can be one of:
  - `super(...);`
  - `this(...);`

---

Figure 9-13. Inheritance and constructors

SW2446.0

### **Notes:**

## ***The superclass in object construction***

---

- Superclass objects are built before the subclass
  - The compiler supplies an implicit `super()` call for all constructors
  - `super(...)` initializes superclass members
- If the first line of your constructor is not a call to another constructor, `super()` is called automatically
  - Zero-argument constructor in the superclass is called as a result
  - This can cause an error if the superclass does not have a zero-argument constructor

---

Figure 9-14. The superclass in object construction

SW2446.0

### **Notes:**

## **Default constructors**

---

- If you do not provide any constructors, a default zero-argument constructor is provided for you
  - The default zero-argument constructor just makes a call to `super()`
- If you implement any constructor, Java will no longer provide you with the default zero-argument constructor
  - You can write your own zero-argument constructor which behaves like the default constructor (that is, just makes an implicit call to `super()`)

---

Figure 9-15. Default constructors

SW2446.0

### **Notes:**

## ***More on this and super***

---

- `this` and `super` can be used in instance methods and constructors in Java
  - They cannot be used in class (static) methods
- `this` and `super` affect method lookup
  - `this`: method lookup starts in the current class
  - `super`: method lookup starts in the immediate superclass
- The keyword `super` lets you use code from a superclass when overriding a method in the current class
  - When a subclass overrides a superclass method, calls to that method will go to the code that overrides the superclass method
  - Using `super` allows you to look up methods starting from the immediate superclass

---

Figure 9-16. More on `this` and `super`

SW2446.0

### **Notes:**

The keyword `this` represents the current object, which is why it cannot be used in class methods. The keyword `super` refers to the superclass of the class in which the keyword is used. As a standalone statement, `super()` is a call to a constructor of the superclass. As a method call, `super.methodName()` is used to call a method in the superclass. Using the `super` keyword to call a method is only needed when calling a method that is overridden in the class where the `super` keyword is used.

## **Polymorphism**

---

- A compact car, luxury car, and sports car are several classes of car that accept the same set of messages, and provide the same service
  - Each car accepts the `accelerate()`, `decelerate()`, `steer()` and `calculateMilesToEmpty()` messages, allowing you to drive to a destination
- The service may be implemented differently by each car, but these classes may be interchanged without affecting the driver who sends messages to the vehicle
  - This principle is known as *polymorphism*

---

Figure 9-17. Polymorphism

SW2446.0

### **Notes:**

## Polymorphism through inheritance

- In Java, classes that inherit from the same superclass inherit the same methods, and can respond to the same messages
  - CompactCar, LuxuryCar and SportsCar can respond to the same messages, and be interchanged without affecting the message sender, a driver
  - Inheritance is one way that Java implements polymorphism
  - See the inheritance relationship between classes that model these cars:

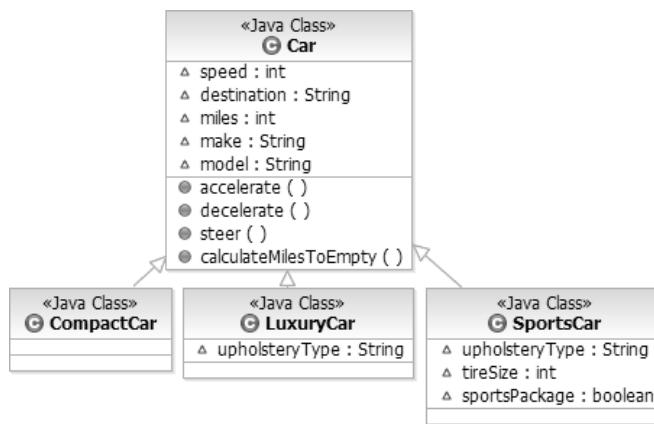


Figure 9-18. Polymorphism through inheritance

SW2446.0

### Notes:

## Implementing polymorphism

---

- A variable can be assigned objects of its declared type, or subtypes of its declared type
  - For example:

```
Car auto = new Car();  
Car auto = new CompactCar();  
Car auto = new LuxuryCar();  
Car auto = new SportsCar();
```
- Assigning an object of one type to an object of another type (higher in the hierarchy), will make the object forget its real type
  - From above, **auto** will no longer know that it is an object of class **CompactCar**, and will only respond to messages for the **Car** class
  - You can get these objects to remember their real type by casting the object to that type, such as:

```
CompactCar cc = (CompactCar) auto;
```
  - Casting to an unrecognized subclass will throw a **ClassCastException**

---

Figure 9-19. Implementing polymorphism

SW2446.0

### **Notes:**

Any object can be assigned to an object of type **Object** since it is at the top of the hierarchy.

For example:

```
Object anything = new AnyType();
```

You can determine the subclass of an object by testing against a known subclass type using the keyword **instanceof**. This is an expensive operation, but it may be used with conditional logic to determine which method call is appropriate if polymorphism has not been implemented.

## Example

- Since the variables forget their type once declared with the type of the superclass, they will only respond to messages in the Car type
- They can respond to the same messages and may be interchanged without affecting the message sender

```
public class Driver {
    public static void main(String[] args) {
        Car auto = new CompactCar();
        auto.accelerate();
        if(stopLight.equals("red")) {
            auto.decelerate();
            auto.accelerate();
        }
        if(corner == true) {
            auto.steer();
        }
    }
}
```

Figure 9-20. Example

SW2446.0

### **Notes:**

In the above example, you could replace the line:

```
Car auto = new CompactCar();
```

with any one of the following lines:

```
Car auto = new LuxuryCar();
Car auto = new SportsCar();
Car auto = new Car();
```

This replacement could be made without affecting the rest of the code in the Driver class, which sends the car the messages telling it when to accelerate, decelerate and steer. Each Car, whether it is a CompactCar, LuxuryCar, SportsCar or Car object, will respond to those messages. Each Car may implement the methods that handle these messages differently, but from a message sender's perspective, these objects are interchangeable. For example, the CompactCar may have an implementation of accelerate() that requires it to push in the clutch and shift the gears on its manual transmission and gently push the gas pedal, while the SportsCar may have an implementation of accelerate() that causes the gas

pedal to hit the floor and the tires to squeal. From the perspective of the Driver class, however, the message to accelerate is sent in the same manner, and the car accelerates in response.

## Checkpoint

1. What are “is-a” and “has-a” relationships, and what do they have to do with class hierarchies?
2. What are the four Java access modifiers, and how do they limit the scope of class members?
3. What is wrong with the following code?

```
public class MyClass extends MySuper {  
    MyClass() {  
        super();  
    }  
    MyClass(int i) {  
        super();  
        this();  
    }  
}
```



Figure 9-21. Checkpoint

SW2446.0

### **Notes:**

## **Unit Summary**

---

- In this unit, you should have learned to:
  - Describe the inheritance of fields and methods
  - Explain the concept of a class hierarchy
  - Outline how subclasses specialize superclasses
  - Explain how method lookup works
  - Create and use subclasses
  - Describe how polymorphism is implemented

---

Figure 9-22. Unit summary

SW2446.0

### **Notes:**

# Unit 10. Design Patterns and Refactoring

## What this unit is about

This unit explains design patterns, and their role in refactoring. Refactoring is also explained, and Application Developer's support for refactoring is outlined.

## What you should be able to do

After completing this unit, you should be able to:

- Explain what design patterns are and how they are used
- Outline the different types of design patterns and list the key elements of a design pattern
- Describe what refactoring is and why it is needed
- Outline when refactoring is performed in the development cycle
- List various strategies and techniques of refactoring
- Use Application Developer to refactor Java classes and elements

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 10.1 Design Patterns and Refactoring

## **Unit objectives**

---

- After completing this unit, you should be able to:
  - Describe what a design pattern is
  - Outline the different types of design patterns and list the key elements of a design pattern
  - Describe what refactoring is and why it is needed
  - Outline when refactoring is done in the development cycle
  - List various strategies and techniques of refactoring
  - Use Application Developer to refactor Java classes and elements

---

Figure 10-1. Unit objectives

SW2446.0

### **Notes:**

## **A focus on structure**

---

- Building a software system involves more than just the concerns of low-level programming
- Thought must be given to the system design as well
  - Design the system before coding
- Planning enhances the quality of the system
  - Plan for system maintenance
  - Plan for change; incorporate extensibility into the system design
- There are many existing solutions to recurring system design problems
  - These solutions are best practices of object-oriented system design, called *design patterns*

---

Figure 10-2. A focus on structure

SW2446.0

### **Notes:**

## ***What is a design pattern?***

---

- A design pattern might be described as:
  - A **named** description  
of a proven **design solution**  
to a **recurring problem**,  
within a given **context**
- Design patterns are essentially an “engineering handbook” for designing software systems
- Design patterns are one example of programming best practices
  - They help code become human-readable, reusable, better performing and easier to maintain

---

Figure 10-3. What is a design pattern?

SW2446.0

### **Notes:**

Design Patterns are abstract concepts, are not specific to a particular industry or type of application, and usually encompass a small number of classes. To apply a specific design pattern, you must simply code your class(es) a certain way, following the pattern described by the design pattern you wish to use.

Design patterns were popularized by the book *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995, written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This book organizes software patterns developed by the authors into a standard format.

A useful online resource is:

<http://www.patterndepot.com/put/8/JavaPatterns.htm>

A book by the same author that is similar to the Web page is:

James W. Cooper, Java Design Patterns: A Tutorial, Addison Wesley, 2000.

## Types of patterns

- Creational patterns – used to decide which class gets instantiated and who does the instantiation
- Structural patterns – control how objects and classes are structured to form larger structures
- Behavioral patterns – relate to the assignment of responsibilities, patterns of communication, and control flow

---

Figure 10-4. Types of patterns

SW2446.0

### **Notes:**

Some examples of creational design patterns are:

- Singleton
- Prototype
- Factory Method
- Builder

Some examples of structural design patterns are:

- Adapter
- Composite
- Proxy
- Façade

Some examples of behavioral design patterns are:

- Observer

- Mediator
- Chain of Responsibility
- Strategy

## Key elements of a pattern

- Intent – what problem this pattern solves
- Motivation – an example that shows how the pattern solves a problem
- Structure – a class, interaction diagrams, or both
- Consequences – benefits and risks of using this pattern

---

Figure 10-5. Key elements of a pattern

SW2446.0

### **Notes:**

This is the format used in the *Design Patterns* book for documenting the patterns.

## Example - Singleton design pattern

- The Singleton creational design pattern ensures that a class is instantiated only one time, and that the class provides access to this object
- An encapsulated static variable references the single instance of the class, and provides that reference to any class that requests this object

Singleton code

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton() {
        // Called only once by the getInstance() method.
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Singleton UML diagram

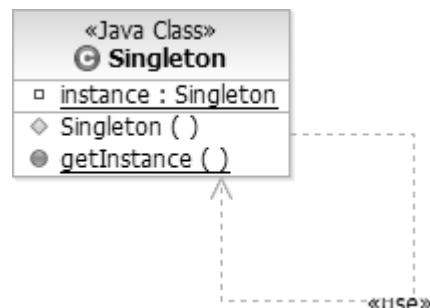


Figure 10-6. Example - Singleton design pattern

SW2446.0

### Notes:

Any class of any name may implement the Singleton design pattern by allowing only one object to be created in this way (the class does not have to be named Singleton). This pattern is very widely used. One situation where the Singleton pattern would be appropriate is if a class is used to maintain a counter or other such state information that is accessed and updated by several parts of the system. In this situation, the data needs to be shared by many different parts of the same system, and there should only ever be one instance of this class. Static fields of a class could be used to hold such data, but there would still be no way to ensure that only one instance of that class is running at one time. Using the Singleton pattern would ensure that only one instance of a class is running, and would ensure that the class handles this responsibility.

What is needed is a way to control how class instances are created and then ensure that only one gets created at any given time. This would give you exactly the behavior you require and free a client from having to know any class details."

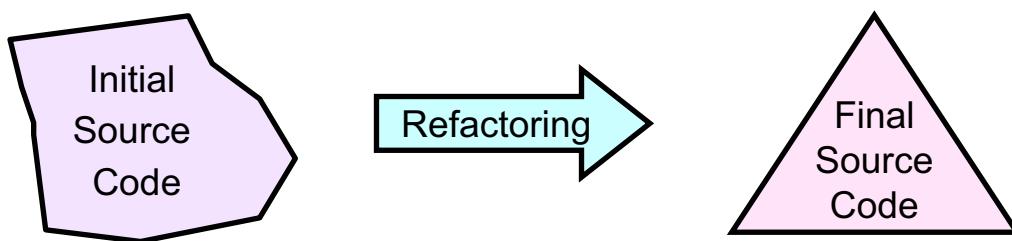
The Singleton design pattern is responsible for creating and providing access to a single instance of a class. The class may only be instantiated one time, and any object requesting

a class which implements the Singleton pattern will be given a reference to the same object.

## What is refactoring?

---

- Refactoring is the process of changing a software system in such a way that the external behavior of the code is unchanged, but its internal structure and architecture are improved
- It is a *behavior-preserving* source code transformation



---

Figure 10-7. What is refactoring?

SW2446.0

### **Notes:**

Even the best designed systems may have weaknesses in design or code implementation that are discovered after the system has been implemented.

Refactoring allows you to improve your code without changing its behavior; it is a way to improve existing code in a regulated way. Each refactoring code transformation may be small, but a series of transformations can produce a major restructuring of the internal structure and architecture of code. At the end of applying a transformation, the code will still have the same behavior.

There are many ways to refactor, and some can be done automatically by tools. Tools to automate many common refactorings exist in IBM Rational Application Developer.

A resource which provides more information on refactoring and how refactorings may be applied is *Refactoring: Improving the Design of Existing Code* by Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts.

## **Why do you need to refactor?**

---

- Refactoring code provides numerous benefits, as the internal structure and architecture of the code is enhanced
  - Unnecessary complexity eliminated
  - Improved readability and understandability
  - Improved maintainability
  - Improved extensibility
  - Enhanced flexibility
  - Improved performance

---

Figure 10-8. Why do you need to refactor?

SW2446.0

### **Notes:**

It may seem that reworking code after it has been written could be expensive or time-consuming, but the benefits provided by refactoring and having higher quality code in a system, in most cases, far outweigh the cost and time it takes to rework code. Refactoring keeps the design of your code simple and avoids needless complexity. Remember that your system may be in existence for many years to come, so spending time to produce high quality code now will save you time and money later when the system needs to be extended and maintained.

## **When do you refactor?**

---

- Refactoring should be done during each stage in the software development life cycle
  - Framework development
  - Application coding
  - Application testing
  - Application maintenance
- Internal code structure and architecture should be examined objectively and analytically at each stage in the development life cycle
  - Refactoring during each stage ensures that the best quality design and code is present during the next development stage
  - Changes are often easier if done early in development, and changes will be smaller if refactoring is done often

---

Figure 10-9. When do you refactor?

SW2446.0

### **Notes:**

A framework is a foundation on which an application will be built.

## The refactoring cycle

- Program source code will go through expansion and contraction phases
  - In an expansion phase, code is added to meet functional requirements
  - In a contraction phase, code is removed and refactored to better address those requirements and plan for the future
- This cycle will be repeated many times during the lifetime of a program

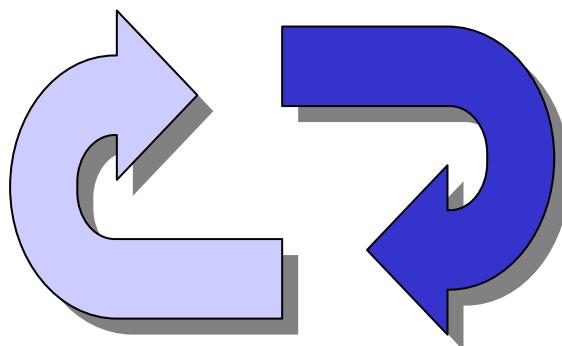


Figure 10-10. The refactoring cycle

SW2446.0

### **Notes:**

Since refactoring simplifies code, it will often result in a contraction of source code.

Refactoring is done on an existing codebase, so after code has been developed to meet functional requirements or to extend system functionality, this source code should be refactored to ensure code is as simple and well-designed as possible. Having code of the highest quality possible will ensure that the next code expansion phase is as easy as possible. Refactoring at the end of each expansion phase ensures that quality code is being worked on during the next expansion phase.

Testing should be done during refactoring to ensure that external behavior of the code is preserved.

## **To refactor or not to refactor**

---

- “Changing the design/code might break the system”
  - Use tests to prove behavior preservation
- “I don't understand how it works now”
  - Learn through the process; learning about the code structure now will be easier than learning it later in a critical situation, or trying to extend or maintain code that is not well understood
  - Build documentation as you refactor and simplify
- “I don't have the time to refactor”
  - Refactoring will pay for itself later, as the system will be more maintainable and extensible

---

Figure 10-11. To refactor or not to refactor

SW2446.0

### **Notes:**

Reasons for neglecting refactoring often include time and cost limitations for a project; remember that having high quality code in the system will reap many long-term benefits and will pay for itself in decreased maintenance cost and time, decreased time to extend the system, and increased system performance. Working with code that is not well-understood or well-designed will make future tasks difficult.

Think long term rather than short term.

## Refactoring strategies and techniques

- Refactoring strategies
  - Eliminating duplicated code
  - Long or complex method refactoring
  - Data clump refactoring
  - Conditional refactoring
- Refactoring techniques
  - Creating and removing classes, methods, variables
  - Moving methods, instance variables up/down a hierarchy
  - Moving methods, instance variables into other classes
  - Reorganizing inheritance, implementation hierarchies
  - Reorganizing method code

Figure 10-12. Refactoring strategies and techniques

SW2446.0

### **Notes:**

A comprehensive list of refactorings can be found at:

<http://www.refactoring.com/catalog/index.html>

This Web site provides a list of refactorings, including what situation each should be applied in, and how the refactoring can be applied to code.

Often during refactoring, design patterns are applied or consulted.

After each refactoring step, you must ensure the integrity of your system with testing. Use a test framework like JUnit. Tests should be passed before and after refactoring. Include both functional and performance tests. Take small steps with frequent tests.

## Refactoring tools

- The refactoring tools of Application Developer make it easier for the developer to refactor
- For example, when renaming a method, they will:
  - Browse for senders of the method
  - Display the proposed change and its implications
  - Make the change
  - Update the senders

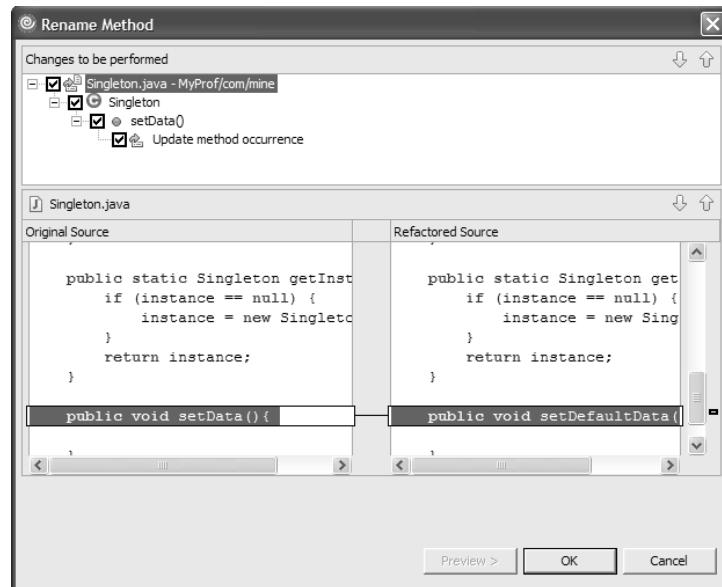


Figure 10-13. Refactoring tools

SW2446.0

### Notes:

## Using the refactoring tools

- Refactoring actions are available by right-clicking Java classes or elements, or by using the **Refactor** menu selection
- UML diagrams are updated automatically as part of refactoring

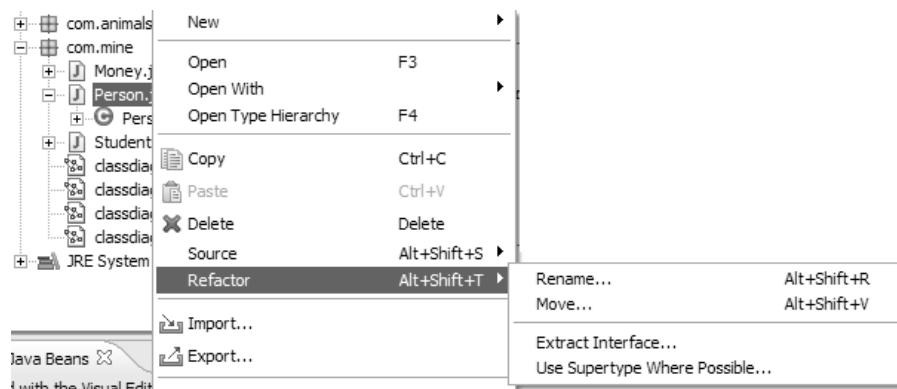


Figure 10-14. Using the refactoring tools

SW2446.0

### **Notes:**

To use the Refactor context menu item of a Java class or element, either:

- Right-click a class in the Package Explorer view
- Right-click a Java element in an editor or the Outline view

## Refactoring example

---

- How can you extend an e-mail system to do more than text and HTML?

```
writeEmailHeader(String headerString, boolean html) {  
    if (html) {  
        out.print("<H1>");  
    }  
    out.print(headerString);  
    if (html) {  
        out.print("</H1>");  
    }  
}
```

---

Figure 10-15. Refactoring example

SW2446.0

### **Notes:**

## Refactored solution

- Refactored using the Strategy design pattern.

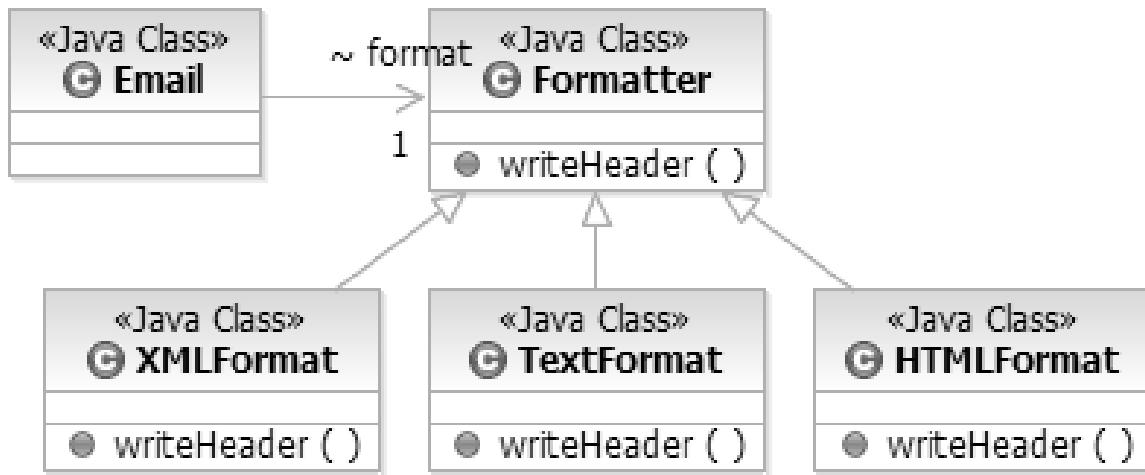


Figure 10-16. Refactored solution

SW2446.0

### Notes:

This chart references the Strategy pattern, which is one of a family of design patterns that may help when refactoring.

The Strategy pattern here defines a family of algorithms to handle XML, HTML and Text; each algorithm is encapsulated into a class, and handles the problem of writing a header in a way appropriate to its format. Notice that each of these classes is interchangeable with the others, indicating that polymorphism may be used to select which is used to format the e-mail header.

The original **Formatter** object used to have a mass of code with logic to handle many special cases to select the type of output, and now, this original object has been split into pieces. These pieces, the three strategy classes with algorithms that handle different types of output, are organized into a class hierarchy so that instead of conditional logic choosing between pieces, an algorithm is selected using polymorphism.

For more information about this pattern, and patterns, in general, see the Pattern Wiki at <http://c2.com/cgi/wiki?StrategyPattern>

## **Checkpoint**

---

1. What is a design pattern?
2. Name the three different types of design patterns
3. What is refactoring?
4. What are some techniques which can be used to refactor?
5. When should refactoring occur?



---

Figure 10-17. Checkpoint

SW2446.0

### **Notes:**

## Unit summary

- In this unit, you should have learned to:
  - Describe what a design pattern is
  - Outline the different types of design patterns and list the key elements of a design pattern
  - Describe what refactoring is and why it is needed
  - Outline when refactoring is done in the development cycle
  - List various strategies and techniques of refactoring
  - Use Application Developer to refactor Java classes and elements

---

Figure 10-18. Unit summary

SW2446.0

### **Notes:**



# Unit 11. Interfaces

## What this unit is about

This unit introduces the concept of interfaces, and discusses their use, benefits, naming conventions, and implementation.

## What you should be able to do

After completing this unit, you should be able to:

- Explain the concept of interfaces
- Declare an interface in Java
- Declare that a class implements one or more interfaces
- Explain what a class implementing an interface must supply

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 11.1 Interfaces

## Unit objectives

---

- After completing this unit, you should be able to:
  - Explain the concept of an interface and outline what it provides
  - Declare an interface in Java
  - Declare that a class implements one or more interfaces
  - Explain what a class implementing an interface must supply
  - Identify programming scenarios in which interfaces are useful

---

Figure 11-1. Unit objectives

SW2446.0

### **Notes:**

## Interfaces

- Declared types in Java are either classes or interfaces
- Interfaces represent a promise of supported services of the objects which implement the interface
  - An interface represents a contract that objects implement
- An interface is simply a list of method declarations
  - Its methods are declared but not implemented
  - An interface is not a class, although it looks remarkably like an abstract class

```
public interface File {  
    public void open(String name);  
    public void close();  
}
```

Figure 11-2. Interfaces

SW2446.0

### Notes:

Like a class, an interface defines methods. However, a class may implement methods, while an interface never implements methods.

Both abstract classes and interfaces can ensure that a concrete class will have all outlined methods, but abstract classes may also contain implemented methods, while interfaces cannot contain any implemented methods. As well, a class that implements an interface **must** implement all its methods, which is not the case when extending abstract classes.

## Protocols

- An interface defines a protocol (a set of methods)
  - If a class implements a given interface, then that class implements that protocol
- An interface can be used to impose a common protocol on a group of classes that are not related by inheritance
  - In the chain of classes related by inheritance, the common protocol is imposed through subclassing

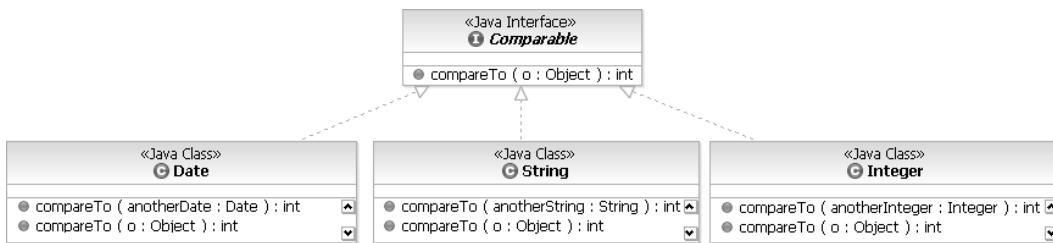


Figure 11-3. Protocols

SW2446.0

### Notes:

A message provides access to a specific service (method) provided by a class. The set of messages a class accepts is called an interface. When multiple classes accept the same set of messages, the interface to the services provided by a class may be abstracted and may stand on its own.

Interfaces contain only method stubs - no functionality. An interface may contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly public, static, and final (an interface can contain constant fields).

By contrast, superclasses can contain methods, with functionality, as well as variables.

## Implementing interface methods

- Methods declared in an interface are implemented in the classes which support that interface

```
public class TextFile implements File {  
    public void open(String name) {  
        // implementation of open method  
    }  
    public void close() {  
        // implementation of close method  
    }  
}
```

Figure 11-4. Implementing interface methods

SW2446.0

### **Notes:**

Any class implementing an interface must provide an implementation for all methods declared in the interface.

## Syntax

---

- In a class declaration, the naming of the superclass precedes any interfaces supported by the class

```
public class Directory extends Secure
    implements File {
    ...
}
```

- If a class implements multiple interfaces, the interfaces are all listed, separated by commas

```
public class Directory
    implements File, Secure {
    ...
}
```

---

Figure 11-5. Syntax

SW2446.0

### Notes:

## ***Implementing an interface***

---

- A class which implements an interface must implement every method defined by that interface
  - If one or more methods are not implemented, the compiler will generate an error
- Subclasses automatically implement all interfaces that their superclass implements
- Some interfaces, such as **Cloneable**, do not contain any methods
  - These act as a signal to Java that a class observes a certain protocol that may not be expressible as a specific set of methods
  - In the case of **Cloneable**, it alerts the JVM that objects of the implementing class can be copied via the `clone()` method

---

Figure 11-6. Implementing an interface

SW2446.0

### **Notes:**

An empty interface is uncommon, but useful in this case.

The `clone()` method performs the actual bitwise duplication of an object. The `clone()` method is part of the `Object` class, which ordinarily would mean that all classes would inherit it. However, a built-in safeguard, `CloneNotSupportedException`, prevents any object from being cloned if it is not an instance of a "cloneable" class. Thus, `Cloneable` is a way of preventing just any and all objects from being cloned.

## Typing and interfaces

---

- A variable's type can be an interface
  - Only objects whose class implements that interface can be bound to that variable
  - Only messages defined by the interface can be used
  - Interfaces cannot appear in a new expression

```
File r = new File(); // Error
File f = new TextFile(); // OK!
```

---

Figure 11-7. Typing and interfaces

SW2446.0

### **Notes:**

In this example, **f** does not “know” it is a **TextFile**, unless it is explicitly cast as one.

Note that it is best practice to code to an interface rather than an implementation, as is shown above.

## Subinterfaces

- Interfaces can be extended
  - Interfaces are not classes, so this hierarchy is independent of the class hierarchy
  - The interface which extends another interface inherits all of its method declarations

```

interface File {
    public void open(String name);
    public void close();
}

interface ReadableFile extends File {
    public byte readByte();
}

interface WritableFile extends File {
    public void writeByte(byte b);
}

interface ReadWriteFile extends ReadableFile,
                               WritableFile {
    public void seek(int position);
}

```

Figure 11-8. Subinterfaces

SW2446.0

### **Notes:**

When subclassing, use the word **extends** with only one superclass.

With interfaces, **extends** can refer to **multiple base interfaces** when building a new interface.

In other words:

- With classes, there is no multiple inheritance in Java.
- With interfaces, there can be multiple inheritance.

