



Integrated Cloud Applications & Platform Services



Java SE 8 Programming

Student Guide - Volume II

D84838GC10

Edition 1.0 | December 2016 | D87758

Learn more from Oracle University at education.oracle.com

ORACLE®

Authors

Anjana Shenoy
Michael Williams
Tom McGinn
Peter Fernandez

Technical Contributors and Reviewers

Pete Daly
Sravanti Tatiraju
Nick Ristuccia
Stuart Marks
Hiroshi Hiraga
Peter Hall
Matthew Slingsby
Marcus Hirt
Irene Rusman
Joanne Sun
Marilyn Beck
Joe A Boulenouar

Editors

Aju Kumar
Malavika Jinka
Arijit Ghosh
Anwesha Ray

Graphic Designer

Divya Thallap

Publishers

Giri Venugopal
Michael Sebastian
Veena Narasimhan

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

1 Introduction

- Course Goals 1-2
- Course Objectives 1-3
- Audience 1-5
- Prerequisites 1-6
- Class Introductions 1-7
- Course Environment 1-8
- Java Programs Are Platform-Independent 1-9
- Java Technology Product Groups 1-10
- Java SE Platform Versions 1-11
- Downloading and Installing the JDK 1-12
- Java in Server Environments 1-13
- The Internet of Things 1-14
- The Java Community 1-15
- The Java Community Process (JCP) 1-16
- OpenJDK 1-17
- Oracle Java SE Support 1-18
- Additional Resources 1-19
- Summary 1-20

2 Java Syntax and Class Review

- Objectives 2-2
- Java Language Review 2-3
- Java Class Structure 2-4
- A Simple Class 2-5
- Java Naming Conventions 2-6
- How to Compile and Run 2-7
- How to Compile and Run: Example 2-8
- Code Blocks 2-9
- Primitive Data Types 2-10
- Numeric Literals 2-11
- Operators 2-12
- Logical Operators 2-13
- if else Statement 2-14
- switch Statement 2-15

while Loop	2-16
do-while Loop	2-17
for Loop	2-18
Arrays and for-each Loop	2-19
Strings	2-20
String Operations: StringBuilder	2-21
A Simple Java Class: Employee	2-22
Methods	2-23
Creating an Instance of a Class	2-24
Constructors	2-25
package Statement	2-26
import Statements	2-27
Java Is Pass-By-Value	2-29
Pass-By-Value for Object References	2-30
Objects Passed as Parameters	2-31
Garbage Collection	2-32
Summary	2-33
Practice 2-1 Overview: Creating Java Classes	2-34
Quiz	2-35

3 Encapsulation and Subclassing

Objectives	3-2
Encapsulation	3-3
Encapsulation: Example	3-4
Encapsulation: Public and Private Access Modifiers	3-5
Encapsulation: Private Data, Public Methods	3-6
Employee Class Refined	3-7
Make Classes as Immutable as Possible	3-8
Method Naming: Best Practices	3-9
Encapsulation: Benefits	3-10
Creating Subclasses	3-11
Subclassing	3-12
Manager Subclass	3-13
Constructors in Subclasses	3-14
Using super	3-15
Constructing a Manager Object	3-16
Overloading Methods	3-17
Overloaded Constructors	3-18
Overloaded Constructors: Example	3-19
Single Inheritance	3-20
Summary	3-21