## Using interfaces

---

- Cross-hierarchy polymorphism can be accomplished using interfaces
- Interfaces allow access to methods in separate class trees
  - Can substitute an object for another object which is not related via the class hierarchy
  - Classes that implement the same interface understand the same messages, regardless of their location in the class hierarchy

---

Figure 11-9. Using interfaces

SW2446.0

### **Notes:**

One of the differences between interfaces and abstract classes is that the use of abstract classes cannot provide access to methods in separate class trees.

Interfaces are a way to relate classes that are not related through inheritance.

## Example

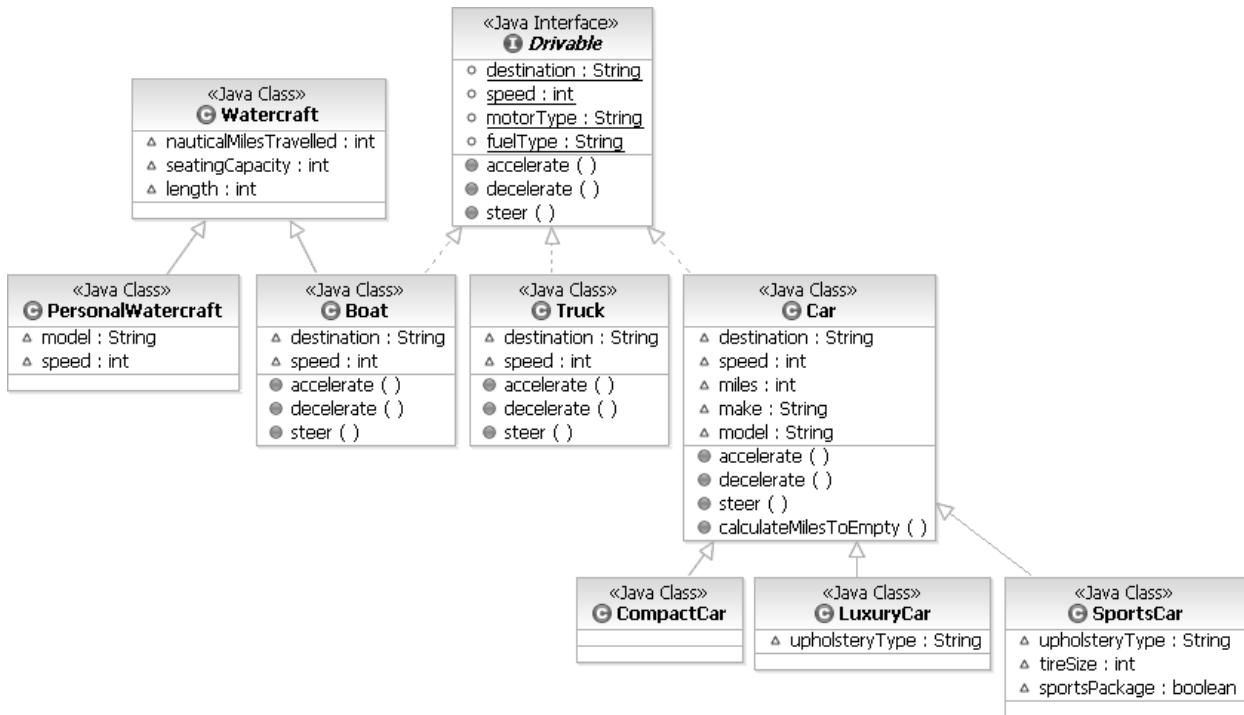


Figure 11-10. Example

SW2446.0

### Notes:

Here, you can see that the Boat class is in fact a subclass of the Watercraft class, and is part of the inheritance tree that includes other Watercraft, such as PersonalWatercraft. As well, the Car class is subclassed by SportsCar, LuxuryCar and CompactCar (which all implement the Drivable interface because their superclass, Car, does), and is in a different class hierarchy than the Watercraft class hierarchy. Classes that are not related through inheritance can be related through interfaces. When it is not natural for classes to be related through inheritance, providing an interface contract that a class may support allows classes that handle the same messages to be related.

Furthermore, since the Boat, Truck and Car classes belong to different class hierarchies, yet respond to the same messages (they fulfill the same contract outlined by the Drivable interface), it is also clear that interfaces are a way to accomplish cross-hierarchy polymorphism. The same messages can be sent to each of these classes that reside in different class hierarchies, and the service provided would be the same: each class would provide the services outlined by the Drivable interface. Classes may implement each method differently, but the service would be the same, even though the classes are not related by inheritance.

## ***More advantages of using interfaces***

---

- Using interfaces allows more control over how objects are used
  - Similarities among classes not related by inheritance can be captured, without forcing an unnatural relationship
  - The developer knows which message the objects will respond to; an object's interface may be revealed without revealing its class
  - Improves reusability of code

---

Figure 11-11. More advantages of using interfaces

SW2446.0

### **Notes:**

## Naming conventions for interfaces

---

- Make "able" interfaces
  - Cloneable, Serializable, and so on
- Name interfaces using proper nouns and provide "Impl" implementation of your interfaces
  - **Bank** interface, **BankImpl** class
  - **BankAccount** interface, **BankAccountImpl** class
  - With this convention, the interface typically contains a definition for all (or most) of the implementation class's public methods
- Prefix interface names with "I" and use proper nouns for your classes
  - **IBank** interface, **Bank** class
  - **IBankAccount** interface, **BankAccount** class

---

Figure 11-12. Naming conventions for interfaces

SW2446.0

### **Notes:**

These naming conventions are suggested, and not intended to be enforced simultaneously.

## **Checkpoint**

---

1. What is the difference between an interface and a class?
2. How are interfaces like abstract classes? How are they not?



---

Figure 11-13. Checkpoint

SW2446.0

### **Notes:**

## **Unit summary**

---

- In this unit, you should have learned to:
  - Explain the concept of an interface and outline what it provides
  - Declare an interface in Java
  - Declare that a class implements one or more interfaces
  - Explain what a class implementing an interface must supply
  - Identify programming scenarios in which interfaces are useful

---

Figure 11-14. Unit summary

SW2446.0

### **Notes:**



# Unit 12. Collections

## What this unit is about

This unit introduces the Java Collections Framework, and the concepts and implementation of collections in Java.

## What you should be able to do

After completing this unit, you should be able to:

- Describe the basic concept of collections and what they represent in Java
- Identify the collection interfaces provided by Java:
  - Interfaces
  - Abstract types
  - Concrete implementations
- List the concrete implementations of the collections, and choose between various concrete implementations of collections

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 12.1 Collections

## **Unit objectives**

---

- After completing this unit, you should be able to:
  - Describe the basic concept of collections and what they represent in Java
  - Identify the collection interfaces provided by Java
    - Interfaces
    - Abstract types
    - Concrete implementations
  - List the concrete implementations of the collections, and choose between various concrete implementations of collections
  - Describe how the “legacy” classes and interfaces fit in with the more modern classes and interfaces

---

Figure 12-1. Unit objectives

SW2446.0

### **Notes:**

## **What is a collection?**

- A collection is an object that groups multiple elements into a single unit
- Collections typically represent data items that form a natural group, such as:
  - A poker hand
    - A collection of cards
  - A mail folder
    - A collection of letters
  - A telephone directory
    - A collection of name-to-phone-number mappings



Figure 12-2. What Is a collection?

SW2446.0

### **Notes:**

Collections are sometimes called *containers*, but this word is already overloaded in the Java world.

## Collections represent data structures

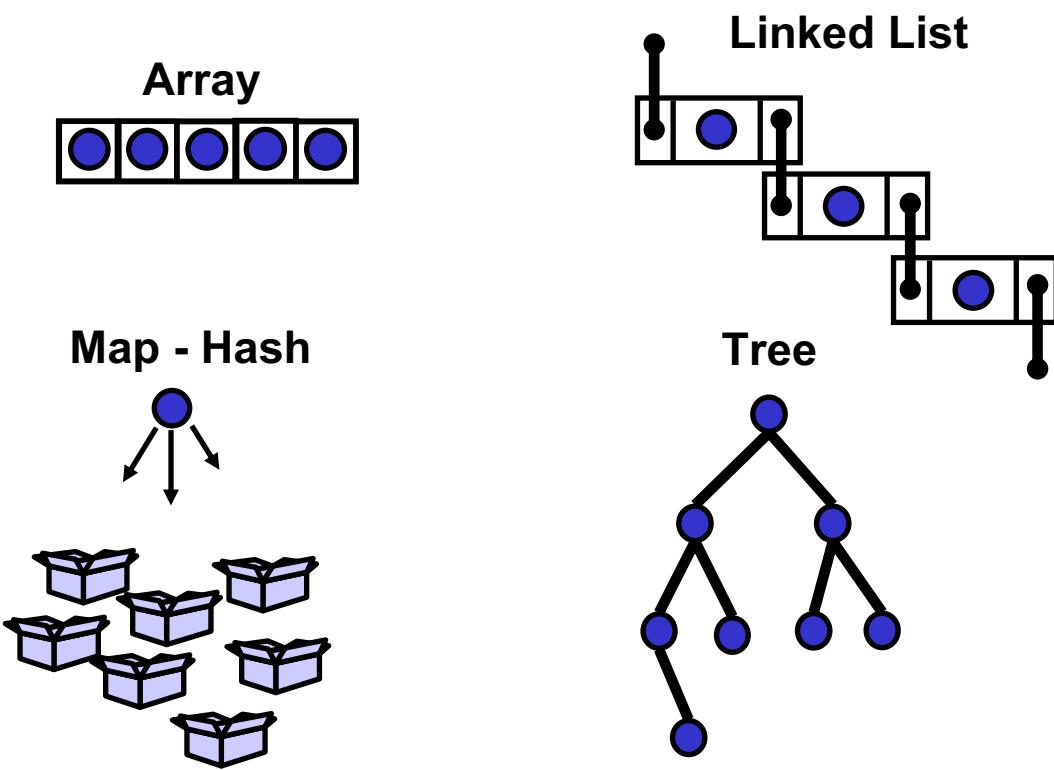


Figure 12-3. Collections represent data structures

SW2446.0

### **Notes:**

An array is a fixed length series of objects.

A linked list is an sequence of objects which allows duplicate elements.

A map is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

A tree is a data structure accessed from the root node.

## ***The Java Collections Framework***

---

- A unified architecture for representing and manipulating collections

- It contains three things:

- Interfaces**

- Abstract data types representing collections
    - Allow collections to be manipulated independently of the details of their representation

- Implementations**

- Concrete implementations of the collection interfaces
    - Reusable data structures

- Algorithms**

- Methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces
    - Reusable functionality via polymorphism; same method can be used on many different implementations of the appropriate collections interface

---

Figure 12-4. The Java Collections Framework

SW2446.0

### **Notes:**

Interfaces provide extension points through which new collection types can be added which provide different implementations of the same methods.

## ***Benefits of a collections framework***

---

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
- Reduces the effort to learn and use new APIs
- Reduces the effort to design new APIs
- Encourages software reuse

---

Figure 12-5. Benefits of a collections framework

SW2446.0

### **Notes:**

## Specific kinds of collections

---

- **Set**

- Cannot contain duplicate elements
    - Examples: employees, processes running on a machine

- **List**

- Ordered collection that can contain duplicates
    - Examples: Web page history, student roster

- **Map**

- A set of key-value pairs where duplicate keys are not allowed
    - Example: dictionary, property sheet

---

Figure 12-6. Specific kinds of collections

SW2446.0

### **Notes:**

Arrays are also considered collections, though they are not part of the Java Collections Framework.

## Interfaces in the framework

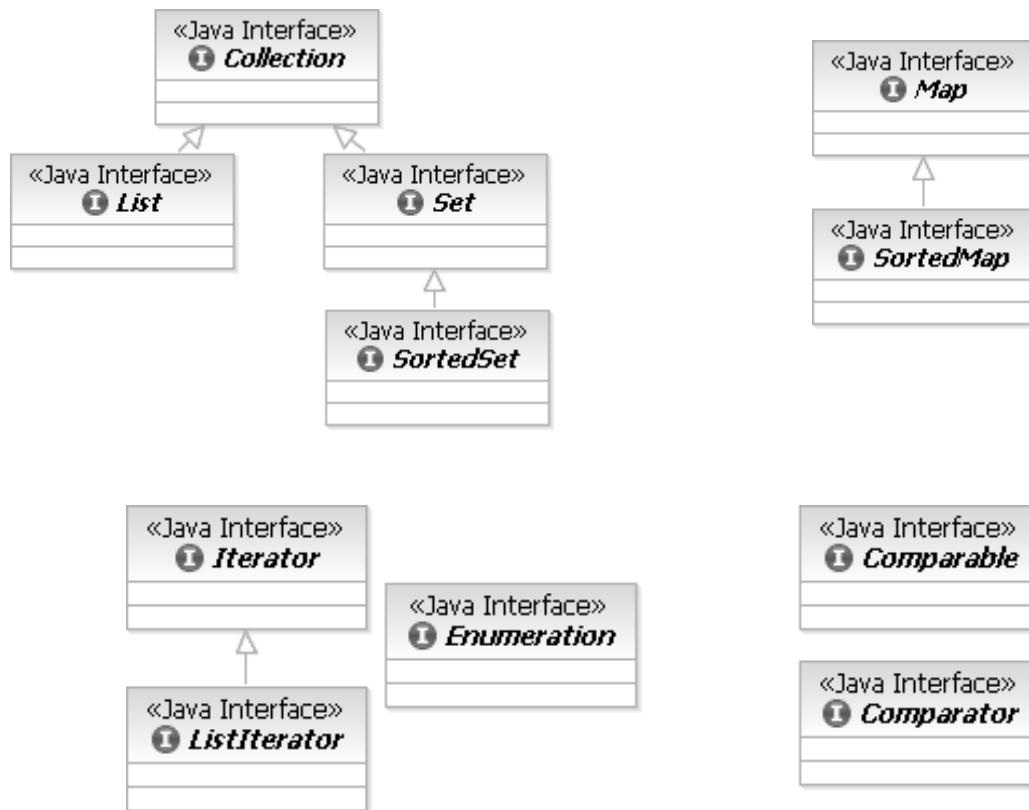


Figure 12-7. Interfaces in the framework

SW2446.0

### Notes:

There are also a number of abstract “helper” classes that aid in the creation of classes that implement these interfaces. For example, **AbstractCollection** is an abstract class that implements **Collection**, and **AbstractList** is an abstract class that implements **List**. These classes provide “skeleton” classes that can be easily extended to implement the desired interfaces.

All of these interfaces (except for **java.lang.Comparable**) are in the **java.util** package, as are their abstract and concrete implementations.

## The collection interface

- Defines the basic interface needed to manipulate a collection of objects
  - Add to collection
  - Remove from collection
  - Test membership
- Contains methods to operate on individual items or in bulk
- Also defines methods to facilitate looping through collections and converting collections to arrays
- Defines the overall “contract” for collections

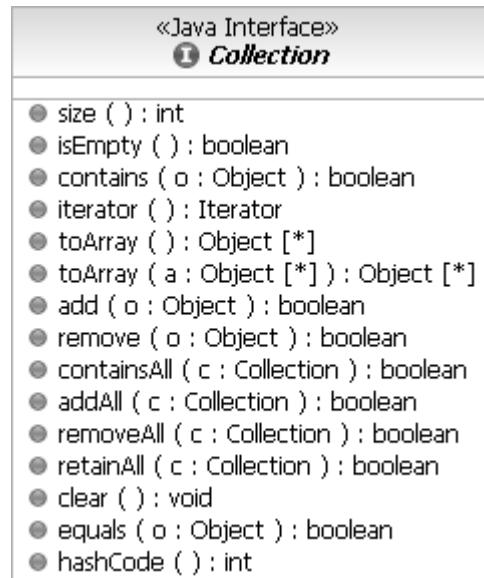


Figure 12-8. The collection interface

SW2446.0

### Notes:

We will discuss **Iterators** later.

# Collections, sets and lists

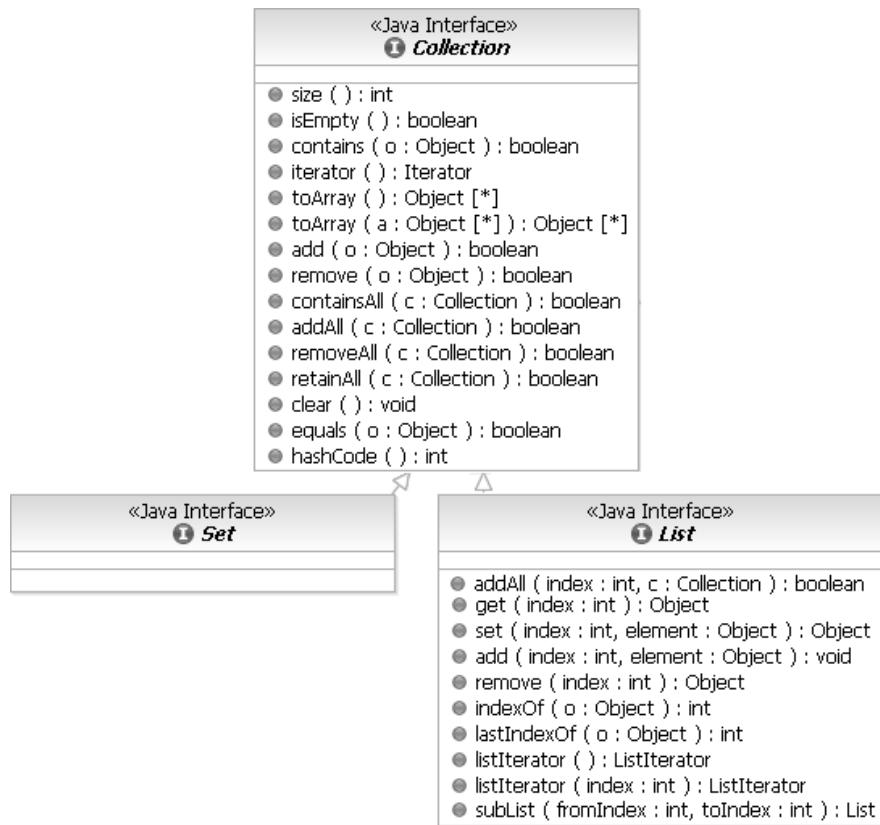


Figure 12-9. Collections, sets and lists

SW2446.0

## Notes:

**Set** does more than duplicate **Collection**. Although it has exactly the same methods as its parent interface, **Set** provides a more restricted contract. For example, the `add()` method in the **Set** interface does not allow duplicate items to be added.

**List** inherits all of the methods of **Collection** methods (not shown here) and adds others that expose **List**'s sequential nature. Notice that most of these either return or take as an argument an *index* representing the position of an item in the ordered **List**.

## The Map interface

- A map holds values that are retrieved by key
  - Cannot contain duplicate keys
- Defines the basic interface needed to manipulate such a collection
  - Add a key-value pair
  - Remove a key-value pair
  - Given a key, get the value
  - Test membership
- Provides three *collection views*, which allow a map's contents to be viewed as:
  - A set of keys
  - A collection of values
  - A set of key-value mappings

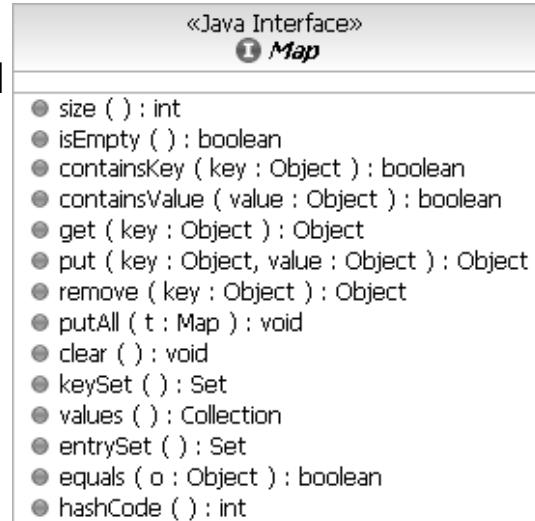


Figure 12-10. The Map interface

SW2446.0

### Notes:

The **Map** interface takes the place of the **Dictionary** class in earlier versions of Java, which was a totally abstract class rather than an interface.

**Map** methods do not directly return iterators, but these can be obtained indirectly via the collection views.

## Comparing objects

---

- In order to sort items in a collection, there must be a way to impose a *total ordering* on the items
  - For any two items in the collection, it must be possible to compare the objects and unambiguously determine whether:
    - Object A comes before object B
    - Object B comes before object A
    - Object A and object B are equal
- The **Comparable** interface can be used to implement a class whose objects are capable of comparing other objects to themselves
- The **Comparator** interface can be used to implement a class whose purpose is to compare other objects to each other



Figure 12-11. Comparing objects

SW2446.0

### Notes:

Both `compareTo()` and `compare()` return an integer which is either negative, zero or positive depending on the order of the objects being compared.

A class that implements the **Comparable** interface instantiates objects that know how to compare themselves to each other; classes that implement the Comparable interface have a *natural ordering*. For example, String objects know how to compare themselves to other String objects alphabetically.

A class that implements the **Comparator** interface instantiates objects that know how to compare two objects of the same type; it can be passed to the sort methods of various collection types. Two objects may compare differently using different comparators.

## Sorted collections

- **SortedSet** is a set with an intrinsic and automatically maintained order
  - Additional methods expose this order
- **SortedMap** is a map with similar properties, based on key order
- In either case, order may be determined by a natural order or a comparator

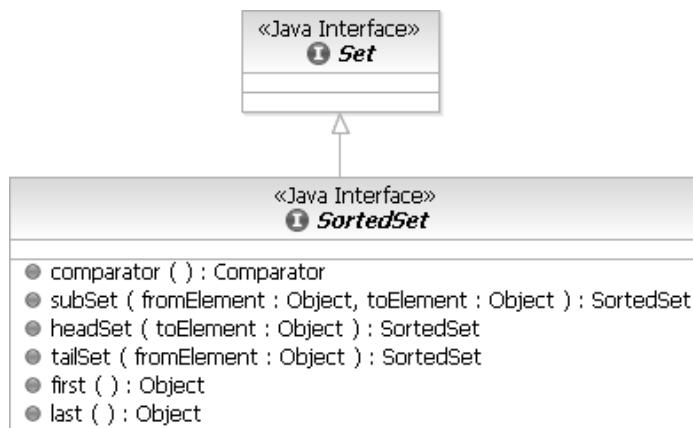


Figure 12-12. Sorted collections

SW2446.0

### Notes:

The methods of **SortedMap** are similar to those of **SortedSet**:

- Comparator `comparator()`
- Object `firstKey()`
- SortedMap `headMap(Object.toKey)`
- Object `lastKey()`
- SortedMap `subMap(Object.fromKey, Object.toKey)`
- SortedMap `tailMap(Object.fromKey)`

## Iterators

- Provide a convenient way to loop through the entire contents of a collection, one object at a time
  - Similar to an SQL cursor
- **ListIterator** adds methods that expose the sequential nature of the underlying list
- **add** and **remove** operations are passed through to the underlying collection
- Iterators of sorted collections loop through the collection according to its underlying order

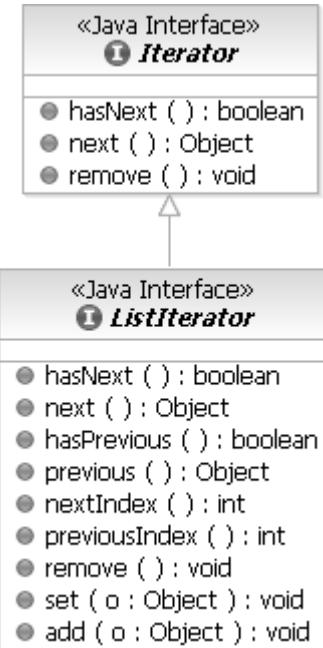


Figure 12-13. Iterators

SW2446.0

### Notes:

**Iterator** replaces **Enumeration** in earlier versions of Java. The main differences are improved method names, and a well-defined **remove()** operation.

**Iterator** is an example of the Iterator design pattern. The purpose of this pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

## The Iterator code pattern

```
Collection c;  
  
// Some code to build the collection  
  
Iterator i = c.iterator();  
while (i.hasNext()) {  
    Object o = i.next();  
    // Process this object  
}
```

Figure 12-14. The Iterator code pattern

SW2446.0

### **Notes:**

This is not a design pattern, as discussed on the previous page, but a code pattern – a template for writing code using **Iterator** that exemplifies standard practice.

## Interfaces and implementations

		IMPLEMENTATIONS				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Legacy
I N T E R F A C E S	Set	HashSet		TreeSet		
	List		ArrayList		LinkedList	Vector, Stack
	Map	HashMap		TreeMap		HashTable, Properties

Best practice: code to the interface rather than the implementation to promote portability

Figure 12-15. Interfaces and implementations

SW2446.0

### Notes:

## Example

```
public class MapExample {  
    public static void main(String args[]) {  
        Map map = new HashMap();  
        Integer ONE = new Integer(1);  
        for (int i=0, n=args.length; i<n; i++) {  
            String key = args[i];  
            Integer frequency =(Integer)map.get(key);  
            if (frequency == null) {  
                frequency = ONE;  
            } else {  
                int value = frequency.intValue();  
                frequency = new Integer(value + 1);  
            }  
            map.put(key, frequency);  
        }  
        System.out.println(map);  
        Map sortedMap = new TreeMap(map);  
        System.out.println(sortedMap);  
    }  
}
```

Figure 12-16. Example

SW2446.0

### Notes:

Run this example by specifying a list of arguments, for example: java MapExample 1 2 3 4 5

## ***Implementation choices***

---

### **•Set/Map**

- HashSet/HashMap
  - Very fast, no ordering
- TreeSet/TreeMap
  - Maintains balanced tree, good for sorted iterations
- Hashtable
  - Synchronized
  - Be sure to use **Map** interface

### **•List**

- ArrayList
  - Very fast
- LinkedList
  - Good for volatile collection, or adding to the front of the list
- Vector
  - Synchronized
  - Be sure to use **List** interface

---

Figure 12-17. Implementation choices

SW2446.0

### **Notes:**

The choice of *initial capacity* and *load factor* important for performance  
**HashSet/HashMap**.

**HashTable** can be used to simulate a **Set** by letting the members of the set be the keys, but this is not recommended.

**HashTable** and **Vector** contain many legacy methods which should no longer be used.

With **ArrayList**, you can use native method **System.arraycopy**.

## Legacy collections

---

- The legacy collection classes are still available, but their implementations have changed
  - java.util.Vector
    - Extendable, shrinkable, indexed list
  - java.util.Stack
    - Extends Vector to allow push and pop on a LIFO
  - java.util.BitSet
    - Expandable set of true/false flags
  - java.util.Dictionary
    - Abstract class now obsolete and replaced by java.util.Map
  - java.util.Hashtable
    - Efficient storage of objects with no natural organization
  - java.util.Properties
    - Stores key-value pairs; the key is the name of a property

---

Figure 12-18. Legacy collections

SW2446.0

### **Notes:**

The java.util.Dictionary class is obsolete, and java.util.Map should be used instead.

## Cloning collections

- You can make a copy of collections with the `clone()` method
  - This creates a new collection but does not clone the objects stored in the collection (called a shallow copy)

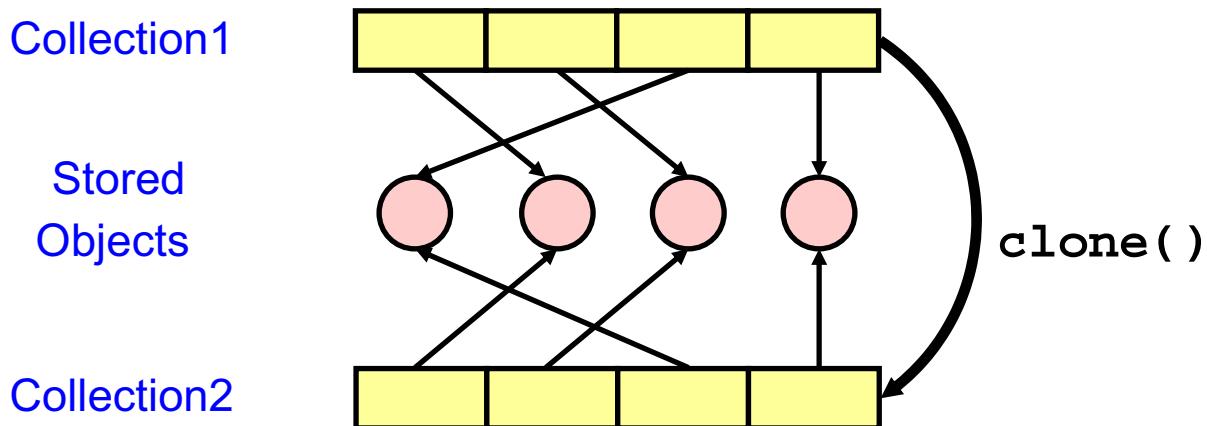


Figure 12-19. Cloning collections

SW2446.0

### **Notes:**

## ***The Collections class***

---

- `java.util.Collections` consists exclusively of static methods that operate on or return collections
- It contains:
  - Polymorphic algorithms that operate on collections, such as:
    - `binarySearch`
    - `copy`
    - `min` and `max`
    - `replace`
    - `reverse`
    - `rotate`
    - `shuffle`
    - `sort`
    - `swap`
  - “Wrappers”, which return a new collection backed by a specified collection
    - Synchronized collections
    - Unmodifiable collections

---

Figure 12-20. The Collections class

SW2446.0

### **Notes:**

The name of this class is **Collections**. There is also a **Collection** interface, and there is no connection between the two.

## Collections examples

```
public class CollectionsExample {

    public static void main(String args[]) {
        List list = new ArrayList();
        for (int i = 0, n = args.length; i < n; i++) {
            list.add(args[i]);
        }
        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);

        List noMods = Collections.unmodifiableList(list);
        noMods.add("error");
    }
}
```

The screenshot shows a Java application window with tabs for Problems, Javadoc, Declaration, Search, Console, Servers, and Database Explorer. The Console tab is active, displaying the following output:

```
<terminated> CollectionsExample [Java Application] C:\Program Files\IBM\Rational\SDP\6.0\runtimes\base_v6\java\jre\bin\javaw.exe (May 2, 2005 9:26:06 PM)
[hang, on, sloopy, sloopy, hang, on, yeah, yeah, yeah]
[hang, hang, on, on, sloopy, sloopy, yeah, yeah, yeah]
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1018)
    at com.ibm.websphere.samples.bank.client.CollectionsExample.main(CollectionsExample.java:40)
```

Figure 12-21. Collections examples

SW2446.0

### Notes:

## Checkpoint

1. What is:
  - A set?
  - A list?
  - A map?
2. What three things does the Java Collections Framework provide?
3. What extra behavior does **ListIterator** provide over **Iterator**?
4. If an object is **Comparable**, what method must it support?
5. Which concrete implementations are sorted?
6. Which concrete implementations map names to values?



Figure 12-22. Checkpoint

SW2446.0

### **Notes:**

## **Unit summary**

---

- In this unit, you should have learned to:
  - Describe the basic concept of collections and what they represent in Java
  - Identify the collection interfaces provided by Java
    - Interfaces
    - Abstract types
    - Concrete implementations
  - List the concrete implementations of the collections, and choose between various concrete implementations of collections
  - Describe how the “legacy” classes and interfaces fit in with the more modern classes and interfaces

---

Figure 12-23. Unit summary

SW2446.0

### **Notes:**

# Unit 13. Threads and Synchronization

## What this unit is about

This unit introduces the concept of Java threads, and its implementation through the Thread class and the Runnable interface. Synchronization and its implementation are also discussed.

## What you should be able to do

After completing this unit, you should be able to:

- Understand the difference between processes and threads
- Outline how threads are handled and implemented in Java
- Subclass the Thread class to create a new thread
- Implement the Runnable interface to create a new thread
- Describe the life cycle of a thread and how its activities may be controlled
- Describe the role of daemon threads and the JVM treatment of these
- Explain the notion of synchronization

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 13.1 Threads and Synchronization

## Unit objectives

---

- After completing this unit, you should be able to:
  - Understand the difference between processes and threads
  - Outline how threads are handled and implemented in Java
  - Subclass the Thread class to create a new thread
  - Implement the **Runnable** interface to create a new thread
  - Describe the life cycle of a thread and how its activities may be controlled
  - Describe the role of daemon threads and the JVM treatment of these
  - Explain the notion of synchronization

---

Figure 13-1. Unit objectives

SW2446.0

### **Notes:**

## Processes and threads

- On most computer systems, simple programs are run as processes by the CPU
- A *process* runs in its own memory address space, and consists of data, a call stack, the code being executed, the heap and other segments
- A *thread* executes instructions and is a path of execution through a program (process)
  - A thread does not carry all of the process information data
  - Many threads may run concurrently through a program, and may potentially share and access the same global data within the program

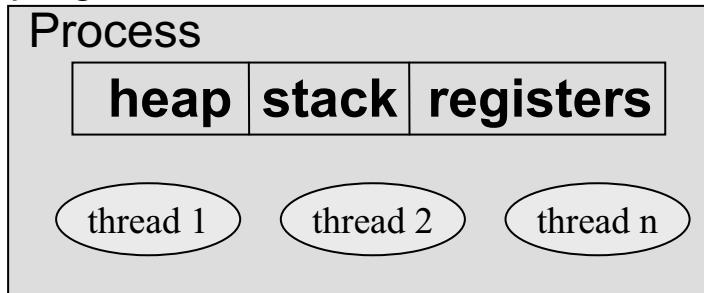


Figure 13-2. Processes and threads

SW2446.0

### **Notes:**

Threads allow different activities a program must perform to take place simultaneously. During times when the program would otherwise be waiting idly, it may perform tasks, helping to increase the responsiveness of the program. One example of this may be an e-mail client program which simultaneously needs to check for new e-mail, wait for GUI events such as a button click, and display an e-mail you have selected to read.

## Threads in Java

- Java has several classes that enable threads to be created
- The **Runnable** interface defines a single method: `run()`.
- The **Thread** class implements the **Runnable** interface and adds methods to manage the thread's priority, interrupt the thread, temporarily suspend the thread and so on.
- The `java.lang.Thread` class is the base for all objects that can behave as threads

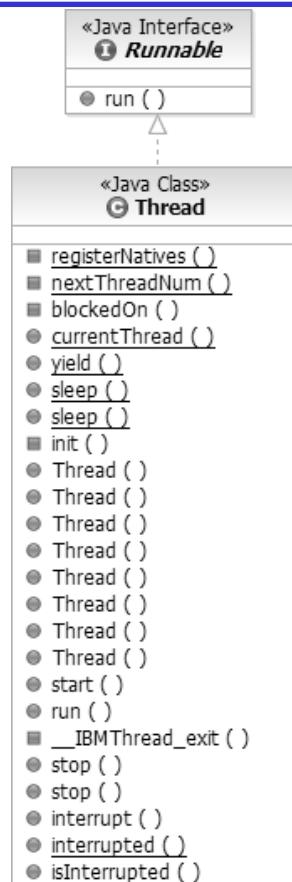


Figure 13-3. Threads in Java

SW2446.0

### Notes:

This is only a partial list of the methods of `Thread`. A thread needs a handle to a `Runnable` object, whose `run()` method it will execute.

## Creating threads in Java

---

- There are two ways to create a new thread using the Thread class
- A class can subclass Thread and override the `run()` method of the Thread class
  - The Thread itself is a Runnable object
- A class can implement the **Runnable** interface and implement the run method
  - This class can be run as a thread by creating a new Thread object and passing the Runnable object into the Thread(Runnable) constructor
  - New activities are started when the Thread object executes the Runnable object's `run()` method

---

Figure 13-4. Creating threads in Java

SW2446.0

### **Notes:**

## **Example: subclassing the Thread class**

- The class TrafficLight extends the Thread class and overrides the inherited method run():

```
class TrafficLight extends Thread {  
    public void run() {  
        // loop, change light color & sleep  
    }  
}
```

- Run the thread using the start() method inherited from the Thread class:

```
...  
TrafficLight tl = new TrafficLight();  
tl.start(); // Indirectly calls run()  
...
```

---

Figure 13-5. Example: subclassing the Thread class

SW2446.0

### **Notes:**

## Example: implementing Runnable

- The class TrafficLight has to provide its implementation for the method run():

```
class TrafficLight implements Runnable {  
    public void run() {  
        // loop, change light color & sleep  
    }  
}
```

- Create a new Thread with the Runnable class as a parameter:

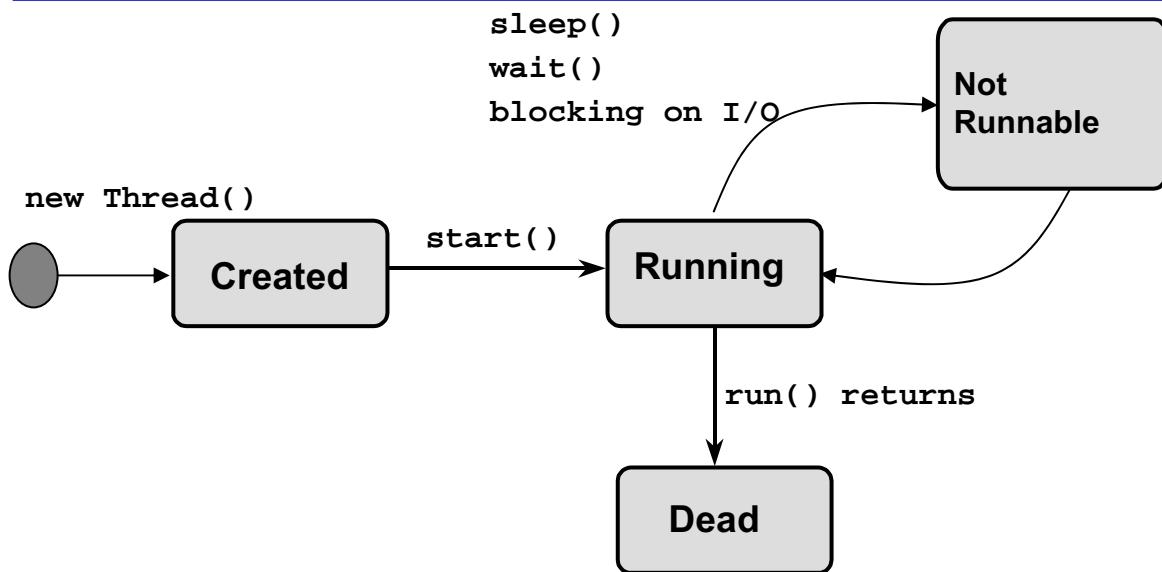
```
...  
TrafficLight tl = new TrafficLight();  
new Thread(tl).start();  
...
```

Figure 13-6. Example: implementing Runnable

SW2446.0

### **Notes:**

## Life cycle of a thread



- A thread that is running or asleep is said to be alive
  - This can be tested with the `isAlive()` method
- Once dead, the thread cannot be restarted
  - However, it can be examined

Figure 13-7. Life cycle of a Thread

SW2446.0

### Notes:

A thread in the not runnable state has its execution temporarily paused.

Calling the `start()` method causes a thread to begin executing. The Java virtual machine calls the `run()` method of this thread. You should never call `run()` directly; instead, you should call the `start()` method of your `Runnable` object. Calling `run()` directly simply runs the code in the same old thread, while calling the `start()` method creates a new execution context for the new thread, and then implicitly calls the `run()` method to begin a new activity in the new thread.

When a thread is running, calling the `sleep()` or `wait()` methods will cause the execution of the thread to temporarily pause. The `sleep()` method will pause the execution of the thread for a given amount of time. The `wait()` method is inherited from `java.lang.Object`, and causes the current thread to wait until another thread calls the `notify()` method or the `notifyAll()` method for this object.

## Controlling activities

- Threads are meant to be controlled
  - Methods exist to tell a thread when to run, when to pause, and so forth
  - `start()` - starts the thread's `run()` method
  - `sleep()` – pauses execution for given amount of time
  - `stop()` - deprecates since it is inherently unsafe
  - `destroy()` - kills a thread without any cleanup
  - `resume()` and `suspend()` - deprecates for the same reason as `stop()`
  - `yield()` - causes the currently executing thread object to pause temporarily, and allow other threads to execute
  - `interrupt()` - causes any kind of wait or sleep to be aborted
  - `setPriority()` - updates the priority of this thread

Figure 13-8. Controlling activities

SW2446.0

### **Notes:**

*Deprecated:* when a method is deprecated, its use will generate a warning error message at compile time. It can still be run, however, and the method will be executed. Deprecated methods may be removed from the Java API in the future, though, and should not be used except for compatibility with existing code.

The `stop()` method has been deprecated it is inherently unsafe and may result in arbitrary behavior. Many uses of `stop()` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its `run` method in an orderly fashion if the variable indicates that it is to stop running.

If the target thread waits for long periods (on a condition variable, for example), the `interrupt()` method should be used to interrupt the wait.

## Stopping threads

---

- When the `run()` method returns, the thread is dead
  - A dead thread cannot be restarted
- A dead Thread object is not destroyed
  - Its data can be accessed
- Set a field to indicate stop condition and poll it often

```
public void run() {  
    stopFlag = false;  
    try {  
        while (!stopFlag) {.....}  
    }  
    catch (InterruptedException e) {...}  
}  
public void finish() {  
    stopFlag = true;  
    .....  
}
```

---

Figure 13-9. Stopping threads

SW2446.0

### **Notes:**

## Daemon threads

- Daemon threads are service threads that run in the background
- Daemon threads are commonly used to:
  - Implement servers polling sockets
  - Loop waiting on an event or input
  - Service other threads
- Since daemon threads exist to support other threads, the JVM terminates a program when no user threads are running and only daemon threads remain
- Applicable control methods on a thread:
  - `isDaemon()` – check the daemon status of a thread
  - `setDaemon()` – set the daemon status of a thread

Figure 13-10. Daemon threads

SW2446.0

### **Notes:**

When the last user thread finishes, any running daemon threads are stopped, and the program is terminated by the JVM. The default daemon status of a thread is the same as that of thread creating it. The daemon status of `main()` is false. If you want all threads in your program to terminate when `main()` finishes, set their daemon status using `setDaemon(true)`.

A thread's status can't be changed after the thread has started.

## ***Multi-threading: need for synchronization***

---

- In many situations, concurrently running threads must share data and consider the state and activities of other threads
  - Example: producer-consumer programming scenarios
    - Producer thread generates data that is needed and consumed by another thread
  - Data may be shared using a common object that both threads access
- In Java, an object can be operated on by multiple threads; it is the responsibility of the object to protect itself from any possible interference
- Objects can be locked to prevent critical sections of code from being simultaneously accessed by multiple threads

---

Figure 13-11. Multithreading: need for synchronization

SW2446.0

### **Notes:**

Each object instance has a unique lock associated with it. All critical sections of code are locked when a thread enters any of these sections. If another thread tries to enter any of these sections of code while the first thread is still executing one of them, it will block and wait until the first thread is finished.

## Synchronization (1 of 2)

- The **synchronized** keyword may be used to synchronize access to an object among the threads using the object
  - The **synchronized** keyword guards critical sections of code
  - Either methods or arbitrary sections of code may be synchronized

```
public synchronized boolean deduct(float t) {  
    // code block associated  
    // with critical section...  
}
```

```
// non-critical section  
  
synchronized (object) {  
    // critical section...  
}  
  
// more non-critical stuff
```

Figure 13-12. Synchronization (1 of 2)

SW2446.0

### **Notes:**

## Synchronization (2 of 2)

- All synchronized sections of code in an object are locked when a thread executes any one synchronized code section
  - No other threads are able to enter a synchronized section of code while it is locked
  - Threads may still access other non-synchronized methods
- If the synchronized method is static, a lock is obtained at the class level instead of the object level
  - Only one thread at a time may use such a method

---

Figure 13-13. Synchronization (2 of 2)

SW2446.0

### **Notes:**

In an object, each critical section (code section that must be protected from multiple simultaneous thread access) is guarded by the synchronized keyword. A thread acquires the lock on **all** synchronized code sections when it passes the synchronized keyword of any such code section. Other threads cannot call a synchronized method on the same object until the object is unlocked (that is, the lock is released by the first thread, when it has exited the critical section).

Also note that unless a synchronized method is static, a lock is associated with the object and not the class, therefore numerous threads may be executing the same method simultaneously, as long as the objects are different (the messages are sent to different objects).

## Synchronization issues

- Use synchronization sparingly
  - Can slow performance by reducing concurrency
  - Can sometimes lead to fatal conditions such as **deadlock**
  
- Other techniques should be used with synchronization to assure optimal performance and to assist threads in coordinating their activities
  - For example, `notifyAll()` and `wait()`

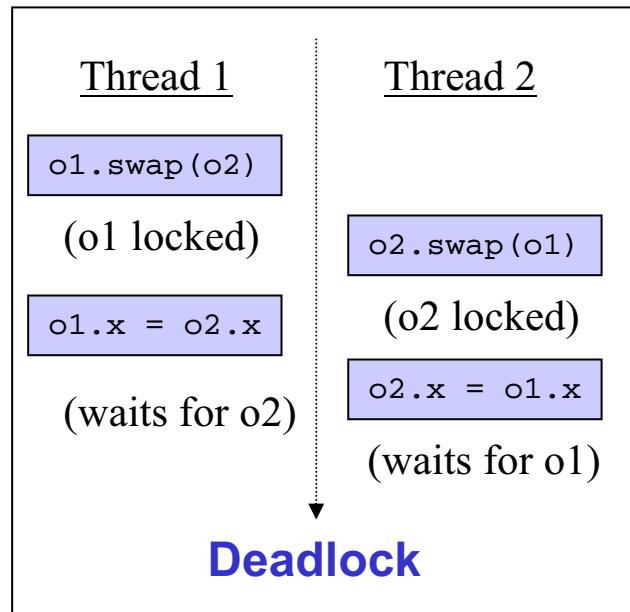


Figure 13-14. Synchronization issues

SW2446.0

### Notes:

In the example, two threads set out to swap two values in two objects using the synchronized `swap()` method. Thread 1 gets a lock on the first object, while thread 2 gets a lock on the second. No further work is possible, because each thread will block on the lock of the other thread. Deadlock is not automatically detected or handled in Java, so a system in which it occurs will simply grind to a halt for no immediately apparent reason (that is, will not terminate with an exception).

Overuse of synchronization may also lead to *race conditions* – cases where results vary and errors occur intermittently depending on the order in which certain threads get executed. This may occur in situations such as a producer/consumer situation in which the consumer consumes results faster than the producer produces them.

**notifyAll()** and **wait()** are two methods which all objects inherit from Object. They can be used to manage concurrency in a cooperative rather than an antagonistic fashion. For instance, one thread can suspend itself while a second thread is doing some critical work or using a critical resource. The second thread can then notify the first thread when the critical task is completed.

For a full discussion of synchronization, see  
<http://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html>.

## **Checkpoint**

1. What is the difference between a process and a thread?
2. What are the two ways to implement a thread?
3. What are the four stages in the life cycle of a thread?
4. What is the keyword that is used to indicate a critical section (section of code that should not be accessed simultaneously by more than one thread) in Java?



Figure 13-15. Checkpoint

SW2446.0

### **Notes:**

## Unit summary

---

- In this module, you should have learned to:
  - Understand the difference between processes and threads
  - Outline how threads are handled and implemented in Java
  - Subclass the Thread class to create a new thread
  - Implement the **Runnable** interface to create a new thread
  - Describe the life cycle of a thread and how its activities may be controlled
  - Describe the role of daemon threads and the JVM treatment of these
  - Explain the notion of synchronization

---

Figure 13-16. Unit summary

SW2446.0

### **Notes:**

# Unit 14. Utility Classes

## What this unit is about

This unit introduces the major utility classes in Java, including those found in the `java.lang` and `java.util` packages. Classes discussed include those providing conversion functions, the `Math` class, the `System` class, and the `Matcher` and `Pattern` classes. Regular expression support is discussed, as provided in the `String` class and in the combined use of the `Matcher` and `Pattern` classes.

## What you should be able to do

After completing this unit, you should be able to:

- Use the methods in the primitive wrapper classes to convert between types
- Use the math functions provided by the `Math` class
- Use the `System` and `Class` classes to get properties and class information
- Use the `String`, `Pattern` and `Matcher` classes to drive pattern matching using regular expressions

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 14.1 Utility Classes

## **Unit objectives**

---

- After completing this unit, you should be able to:
  - Use methods in the primitive wrapper classes to convert between types
  - Use the math functions provided by the Math class
  - Use the System and Class classes to get properties and class information
  - Use the String, Pattern and Matcher classes to do pattern matching using regular expressions

---

Figure 14-1. Unit objectives

SW2446.0

### **Notes:**

## Conversion functions of wrapper classes

- Use `toString()` to convert numeric values to strings
- Use `parse<type>()` and `valueOf()` to convert strings to numeric values
- Use explicit conversion methods to convert between numeric types

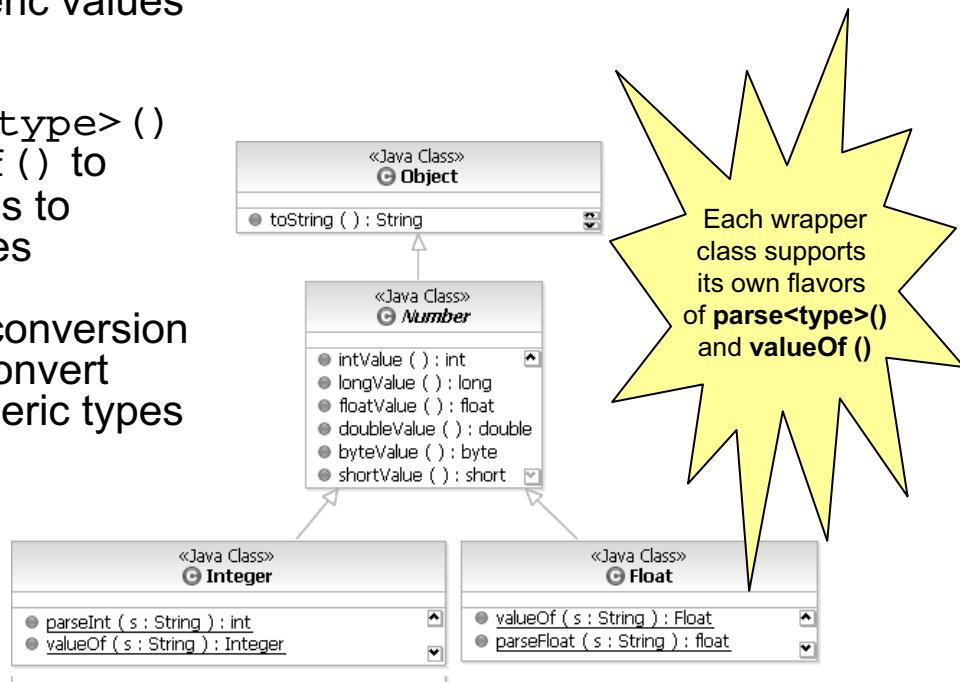


Figure 14-2. Conversion functions of wrapper classes

SW2446.0

### Notes:

Methods exist in the primitive wrapper classes (which subclass `Number`) to convert between types; for example, the `floatValue()` method in `Integer` returns the value of that `Integer` as a float, and similar `<type>Value()` methods exist to convert `Integer` values to other primitive types. As well, the `parseInt()` method in the `Integer` class parses the string argument as a signed decimal integer, and the `valueOf()` method returns an `Integer` object holding the value of the specified String.

## Numeric constants

---

### • Boolean

- Boolean FALSE
- Boolean TRUE

### • Byte

- byte MIN\_VALUE
- byte MAX\_VALUE

### • Character

- int MAX\_RADIX
- char MAX\_VALUE
- int MIN\_RADIX
- char MIN\_VALUE
- Unicode classification constants

### • Double

- double MAX\_VALUE
- double MIN\_VALUE
- double NaN
- double NEGATIVE\_INFINITY
- double POSITIVE\_INFINITY

### • Float

- float MAX\_VALUE
- float MIN\_VALUE
- float NaN
- float NEGATIVE\_INFINITY
- float POSITIVE\_INFINITY

### • Integer

- int MIN\_VALUE
- int MAX\_VALUE

### • Long

- long MIN\_VALUE
- long MAX\_VALUE

### • Short

- short MIN\_VALUE
- short MAX\_VALUE

Figure 14-3. Numeric constants

SW2446.0

### Notes:

All of these fields are declared as **public static final**.

Constants like **pi** are in the **java.lang.Math** class.

The **String** class also contains an interesting static field – though not a numeric constant.

**String.CASE\_INSENSITIVE\_ORDER** returns a **Comparator** which can be used to perform a case-insensitive string comparison.

## **Examples**

---

```
float f = Float.parseFloat("3.14159");
System.out.println(f); // 3.14159
```

```
int i = (new Float(f)).intValue();
String s = (new Integer(i)).toString();
System.out.println(s); // 3
```

```
double d = (new Integer(Integer.MAX_VALUE))
    .doubleValue();
System.out.println(d); // 2.147483647E9
```

---

Figure 14-4. Examples

SW2446.0

### **Notes:**

## The Math class

- `java.lang.Math` provides static versions of:
  - Mathematical constants:
    - `Math.E`
    - `Math.PI`
  - Mathematical functions:
    - `max`, `min`...
    - `abs`, `floor`, `ceil`...
    - `sqrt`, `pow`, `log`, `exp`...
    - `cos`, `sin`, `tan`, `acos`, `asin`, `atan`...
    - `random`
- Most functions take doubles as arguments and return double values.

$$e^{\sqrt{2\pi}}$$

```
Math.pow(Math.E, 2.0*Math.sqrt(Math.PI))
```

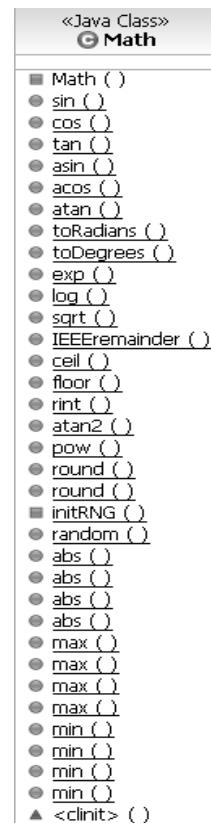


Figure 14-5. The Math class

SW2446.0

### Notes:

## The System class

- `java.lang.System` contains a variety of useful utility functions
  - Standard I/O handles
    - `InputStream in` and `PrintStreams out` and `err`
    - May be reset with `setIn()`, `setOut()` and `setErr()`
  - `arraycopy()`
    - Provides high performance array copy or subset
  - `currentTimeMillis()`
    - Returns current time in milliseconds
  - `exit()`
    - Terminates the currently running Java Virtual Machine
  - `gc()`
    - Requests the garbage collector to run
  - System property methods
    - Can retrieve properties such as the Java Runtime Environment version, the Java installation directory, and so on

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

Figure 14-6. The System class

SW2446.0

### Notes:

The `InputStream in` and `PrintStreams out` and `err` are fields in the `System` class.

The `arraycopy()` method copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

`gc()` does not actually run the garbage collector, but rather suggests to the garbage collector that it should run. There is no guarantee that the garbage collector will run.

The `System` class cannot be instantiated.

## The Class class

- A Class object is created for each class Java loads at runtime
  - The Class class describes the Java class or interface, and provides some useful methods
- You can obtain the Class object at runtime in several ways:

```
TextFile file = new TextFile();
Class c = file.getClass();
c = TextFile.class;
c = Class.forName("TextFile");
```

- You can obtain a Class object representing the superclass using the getSuperclass() method:

```
TextFile.class.getSuperclass();
```

---

Figure 14-7. The Class class

SW2446.0

### **Notes:**

## Pattern matching

- Java 1.4 introduced powerful new techniques for matching patterns in Strings
  - Based on *regular expressions*
- Matcher objects consume patterns and Strings to find and process matching subsets
  - Can also be used with `String.split()` method



Figure 14-8. Pattern matching

SW2446.0

### **Notes:**

Regular expressions are discussed on the next slide.

## Regular expressions

---

- Regular expressions describe a set of strings through a concise language that includes:
  - Characters
  - Metacharacters
  - Character classes
- Metacharacters can represent:
  - Boundaries
    - ^ means *line beginning*
  - Quantifiers
    - X\* means *one or more occurrences of X*
  - Operators
    - X|Y means *X or Y*
- Character classes represent groups of characters, such as:
  - [a-f] (lowercase characters “a” through “f”)
  - \s (any whitespace character)
  - \d (any digit)

“^ [Hh] ello, ?\s+ [Ww] orld!?\$”

---

Figure 14-9. Regular expressions

SW2446.0

### Notes:

## Some useful constructs

Construct	Definition	Example	Matches
.	Any character	a.a	“aaa”, “aba”, “aca”, ...
[ ]	Class	[ade]	‘a’ or ‘d’ or ‘e’
[^]	Negation	[^abc]	any char except [abc]
^	Start of line	^abc	“abc...”
\$	End of line	abc\$	“...abc”
\b	Word boundary	\bxyz\b	“xyz” but not “axyzw”
( )	Group	(abc)	“abc”
	Logical or	(abc) (xyz)	“abc” or “xyz”
*	Zero or more times	(abc)*	“, “abc”, “abcabc”, ...
+	One or more times	(abc)+	“abc”, “abcabc”, ...
?	Once or not at all	(abc)?	“abc”, “”
\d	A digit	\dabc	“0abc”, “1abc”, ...
\s	White space character	\s+	Any white space

Figure 14-10. Some useful constructs

SW2446.0

### Notes:

For a complete list and references, see <http://java.sun.com/j2se/1.4.2/docs/api/index.html> (look under **java.util.regex.Pattern**).

When representing metacharacters with backslashes inside Java string literals, be sure to use a double backslash – once as called for by the rules of the regular expression, and once to escape the backslash so that Java doesn't see it as a special character. For example, the Java compiler will complain that “\bxyz\b” is not a valid string (because \b is not a valid escape sequence in Java, and so it doesn't mean anything to Java). The correct way to represent this regular expression in a Java string literal would be “\\bxyz\\b”.

# **Using regular expressions with Strings**

- The **String** class provides several methods which support regular expressions:
    - `boolean matches(String regex)`
      - True if this string matches the given regular expression
    - `String replaceAll(String regex, String replacement)`
      - Replaces each substring of this string that matches the given regular expression with the given replacement
    - `String replaceFirst(String regex, String replacement)`
      - Replaces the first substring of this string that matches the given regular expression with the given replacement
    - `String[] split(String regex)`
      - Splits this string around matches of the given regular expression
    - `String[] split(String regex, int limit)`
      - Splits this string around matches of the given regular expression

Figure 14-11. Using regular expressions with Strings

---

SW2446.0

## **Notes:**

In the second version of the **split()** method, the limit parameter controls the number of times the pattern is applied and therefore affects the length of the resulting String array that is returned (the String array contains each substring of this string that is terminated by another substring that matches the given expression or is terminated by the end of the string). If the limit  $n$  is greater than zero then the pattern will be applied at most  $n - 1$  times, the array's length will be no greater than  $n$ , and the array's last entry will contain all input beyond the last matched delimiter. If  $n$  is non-positive then the pattern will be applied as many times as possible and the array can have any length. If  $n$  is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded.

**StringBuffer** does not provide any methods which support regular expressions.

## **Example**

---

```
String x = "Hello, World!";
boolean b =
    x.matches ("^ [Hh] ello, ?\\s+ [Ww] orld! ?$") ;
```

### **Results:**

b = true

```
String x = "apples,oranges,    pears";
String[] sa = x.split(",\\s*");
```

### **Results:**

sa[0] = “apples”  
sa[1] = “oranges”  
sa[2] = “pears”

---

Figure 14-12. Example

SW2446.0

### **Notes:**

## ***More extensive operations***

---

- `java.util.regex` contains two classes for pattern matching operations
- Matcher provides an engine that can perform sophisticated searching and parsing based on regular expressions; there are three different types of matching:
  - `matches()` method attempts to match the entire input sequence against the pattern
  - `lookingAt()` method attempts to match the input sequence, starting at the beginning, against the pattern
  - `find()` method scans the input sequence looking for the next subsequence that matches the pattern
- Pattern provides:
  - A way to compile regular expressions so that they can be reused with good performance
  - A factory for Matcher objects
  - Versions of `split()` and `matches()` which work against objects of type **CharSequence**

---

Figure 14-13. More extensive operations

SW2446.0

### **Notes:**

**java.lang(CharSequence)** is an interface implemented by **String**, **StringBuffer** and **CharBuffer** (as of Java 1.4)

## **Example**

---

- Typical example of using the Pattern and Matcher classes together:

```
Pattern p = Pattern.compile("abc*b");  
Matcher m = p.matcher("abccccccb");  
boolean b = m.matches();
```

### **Results:**

b = true

---

Figure 14-14. Example

SW2446.0

### **Notes:**

The Pattern.compile() method compiles the given regular expression into a pattern ("abc\*b"). The Pattern.matcher() method creates a matcher that will match the given input, "abccccccb", against this pattern. The Matcher.matches() method attempts to match the entire input sequence against the pattern.

In this case, you could instead have used:

boolean b = Pattern.matches("abc\*b", "abccccccb"); but for repeated matches it is less efficient since it does not allow the compiled pattern to be reused.

## ***Checkpoint***

---

1. Which method in the Integer class converts a String to an int? An int to a String?
2. Which method creates a random value?
3. Which predefined output channels does System provide?
4. Interpret this regular expression:  
 $\backslash d\{5\}(-\backslash d\{4\})?$
5. What does a matcher do? How are matchers created?



---

Figure 14-15. Checkpoint

SW2446.0

### **Notes:**

## Unit summary

---

- In this module, you should have learned to:
  - Use methods in the primitive wrapper classes to convert between types
  - Use the math functions provided by the Math class
  - Use the System and Class classes to get properties and class information
  - Use the String, Pattern and Matcher classes to do pattern matching using regular expressions

---

Figure 14-16. Unit summary

SW2446.0

### **Notes:**



# Unit 15. Exceptions and Exception Handling

## What this unit is about

This unit describes Java facilities for raising and handling errors and other conditions that disrupt execution flow.

## What you should be able to do

After completing this unit, you should be able to:

- Explain what a Java exception is and explain the Java Exception hierarchy
- Describe the conditions that act as the source of exceptions
- Use try/catch/finally blocks to properly handle exceptional events
- Use the throw keyword to throw a predefined Throwable object or your own Exception subtype
- Describe and use assertions

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 15.1 Exceptions and Exception Handling

## **Unit objectives**

---

- After completing this unit, you should be able to:
  - Explain what a Java exception is and describe the benefits of object-oriented exception handling
  - Describe the conditions that act as the source of exceptions
  - Use try/catch/finally blocks to catch and handle specific exceptions
  - Use the `throw` keyword to throw a predefined `Throwable` object or your own `Exception` subtype
  - Describe and use assertions

---

Figure 15-1. Unit objectives

SW2446.0

### **Notes:**

## Exceptions

---

- An *exception* is an event or condition that disrupts the normal flow of execution in a program
  - Exceptions are errors in a Java program
  - The condition causes the system to *throw* an exception
  - The flow of control is interrupted and a handler will *catch* the exception

---

Figure 15-2. Exceptions

SW2446.0

### **Notes:**

## Exception handling

---

- Exception handling is object-oriented
  - It encapsulates unexpected conditions in an object
  - It provides an elegant way to make programs robust
  - It isolates abnormal from regular flow of control

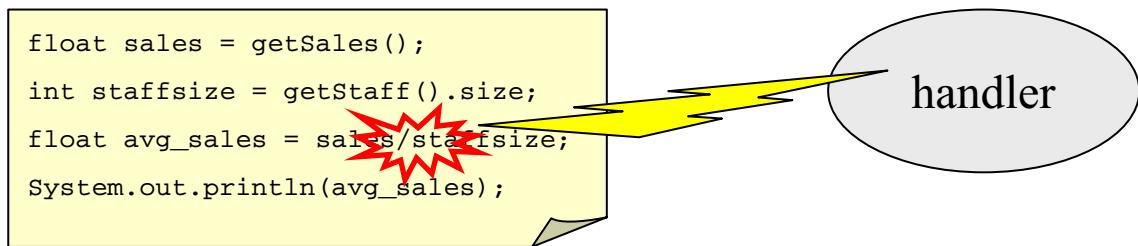


Figure 15-3. Exception handling

SW2446.0

### **Notes:**

## Exception sources

- JVM can detect unrecoverable conditions
  - Examples:
    - Class cannot be loaded
    - Null object reference used
- Both core classes and code that you write can throw exceptions
  - Examples:
    - IO error
    - Divide by zero
    - Data validation
    - Business logic exception
- Exceptions terminate execution unless they are handled by the program

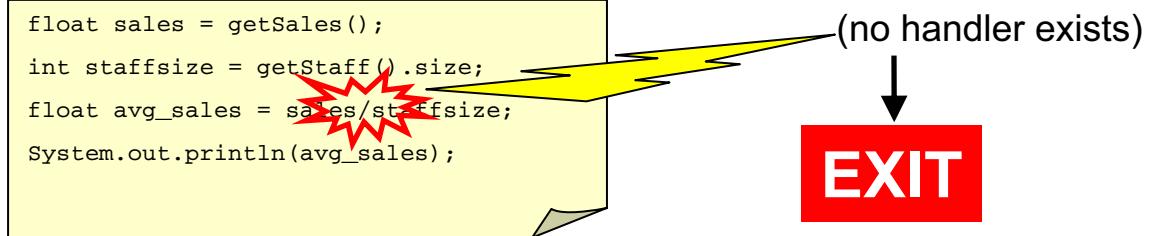


Figure 15-4. Exception sources

SW2446.0

### Notes:

## The exception hierarchy

- **Throwable** is the base class, and provides a common interface and implementation for most exceptions
- **Error** indicates serious problems that a reasonable application should not try to catch, such as:
  - **VirtualMachineError**
  - **CoderMalfunctionError**
- **Exception** heads the class of conditions that should usually be either caught or specified as thrown
- A **RuntimeException** can be thrown during the normal operation of the JVM
  - Methods may choose to catch these but need not specify them as thrown
  - Examples:
    - **ArithmaticException**
    - **BufferOverflowException**

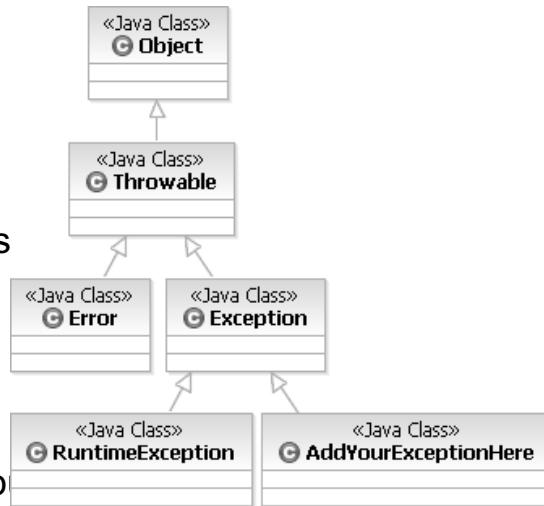


Figure 15-5. The exception hierarchy

SW2446.0

### Notes:

All exception classes inherit from one of the classes shown. Typically, related exceptions are grouped in the hierarchy. Developers are free to create their own sub-hierarchies of exceptions. Every exception is represented by an instance of the class **Throwable** or one of its subclasses. These objects can transfer information from the point at which an exception occurs to the handler that catches it.

Because Java does not require methods to catch or specify runtime exceptions, some programmers write code that throws only runtime exceptions or make all of their exception subclasses inherit from **RuntimeException**. These are poor coding practices known as “exception abuse.”

**Error** exceptions and subclasses indicate serious problems, and these are considered non-recoverable exceptions. They are exempt from compile-time checking and are *unchecked exceptions* because they may occur at many parts of the program and recovery is difficult or impossible. Unchecked exceptions do not have to be handled in your Java code.

**Exception** indicates recoverable problems; RuntimeExceptions occur at run time, and RuntimeException and its subclasses are also *unchecked exceptions*. These are usually errors that happen because the developer may not have checked for certain conditions; these errors will generate exceptions such as the NullPointerException and ArrayIndexOutOfBoundsException. The “AddYourExceptionHere” class indicates that you may add your own exceptions in place of that class; there are also many other subclasses to the Exception class. All exceptions other than RuntimeException and Error exception and their subclasses are considered *checked exceptions* and you must handle checked exceptions in your code.

## Handling exceptions

- Checked exceptions must be either handled in the method where they are generated, or delegated to the calling method

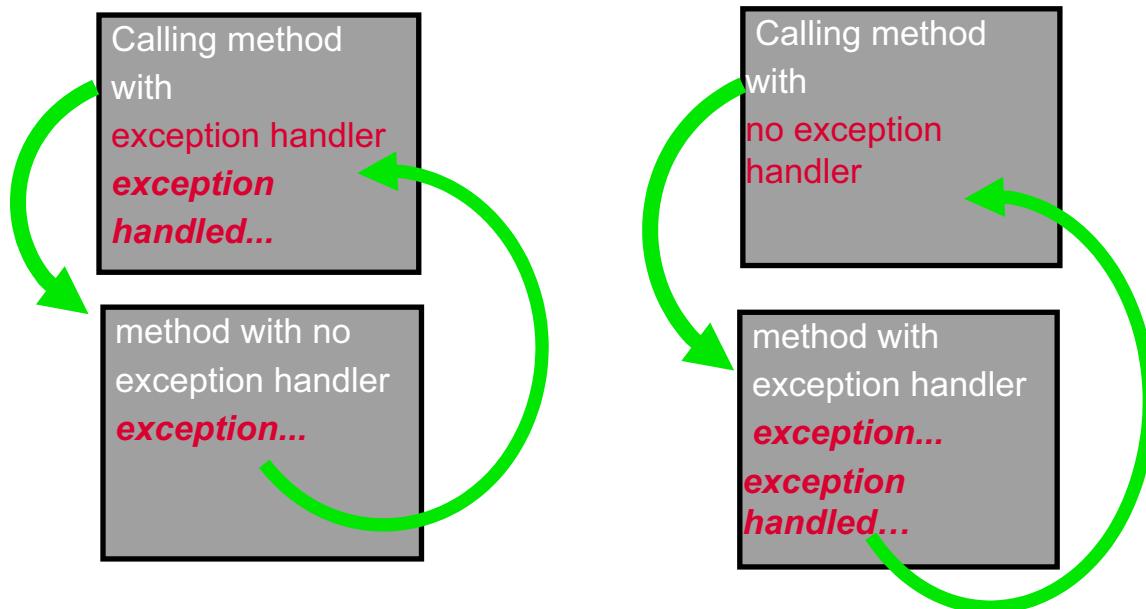


Figure 15-6. Handling exceptions

SW2446.0

### Notes:

## Keywords

- **throws**
  - A clause in a method declaration that lists exceptions that may be delegated up the call stack
    - Example: `public int doIt() throws SomeException, ...`
- **try**
  - Precedes a block of code with attached exception handlers
  - Exceptions in the `try` block are handled by the exception handlers
- **catch**
  - A block of code to handle a specific exception
- **finally**
  - An optional block which follows `catch` clauses
  - Always executed regardless of whether an exception occurs
- **throw**
  - Launches the exception mechanism explicitly
    - Example: `throw (SomeException)`

---

Figure 15-7. Keywords

SW2446.0

### **Notes:**

## ***try/catch blocks***

---

- To program exception handling, you must use try/catch blocks
- Code that might produce a given error is enclosed in a try block
- The catch clause must immediately follow the try block

```
try{  
    // Code that reads input from a file  
} catch (IOException ioe) {  
    // Some code that deals with I/O problems  
}
```

Figure 15-8. try/catch blocks

SW2446.0

### **Notes:**

```
public class Test{  
    public static void main(String[] args) {  
        InputStreamReader isr = new InputStreamReader(System.in);  
        try {  
            isr.read();  
        } catch(IOException ioe) {  
            ioe.printStackTrace()  
        }  
    }  
}
```

The read method inside of the InputStreamReader class throws an IOException; the compiler will force you to put the read method in a *try* block. If no errors occur, the program

will proceed and the catch block will not be executed. If an error occurs, the read method will create an IOException object, and the exception handler will look for the appropriate place to continue execution. In this case the Test program will take control and execute the code inside the *catch* block. All the code here will do is print out a Stack Trace to the console and exit the program.

## The catch clause

---

- The clause always has one argument that declares the type of exception to be caught
- The argument must be an object reference for the class **Throwable** or one of its subclasses
- Several catch clauses may follow one try block

```
catch (MyException me) {  
    ...  
}
```

---

Figure 15-9. The catch clause

SW2446.0

### **Notes:**

## Example

```
class MultiCatch {  
  
    public static void main( String args[] ) {  
  
        try {  
            // format a number  
            // read a file  
            // something else...  
        }  
        catch(IOException e) {  
            System.out.println("I/O error " + e.getMessage());  
        }  
        catch(NumberFormatException e) {  
            System.out.println("Bad data " + e.getMessage());  
        }  
        catch(Throwable e) { // catch all  
            System.out.println("error: " + e.getMessage());  
        }  
  
    }  
  
}
```

Figure 15-10. Example

SW2446.0

### Notes:

Remember that each of the **try** and **catch** blocks have separate scope – variables declared inside one are not known to the others. If you want to have access to the same variable inside both the **try** and **catch** blocks, be sure to declare the variable outside of the **try** block.

## The **finally** clause

- Optional clause that allows cleanup and other operations to occur whether an exception occurs or not
  - May have **try/finally** with no **catch** clauses
- Executed after any of the following:
  - try** block completes normally
  - catch** clause executes
    - Even if **catch** clause includes **return**
  - Unhandled exception is thrown, but before execution returns to calling method

```
try {  
    // File processing  
} catch (IOException e) {  
    // Handle exception  
} finally {  
    // Close files  
}
```

Figure 15-11. The **finally** clause

SW2446.0

### **Notes:**

The previous comments on scope of **try** and **catch** blocks also apply to **finally** blocks.

## Example

```
public class MyTest {  
  
    public static void main(String[] args) {  
        System.out.println(exceptionTest());  
    }  
    public static int exceptionTest() {  
        try {  
            int x = 0;  
            int y = 1;  
            System.out.println("Result=" + (y/x));  
        } catch (Exception e) {  
            e.printStackTrace();  
            return 1;  
        } finally {  
            System.out.println("Finally!");  
        }  
        return 2;  
    }  
}
```

Figure 15-12. Example

SW2446.0

### **Notes:**

1. As is, the program will catch the divide-by-zero exception, print a stack trace, print "Finally!", and return and print 1.
2. Comment out the catch block. The program will print "Finally!", and print a stack trace (because the divide-by-zero exception was uncaught).
3. Change "0" to "1". The program will print "Result=1", print "Finally!", and return 2.
4. Move the "return 2" statement inside the finally block, and uncomment the catch block. The program will catch the divide-by-zero exception, print a stack trace, print "Finally!", and return and print 2.

## Nested exception handling

- It may be necessary to handle exceptions inside a catch or finally clause
  - For example, you may want to log errors to a file, but all I/O operations require **IOException** to be caught.
- Do this by nesting a try/catch (and optional finally) sequence inside your handler

```
try {  
    // Processing  
} catch (MyException) {  
    try {  
        // Log error  
    } catch (IOException ioe) {  
        ioe.printStackTrace();  
    } finally {  
        // Close error log file  
    }  
}
```

---

Figure 15-13. Nested exception handling

SW2446.0

### **Notes:**

Exceptions can still be generated within catch and finally blocks, so they must be caught and handled as well.

## The throw keyword

- Not to be confused with keyword throws
- Can be used in a try block when you want to deliberately throw an exception
- You can throw a predefined Throwable object or your own Exception subtype
- Create a new instance of the exception class to encapsulate the condition
- The flow of the execution stops immediately after the throw statement, and the next statement is not reached
  - A finally clause will still be executed if present

```
throw new java.io.IOException("msg");
```

Figure 15-14. The throw keyword

SW2446.0

### Notes:

The throw keyword can be used to intentionally generate an exception.

Below is an example:

```
public class InputStreamReader extends Reader {  
    .....  
    public int read() throws IOException {  
        if(.....) {  
            throw new IOException();  
        }  
    }  
}
```

Again look at the method declaration. It has the throws keyword which tells the compiler that this read() may throw an IOException. The code inside the method creates the actual IOException object and signals the exception condition with the throw keyword.

## Handling runtime exceptions

---

- What happens when something goes wrong in the JVM?
  - It throws an error derived from Error depending on the type of problem
- What happens if RuntimeException is thrown?
  - Methods are not forced to declare RuntimeException in their **throws** clauses; the exception is passed to the JVM
- The JVM does the necessary cleaning and terminates the application or applet

---

Figure 15-15. Handling runtime exceptions

SW2446.0

### **Notes:**

## Assertions

---

- An *assertion* is a Java statement that allows you to test your assumptions about your program
  - In a traffic simulator, you might want to assert that a speed is positive, yet less than a certain maximum
- An assertion contains a boolean expression that you believe will be true when the assertion executes – if not true, the system throws an error
  - By verifying that the boolean expression is true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors
- Benefits:
  - Writing assertions while programming is a quick and effective way to detect and correct bugs
  - Assertions document the inner workings of your program, enhancing maintainability

---

Figure 15-16. Assertions

SW2446.0

### Notes:

The assertion facility is available starting with Java 1.4.

## Using assertions

---

- Two forms:

- `-assert <boolean expression> ;`
- `-assert <boolean expression> : <value expression> ;`

- If the boolean expression is false:

- Form 1 throws an **AssertionError** with no message
- Form 2 throws an **AssertionError** with a message defined by evaluating the second expression

- Assertion checking is disabled by default.

- Must be enabled at Java command line using the `enableassertions` switch
- Assertions can be enabled or disabled on a package-by-package or class-by-class basis
- `assert` statements are ignored if not enabled

---

Figure 15-17. Using assertions

SW2446.0

### **Notes:**

Note that **AssertionError** is a subclass of **Error** – don't check for failed assertions with **try/catch** blocks.

## ***When to use assertions***

---

- Do not use assertions:
  - For argument checking in public methods
  - To do any work that your application requires for correct operation
  
- Use assertions to test:
  - Internal invariants (values that should not change)
    - For example, place default: `assert false` at the end of switch statements with no default
  - Control-flow invariants
    - For example, place `assert false` at locations that should never be reached
  - Preconditions, postconditions, and class invariants
    - For example, argument checking in private methods

---

Figure 15-18. When to use assertions

SW2446.0

### **Notes:**

Remember that by default assertion checking is disabled. Therefore, do not take advantage of assertion “side-effects” (that is, do not assume that either of the expressions in an assert statement will actually be evaluated).

The fact that **AssertionError** is a subclass of **Error** provides a clue to the use of assertions: assertion errors are not expected to occur in the normal course of a running program, and when they do occur, they are not expected to be recoverable. Use them to detect and signal conditions that should never happen, values that should never occur, and locations that execution should never reach.

## **Checkpoint**

1. What are the four main classes in the Exception hierarchy?
2. What is a checked exception, and how does Java deal with them?
3. What is the difference between the `throw` and `throws` keywords?
4. What exception gets thrown by a failed `assert` statement?



Figure 15-19. Checkpoint

SW2446.0

### **Notes:**

## **Unit summary**

---

- In this unit, you should have learned to:
  - Explain what a Java exception is and describe the benefits of object-oriented exception handling
  - Describe the conditions that act as the source of exceptions
  - Use try/catch/finally blocks to catch and handle specific exceptions
  - Use the `throw` keyword to throw a predefined `Throwable` object or your own `Exception` subtype
  - Describe and use assertions

---

Figure 15-20. Unit summary

SW2446.0

### **Notes:**

# Unit 16. I/O and Serialization

## What this unit is about

This unit describes the classes, methods and techniques used in Java to perform various I/O operations, particularly the serialization of data.

## What you should be able to do

After completing this unit, you should be able to:

- Describe the concept of streams and describe the major functions of stream-based I/O in Java
- Outline the differences between stream-based I/O and new I/O
- Explain the role of channels and buffers in new I/O
- Explain the concepts of serialization and externalization
- Write code that serializes and de-serializes objects

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 16.1 I/O and Serialization

## Unit objectives

---

- After completing this unit, you should be able to:
  - Describe the concept of streams and use the stream classes of the Java language
  - Describe the major functions of stream-based I/O in Java
  - Describe Java's new I/O paradigm
  - List the differences between stream-based I/O and new I/O
  - Explain the role of channels and buffers in new I/O
  - Explain the concepts of serialization and externalization
  - Describe how serialization may be implemented by a class, and what the `transient` keyword does
  - Write code that serializes and deserializes objects

---

Figure 16-1. Unit objectives

SW2446.0

### **Notes:**

## Streams

- Streams allow data to be written to or read from an external source
- Byte streams (8-bit) operate on bytes and byte arrays
  - InputStream/OutputStream classes and subclasses
- Character streams (16-bit Unicode) operate on characters, character arrays and strings
  - Reader/Writer classes and subclasses



Figure 16-2. Streams

SW2446.0

### **Notes:**

Unicode is a character encoding standard for text written in any language of the modern world, including mathematical symbols.

I/O in Java is stream-based. What this means is that data read and written is treated like a stream of data, continuous bytes that keep coming in, or going out. At the lowest level of the I/O are a set of streams that read and write these bytes to/from files/sockets/Internet connections, and are device independent.

Streams are the lowest level of I/O that you can deal with in Java. As such, reading and writing to streams is awkward at best. Dealing with the devices at the byte-by-byte level can get complicated and confusing very quickly. There are many different kinds of streams available in Java to access low level I/O. In order to avoid the difficulty of dealing with I/O at this low level, Java provides Readers and Writers that raise the abstraction level of I/O. Readers and Writers deal with chars instead of with bytes. The chars are Unicode characters versus the bytes that are ASCII.

## Operations on streams

---

- Sequential operations on streams usually follow the same pattern, regardless of:
  - Data source
  - Data type
  - Data destination

```
stream.open()
while (more information) {
    read/write information
}
stream.close()
```

---

Figure 16-3. Operations on streams

SW2446.0

### **Notes:**

## Class hierarchy: byte stream input

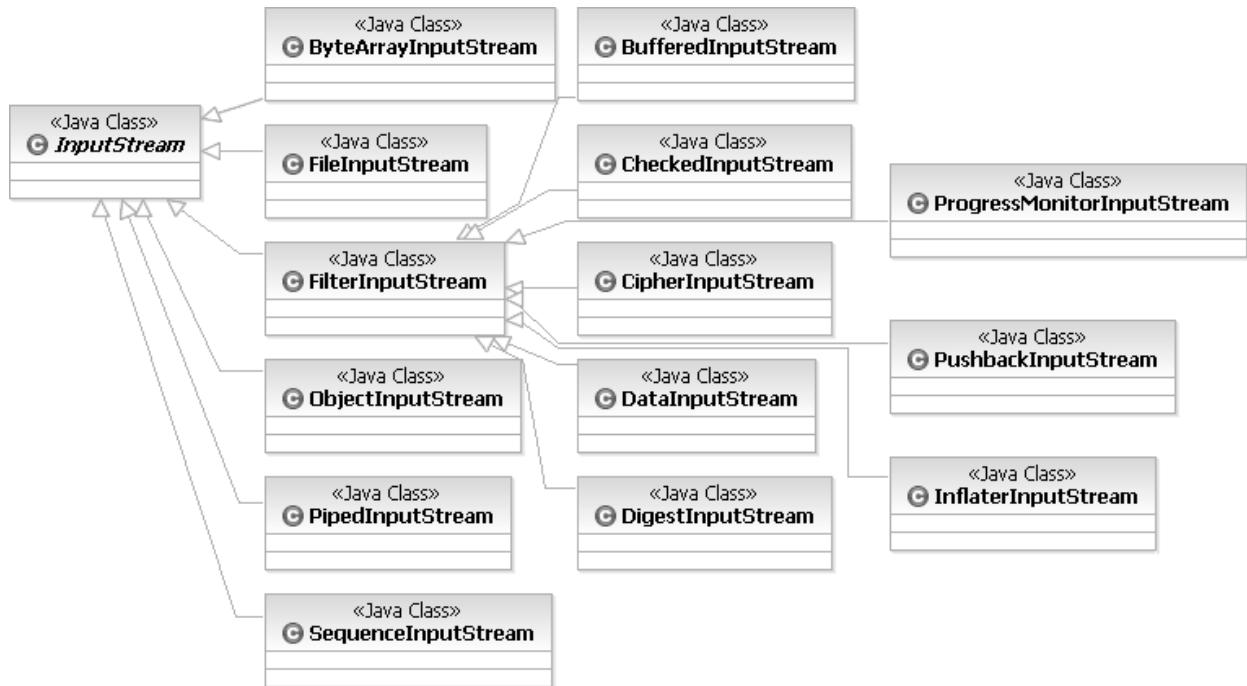


Figure 16-4. Class hierarchy: byte stream input

SW2446.0

### Notes:

**InputStream** is the abstract superclass of all classes representing an input stream of bytes. Applications that need to define a subclass of **InputStream** must always provide a method that returns the next byte of input.

A **ByteArrayInputStream** contains an internal buffer that contains bytes that may be read from the stream. An internal counter keeps track of the next byte to be supplied by the read method.

A **FileInputStream** obtains input bytes from a file in a file system. It is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using a **FileReader**.

A **FilterInputStream** contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class **FilterInputStream** itself simply overrides all methods of **InputStream** with versions that pass all requests to the contained input stream. Subclasses of **FilterInputStream** may further override some of these methods and may also provide additional methods and fields, as follows:

- A **BufferedInputStream** adds functionality to another input stream, namely, the ability to buffer the input and to support the mark and reset methods.
- A **CheckedInputStream** provides an input stream that also maintains a checksum of the data being read. The checksum can then be used to verify the integrity of the input data.
- A **CipherInputStream** is composed of an **InputStream** and a **Cipher** so that read methods return data that are read in from the underlying **InputStream** but have been additionally processed by the **Cipher**.
- A **DataInputStream** lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a **DataOutputStream** to write data that can later be read by a **DataInputStream**.
- A **DigestInputStream** provides a transparent stream that updates the associated message digest using the bits going through the stream.
- A **InflaterInputStream** implements a stream filter for uncompressing data in the “deflate” compression format. It is also used as the basis for other decompression filters, such as **GZIPInputStream**.
- A **ProgressMonitorInputStream** monitors the progress of reading from some **InputStream**. If it's taking a while, a **ProgressDialog** will be popped up to inform the user.
- A **PushbackInputStream** adds functionality to another input stream, namely the ability to “push back” or “unread” one byte.

An **ObjectInputStream** de-serializes primitive data and objects previously written using an **ObjectOutputStream**.

A **PipedInputStream** provides whatever data bytes are written to a **PipedOutputStream** to which it is connected.

A **SequenceInputStream** represents the logical concatenation of other input streams.

**NOTE:** this list is incomplete. It omits deprecated classes and subclasses in packages other than **java.io**.

## Class hierarchy: byte stream output

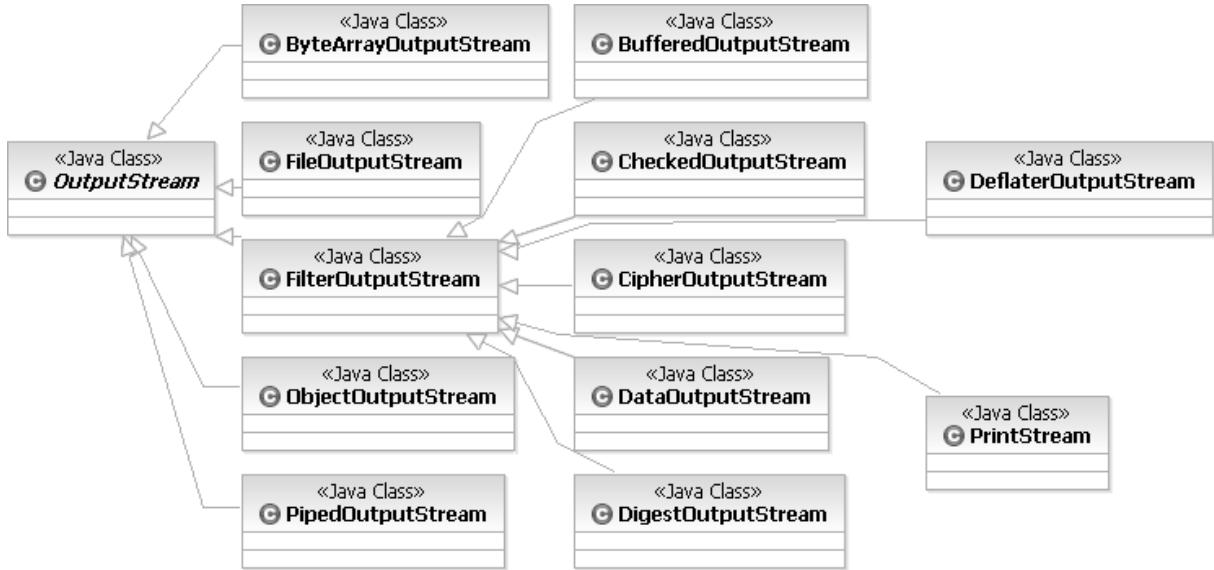


Figure 16-5. Class hierarchy: byte stream output

SW2446.0

### Notes:

**OutputStream** is the abstract superclass of all classes representing an output stream of bytes.

A **ByteArrayOutputStream** implements an output stream in which the data is written into a byte array.

A **FileOutputStream** writes data to a **File** or to a **FileDescriptor**. It is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using **FileWriter**.

A **FilterOutputStream** contains some other output stream, which it uses as its basic sink of data, but possibly transforming the data along the way or providing additional functionality. The class **FilterOutputStream** itself simply overrides all methods of **OutputStream** with versions that pass all requests to the contained output stream. Subclasses of **FilterOutputStream** may further override some of these methods and may also provide additional methods and fields, as follows:

- A **BufferedOutputStream** adds functionality to another output stream, namely, the ability to buffer the output.
- A **CheckedOutputStream** provides an output stream that also maintains a checksum of the data being written. The checksum can then be used to verify the integrity of the output data.
- A **CipherOutputStream** is composed of an **OutputStream** and a **Cipher** so that **write** methods store data that is written to the underlying **OutputStream** but have been additionally processed by the **Cipher**.
- A **DataOutputStream** lets an application write primitive Java data types to an underlying output stream in a machine-independent way. An application uses a **DataOutputStream** to write data that can later be read by a **DataInputStream**.
- A **DigestOutputStream** provides a transparent stream that updates the associated message digest using the bits going through the stream.
- A **DeflaterOutputStream** implements a stream filter for compressing data in the “deflate” compression format. It is also used as the basis for other compression filters, such as **GZIPOutputStream**.
- A **PrintStream** adds functionality to another output stream, namely the ability to print representations of various data values conveniently.

An **ObjectOutputStream** serializes primitive data and objects for later reading using an **ObjectInputStream**.

A **PipedOutputStream** writes bytes to be read by a **PipedInputStream** to which it is connected.

NOTE: this list is incomplete. It omits subclasses in packages other than **java.io** and **LogStream**, a subclass of **PrintStream**.

## Class hierarchy: character stream input

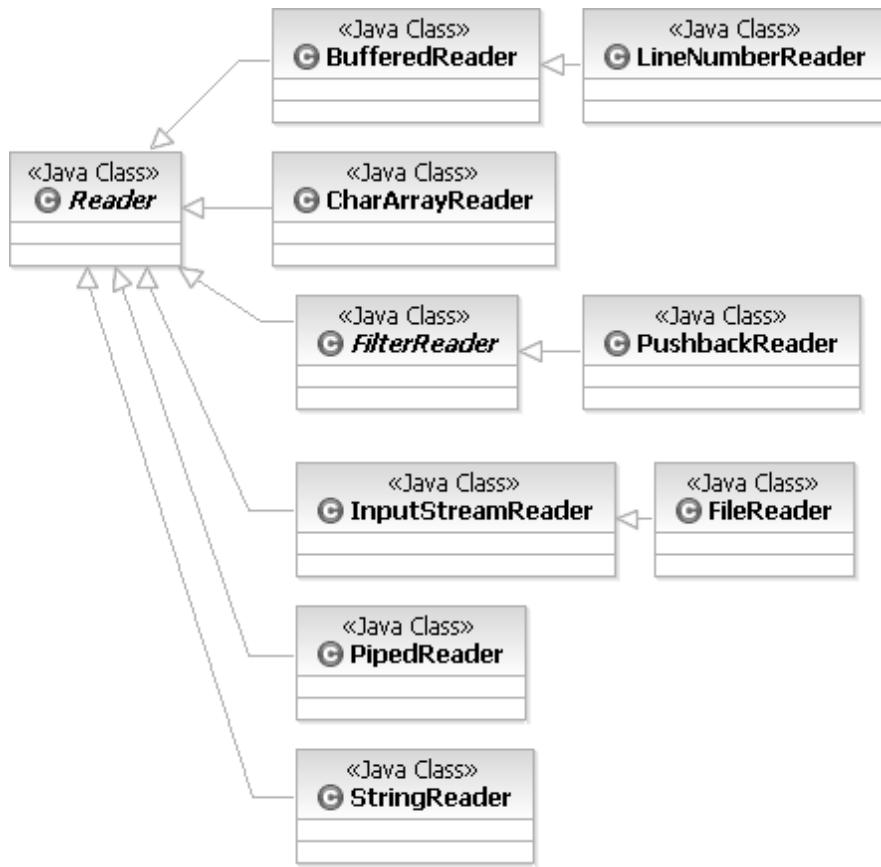


Figure 16-6. Class hierarchy: character stream input

SW2446.0

### Notes:

**Reader** is the abstract class for reading character streams. The only methods that a subclass must implement are **read(char[], int, int)** and **close()**. Most subclasses, however, will override other methods in order to provide higher efficiency, additional functionality, or both.

A **BufferedReader** reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. It may be used to wrap classes with costly operations, improving performance by adding buffering. A **LineNumberReader** is a **BufferedReader** that keeps track of line numbers.

**CharArrayReader** allows a character buffer to be used as a character-input stream.

**FilterReader** is an abstract class for reading filtered character streams. Its subclass **PushbackReader** is a character-stream reader that allows characters to be pushed back into the stream.

**InputStreamReader** is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. Its subclass **FileReader** is a convenience class for reading character files.

**PipedReader** reads from piped character-input streams.

**StringReader** reads from a string.

## Class hierarchy: character stream output

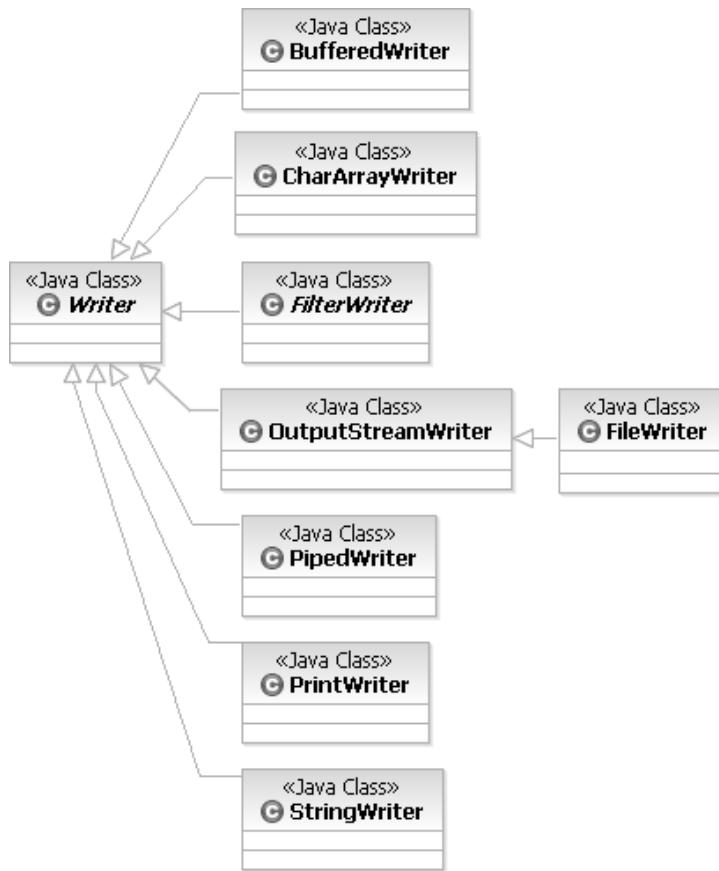


Figure 16-7. Class hierarchy: character stream output

SW2446.0

### Notes:

**Writer** is the abstract class for writing character streams. The only methods that a subclass must implement are **write(char[], int, int)**, **flush()** and **close()**. Most subclasses, however, will override other methods in order to provide higher efficiency, additional functionality, or both.

A **BufferedWriter** writes text to a character-output stream, buffering characters so as to provide for the efficient writing of characters, arrays, and lines. It may be used to wrap classes with costly operations, improving performance by adding buffering.

**CharArrayWriter** allows a character buffer to be used as a character-output stream.

**FilterWriter** is an abstract class for writing filtered character streams.

**OutputStreamWriter** is a bridge from byte streams to character streams: It writes characters and encodes them into bytes using a specified charset. Its subclass **FileWriter** is a convenience class for writing character files.

**PipedWriter** writes to piped character-output streams.

**PrintWriter** prints formatted representations of objects to a text-output stream. This class implements all of the print methods found in **PrintStream**.

**StringWriter** writes to a string.

## Creating streams

### •**FileInputStream**

```
FileInputStream f = new FileInputStream(  
    "C:\\data\\hello.txt");  
f.read(); // next byte in the file  
f.available(); // number of remaining bytes
```

### •**FileReader**

```
FileReader f = new FileReader(  
    "C:\\data\\hello.txt");  
f.read(); // next character in the file
```

Figure 16-8. Creating streams

SW2446.0

### **Notes:**

'\' is an escape character in java, so '\\' represents a single slash in the path.

To be platform independent, the system separator should be read using the **File** class, then the path built using that separator:

```
String separator = File.separator;  
File theFile = new File (separator + "data" + separator + "hello.txt");  
FileReader f = new FileReader(theFile);
```

## ***Filter streams***

- Filter streams filter data as it is read from or written to a stream
  - They are constructed on another *underlying* stream
- Input and output operations may be filtered
  - In a readable filter stream, the `read()` method reads input from the underlying stream, filters it, and passes on the filtered data to the caller
  - In a writable filter stream, the `write()` method filters the data and then writes it to the underlying stream
- Examples:
  - Data buffering (for example, **BufferedOutputStream**)
  - Data conversion (for example, **DeflaterOutputStream**)



Figure 16-9. Filter streams

SW2446.0

### **Notes:**

## Filter stream example: BufferedReader

- **BufferedReader** reads a line, not single characters
- It is built on the simpler **FileReader**

```
BufferedReader in = new BufferedReader(
    new FileReader ("theFile.txt"));
in.readLine();
```



Figure 16-10. Filter stream example: BufferedReader

SW2446.0

### Notes:

In general, every time a Reader makes a read request, a corresponding read request is made of the underlying character or byte stream. Thus, it is advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReader and InputStreamReader.

## Filter stream example: BufferedWriter

- **BufferedWriter** provides for efficient writing of single characters, arrays, and strings
- Wrap a **BufferedWriter** around any **Writer** whose write operations may be costly, such as **FileWriters** and **OutputStreamWriters**. For example:

```
PrintWriter out = new PrintWriter(
    new BufferedWriter(
        new FileWriter("myFile.txt")));
```

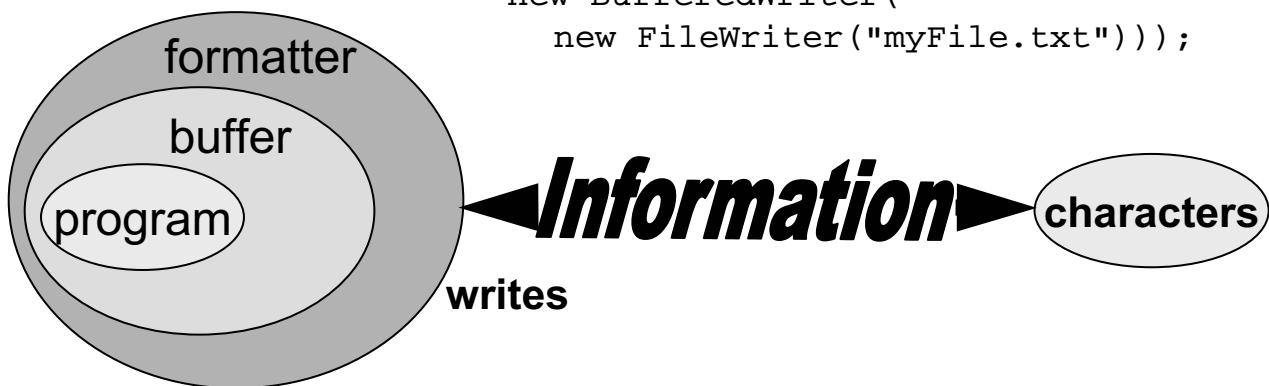


Figure 16-11. Filter stream example: BufferedWriter

SW2446.0

**Notes:**

Wrapping one I/O class around another is a common way to combine functionality.

## ***Dealing with files***

---

- Use streams to read or write files
- Use the **File** class to discover a file's properties
- Useful static fields:
  - `pathSeparator` (“/” in Windows)
  - `separator` (“.” in Windows)
- Sample methods:
  - `delete()`
  - `exists()`
  - `isDirectory()`
  - `isFile()`
  - `mkdir()`
  - ...

---

Figure 16-12. Dealing with files

SW2446.0

### **Notes:**

The File class is an abstraction of file and directory path names.

## ***Midway checkpoint 1***

---

1. What is the package that contains most Java input and output classes?
2. Which streams read characters, and which read bytes?
3. Give an example of three filtered stream classes.



---

Figure 16-13. Midway checkpoint 1

SW2446.0

### **Notes:**

## Why new I/O?

- With stream-based I/O, Java followed the view of an I/O channel as a stream of bytes that was popularized by C and Unix in the 1970s and 1980s
  - Simple
  - Extensible
  - Machine-independent
  - Slow (at some level, one-byte-at-a-time processing)
- Block-oriented I/O:
  - Is inherently faster
    - Some systems can read or write a large block of data almost as fast as they can read or write a single byte
  - Allows for additional features not tolerated by the stream-oriented model
  - Lacks the elegance and simplicity of stream-oriented I/O

---

Figure 16-14. Why new I/O?

SW2446.0

### **Notes:**

In Java 1.4, **java.io** classes have been reimplemented in terms of **java.nio** classes to take advantage of the improved performance.

## Key classes

- *Channels* and *buffers* are the central objects in NIO, and are used for just about every I/O operation
- Channels are analogous to streams in the I/O package
  - All data that goes anywhere (or comes from anywhere) must pass through a Channel object
- A buffer is a container object
  - Data that is sent to a channel must first be placed in a buffer
  - Data that is read from a channel is read into a buffer



Figure 16-15. Key classes

SW2446.0

### Notes:

A buffer is an array. Generally, it is an array of bytes, but other kinds of arrays can be used. But a buffer is more than *just* an array. A buffer provides structured access to data and also keeps track of the system's read/write processes.

## Buffers

- A buffer is a linear, finite sequence of elements of a specific primitive type
- The essential properties of a buffer are:
  - Capacity - the number of elements it contains
  - Capacity is never negative and never changes
  - Limit - the index of the first element that should not be read or written
    - Limit is never negative and is never greater than the capacity
  - Position - the index of the next element to be read or written
    - Position is never negative and is never greater than the limit

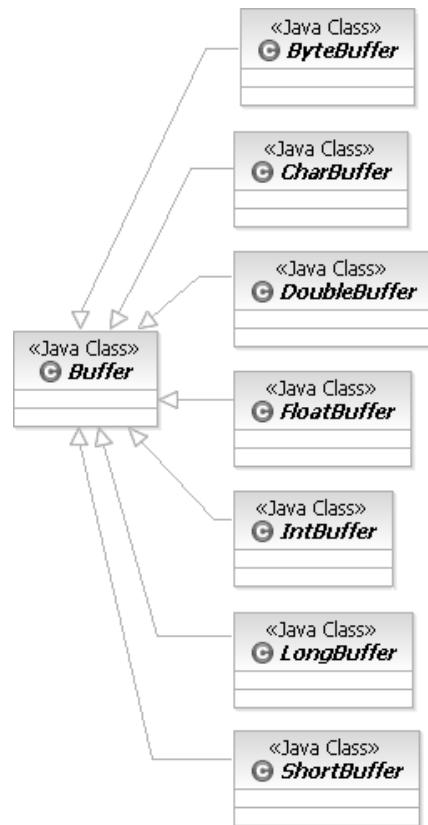


Figure 16-16. Buffers

SW2446.0

### Notes:

There is one subclass of **Buffer** for each non-boolean primitive type.

With new I/O, you write to buffers, and not streams.

## The Channel interface

---

- A channel is an object from which you can read data and to which you can write data
  - A channel is similar to a stream
- All data is handled through Buffer objects
  - You never write a byte directly to a channel; you write to a buffer containing one or more bytes
  - You never read a byte directly from a channel; instead read from a channel into a buffer, and then get the bytes from the buffer
- Objects implementing the channel interface are bi-directional
  - Unlike streams, a channel can be opened for reading, for writing, or for both
  - Closer to the reality of underlying operating system



---

Figure 16-17. The Channel interface

SW2446.0

### Notes:

Channels are defined in the **java.nio.channels** package. **Channel** itself is an interface, not a class; this is not to be confused with the **Channels** class in the **java.nio.channels** package, which provides utility classes for channels and streams.

## **Three steps to read from a Channel**

- Get a channel from a **FileInputStream**:

```
FileInputStream fin = new  
    FileInputStream("readandshow.txt");  
FileChannel fc = fin.getChannel();
```

- Create a buffer:

```
ByteBuffer buff = ByteBuffer.allocate(1024);
```

- Read from the channel into the buffer:

```
buff.clear();  
int bytes_read = fc.read(buff);
```

Figure 16-18. Three steps to read from a channel

SW2446.0

### **Notes:**

The **clear()** method resets the buffer, making it ready to have data read into it.

`bytes_read` will be -1 if the channel has reached end-of-stream.

## Writing to a file

---

- Get a channel from a **FileOutputStream**:

```
FileOutputStream fout = new FileOutputStream(  
    "writesomebytes.txt" );  
FileChannel fc = fout.getChannel();
```

- Create a buffer and put some data in it; in this case, the data will be taken from a byte array called *message*:

```
ByteBuffer buff = ByteBuffer.allocate(1024);  
for (int i=0; i<message.length; i++) {  
    buff.put(message[i]);  
}
```

- Write to the buffer:

```
buff.flip();  
int bytes_written = fc.write(buff);
```

---

Figure 16-19. Writing to a file

SW2446.0

### **Notes:**

The **flip()** method prepares the buffer to have the newly-read data written to another channel.

## **Midway checkpoint 2**

---

1. java.io is based on streams. What is java.nio based on?
2. What are the three basic steps in reading or writing with New IO?
3. Name three high-performance techniques that are enabled by NIO.



---

Figure 16-20. Midway checkpoint 2

SW2446.0

**Notes:**

## Representing objects outside the JVM

- At some time, there will be a need to persist objects in storage
  - Objects can be placed into storage with the same state they have in memory of the JVM

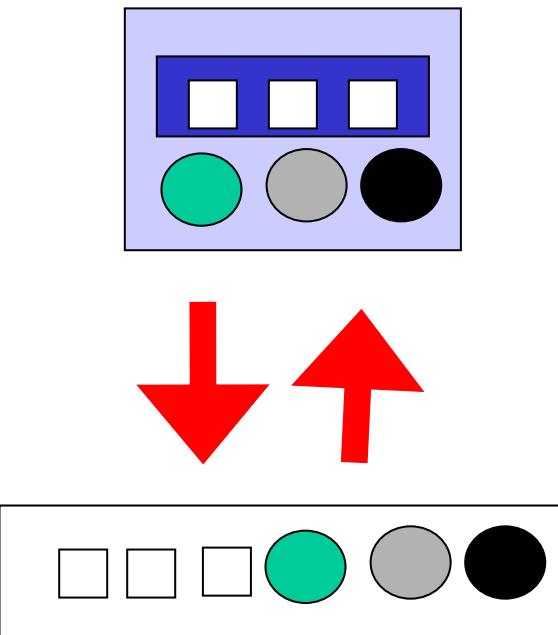


Figure 16-21. Representing objects outside the JVM

SW2446.0

### **Notes:**

Persistence is the ability to record an object's complete state which can be recovered at a later time or at a different site.

## Serialization and externalization

- Java provides two mechanisms for persisting objects:
  - Serialization
  - Externalization
- Serialization or externalization of objects produces an object state which can be reconstructed at a later time, a different site, or both
  - Provides a self-defining basis for object persistence
- Serialization is the *get it all, in a standard way* solution which is the easiest and most common form of persisting objects
- Externalization permits application-specific formatting by requiring the implementation to take full responsibility for saving and restoring state

Figure 16-22. Serialization and externalization

SW2446.0

### **Notes:**

The sets of terms marshall and unmarshall, serialize and de-serialize, and flatten and resurrect, are all used interchangeably.

Much of an object's state consists of references to other objects. Serialization automates the serialization process for all referenced objects of that object which results in much overhead.

Externalization incurs much less overhead since no objects are automatically serialized. Externalizable is a subinterface of serialization. It provides a more direct programming model when objects require significant customized serialization.

## **Serializable interface**

---

- For state to be serialized, a class must declare that it implements `java.io.Serializable`:

```
public class ObjType implements java.io.Serializable
```

- Use `java.io.ObjectInputStream` and `java.io.ObjectOutputStream`:

```
// Save anObj out to a file
ObjectOutputStream out = new ObjectOutputStream
    (new FileOutputStream("saveit.ser"));
out.writeObject (anObj);

// Restore anObj from a file
ObjectInputStream in = new ObjectInputStream
    (new FileInputStream("saveit.ser"));
ObjectType anObj = (ObjectType) in.readObject();
```

---

Figure 16-23. Serializable interface

SW2446.0

### **Notes:**

Serializability of a class is enabled by the class implementing the **java.io.Serializable** interface. Classes that do not implement this interface will not have any of their state serialized or de-serialized. However, the serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

## Transient and readObject()

- Some fields should not be serialized
  - Data which can easily be regenerated
  - Very large data items, which can be instantiated as needed
- Fields marked transient are not serialized
- You can supply code to initialize transient fields by implementing a readObject() method

```
private void readObject(ObjectInputStream ois)
                      throws IOException {
    ois.defaultReadObject();
    // Additional code to initialize transient fields
}

private void writeObject ( ObjectOutputStream os )
                         throws IOException {
    os.defaultWriteObject();
    if (...) {
        //write out additional object state data
    }
}
```

Figure 16-24. Transient and readObject()

SW2446.0

### Notes:

The ObjectInputStream.defaultReadObject() method can reads the non-static and non-transient fields of the current class from the stream, and can only be called from the readObject() method of the class being de-serialized. It will throw the NotActiveException if it is called otherwise.

Threads and file handles should never be serialized.

When implementing Serializable, everything is serialized except for fields marked as transient. By supplying readObject() and writeObject() methods, you can customize this process by specifying conditional code for transient variables. For example, when writing an object, you may want to test to make sure a transient variable that references an image is not null before serialization; this may be a good reason to implement a writeObject() method. The above example calls the defaultWriteObject() method first, then writes any additional data you may want using os.writeObject().

## Serialization object graph

- All non-static and non-transient fields (data members) are serialized
  - This includes the object fields of all super classes
- An object graph is saved
- To prevent cycles, the ObjectOutputStream acts as a container
- Subsequent (embedded) references to a previously serialized object refer to this copy (**not** to serialize it again)

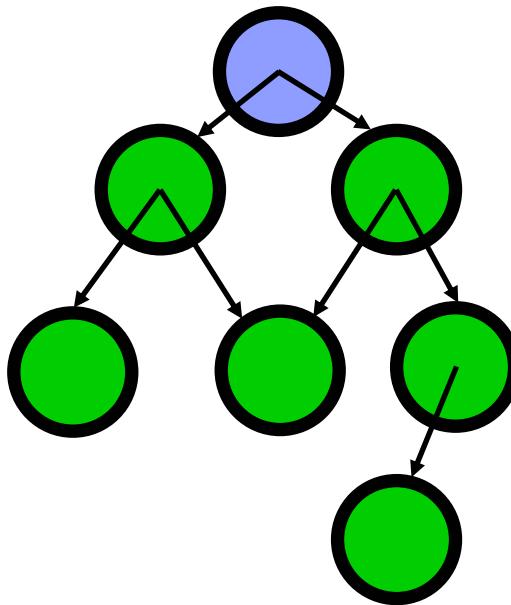


Figure 16-25. Serialization object graph

SW2446.0

### Notes:

When writing several objects to the ObjectOutputStream, a specific object is likely to be referenced multiple times. It is necessary to ensure that only one copy of the object is persisted.

To manage this, the stream acts as a container, giving internal (VM neutral) references or IDs to each object it writes out. Subsequent references to that object will have this ID written out as the placeholder. In fact all embedded object references are written out as an ID.

## Stream unique identifier - SUID

- Sometimes the same bytecode which was used to produce an object is not available when deserializing
- You need a way to ensure that the serialized object is compatible with the available bytecode
- Class information is written out as part of the stream
  - One field is serialVersionUID, the Stream Unique Identifier
  - By default, this is a computed hash signature of the class
  - When deserializing an object, the stream's serialVersionUID for the corresponding class is compared to the available class in the JVM
  - InvalidClassException is raised if the versions do not match

Figure 16-26. Stream unique identifier - SUID

SW2446.0

### **Notes:**

A class may have been modified since the object was serialized, or deserialization may occur in a VM whose CLASSPATH points to different bytecode. There must be a mechanism to ensure that the de-serialized object is compatible with the available bytecode.

Class information is also written out as part of the stream; one field is serialVersionUID. When deserializing an object, the stream's serialVersionUID for the corresponding class is compared to the available class in the JVM. This is the way that you can ensure that the de-serialized object is compatible with the available bytecode. An InvalidClassException is raised if the versions do not match. By default, serialVersionUID is a computed hash signature of the class, but you may instead explicitly define a serialVersionUID (define it as a private final static long). Explicitly defining a serialVersionUID allows you to change the implementation of methods as long as the arguments and return type remain unchanged; you will not get an InvalidClassException upon demarshalling an object.

## Example (1 of 2)

---

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

public class SaveRestoreArrayList {
    public static void main (string [] args) {
        ArrayList names;
        try {
            FileInputStream fis = new
                FileInputStream("arraylist.ser");
            ObjectInputStream ois = new
                ObjectInputStream(fis);
            System.out.println("Reading old names ArrayList");
            names = (ArrayList)ois.readObject();
        }

        //if there is a problem, make a new ArrayList
        catch(Exception e) {
            System.out.println(e);
            System.out.println("Making new names ArrayList.");
            names = new ArrayList();
        }
    }
}
```

Figure 16-27. Example (1 of 2)

SW2446.0

### **Notes:**

**java.util.ArrayList implements Serializable.**

## Example (2 of 2)

```

        System.out.println("Starting contents of ArrayList
                           is\n" + names);
        System.out.println("Adding command line args to the
                           ArrayList");
        for (int i = 0; i < args.length; i++) {
            System.out.println(args [i]);
            names.add(args[i]);
        }
        System.out.println("Ending contents of ArrayList
                           is\n" + names);

        try {
            FileOutputStream fos = new
                FileOutputStream("arraylist.ser")
            ObjectOutputStream oos = new
                ObjectOutputStream(fos);
            oos.writeObject(names);
            System.out.printIn("ArrayList Serialized on disk.")
        } catch(Exception e) {
            System.out.printIn(e);
        }
    }
}

```

Figure 16-28. Example (2 of 2)

SW2446.0

### **Notes:**

Prepare a .ser file as an **ObjectOutputStream**

- Create an instance of a class that implements the **Serializable** interface
- Assign values to properties
- Invoke **writeObject()**
  - Values of properties are saved
  - Values of transient properties are not saved
- Close the output stream

## Externalizable

---

- **Externalizable** is a subinterface of **Serializable**

```
interface Externalizable extends Serializable {}
```

- An object which implements **Externalizable** must provide:

```
public void readExternal (ObjectInput stream)
                         throws IOException {}
public void writeExternal (ObjectOutput stream)
                          throws IOException {}
```

- **Externalizable** permits customized placement of fields to or from a stream

---

Figure 16-29. Externalizable

SW2446.0

### **Notes:**

When there are many manual writes/reads to a serialization stream for a class it is better to use the Externalizable interface. This signals to all that the (persistent) form is highly specialized.

By default only the class identity is saved to the stream when implementing Externalizable. It is the responsibility of the class to write and restore all contents of any referenced instances. You write your own methods to save and restore objects.

## Checkpoint

1. What methods must a class implement if it implements **Serializable**?
2. What are the main differences between the **Serializable** and **Externalizable** interfaces?
3. What field can be used to detect version mismatches in serialized files?



Figure 16-30. Checkpoint

SW2446.0

### **Notes:**

## **Unit summary**

---

- In this unit, you should have learned to:
  - Describe the concept of streams and use the stream classes of the Java language
  - Describe the major functions of stream-based I/O in Java
  - Describe Java's new I/O paradigm
  - List the differences between stream-based I/O and new I/O
  - Explain the role of channels and buffers in new I/O
  - Explain the concepts of serialization and externalization
  - Describe how serialization may be implemented by a class, and what the `transient` keyword does
  - Write code that serializes and deserializes objects

---

Figure 16-31. Unit summary

SW2446.0

### **Notes:**

# Unit 17. JavaBeans

## What this unit is about

This unit describes how Java bean software components are created using the JavaBeans API.

## What you should be able to do

After completing this unit, you should be able to:

- Explain what a Java bean is and outline the three feature sets they expose
- Describe what is provided by the JavaBeans API
- Explain several key concepts and terms associated with beans
- Describe the types of properties a bean may export, and outline what methods of a bean are exported
- Explain what an event source and event listener are
- Describe how bean information can be obtained through introspection and the BeanInfo class

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 17.1 JavaBeans

## Unit objectives

---

- After completing this unit, you should be able to:
  - Define the term *Java bean*, and outline the three feature sets that Java beans expose
  - Describe what is provided by the JavaBeans API
  - Explain several key concepts and terms associated with beans
  - Describe the three levels of use of Java beans
  - Describe the types of properties a bean may export
  - Outline what methods of a bean are exported
  - Explain what an event source and event listener are
  - Describe how bean information can be obtained through introspection and the BeanInfo class

---

Figure 17-1. Unit objectives

SW2446.0

### **Notes:**

## What is a Java bean?

- Java bean:
  - A reusable *software component* that may be manipulated visually in a builder tool
  - A Java class that adheres to specific rules and naming conventions to make tool-based manipulation easy
- Java beans expose three feature sets to builder tools so that they may be visually manipulated:
  - Properties
  - Methods
  - Events

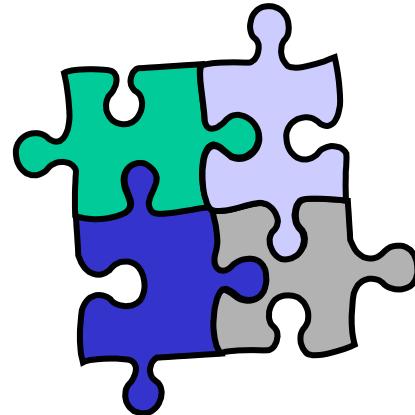


Figure 17-2. What is a Java bean?

SW2446.0

### **Notes:**

A Java class that adheres to specific conventions for its property and event interface definitions can be a Java bean. Java beans, also referred to as beans, can be manipulated visually in a builder tool and assembled into applications.

A Java bean's feature set (properties, methods and events) are exposed to a visual builder tool through standard naming conventions used by a Java bean class; a visual builder tool can recognize these naming patterns and use them to analyze and discover a bean's properties and behavior, and thus allow these features to be visually manipulated. The process in which a builder tool discovers this information about a bean is called introspection, and this capability is what differentiates Java bean classes from other Java classes.

The most typical use of a Java bean is as a graphical user interface component, for example, components in the `java.awt` and `javax.swing` packages; however, though all Java beans may be manipulated visually in a builder tool, not every bean has a visual representation at run time.

## **JavaBeans as software components**

---

- The JavaBeans architecture allows reusable software components to be built, called Java beans
- As code evolves through development and refactoring stages, the goal is to reduce the complexity of code
  - Another goal is to reuse code wherever possible
    - Similar functionality is often needed in different systems, such as labels or buttons on interfaces
- Similar functionality needed across many systems can be encapsulated into reusable components
- These reusable components can be put together to build an application

---

Figure 17-3. JavaBeans as software components

SW2446.0

### **Notes:**

## ***JavaBeans API***

---

- The JavaBeans API is a framework that allows embeddable, modular, and reusable software components to be constructed
- Bridges and wrappers allow bean components to be used by other component technologies
- Bean specification Web address:  
–<http://java.sun.com/products/javabeans/index.jsp>

---

Figure 17-4. JavaBeans API

SW2446.0

### **Notes:**

The JavaBeans API is made up of two main packages: `java.bean` and `java.beans.beancontext`. Also important to the JavaBeans API are several naming and API conventions to which JavaBeans and visual builder tools must adhere.

## **JavaBeans terminology (1 of 2)**

---

- Reflection
  - A Java API `java.lang.reflect` for finding dynamic information about a class (metadata)
- Introspection
  - A bean protocol for discovering dynamically at runtime the exported features of a bean (uses reflection)
- Visual bean
  - A bean that has a visual representation at runtime, such as list, button, or frame
- Invisible bean
  - A bean that does not have a visual representation at runtime, such as Employee or Car

---

Figure 17-5. JavaBeans terminology (1 of 2)

SW2446.0

### **Notes:**

Introspection is used by visual builder tools to determine the features (properties, methods and events) of a bean. The Introspector class recognizes and discovers bean properties, events and methods, using the Reflection API and the JavaBeans naming conventions. Introspection is supported by beans in two ways: bean feature names adhere to specific naming conventions that allow visual builder tools to recognize and discover properties, events and methods, and beans may also provide information on properties, methods, and events explicitly in a bean information (BeanInfo) class. For example, the bean class method names may be examined to determine if they get and set property accessor methods.

Recall that beans may not have a visual representation at run time. However, all beans may have their properties and behavior manipulated at design time. The reason beans may be visually manipulated at design time is because they follow rules and method naming conventions that allow tools to discover and expose information about their properties and behavior.

## JavaBeans terminology (2 of 2)

- Properties

- Attributes that relate to the behavior or appearance of the bean
- Visual builder tools use introspection to discover a bean's properties; exposed properties may be manipulated

- Methods

- Normal Java methods
- Called by other beans or by scripting languages

- Events

- Facilitate communication with other beans
- Listener beans receive events after registering an interest in the event with the event source bean
- Builder tools can examine a bean to discover the events that a bean can fire and receive

---

Figure 17-6. JavaBeans terminology (2 of 2)

SW2446.0

### **Notes:**

**Properties** - Properties determine the appearance and behavior of a bean; properties are examined by visual builder tools, and exposed for customization at design time. They are part of the bean's internal state, and are accessible through accessor and mutator methods.

**Methods** - The methods of a Java bean are normal Java methods; the public methods of a bean are exported, excluding methods that get and set property values and register and remove event listeners. Beans are designed for use by visual builder tools, however the APIs are also easily understood and used by programmers, allowing beans to be manipulated using text editors.

**Events** - Events allow beans to communicate with each other. Source beans fire (send) an event; other beans may be interested in a particular event, and can register themselves as listeners for an event. Listener beans will receive and handle an event when it is fired. Introspection is performed by visual builder tools to discover which events a bean fires and handles.

## ***Three levels of JavaBeans usage***

---

- Tool developers
  - Create builder tools and Integrated Development Environments (IDEs) that allow applications to be built with beans
- Bean developers
  - Create the actual bean components
- Bean consumers
  - Build applications using predefined bean components

---

Figure 17-7. Three levels of JavaBeans usage

SW2446.0

### **Notes:**

Three categories of programmers use JavaBeans:

**Tool developers** - The JavaBeans API is used by tool developers to manipulate beans within the visual builder tool. The JavaBeans naming conventions must also be well-known.

**Bean developers** - Bean developers create Java beans and must know the JavaBeans naming and packaging conventions.

**Bean consumers** - Bean consumers compose pre-existing beans into an application, and must be familiar with the general JavaBeans concepts.

## **Bean basics**

- Any Java class can become a bean
- No special sub-classing is required; instead, there is a set of protocols that must be implemented:
  - Bean class must have a zero-argument constructor
  - The class must implement the **Serializable** interface
  - Properties must be encapsulated via accessors and mutators
  - Properties, methods and events must follow bean naming conventions

Figure 17-8. Bean basics

SW2446.0

### **Notes:**

Beans must follow a set of rules and conventions; they are not a part of the JavaBeans API, but are important rules beans must adhere to. These conventions are sometimes also called design patterns because they are naming and design patterns for Java beans.

Visual builder tools rely on these conventions being used to be able to perform introspection. Important information is provided through naming conventions, though in some cases alternatives to using the naming conventions exist; however, the information typically provided through the use of naming conventions must still be present, and is typically included in the BeanInfo class (discussed in a later slide) if a naming convention is not used.

Properties are accessible through accessor and mutator (get and set) methods. For example, if a bean has a property *prop* of type *Type*, the accessor (get) method would have the following signature:

```
public Type getProp()
```

The mutator (set) method would have the following signature:

```
public void setProp(Type)
```

## Exported bean features

- Features of a Java bean, including properties, methods and events, are exported by a visual builder tool so they may be manipulated
  - Features are discovered through introspection
- Internal state is set and queried by *exported properties*
- Bean components can be manipulated with *exported methods*
  - All public methods are exported by default, excluding methods that get and set property values and register and remove event listeners
- Beans communicate with other components using *exported events*

---

Figure 17-9. Exported bean features

SW2446.0

### **Notes:**

## ***Bean properties***

---

- A property of a Java bean is a part of a bean's internal state that determines the bean's appearance or behavior
  - Accessor and mutator methods are used to get and set the values of properties
- There are different types of properties
  - Simple properties: represent a single value, for example, a boolean property
  - Indexed properties: represent an array of values
  - Bound properties: notifies other objects when its value changes through the firing of a PropertyChangeEvent
  - Constrained: fires a VetoableChangeEvent when its value changes, allowing those listening for this event to veto the change of the property value

---

Figure 17-10. Bean properties

SW2446.0

### **Notes:**

Indexed properties have an array value, in addition to accessor and mutator methods that allow the array and its individual elements to be accessed.

Bound properties send a PropertyChangeEvent to any interested PropertyChangeListener objects when a change in the property value occurs.

Constrained properties can have any changes to its property vetoed by registered listener objects. A PropertyChangeEvent is fired and sent to all registered VetoableChangeListener objects when the value of a constrained property changes. If any of the registered listener objects throws a PropertyVetoException, the constrained property value cannot be changed; the PropertyVetoException propagates back to the mutator method of the property.

## **Bean methods and events**

---

- Exported Methods
  - All public methods are exported, excluding property accessor/mutator methods and methods that register and remove event listeners
- Events are a core feature of the JavaBeans architecture
  - Events are supported by the Java 1.1 event handling model, which is based on the idea of an event source and event listener
- Events are a mechanism for propagating state change notifications between an event source and an event listener
- An event source is a bean capable of generating events
- An event listener is an application or bean capable of responding to events

---

Figure 17-11. Bean methods and events

SW2446.0

### **Notes:**

As provided by the Java 1.1 event handling model, objects that generate or fire events are labeled event source objects. Objects interested in receiving or handling the event register themselves as event listener objects. Source objects maintain a list of listener objects, and include methods to add and remove objects from this list. Most of the time, events are sent to all registered listeners; however, there may be cases where an event is sent to only one listener.

## **Bean example - Counter**

---

- Beans have properties
  - Counter has a bound currentValue property, with setter/getter methods
    - public int getCurrentValue();
    - public void setCurrentValue(int aValue);
- Beans can have methods
  - Counter has increment and decrement methods
    - public void increment();
    - public void decrement();
- Beans can fire events
  - PropertyChangeEvent for currentValue

---

Figure 17-12. Bean example - Counter

SW2446.0

### **Notes:**

## ***Bean information***

---

- Java beans provide information about themselves in two ways:
  - Beans adhere to rules and naming conventions which allow introspection to be used
    - Introspection uses the Java reflection package (JDK 1.1) builder tools to interrogate Java class methods and fields
  - BeanInfo class can be defined to export information
    - This class can be used by the builder tool to discover properties, methods, events, and other information about the bean
    - If the visual builder tool is unable to find the BeanInfo class associated with a bean, the tool will use introspection to discover bean information based on naming conventions
    - The BeanInfo class has a name which appends “BeanInfo” to the bean class name
    - The BeanInfo class implements java.beans.BeanInfo interface

---

Figure 17-13. Bean information

SW2446.0

### **Notes:**

The BeanInfo or bean information class, provides property, method, and event information regarding the bean. The BeanInfo class implements a BeanInfo interface, and this class explicitly outlines the features that should be exposed to visual builder tools. A BeanInfo class may only outline part of the bean's behavior, and builder tools look for information about a bean in several sources.

## **Checkpoint**

---

1. What is a Java bean?
2. What three feature sets are exposed by Java beans?
3. Name four types of properties a Java bean may have.
4. What is the difference between an event source and an event listener?



---

Figure 17-14. Checkpoint

SW2446.0

### **Notes:**

## Unit summary

---

- After completing this unit, you should have learned to:
  - Define the term *Java bean*, and outline the three feature sets that Java beans expose
  - Describe what is provided by the JavaBeans API
  - Explain several key concepts and terms associated with beans
  - Describe the three levels of use of Java beans
  - Describe the types of properties a bean may export
  - Outline what methods of a bean are exported
  - Explain what an event source and event listener are
  - Describe how bean information can be obtained through introspection and the BeanInfo class

---

Figure 17-15. Unit summary

SW2446.0

### **Notes:**



# Unit 18. JDBC Overview

## What this unit is about

This module introduces JDBC, the Java API for relational database access. JDBC concepts and implementation are explained, and examples of statement execution are provided.

## What you should be able to do

After completing this unit, you should be able to:

- Describe how Java applications access relational databases
- Explain the JDBC architecture
- Execute a JDBC query
- Access data returned from a query in a ResultSet
- List the Java types that SQL types map to
- Use PreparedStatements and CallableStatements to create efficient queries

## How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises



## 18.1 JDBC Overview

## Unit objectives

---

- After completing this unit, you should be able to:
  - Describe how Java applications access relational databases
  - Explain the JDBC architecture
  - Execute a JDBC query
  - Access data returned from a query in a ResultSet
  - List the mappings between Java types and SQL types
  - Use PreparedStatements and CallableStatements to create efficient queries

---

Figure 18-1. Unit objectives

SW2446.0

### **Notes:**

## **Java and databases**

---

- Java provides the JDBC API for working with relational data
- JDBC provides a standard way for Java programs to use a variety of relational data sources; it is the Java API that allows SQL statements to be executed in database-independent ways
- JDBC allows Java programs to:
  - Connect to a database
  - Query databases using SQL statements
  - Process the query results

---

Figure 18-2. Java and databases

SW2446.0

### **Notes:**

JDBC allows you to access and process any tabular data, typically data found in a relational database. The term “data source” refers to any tabular data source, typically a relational database.

JDBC may be thought of as standing for “Java Database Connectivity”, but it is not actually an acronym. The name comes from people saying that JDBC was “like ODBC, but for Java”.

## **JDBC 3.0 packaging**

---

- The `java.sql` package provides the API for working with data stored on a data source
  - Allows data stored in a relational database to be accessed and processed using Java
  - Allows SQL statements to be executed against a database
  - Different database drivers can be dynamically registered to allow various databases to be accessed
- `javax.sql` is an optional package API, used for server-side database access and resource management
  - Extends `java.sql` to include server-side data source access and processing
  - Offers connection pooling and `DataSource` support
  - Provides an interface for distributed transactions

---

Figure 18-3. JDBC 3.0 packaging

SW2446.0

### **Notes:**

Both the `java.sql` and `javax.sql` packages are included in the JDK 1.4 platform.

Connection pooling and `DataSource` support will be discussed in the `DataSource` lecture.

Essentially, the `javax.sql` package provides many facilities widely used in enterprise applications. Distributed transaction support allows data sources running on multiple database servers to be involved in a single transaction.

## JDBC architecture

- JDBC is based on the X/Open SQL Call Level Interface (CLI) and Microsoft's ODBC
- JDBC is intended for direct use from Java

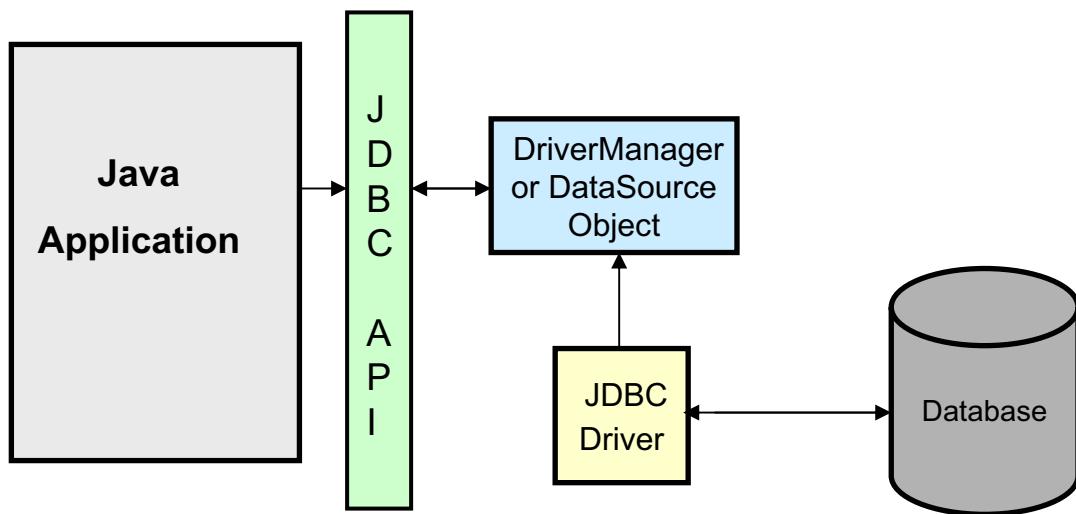


Figure 18-4. JDBC architecture

SW2446.0

### **Notes:**

The standardization of SQL regulated the commands that get sent to a database to perform queries; however, before SQL commands can be issued, you must connect to the database. A major issue was that each database vendor has their own interface for connections; each vendor also extended SQL in various ways. The ODBC API on which JDBC is partly based is a C interface (procedural programming language), and it provided an interface which standardized the way programs could communicate with a database system. SQL and ODBC together allow databases to be connected to and communicated with in a manner independent of the database vendor.

JDBC provides a database-independent way of connecting to and communicating with relational databases, but is intended for direct use from Java (an object-oriented programming language). Key concept - the application developer codes to the JDBC interfaces; each vendor supplies its own set of implementation classes. The application developer does not need to be concerned with the actual class names (except for loading the driver).

There are two main interfaces in the JDBC API. One API is used by application developers in connecting to and working with a database. The second API is a lower-level JDBC driver API that is used by driver writers.

The DriverManager is a class that is responsible for managing a set of JDBC drivers (DataSources will be discussed in a later lecture, and are the preferred method of connecting to a database). The JDBC Driver is a class which implements the JDBC Driver interface. The JDBC Driver takes messages passed in through the JDBC API and converts them to the database-specific protocol, permitting a specific database to be accessed.

## Using JDBC to execute an SQL query

- The steps to executing a database query are:
  - DriverManager loads the JDBC Driver
  - Get a database connection using the Connection object
  - Create an SQL query using the Statement object
  - Execute the query

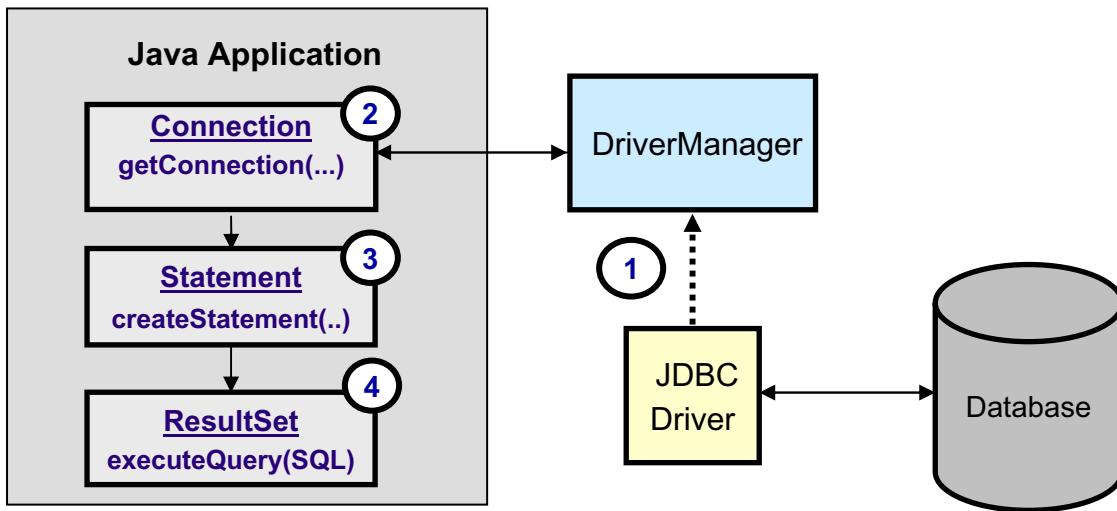


Figure 18-5. Using JDBC to execute an SQL query

SW2446.0

### Notes:

In the diagram above:

1. The DriverManager loads the JDBC Driver that is needed in order for the commands that are issued by the program to be translated into the vendor-specific commands.
2. The Java application obtains a database connection by using the `DriverManager.getConnection()` method, passing in a database URL (and possibly a userid and password) for connecting to the database. A `Connection` object is returned, if a connection to the database was successfully established.
3. An SQL statement is built using the `Connection.createStatement()` method; a `Statement` object is returned, and you are now ready to query the database.
4. An SQL query is run against the specified database by calling one of the methods of the `Statement` object, possibly `Statement.executeQuery()` or `Statement.executeUpdate()`, specifying the SQL statement as a String parameter.
5. In this case, since the `executeQuery()` method of the `Statement` object was used, a `ResultSet` will be returned and will provide the results of the SQL query.

## Query example

---

- This class loads a database driver, creates a connection to the database, creates a Statement, then executes a query on the database:

```
public void sqlQuery() {  
    Connection conn = null;  
  
    try {  
        // Load a driver and create a connection  
        Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");  
        conn = DriverManager.getConnection("jdbc:db2:Employee",  
                                         "user", "password");  
  
        // Create a statement and execute a query  
        Statement query = conn.createStatement();  
        ResultSet result =  
            query.executeQuery("SELECT * FROM Employee");  
  
    } catch (ClassNotFoundException e) {  
        System.out.println("Could not find driver" + e);  
    } catch (SQLException e) {  
        System.out.println("Connection creation failed" + e);  
    }  
}
```

---

Figure 18-6. Query example

SW2446.0

### **Notes:**

Note that the try/catch block should have an accompanying finally clause; in the finally clause, the ResultSet, Statement, and Connection resources (objects) should be closed using their respective close() methods. This will be discussed more in a later slide.

The JDBC Driver, COM.ibm.db2.jdbc.app.DB2Driver in this case, is a class which implements the JDBC Driver interface. The JDBC Driver takes messages passed in through the JDBC API and converts them to the database-specific protocol.

If the SQL query does not return a result set (that is, is not a SELECT statement), executeUpdate may be used instead. Not only are INSERT, DELETE and UPDATE statements executed this way, but also data definition statements (DDL), such as “CREATE TABLE”.

## **ResultSet**

---

- A ResultSet provides access to a table of data generated by executing a database query
  - Maintains a cursor pointing to its current row of data
  - Initially the cursor is positioned before the first row
- The `next()` method moves the cursor to the next row
  - A boolean value of `false` is returned if there are no more rows
  - Table rows are retrieved in sequence

---

Figure 18-7. ResultSet

SW2446.0

### **Notes:**

**ResultSet** was enhanced starting with Java 1.3. Methods now exist to scroll backwards, and to absolute or relative rows, if the underlying database system supports it.

## ***ResultSet methods***

---

- ResultSet has methods to retrieve column values for the current row
  - Within a row, column values can be accessed in any order
- The methods all follow these templates:
  - Retrieve value at column with a given index (column numbering begins at 1):  
`<Type> get<Type>(int columnIndex)`
  - Retrieve value at column with a given name:  
`<Type> get<Type>(String columnName)`
- The access methods perform the necessary type conversion between SQL types and Java types
  - `getString()` can be performed on any column
  - `String age = resultSet.getString("AGE");`

---

Figure 18-8. ResultSet methods

SW2446.0

**Notes:**

## ResultSet example

- This example extends the previous example, by retrieving results from the ResultSet within a while loop

```

// Load driver and get connection
// as in previous example. Then...
Statement query = conn.createStatement();
ResultSet result = query.executeQuery(
    "SELECT * FROM Employee");

// Iterate through result
while (result.next()) {
    System.out.println(
        result.getString("firstname") + " " +
        result.getDate("birthdate"));
}

```

Figure 18-9. ResultSet Example

SW2446.0

### **Notes:**

Recall that the ResultSet maintains a cursor pointing to its current row of data, and initially the cursor is positioned before the first row. Therefore, when the while loop initially evaluates `result.next()`, it moves the cursor to the first row of data returned by the query. When no more rows exist in the ResultSet, a boolean value of `false` is returned and the loop will exit.

Starting with JDBC 2.0, ResultSets allow you to scroll forward or backwards through the results. As well, read-only and updateable result sets are allowed. Parameters that set these and other options regarding the ResultSet are specified when the SQL statement is created. See the Java documentation on the ResultSet class for more information on setting these.

## SQL exceptions

---

- Most of the methods in Statement and Connection can throw an SQLException
  - Checked exceptions must be caught
- This can be handled in one of three ways:
  - Catch the exception and recover locally
  - Delegate the exception to the calling method
    - May lose an opportunity to provide detailed information on error location and context
  - Catch and "convert" the exception by throwing another exception
    - Wrap the original exception in the new one:

```
catch (SQLException sqle) {  
    throw new MySQLException(  
        "Select for employee failed in getEmp",  
        sqle);  
}
```

---

Figure 18-10. SQL exceptions

SW2446.0

### **Notes:**

Catching the SQLException and recovering locally simply means that your catch block has code within it to handle the SQLException.

Delegating the exception to the calling method means that your method throws the exception itself, forcing the calling method to handle it.

Catching and “converting” the exception by throwing another exception involves throwing a new exception within the catch block code. Instead of using the catch block to handle the SQLException, you use it to throw a new exception, wrapping the original SQLException in the new exception so information is not lost. Code for this situation is shown in the slide.

## PreparedStatements (1 of 2)

- Using the Statement object, the SQL statement is compiled each time it is executed
  - This can be inefficient if the same SQL statement is run many times with different parameters
- To avoid this costly overhead, an SQL statement can be compiled once, then run many times with different search parameters.
  - The query is pre-compiled and stored in a PreparedStatement object
- This object can then be used to efficiently execute this statement many times with different values

Figure 18-11. PreparedStatements (1 of 2)

SW2446.0

### **Notes:**

Use PreparedStatements when you are going to execute the same Statement numerous times (parameters of the query may change each time). One benefit to using PreparedStatements is that they allow the SQL statement to be immediately sent to the database and compiled; thus, the PreparedStatement contains an SQL statement that has been *precompiled*. While Statement objects require the SQL statement to be compiled each time it is executed, this costly operation has been performed in advance with the PreparedStatement, and this overhead is saved when the query is executed. Another benefit is increased clarity in the code: variables are separated from the SQL. This makes the code easier to maintain.

Typically PreparedStatements are used for SQL statements that have parameters; these parameters may change on each execution of the query.

## **PreparedStatements (2 of 2)**

---

- Variable parameter data is denoted with “?”
  - Parameters may be substituted wherever a literal value would be allowed
  - Set the variable parameters and re-execute the statement

```
PreparedStatement pstmt =  
con.prepareStatement("INSERT  
    INTO AddressBook (NAME, PHONENO, EMAIL)  
    VALUES (?, ?, ?)");  
  
pstmt.setString(1, "Jane Doe");  
pstmt.setString(2, "555-1212");  
pstmt.setString(3, "jdoe@mytown.com");  
pstmt.executeUpdate();
```

---

Figure 18-12. PreparedStatements (2 of 2)

SW2446.0

### **Notes:**

Unlike the Statement object, an SQL statement is given to the PreparedStatement object when it is created.

## **PreparedStatement methods**

---

- Methods in PreparedStatement allow you to:
  - Set parameter values
    - One `set<Type>()` method per Java datatype, such as `setInt(int, int)`, `setDate(int, Date)`.
    - When using the `set<Type>()` methods to set the values of each query parameter, the first argument specifies the index of the parameter (parameter numberings start with 1)
  - Execute a SELECT SQL statement
    - `executeQuery()`
  - Execute an INSERT, UPDATE or DELETE SQL statement
    - `executeUpdate()`

---

Figure 18-13. PreparedStatement methods

SW2446.0

### **Notes:**

Do not use PreparedStatements for SQL statements that cannot contain parameters. In particular, do not use PreparedStatements for data definition language (DDL) such as CREATE TABLE.

## **CallableStatement**

---

- SQL stored procedures can also be executed from Java
  - CallableStatement facilitates calls to SQL stored procedures
  - CallableStatement inherits Statement methods that allow it to work with SQL, and also inherits PreparedStatement methods that allow it to work with input parameters
- Calls to stored procedures using the CallableStatement can be faster than using PreparedStatements
  - The stored procedure resides with the database, and is specific to that database
  - JDBC provides a standard way to execute stored procedures for all relational databases
- CallableStatements are executed using the methods inherited from PreparedStatement

---

Figure 18-14. CallableStatement

SW2446.0

### **Notes:**

Stored procedures are used to reduce network traffic between the database client and the database, improve database efficiency, and enhance database security.

Stored procedures are user-generated functions or procedures that are registered with the database and can be called by client applications. They are named SQL statements, and you may have many of them.

## CallableStatement input/output (1 of 2)

- Input (IN parameters)
  - IN parameters supply input values to the stored procedure
  - IN parameters are set using the set methods inherited from PreparedStatement
    - The set<Type> () methods that correspond to Java datatypes, such as `setInt(int, int)`,  
`setDate(int, Date)`
    - When using the set<Type> () methods to set the values of each stored procedure parameter, the first argument specifies the index of the parameter (numberings start with 1)

---

Figure 18-15. CallableStatement input/output (1 of 2)

SW2446.0

### **Notes:**

Calls to stored procedures can have two formats, one format which allows a return value/output (OUT parameter) for the stored procedure to be specified, and the other format which does not allow a return value to be received. Input (IN parameters) are allowed to be specified for either stored procedure format, and stored procedures may have parameters for input values (IN parameters), output values (OUT parameters), or both input and output values (INOUT parameters). Just as in PreparedStatements, a question mark is used to indicate that a parameter will be specified as a value.

## **CallableStatement input/output (2 of 2)**

- Output (OUT parameters)
  - OUT parameters allow output values to be returned by the stored procedure
  - OUT parameter types (SQL types) must be registered prior to executing the stored procedure
  - Values are retrieved by `get<Type>(int)` methods
    - For example, <Type> can be `String`
- Input/Output (INOUT parameters)
  - INOUT parameters supply input *and* allow output values to be returned by the stored procedure
  - Input values must be set using the proper `set<Type>` method
  - Output must be registered with the `registerOutParameter` method prior to stored procedure execution

---

Figure 18-16. CallableStatement input/output (2 of 2)

SW2446.0

### **Notes:**

The Java syntax for calling a stored procedure takes on several forms. These formats are outlined below, and parameters appear within the square brackets, following the stored procedure name, and are optional. The question mark denotes a parameter, and can be used to specify IN, OUT or INOUT parameters; the type of parameter denoted by the question mark depends on the stored procedure, `procedure_name` in this case.

The syntax for a stored procedure call that does not return an output parameter is:

```
{call procedure_name[(?, ?, ...)]}
```

The syntax for a stored procedure call that return output parameters is:

```
{? = call procedure_name[(?, ?, ...)]}
```

The syntax for a stored procedure call that contains no input or output parameters is:

```
{call procedure_name}
```

## CallableStatement example (1 of 2)

```
CallableStatement cstmt = con.prepareCall  
    ("{call getTestData(?, ?)}");  
  
// Set input parameters  
cstmt.setByte(1, 25);  
cstmt.setBigDecimal(2, 83.75);  
  
// Register output parameters  
cstmt.registerOutParameter(1,  
    java.sql.Types.TINYINT);  
cstmt.registerOutParameter(2,  
    java.sql.Types.NUMERIC);  
ResultSet rs = cstmt.executeQuery();
```

Figure 18-17. CallableStatement example (1 of 2)

SW2446.0

### **Notes:**

The set<Type>() methods are inherited from PreparedStatement.

The OUT parameters need to be registered before the statement is executed.

Recall that the executeQuery() method will return a ResultSet, so create one here to access returned results.

For the output parameter registration, the first int argument is the index of the parameter returned from the procedure execution. The second int is the JDBC type returned from the database. This will determine which get<Type>() is needed for retrieval.

## **CallableStatement example (2 of 2)**

---

```
// Retrieve and print values of the ResultSet
while (rs.next()) {
    String name = rs.getString(1);
    int score = rs.getInt(2);
    int percentile = rs.getInt(3);
    System.out.print("name = " + name + ", "
                     + score);
    System.out.println(", percentile = " +
                     percentile);
}

// Retrieve output parameters
byte x = cstmt.getByte(1);
java.math.BigDecimal n =
    cstmt.getBigDecimal(2);
```

---

Figure 18-18. CallableStatement example (2 of 2)

SW2446.0

### **Notes:**

Note that the ResultSet should be processed before the OUT parameters, due to the limitations of some databases.

The get<Type>() passes the index of the OUT parameter being retrieved. The CallableStatement.get<Type> method that should be used is the one with the Java type that corresponds to the JDBC type that was registered for the parameter.

## **JDBC efficiency**

---

- Using JDBC efficiently depends on a number of factors:
  - Use the right driver
  - Use JDBC's PreparedStatement and CallableStatement when appropriate
  - Always close resources, such as Statement, PreparedStatement or CallableStatement objects, ResultSets and Connections when finished using them
    - Usually done in a finally clause to ensure execution even in the face of errors
- Database access is the performance bottleneck in most Java applications that use a database

---

Figure 18-19. JDBC efficiency

SW2446.0

### **Notes:**

Many types of drivers exist, and some have more overhead than others and are not designed for specific environments; choose the fastest one possible for your application.

Closing resources such as Statement, ResultSets and Connections releases the database and JDBC resources instead of waiting for it to happen automatically; this is best practice and avoids resources being held unnecessarily.

Creating a database connection is the most resource-intensive step in this process; however, once this connection has been established, other operations using it are much less expensive and it would be preferable if this connection could be reused (similar to the way PreparedStatements allow the same SQL statement to be reused, being executed with different parameters). The lecture on DataSources will discuss this further.

## **Checkpoint**

---

1. What is the primary method for moving between rows of a ResultSet? How do you know when there are no more rows?
2. Name two techniques for increasing performance when performing the same query more than once.
3. Name three types of parameters that handle the input and output of a stored procedure call.
4. What are three factors in JDBC efficiency?



---

Figure 18-20. Checkpoint

SW2446.0

### **Notes:**

## Unit summary

- In this unit, you should have learned:
  - How Java applications access relational databases
  - An overview of the JDBC architecture
  - How JDBC queries can be executed
  - How to access data returned from a query in a ResultSet
  - The Java types to which SQL types map
  - How to use PreparedStatements and CallableStatements to create efficient queries

---

Figure 18-21. Unit summary

SW2446.0

### **Notes:**

## Appendix: mapping SQL types

<u>SQL TYPE</u>	<u>ResultSet method</u>	<u>Java Type</u>
CHAR	getString	String
VARCHAR	getString	String
LONGVARCHAR	getString, getAsciiStream, getUnicodeStream	String InputStream
NUMERIC	getBigDecimal	java.math.BigDecimal
DECIMAL	getBigDecimal	java.math.BigDecimal
BIT	getBoolean	boolean
TINYINT	getByte	byte
SMALLINT	getShort	short
INTEGER	getInt	int
BIGINT	getLong	long
REAL	getFloat	float
FLOAT	getDouble	double
DOUBLE	getDouble	double
DATE	getDate	java.sql.Date
TIME	getTime	java.sql.Time
TIMESTAMP	getTimestamp	java.sql.Timestamp

Figure 18-22. Appendix: mapping SQL types

SW2446.0

### **Notes:**

LONGVARCHAR is a relic of the earlier days of DB2, and should be avoided. Use CLOB for large amounts of character data, and BLOB for binary objects.

NUMERIC and DECIMAL are synonyms.

Recall that the ResultSet access methods perform the necessary type conversion between SQL types and Java types.

# Unit 19. Java Naming and Directory Interface

## What this unit is about

This unit introduces naming and directory services, and the Java Naming and Directory Interface which allows Java programs to access these services. An example showing how JNDI is used to access a naming service is provided.

## What you should be able to do

After completing this unit, you should be able to:

- Explain the benefits and use of naming and directory services
- Define JNDI and explain what it is used for
- Use JNDI to access a naming service and look up a resource

## How you will check your progress

Accountability:

- Checkpoint questions



## 19.1 Java Naming and Directory Interface

## **Unit objectives**

---

- After completing this unit, you should be able to:
  - Explain how and why naming services are used
  - Explain how and why directory services are used
  - Define JNDI and explain how and why it is used
  - Use JNDI to look up a resource

---

Figure 19-1. Unit objectives

SW2446.0

### **Notes:**

## The problem: finding resources

- In a distributed application, many components of the system may reside on different machines, in different locations
  - These components must work together to provide the functionality of the system
  - If they cannot find each other, they cannot work together
  
- One application may use the services of another application
  - In order to use the resources provided by another application, there must be some way of looking up those resources

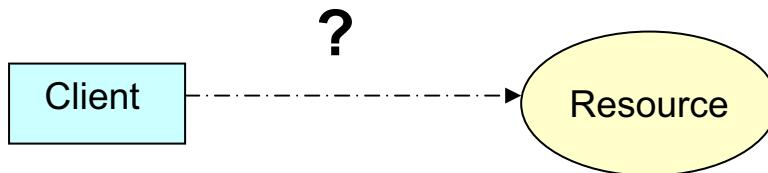


Figure 19-2. The problem: finding resources

SW2446.0

### **Notes:**

If a program needs to use a resource, such as a database, it will need to know the location of the resource. If this information is hardcoded into the program, the source code will need to be updated and recompiled if the resource is relocated. It would be beneficial if there was another way of accessing a resource that did not require the client program to know and hardcode the exact location of the resource.

## The solution: naming and directory services

- Naming and directory services make it possible for resources (objects) to be looked up
  - These services are like a library card catalog: you know the name of the book you want to borrow, and you use the card catalog to find out where the book is on the shelves
  - A card catalog also provides additional information about the book on its cards, and directory services provide this functionality

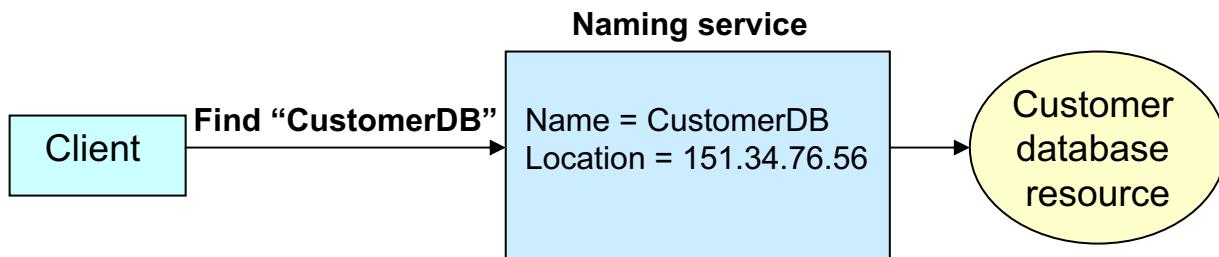


Figure 19-3. The solution: naming and directory services

SW2446.0

### Notes:

The naming service knows where a resource can be found, and maps a name to that location. The client program only needs to know the name of a resource, and can ask the naming service to find the resource with a given name.

Note that the word object in this context does not necessarily refer to a Java object; it refers to a software entity that can represent diverse system resources such as printers, servers, executable software components, users, access control information, and software connections to databases or external systems. In the next unit, you will learn how to look up and use a JDBC resource called a DataSource.

## **Uses of naming and directory services**

---

- Naming services and directory services provide complementary functionality
  - Most products provide both naming and directory services
- Naming services typical usage:
  - Enterprise Java applications access components and resources by dynamically looking up their logical name at run time
- Directory services typical usage:
  - Names and passwords used for user authentication and access control are registered in a directory service
  - System resources such as printers and workstations are tracked
  - Java applications do not generally use directory services directly, though they may
    - Java applications typically rely on middleware such as application servers to provide services such as security
    - Middleware facilitates services such as access control of applications using directory services

---

Figure 19-4. Uses of naming and directory services

SW2446.0

### **Notes:**

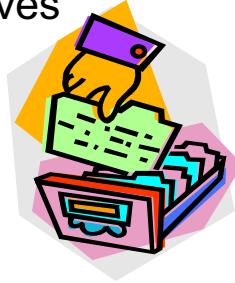
Naming services supply a name-to-object mapping and provide object lookup services based on name, while directory services supply information (called attributes) about the objects and provide facilities to search for objects based on name or attributes.

Generally, middleware and administrators make use of directory services; for example, an application server will use directory services to register access control information, in order to control access to applications. Thus, enterprise Java applications, which run on these application servers, do not use directory services directly, but instead rely on application servers to provide these services.

## ***What is a naming service?***

---

- Consider a card catalog at a library
  - Each card has a book name and a location telling you where to find it on the shelves



- A naming service is a card catalog for computer resources
  - It maps human-friendly names to computer resources (objects)
  - The name is *bound* to the actual resource or a resource reference, telling you where to find the resource
  - An example of a name/computer resource mapping is the mapping of machine names (such as www.ibm.com) to IP addresses provided by the Domain Name System (DNS)

---

Figure 19-5. What is a naming service?

SW2446.0

### **Notes:**

The DNS maps machine names (such as www.ibm.com) to IP addresses; IP addresses specify a location on the Internet where a machine may be found. The IP address is a reference to the resource (which is a computer).

Typically, you will only be concerned with looking up/accessing objects that have already been bound to a name; you will not usually have to do the binding yourself.

## Some naming service terms

- A *naming context* represents a set of name/object mappings (bindings)
  - Every context has a naming convention, and all name/object mappings in a context follow it
  - All names are relative to a context
- A *naming system* is a set of contexts which follow the same naming convention
  - A naming system provides services to manage and manipulate the bindings within the contexts
- A *naming service* is the product (actual code) that implements the naming system, providing a service
  - A naming service is accessible through its *interface*

Figure 19-6. Some naming service terms

SW2446.0

### **Notes:**

A naming context (also known as a context) provides a service to look up the resource associated with a specific name; it may also provide services for binding and unbinding names, as well as listing all names that are bound in the context.

An initial context is a naming context that provides a starting point for naming operations.

Using the DNS example, all machine name/IP address mappings use the DNS naming convention, which requires the parts of the DNS name to be ordered from right to left and separated by a ". ". An example naming **system** would be any system that implements the Domain Name System. An example of a naming **service** is the DNS, which offers a naming service that binds machine names to IP addresses.

## Naming subcontexts

---

- A *namespace* is the set of all names in a naming system
  - JNDI defines a hierarchical namespace for naming objects
- Within a naming context, a name can be bound to another context object
  - This is similar to a file system, where a directory can have subdirectories
  - A context object bound to a name within a context is referred to as a *subcontext*
  - In this way, resource names are hierarchical

---

Figure 19-7. Naming subcontexts

SW2446.0

### **Notes:**

A namespace represents all names in a naming system, and acts as a registry of bound objects.

A subcontext will follow the same naming conventions of the context it resides within. For example, using the DNS example, within the DNS domain ibm.com, the com DNS domain is a context, while ibm is a subcontext of com.

## How does a naming service work?

- Naming services allow you to look up an object given its name
  - Object name must adhere to a naming convention for that resource type

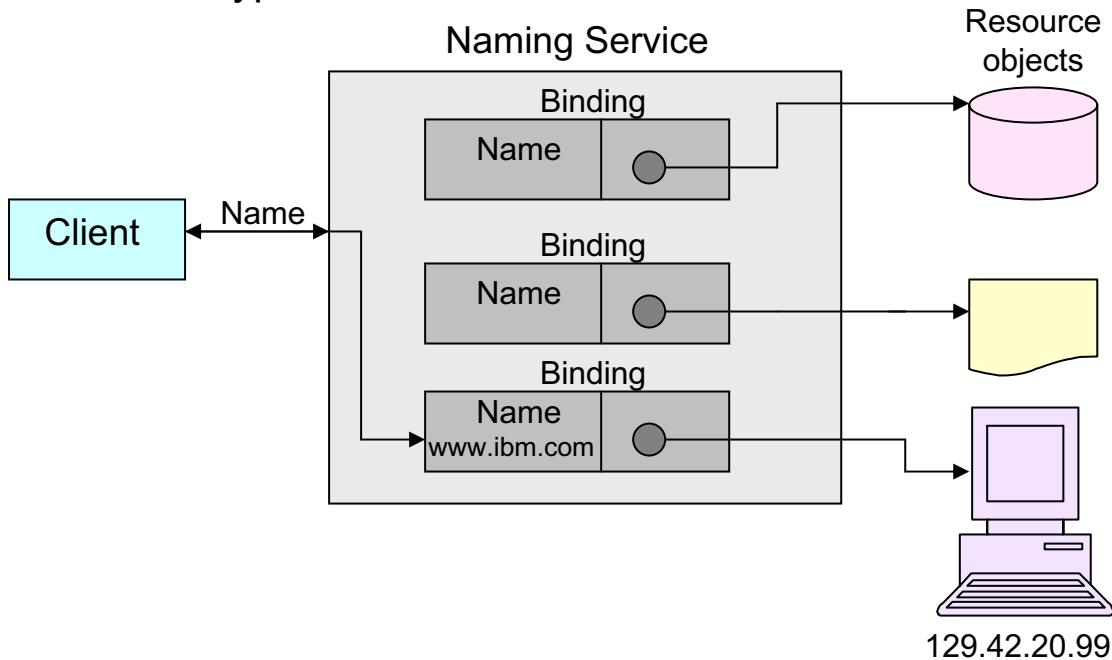


Figure 19-8. How does a naming service work?

SW2446.0

### **Notes:**

This diagram shows names that are bound to object references.

In the case of a DNS name, the name `www.ibm.com` adheres to the DNS naming convention; this name is mapped to a reference to the computer with an Internet location of `129.42.20.99`. The name is not bound to the computer object itself, but rather, the reference to this computer allows the naming service to know how to access and retrieve it.

## ***What is a directory service?***

---

- Recall the library card catalogue metaphor
  - Each card has the book name and its location on library shelves
  - Other information about the book is also included on the card
    - Author name, ISBN number, date of book publishing, and possibly other information
- Similarly, a directory service is a catalog of information about objects
  - Directory services have *entries* which associate a name with an object
  - Objects are permitted to have *attributes* to hold other information about the object
    - An object such as a printer would have information about its resolution and speed

---

Figure 19-9. What is a directory service?

SW2446.0

### **Notes:**

While a library card catalogue contains cards with information mapping a book to information about that book, such as its location, ISBN number, author name and so on, a directory service allows information to be associated with objects. The information associated with an object is called its attributes. A person object may have a name, a mailing address, and other information, such as a phone number, e-mail address and so on.

Objects in a directory often map to objects of the real world, such as a printer or a person.

Example directory products:

IBM Directory Server

Novell Directory Service

## An issue with directory services

- There are many different directory services
  - Each directory has its own interface
- A company may use different directories for different resources
- Developers must learn and support multiple interfaces

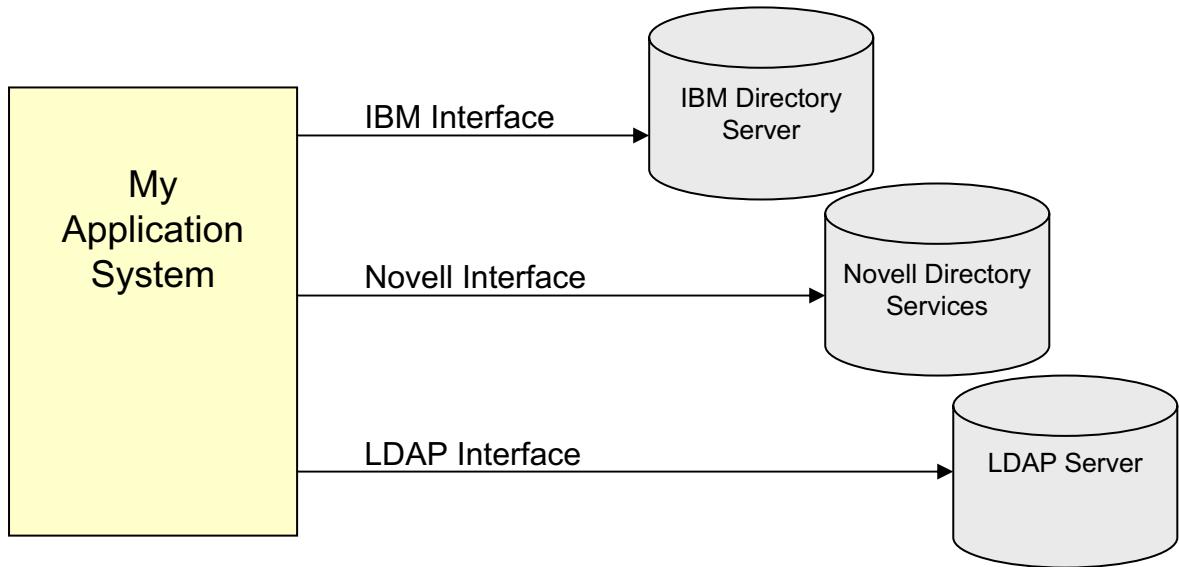


Figure 19-10. An issue with directory services

SW2446.0

### **Notes:**

## JNDI

---

- Java Naming and Directory Interface (JNDI)
  - A Java API to access naming and directory services
  - Provides a common interface to access naming and directory services
  - Also allows Java applications to store and retrieve Java objects
    - A directory containing a printer object may allow a Java program to look up a printer; a stream of data may be sent to the printer to be printed

---

Figure 19-11. JNDI

SW2446.0

### Notes:

JNDI is widely used by JDBC DataSources (discussed in the next lecture), and by Enterprise JavaBeans (EJBs).

## Why do you need JNDI? (1 of 2)

- Java applications accessing directory services would need to code to each unique interface and have code for each naming convention

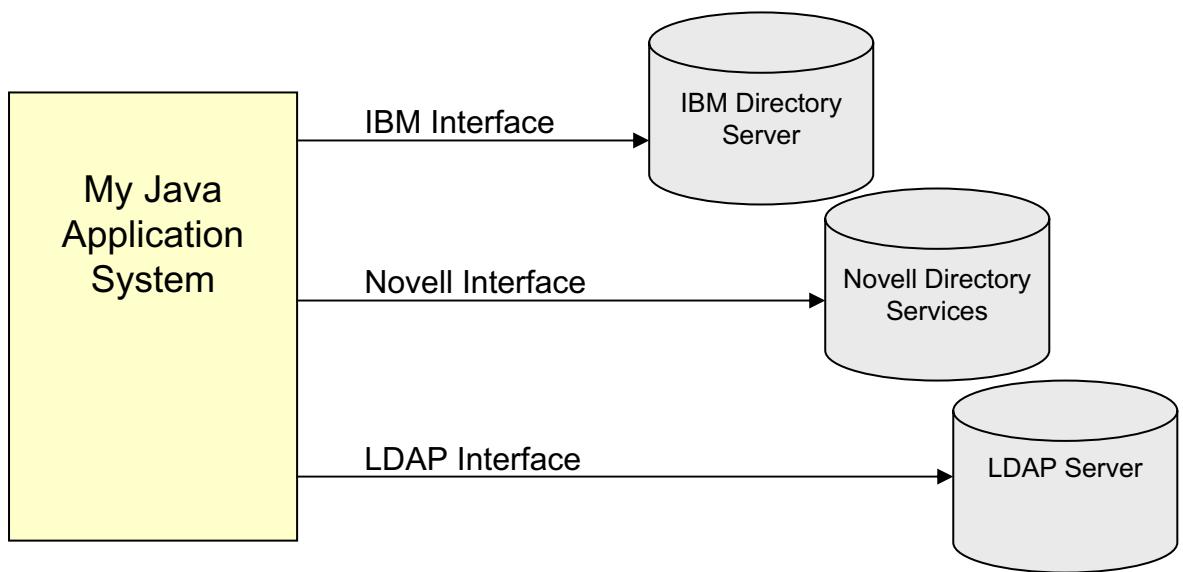


Figure 19-12. Why do you need JNDI? (1 of 2)

SW2446.0

### **Notes:**

An advantage of JNDI is that a single API works on both naming and a variety of directory services.

## Why do you need JNDI? (2 of 2)

- Java applications using JNDI to access the directory services only need to code to the JNDI interface

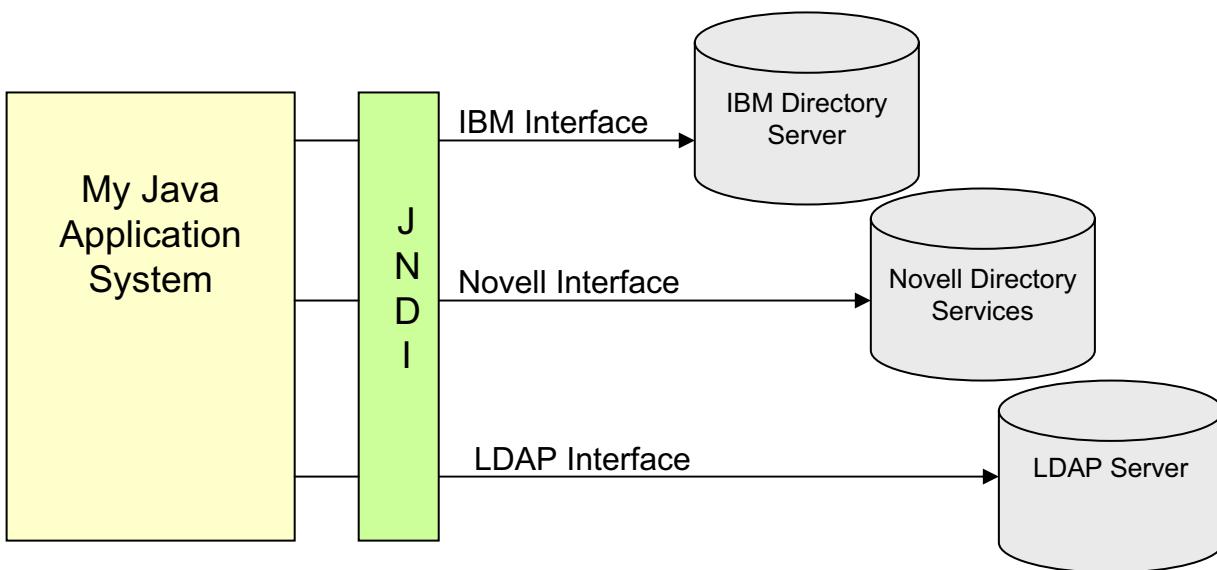


Figure 19-13. Why do you need JNDI? (2 of 2)

SW2446.0

### **Notes:**

Names are specified relative to a Context, and most Context methods accept names specified using String or Name objects.

## ***System component: JNDI***

- Java applications using JNDI to access the naming and directory services only need to code to the JNDI API
- The JNDI implementation code provides a service provider interface (SPI), which allows each naming and directory service to plug in its own service provider (SP)

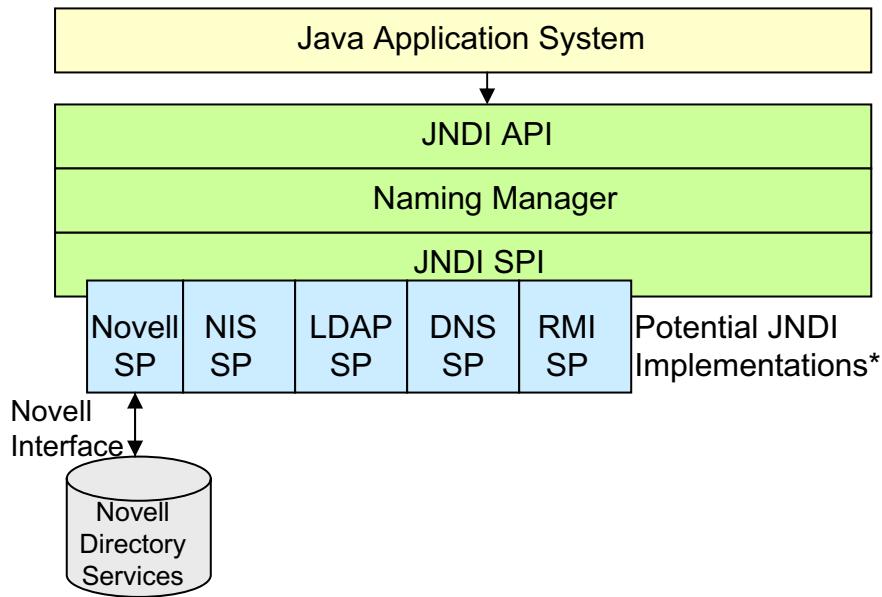


Figure 19-14. System component: JNDI

SW2446.0

### **Notes:**

\*More JNDI implementation possibilities exist.

The JNDI packages from Sun contain Service Providers for:

- LDAP
- CORBA COS
- RMI registry
- File system

You can write your own service provider, download one from the vendor, or from the JNDI Web site, <http://java.sun.com/products/jndi/serviceproviders.html>

The Service Provider Interface (SPI) outlines what service providers must provide in order for their product to plug into this architecture.

## **JNDI API: main packages**

---

- Two packages exist to allow programmers to use the functionality provided by naming and directory services
- Main packages used:
  - **javax.naming** – Naming services package
    - Contains classes and interfaces that allow you to access naming services
    - Includes Context interface, InitialContext class, lookup operations, object references and NamingException
  - **javax.naming.directory** – Directory services package
    - Adds support for directory services
    - Includes DirContext interface, InitialDirContext class, attribute manipulation operations, and search operations

---

Figure 19-15. JNDI API: main packages

SW2446.0

### **Notes:**

Additional packages, including the Event package (javax.naming.event), the LDAP package (javax.naming.ldap), and the Service Provider package (javax.naming.spi), are also included in the JNDI API.

You will typically only make use of several classes within the javax.naming package, which provides functionality to access naming services.

The Context interface in the javax.naming package provides an interface for looking up, binding/unbinding, renaming objects, and creating and destroying subcontexts. All naming and directory operations in JNDI are performed relative to a context, therefore, the InitialContext class is provided as an initial point from which naming operations can be performed.

Naming and directory services can store objects in various ways; some may store Java objects. However, Java applications may not be the only type of application that accesses the objects. The Reference class represents a reference to an object found outside of the naming and directory system, providing address information for objects outside the system. The Reference to an object also knows how to make a copy of the object, in order to return

the Java objects represented by the Reference when clients request them, creating the impression that Java objects are stored.

## JNDI naming subcontexts

---

- JNDI has many naming subcontexts
- JNDI resource names are organized in a hierarchy
  - For example, names of JDBC resources must be in the `java:comp/env/jdbc` subcontext of the context `java:comp/env`
  - An employee database may have a logical JNDI name of `employeedb`, and the resource name for this database would be `java:comp/env/jdbc/employeedb`

---

Figure 19-16. JNDI naming subcontexts

SW2446.0

### **Notes:**

## Using JNDI: naming operations (1 of 5)

- This example takes a file system directory name and a file name on the command line
- The file name is treated as a resource name, and is looked up in the namespace persisted to the directory, and the results of this lookup are output

- To start, import the packages required for naming operations:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

- Specify the directory path to be used in creating the initial context, and the file name to be looked up, retrieved from the command line arguments:

```
String localDirectory = args[0];
String name = args[1];
```

Figure 19-17. Using JNDI: naming operations (1 of 5)

SW2446.0

### **Notes:**

The file system can be made to look like a naming service; a directory represents a context, and has a set of bindings between file names and objects (file handles) that allow the system to work with the files. Further, if a directory contains a name bound to another directory object (a subdirectory), the subdirectory is called a subcontext.

The file system naming service is provided by Sun. The file system context is useful to show how to use a naming service when no application server is available, but typically when developing Java applications that need to use naming services, they will be enterprise Java applications and these will run on an application server. When running your enterprise Java application on WebSphere Application Server, you should use the JNDI service provided by WebSphere Application Server.

## ***Using JNDI: naming operations (2 of 5)***

---

- Set up the initial context environment
  - You may have to provide information to the InitialContext class in order for it to build the appropriate initial context
- If environment properties are needed to create the initial context, a Hashtable or one of its subclasses, such as the Properties class, may be used to specify this information
  - Environment properties are needed in this example
- Common environment properties are:
  - The JNDI service provider to be used, specified by the fully qualified class name of the class that should be used to create an initial context
    - The class com.sun.jndi.fscontext.RefFSContextFactory specifies that the Sun file system service provider will be used to create the initial context in this example
  - Other property values may be specified to provide needed information for the class that will be used to create the initial context
    - The file system service provider class needs to know what path to use, and this is specified by the PROVIDER\_URL property

---

Figure 19-18. Using JNDI: naming operations (2 of 5)

SW2446.0

### **Notes:**

You may not need to set up the initial context environment if you intend to use the default initial context; the environment may already be configured and you will not have to explicitly set property values to pass to the InitialContext constructor.

A java.util.Hashtable, or one of its subclasses such as the Properties class, is used to store the environment properties needed to create the initial context. The Hashtable stores information that tells the InitialContext object what class in the JNDI service provider to use to create the initial context (specified by the Context.INITIAL\_CONTEXT\_FACTORY property), and may also store other information need by the class being used to create the initial context. The name-value pairs of the Hashtable specify how the Context should be created.

In general, JNDI clients using the naming service provided by WebSphere Application Server should assume the correct environment is already configured so there is no need to explicitly set property values and pass them to the InitialContext constructor. JNDI clients running in the WebSphere runtime environment should not have to set properties because they are already set; if you need to set the service provider property from a program, you

can use the Java constant, `javax.naming.Context.INITIAL_CONTEXT_FACTORY`, specifying this property value as `com.ibm.websphere.naming.WsnInitialContextFactory` (the initial context factory for WebSphere Application Server). Though the default initial context will usually be what is needed, a JNDI client may need to access a namespace other than the one identified in its environment. In this case, the `PROVIDER_URL` property may be set to indicate a URL string that specifies configuration information for the service provider to use.

## Using JNDI: naming operations (3 of 5)

- Code to set up the environment that will be used to create the initial context:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, localDirectory);
```

---

Figure 19-19. Using JNDI: naming operations (3 of 5)

SW2446.0

### **Notes:**

In this example, the line

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");
```

sets a parameter in the Hashtable environment telling the InitialContext what class in the JNDI service provider should be used to create the initial context. This parameter indicates what service provider should be used; in this case, the file system service provider is being used, which essentially makes the file system appear as a naming service.

The line env.put(Context.PROVIDER\_URL, localDirectory); specifies the path that should be used for the creation of the initial context, in URL form. In this case, a directory in the form of a URL is specified on the command line.

The InitialContext constructor uses this Hashtable when creating the initial context. The Hashtable allows various properties to be specified, which allows the InitialContext to be tailored to a specific environment.

## Using JNDI: naming operations (4 of 5)

- Create the initial context, passing in the environment properties Hashtable that will be used to create the initial context
  - The initial context object that is returned is the entry point into the file system naming system

```
Context ctx = new InitialContext(env);
```

- Using the file name as a resource name, perform a lookup on this name in the namespace persisted to the directory, and cast the retrieved object to a File object (surround with a try/catch block to catch a NamingException if the file name is not found)

```
File file = (File)ctx.lookup(name);
```

- Print the results, then close the context:

```
System.out.println("Name: " + name + " is bound  
to file: " + file);  
ctx.close();
```

Figure 19-20. Using JNDI: naming operations (4 of 5)

SW2446.0

### Notes:

Recall that you must provide the **env** argument to the `InitialContext` constructor to use a non-default name service initial context, in this case the file system context. Also recall that when running your enterprise Java application on WebSphere Application Server, you should use the JNDI service provided by WebSphere Application Server. When accessing the default WebSphere Application Server namespace on the local WebSphere Application Server server, use the no-argument constructor of `InitialContext`.

A Hashtable is used to store the environment properties needed to create the initial context. This Hashtable is then passed to the `InitialContext` object, so that the `InitialContext` knows what JNDI provider is being used (in this case, the file system service provider), and so that other environment properties needed to create this initial context can be located.

The initial context is created, and recall that this provides the context from which all naming operations are performed. It contains a set of name-object mappings. The JNDI lookup operation is performed with a call to the `lookup` method on the `InitialContext` object. The resource name is passed to the `lookup` operation. Note that the name has to conform to the naming conventions of the service provider that you specified you were using (the name in

this case must adhere to the naming conventions of the local file system). An object is returned by the lookup operation, and because the file system is the service provider, a File object should be returned by this lookup, and the returned object is cast to a File object.

In this example, if the file name is not found, it will cause a NamingException to be thrown, and this must be handled. The root naming exception, javax.naming.NamingException, is a superclass of the exceptions thrown by the operations in the Context and DirContext interfaces.

## Using JNDI: naming operations (5 of 5)

```

public class JNDINaming {
    public static void main(String[] args) {
        // Specify a local directory as the first command-line argument
        String localDirectory = args[0];
        // Specify a file name within the directory specified
        String name = args[1];

        // Set up the environment; this specifies JNDI provider to use
        // and the initial URL to use, a directory in URL form here
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
        env.put(Context.PROVIDER_URL, localDirectory);

        try {
            // Create the Initial Context
            Context ctx = new InitialContext(env);
            // Look up an object, and cast it back to the expected type
            File file = (File)ctx.lookup(name);
            // Print the results
            System.out.println("Name: " + name + " is bound to file: " +
                file);
            // Close the context once we're finished with it
            ctx.close();
        } catch (NamingException ne) {
            // Handle the NamingException if the name can't be located
            System.err.println("Unable to look up " + name + " " + ne);
        }
    }
}

```

Figure 19-21. Using JNDI: naming operations (5 of 5)

SW2446.0

### **Notes:**

This is the resulting code for the JNDI naming services example. Note that the import statements have been excluded for simplicity.

## **Checkpoint**

---

1. What is a naming service?
2. What is a directory service?
3. What is the difference between a naming service and a directory service?
4. What is JNDI?
5. What are the benefits of using JNDI?



---

Figure 19-22. Checkpoint

SW2446.0

### **Notes:**

## Unit summary

---

- In this unit, you should have learned to:
  - Explain how and why naming services are used
  - Explain how and why directory services are used
  - Define JNDI and explain how and why it is used
  - Use JNDI to look up a resource

---

Figure 19-23. Unit summary

SW2446.0

### **Notes:**



# Unit 20. DataSources

## What this unit is about

This unit introduces the rationale for connection pooling, and explains how DataSources implement connection pooling and may be accessed from Java programs using JNDI.

## What you should be able to do

After completing this unit, you should be able to:

- Explain the rationale for connection pooling
- Describe what a connection pool provides
- Describe DataSources as the implementation of connection pooling
- Outline the benefits of using DataSources to connect to a database
- Write code to access a DataSource using JNDI

## How you will check your progress

Accountability:

- Checkpoint questions



## 20.1 DataSources

## Unit objectives

---

- After completing this unit, you should be able to:
  - Explain the rationale for connection pooling
  - Describe what a connection pool provides
  - Describe DataSources as the implementation of connection pooling
  - Outline the benefits of using DataSources to connect to a database, in comparison to using the DriverManager to obtain database connections
  - Write code to access a DataSource

---

Figure 20-1. Unit objectives

SW2446.0

### **Notes:**

## **Rationale for connection pooling**

---

- Opening a database connection is the most expensive and time-consuming operation in a database transaction
- Web-driven database interactions are typically very short in duration
- For performance and scalability, you want to:
  - Minimize database connection setup and tear-down
  - Cap the maximum number of active sessions
- Connection pooling addresses these issues

---

Figure 20-2. Rationale for connection pooling

SW2446.0

### **Notes:**

There are usually only a small number of users performing a database transaction compared to the number of active users at a particular point in time, and database interactions are typically short-lived. Obtaining a database connection can take much longer than completing the database transaction, so it is beneficial to have a mechanism which allows a pre-existing set of active database connections to be reused. This allows the overhead of obtaining a new database connection to be done once while the connection is used many times, and would cap the number of simultaneous database connections.

## **What is connection pooling?**

---

- A connection pool is a managed group of open database connections that are available for use and reuse
  - A group of JDBC Connection objects are shared among those requiring access to a particular database, and are managed by the connection pool
- A database connection is obtained from this pool when needed, and then returned to the pool when it is no longer needed so that the connection can be reused
- A connection pool:
  - Maintains a number of persistent connections to the database
  - Minimizes connection overhead
  - Spreads out connection cost over repeated uses

---

Figure 20-3. What is connection pooling?

SW2446.0

### **Notes:**

Recall that JDBC allows applications to access any tabular data source, typically a relational database.

## ***Connection pooling in JDBC 3.0***

---

- JDBC 3.0 supports connection pooling through the classes and interfaces in the javax.sql package
  - The DataSource interface represents a physical data source
- Using the DataSource interface provided by the javax.sql package is the preferred way of connecting to a relational data source
  - When using a DataSource, a Connection object is obtained from a DataSource class rather than the DriverManager class

---

Figure 20-4. Connection pooling in JDBC 3.0

SW2446.0

### **Notes:**

## DataSource implementation

---

- The DataSource interface is implemented by driver or application server vendors
- One type of implementation provides for connection pooling
- DataSource objects of the connection pooling implementation work with middleware, such as application servers, to support connection pooling

---

Figure 20-5. DataSource implementation

SW2446.0

### **Notes:**

The DataSource interface is implemented by a driver vendor and may be implemented in several different ways. One implementation of the DataSource interface is the connection pooling implementation, and this implementation provides Connection objects that are involved in connection pooling. The connection pooling implementation is the DataSource implementation that this unit examines.

Two other implementations of the DataSource interface exist that will not be discussed in this unit: the basic implementation, which provides the same connection as when the DriverManager class is used to create a connection, and the distributed transaction implementation, which provides Connection objects that can be used in distributed transactions, and are typically also involved in connection pooling.

Applications use the DataSource API, and the connection pooling is done transparently and internally by middleware that manages the connection pool. Other classes are included in the JDBC 3.0 javax.sql package and facilitate connection pooling, but these classes aren't seen or used by the client requiring a data source connection; instead, they are used by middleware to provide for connection pooling.

## Using the DriverManager

- The DriverManager loads the JDBC driver for a specific database
- A single connection to a database is created
- SQL statements (queries/updates) are executed
- The Connection is destroyed
- Connection objects are created, used, and destroyed as needed

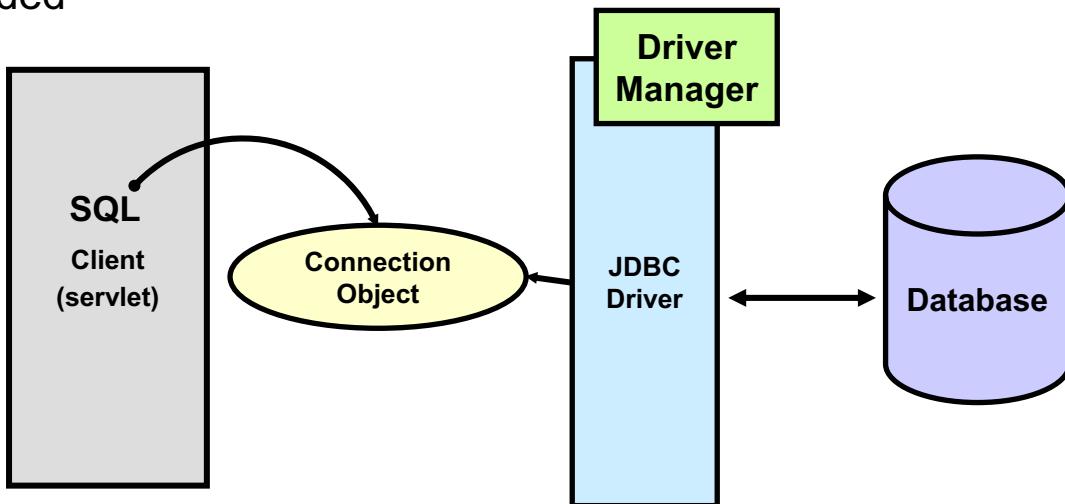


Figure 20-6. Using the DriverManager

SW2446.0

### **Notes:**

When connecting to a database using the DriverManager, your code first tells the DriverManager to load the JDBC driver for a specific database. Next, you ask the DriverManager to create a connection to that database, specifying the database URL, user name and password (if necessary). This information is hardcoded into your Java class, and if the database or its properties change, your code would have to reflect changes to this information.

## Using a DataSource

- Look up the DataSource using JNDI
- Get a Connection object from the DataSource
- Execute SQL (queries/updates) using the Connection object
- Close the Connection (return it to the pool)
- Acquire and return connections from the connection pool as needed

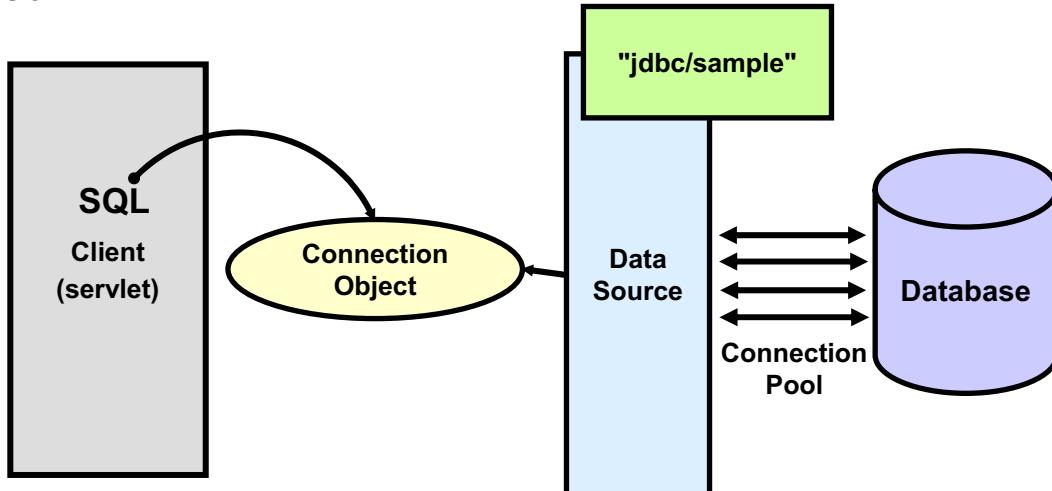


Figure 20-7. Using a DataSource

SW2446.0

### Notes:

A database driver that is accessed through a DataSource object is not registered with the DriverManager; instead, a JNDI lookup is performed on the name of the DataSource, and the DataSource is used to obtain a Connection object. This diagram depicts a DataSource object of the connection pooling implementation, and thus, the Connection object obtained is involved in connection pooling.

One of the major advantages to using a DataSource object to retrieve connections to a relational data source is that the data source's properties may change without affecting the code that accesses and uses it, unlike the approach that uses the DriverManager. The driver class name is never specified or loaded by the DriverManager in your code.

Typically, the Connection object is returned to the connection pool when it will not be used again in the near future, allowing it to be reused by others needing a connection to that data source.

## **How does a *DataSource* work? (1 of 2)**

- Driver or application server vendors supply the implementation for the *DataSource* class
  - A *DataSource* object represents a specific physical data source
  - The *DataSource* object implemented with connection pooling works with middleware to provide connection pooling services
  - A Connection obtained from a *DataSource* object is an active connection to the specific physical data source represented by the *DataSource* object
  
- A logical name is bound to the *DataSource* object by a naming service
  - The *DataSource* may be retrieved by an application through a JNDI lookup for the logical name associated with the *DataSource*

---

Figure 20-8. How does a *DataSource* work? (1 of 2)

SW2446.0

### **Notes:**

Typically, a system administrator will bind the logical name to the *DataSource*; clients using a *DataSource* will only contain code to look up a *DataSource*, then obtain and use a database connection provided by the *DataSource*.

A *DataSource* object is looked up and retrieved using JNDI; code to look up a *DataSource* will be shown in later slides.

## ***How does a DataSource work? (2 of 2)***

---

- Once the DataSource object has been retrieved, you may obtain a Connection to the data source
- Use the Connection to execute SQL statements, and when the Connection will not be needed again in the near future, return it to the connection pool by using the Connection.close() method
  - The data source Connection will not be closed, but instead will be returned to the pool
  - Closing the Connection after use is important to ensure that other users can use it

---

Figure 20-9. How does a DataSource work? (2 of 2)

SW2446.0

### **Notes:**

The Connection object that is returned by the DataSource.getConnection() method is actually a handle to a data source connection, instead of being a physical connection; this is transparent to the client. Calling the Connection.close() method on this Connection object allows the middleware (which handles the connection pool) to invalidate the handle to that Connection and return the Connection to its pool.

## Example: coding for DataSources

- Create the initial naming context
  - This sample data source is configured on a WebSphere Application Server
  - The WebSphere Application Server initial context factory class, com.ibm.websphere.naming.WsnInitialContextFactory, is typically used by WebSphere Application Server applications to perform JNDI operations

```
Hashtable parms = new Hashtable();
parms.put(Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.websphere.naming.WsnInitialContextFactory");
Context ctx = new InitialContext(parms);
```

- Use the context to perform a JNDI lookup, retrieving a DataSource object

```
DataSource ds =
(DataSource)ctx.lookup("jdbc/sample");
```

Figure 20-10. Example: coding for DataSources

SW2446.0

### **Notes:**

Before a DataSource can be retrieved, an administrator must create and configure a data source on the middleware, such as an application server. In this course, a data source configured on WebSphere Application Server V6.0 is used; later slides provide information on configuring a data source on WebSphere Application Server V6.0.

These two snippets of code that create the initial naming context and obtain the DataSource object typically appear in the init() method of a servlet or in the constructor for a Bean.

When creating the initial naming context for use with a data source configured on WebSphere Application Server, you may choose not to specify the initial context factory and use the no-argument constructor of the InitialContext instead, if you intend to use the default initial naming context:

```
Context initialContext = new InitialContext();
```

The WebSphere Application Server runtime environment is set up to use this WebSphere Application Server initial context factory if one is not specified explicitly by the JNDI client.

## Obtaining a DataSource connection

- Get a connection

```
con = ds.getConnection("userid", "password");
```

- Perform normal SQL processing

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sqlString);
```

- Close the connection (return it to the connection pool)

- Done in a `finally` clause to ensure execution in the face of errors

```
con.close();
```

---

Figure 20-11. Obtaining a DataSource connection

SW2446.0

### **Notes:**

Typically, a database connection is obtained, SQL processing is done, and the connection is returned to the connection pool (`Connection.close()` is called) in a servlet's `doGet/doPost` method, or in a Bean's business logic method.

Recall that the `Connection` object that is returned by the `DataSource.getConnection()` method is actually a handle to a database connection, instead of being a physical connection; thus, calling the `Connection.close()` method on this `Connection` object allows the middleware to return the `Connection` to its pool, rather than closing the database connection.

## DataSource example

```

Context ctx = null;
try {
    Hashtable params = new Hashtable();
    params.put(Context.INITIAL_CONTEXT_FACTORY,
               "com.ibm.websphere.naming.WsnInitialContextFactory");
    ctx = new InitialContext(params);
} catch (NamingException ne) {    System.out.println(ne);
}
// Define an SQL statement ...
try {
    DataSource ds = (DataSource)ctx.lookup("jdbc/sample");
    con = ds.getConnection();

    stmt = con.createStatement();
    rs = stmt.executeQuery(sqlString);
} catch (SQLException sqle) {
System.out.println(sqle);
} catch (NamingException ne) {    System.out.println(ne);
} finally {
    // Close the ResultSet and the Statement first ...
    if(con != null){
        try {
            con.close();
        } catch (SQLException e) {
System.out.println(e);
        }
    }
}
}

```

Figure 20-12. DataSource example

SW2446.0

### **Notes:**

Code that relates to typical SQL processing has been omitted, and is denoted with a comment that ends in “...”.

In this example, the initial context is created using the com.ibm.websphere.naming.WsnInitialContextFactory class. Once the InitialContext has been created, a JNDI lookup is performed to find DataSource mapped to the name jdbc/sample. Once the DataSource has been retrieved, a Connection is obtained from it. An SQL query is run on the data source, and then possible exceptions resulting from these operations are caught. The Connection to the data source is returned to the pool using the Connection.close() call, in the finally clause.

## DataSource exceptions

- An application server will manage the connection pool, and may throw different exceptions to allow you to recover from specific problems
  - ConnectionWaitTimeoutException
  - StaleConnectionException
- These exceptions allow you to write appropriate exception-handling code
  - Usually involves re-acquiring the database connection

---

Figure 20-13. DataSource exceptions

SW2446.0

### **Notes:**

WebSphere Application Server maps several database-specific SQL exceptions to WebSphere-specific exceptions to ease development; instead of requiring you to know all of the database-specific SQL exceptions that could be thrown in typical situations, you must be familiar with several WebSphere-specific exceptions.

## **Checkpoint**

1. What is a connection pool?
2. Name two benefits of using connection pooling.
3. What class in the javax.sql package implements connection pooling and what is a benefit of using it?
4. Name the steps to access a DataSource, and use it to get a connection to a database.
5. How do you create a data source so that your program can access it?



Figure 20-14. Checkpoint

SW2446.0

### **Notes:**

## ***Unit summary***

---

- In this unit, you should have learned:
  - The rationale for connection pooling
  - What a connection pool provides
  - That DataSources implement connection pooling
  - The benefits of using DataSources to connect to a database, in comparison to using the DriverManager to obtain database connections
  - How to write code to access a DataSource

---

Figure 20-15. Unit summary

SW2446.0

### **Notes:**

See the information in the appendix section of this unit for more information on support for data sources in Application Developer, and for information on how to create and configure a data source and connection pool.

## Appendix: data sources support

- Data source must be set up as a resource on the application server in order for you to access it.

– Use the WebSphere Application Server V6.0 administrative console

- To define a new data source, click the **Data sources** link after configuring the JDBC provider

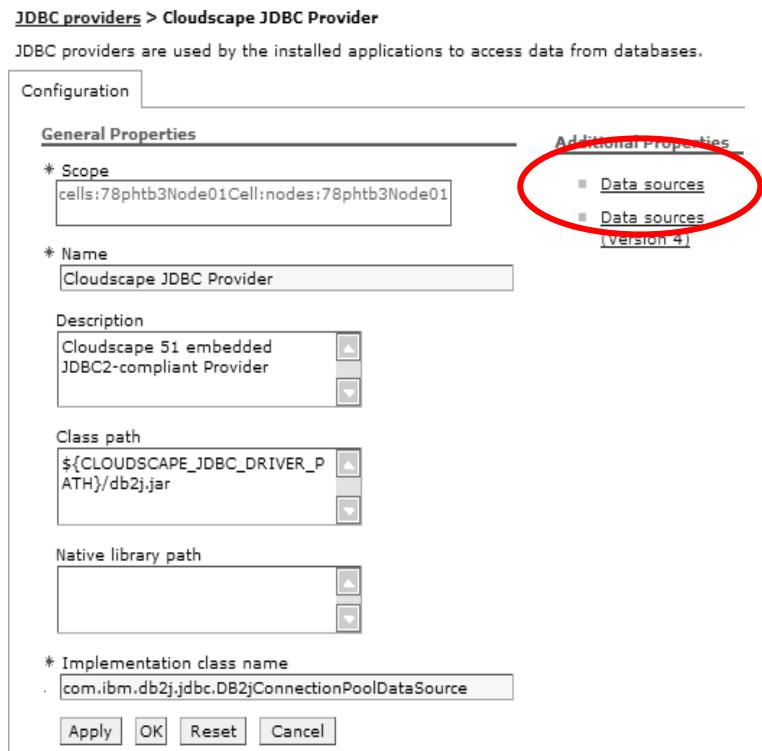


Figure 20-16. Appendix: data sources support

SW2446.0

### Notes:

In order to use a data source, one must first be configured on WebSphere Application Server V6.0. This will typically be done by the administrator of an application server, but it is useful to create a data source during development for testing purposes. To create and configure a data source, you must use the WebSphere Application Server administrative console, which is accessible from within Application Developer.

To configure a data source, first start the server from the Servers view by right-clicking the server and choosing **Start**. Once the server has started, right-click this server and select **Run administrative console**. After the administrative console has opened, log in. Expand **Resources** on the left side, and select **JDBC Providers**. Specify the properties, including implementation class name (which is the Java class name for your JDBC driver implementation), for your JDBC provider.

## Appendix: creating a new data source

---

- Click the **New** button to create a new data source

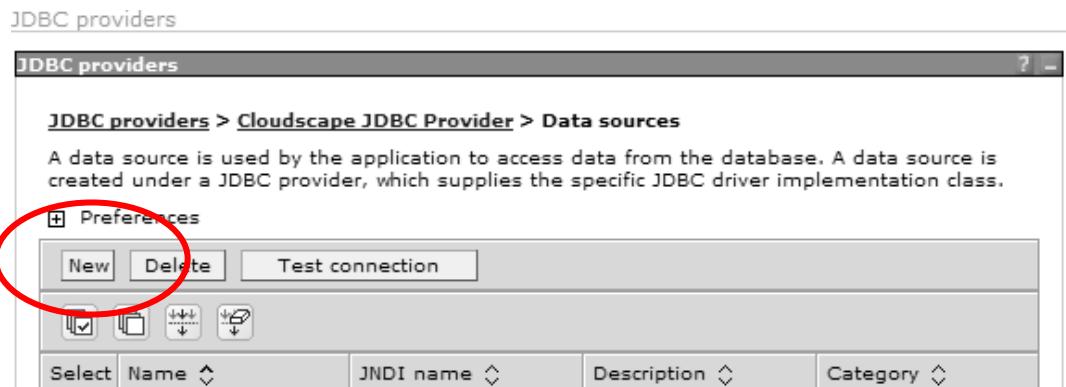


Figure 20-17. Appendix: creating a new data source

SW2446.0

### **Notes:**

This screen will list all the data sources which use a particular JDBC driver implementation class.

## Appendix: configuring a data source

- Specify the properties for this data source
- Click the **Connection pool properties** link to specify information regarding the connection pool

JDBC providers > Cloudscape JDBC Provider > Data sources > Cloudscape JDBC Driver DataSource

A data source is used by the application to access data from the database. A data source is created under : JDBC provider, which supplies the specific JDBC driver implementation class.

Configuration

General Properties

- \* Scope: cells:78phbt3Node01Cell:nodes:78phbt3Node01
- \* Name: Cloudscape JDBC Driver DataSource
- JNDI name: jdbc/sample
- Use this Data Source in container managed persistence (CMP)
- Description: New JDBC Datasource
- Category:
- Data store helper class name:
  - Select a data store helper class: Data store helper classes provided by WebSphere Application Server
    - Cloudscape data store helper (com.ibm.websphere.rssadapter.CloudscapeDataStoreHelper)
  - Specify a user-defined data store helper: Enter a package-qualified data store helper class name

Additional Properties

- Connection pool properties** (highlighted with a red circle)
- WebSphere Application Server data source properties
- Custom properties

Related Items

- J2EE Connector Architecture (J2C) authentication data entries

...continued

Component-managed authentication alias

Container-managed authentication

Cloudscape data source properties

- \* Database name: C:\labfiles60\Cloudscape-DB

Apply OK Reset Cancel

Figure 20-18. Appendix: configuring a data source

SW2446.0

### Notes:

Configure the properties for your data source in this screen, including the display name for this data source resource, the JNDI name which will be used to look up this resource, and the name of the database to which this data source connects.

Links under the **Additional Properties** heading allow you to specify properties related to this data source.

## Appendix: configuring a connection pool

- Specify the properties for the connection pool

JDBC providers > Cloudscape JDBC Provider > Data sources > Cloudscape JDBC Driver  
 DataSource > Connection pools

Connection pool properties that can be modified to change the behavior of the J2C connection pool manager. Default values are provided for non-production use. Review and possible modification of these configuration values are recommended.

Configuration

<b>General Properties</b>		<b>Additional Properties</b>
<b>Scope</b> <input type="text" value="cells:78phtb3Node01Cell:nodes:78phtb3Node01"/>		<ul style="list-style-type: none"> <li>■ <a href="#">Advanced connection pool properties</a></li> <li>■ <a href="#">Connection pool custom properties</a></li> </ul>
Connection timeout	<input type="text" value="180"/>	seconds
Maximum connections	<input type="text" value="10"/>	connections
Minimum connections	<input type="text" value="1"/>	connections
Reap time	<input type="text" value="180"/>	seconds
Unused timeout	<input type="text" value="1800"/>	seconds
Aged timeout	<input type="text" value="0"/>	seconds
Purge policy	<input type="button" value="EntirePool"/>	
<input type="button" value="Apply"/> <input type="button" value="OK"/> <input type="button" value="Reset"/> <input type="button" value="Cancel"/>		

Figure 20-19. Appendix: configuring a connection pool

SW2446.0

### Notes:

Configure the properties for the connection pool associated with this data source in this screen. Specify the connection timeout (interval after which a connection request times out), maximum number of connections that can be created in this pool, the minimum number of connections that must be maintained in this pool, and other information. Hover over the field name to view information on what a field specifies, or click inside a field's text box to have this information appear in the Help section on the right of the screen.

Links under the **Additional Properties** heading allow you to specify properties related to this connection pool.

Once your data source and its connection pool have been configured, save the changes you've made and close the administrative console.

## Configuring project resources

- Configure your project to refer to the data source resource that you have created

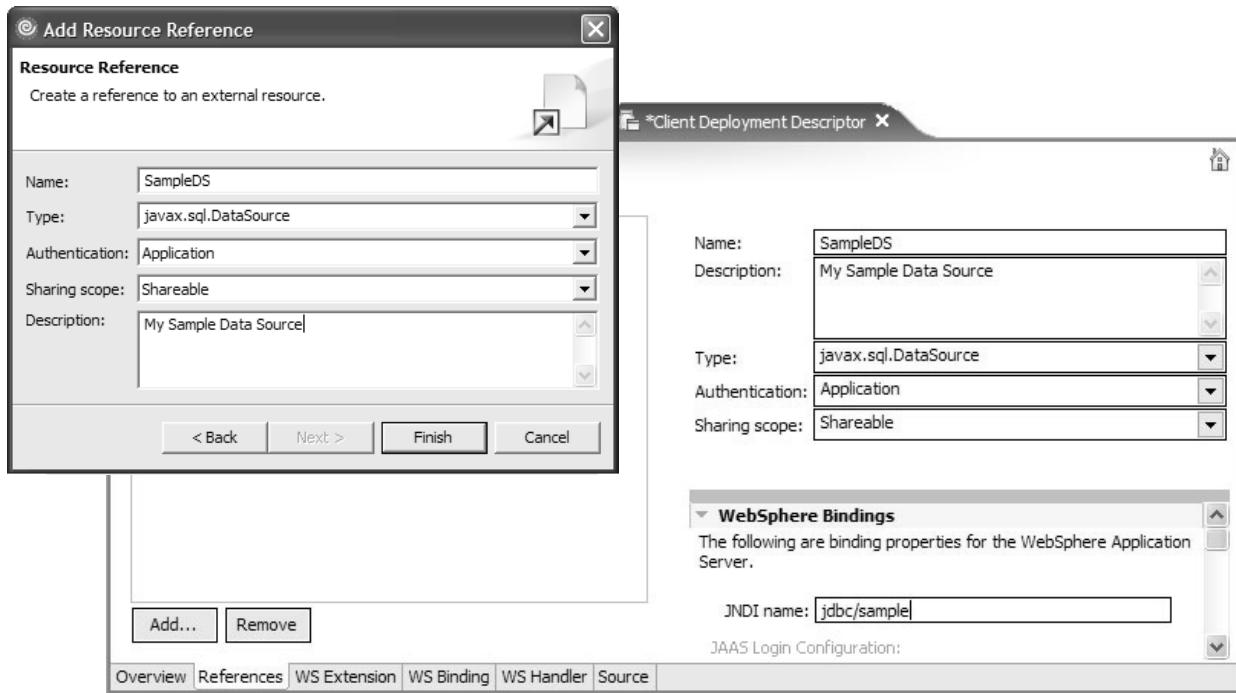


Figure 20-20. Configuring project resources

SW2446.0

### Notes:

A deployment descriptor is a special file that contains information needed to properly deploy your project to a server. Here, the deployment descriptor is a file called application-client.xml, which can be accessed by clicking a file called “Deployment Descriptor” in your project (this file is a shortcut to the XML file).

Click the **References** tab in the Deployment Descriptor editor, and click the **Add...** button to add a resource reference to your project. Choose to add a **Resource Reference** to your project. Fill in the information for your data source. When this is complete, add the **JNDI name** for this resource under the WebSphere Bindings section of the editor.



# Unit 21. Wrap-Up

## What this unit is about

This unit concludes the course and provides an outline of resources that may be appropriate for those wishing to study course topics in more detail.

## What you should be able to do

After completing this unit, you should be able to:

- Give an overview of the topics covered by this course
- List Web sites and books that provide more information on the topics discussed in this course



## 21.1 Wrap-Up

## ***What you have seen***

---

- Object-oriented programming
  - Concepts and design principles
- Java programming language
  - Basic syntax
  - Classes and primitive types
  - Inheritance
  - Interfaces
  - Refactoring
  - Threads
  - Collections
  - JavaBeans
  - Serialization
  - Exceptions
  - Coding for databases: JDBC, JNDI, and DataSources
- IBM Rational Application Developer V6
  - Workbench, Java tooling, Debug perspective

---

Figure 21-1. What you have seen

SW2446.0

### **Notes:**

## **Resources**

---

- IBM Web sites
  - <http://www.ibm.com/WebSphere>
  - <http://www.ibm.com/developer> (developerWorks)
  - <http://www.redbooks.ibm.com/>
- Sun Web sites
  - <http://java.sun.com/j2se/> (Standard Edition)
  - <http://java.sun.com/j2ee/> (Enterprise Edition)
  - <http://java.sun.com/j2se/1.4.2/docs/api/> (JavaDoc)
- Kyle Brown's public Web site
  - Lots of information on various topics
  - <http://hometown.aol.com/kgb1001001/index.html>
- Java Wiki
  - Community-modifiable Web site
  - <http://www.c2.com/cgi/wiki?TopicJava>
    - Get started by clicking **LikePages** or **FindPage** and then typing *Java* as a search parameter

---

Figure 21-2. Resources

SW2446.0

### **Notes:**

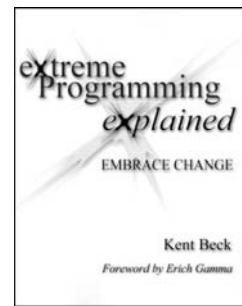
This list is the same as the one you saw at the end of Unit 2.

## Advanced reading

---

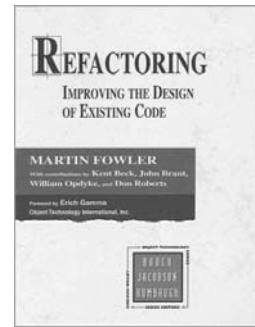
- *Extreme Programming Explained: Embrace Change*

– by Kent Beck  
– Addison-Wesley Publishing  
– ISBN: 0201616416



- *Refactoring : Improving the Design of Existing Code*

– by Martin Fowler  
– Addison-Wesley Publishing  
– ISBN: 0201485672



---

Figure 21-3. Advanced reading

SW2446.0

### **Notes:**

## More advanced reading

- *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*
  - by William Brown,  
Raphael C. Malveau,  
Hays W. McCormick III,  
Thomas J. Mowbray
  - John Wiley & Sons
  - ISBN: 0471197130

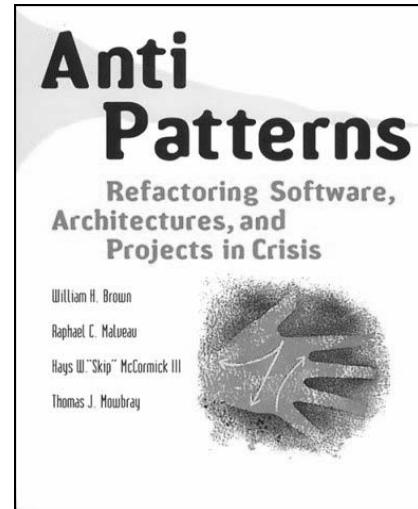


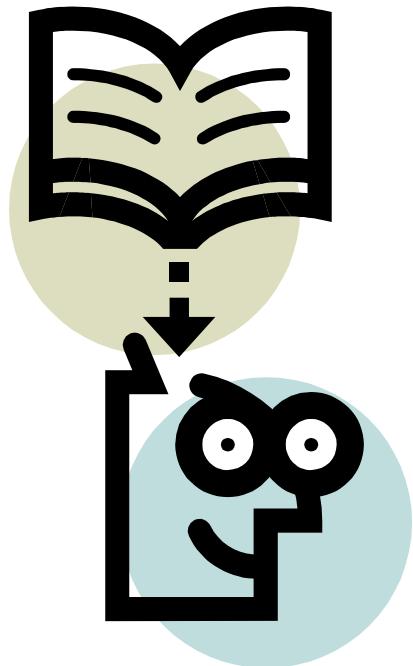
Figure 21-4. More advanced reading

SW2446.0

### **Notes:**

## Goodbye

---



---

Figure 21-5. Goodbye

SW2446.0

### Notes:



**IBM**<sup>®</sup>