Practice 3-1 Overview: Creating Subclasses 3-22
Quiz 3-23

4 Overriding Methods, Polymorphism, and Static Classes

Objectives 4-2

Using Access Control 4-3

Protected Access Control: Example 4-4

Access Control: Good Practice 4-5

Overriding Methods 4-6

Invoking an Overridden Method 4-8

Virtual Method Invocation 4-9

Accessibility of Overriding Methods 4-10

Applying Polymorphism 4-11

Using the instanceof Keyword 4-13

Overriding Object methods 4-14

Object toString Method 4-15

Object equals Method 4-16

Overriding equals in Employee 4-17

Overriding Object hashCode 4-18

Methods Using Variable Arguments 4-19

Casting Object References 4-21

Upward Casting Rules 4-22

Downward Casting Rules 4-23

static Keyword 4-24

Static Methods 4-25

Using Static Variables and Methods: Example 4-26

Implementing Static Methods 4-27

Calling Static Methods 4-28

Static Variables 4-29

Defining Static Variables 4-30

Using Static Variables 4-31

Static Initializers 4-32

Static Imports 4-33

Design Patterns 4-34

Singleton Pattern 4-35

Singleton: Example 4-36

Immutable Classes 4-37

Example: Creating Immutable class in Java 4-38

Summary 4-39

Practice 4-1 Overview: Overriding Methods and Applying Polymorphism 4-40

Practice 4-2 Overview: Overriding Methods and Applying Polymorphism 4-41

Practice 4-3 Overview: Applying the Singleton Design Pattern 4-42
Quiz 4-43

5 Abstract and Nested Classes

Objectives 5-2
Modeling Business Problems with Classes 5-3
Enabling Generalization 5-4
Identifying the Need for Abstract Classes 5-5
Defining Abstract Classes 5-6
Defining Abstract Methods 5-7
Validating Abstract Classes 5-8
Final Methods 5-9
Final Classes 5-10
Final Variables 5-11
Declaring Final Variables 5-12
Nested Classes 5-13
Example: Member Class 5-14
Enumerations 5-15
Enum Usage 5-16
Complex Enums 5-17
Summary 5-19
Practice 5-1 Overview: Applying the Abstract Keyword 5-20
Practice 5-2 Overview: Using Inner Class As a Helper Class 5-21
Practice 5-3 Overview: Using Java Enumerations 5-22
Quiz 5-23

6 Interfaces and Lambda Expressions

Objectives 6-2
Java Interfaces 6-3
A Problem Solved by Interfaces 6-4
CrushedRock Class 6-5
The SalesCalcs Interface 6-6
Adding an Interface 6-7
Interface References 6-8
Interface Reference Usefulness 6-9
Interface Code Flexibility 6-10
default Methods in Interfaces 6-11
default Method: Example 6-12
static Methods in Interfaces 6-13
Constant Fields 6-14
Extending Interfaces 6-15

Implementing and Extending	6-16
Anonymous Inner Classes	6-17
Anonymous Inner Class: Example	6-18
String Analysis Regular Class	6-19
String Analysis Regular Test Class	6-20
String Analysis Interface: Example	6-21
String Analyzer Interface Test Class	6-22
Encapsulate the for Loop	6-23
String Analysis Test Class with Helper Method	6-24
String Analysis Anonymous Inner Class	6-25
String Analysis Lambda Expression	6-26
Lambda Expression Defined	6-27
What Is a Lambda Expression?	6-28
Lambda Expression Shorthand	6-31
Lambda Expressions as Variables	6-32
Summary	6-33
Practice 6-1: Implementing an Interface	6-34
Practice 6-2: Using Java Interfaces	6-35
Practice 6-3: Creating Lambda Expression	6-36
Quiz	6-37

7 Generics and Collections

Objectives	7-2
Topics	7-3
Generics	7-4
Simple Cache Class Without Generics	7-5
Generic Cache Class	7-6
Generics in Action	7-7
Generics with Type Inference Diamond	7-8
Collections	7-9
Collection Types	7-10
Collection Interfaces and Implementation	7-11
List Interface	7-12
ArrayList	7-13
Autoboxing and Unboxing	7-14
ArrayList Without Generics	7-15
Generic ArrayList	7-16
Generic ArrayList: Iteration and Boxing	7-17
Set Interface	7-18
TreeSet: Implementation of Set	7-19
Map Interface	7-20

Map Types	7-21
TreeMap: Implementation of Map	7-22
Deque Interface	7-23
Stack with Deque: Example	7-24
Ordering Collections	7-25
Comparable: Example	7-26
Comparable Test: Example	7-27
Comparator Interface	7-28
Comparator: Example	7-29
Comparator Test: Example	7-30
Summary	7-31
Practice 7-1 Overview: Counting Part Numbers by Using a HashMap	7-32
Practice 7-2 Overview: Implementing Stack by Using a Deque Object	7-33
Quiz	7-34

8 Collections, Streams, and Filters

Objectives	8-2
Collections, Streams, and Filters	8-3
The Person Class	8-4
Person Properties	8-5
Builder Pattern	8-6
Collection Iteration and Lambdas	8-7
RoboCallTest07: Stream and Filter	8-8
RobocallTest08: Stream and Filter Again	8-9
SalesTxn Class	8-10
Java Streams	8-11
The Filter Method	8-12
Method References	8-13
Method Chaining	8-14
Pipeline Defined	8-16
Summary	8-17
Practice Overview	8-18

9 Lambda Built-in Functional Interfaces

Objectives	9-2
Built-in Functional Interfaces	9-3
The java.util.function Package	9-4
Example Assumptions	9-5
Predicate	9-6
Predicate: Example	9-7
Consumer	9-8

Consumer: Example 9-9
Function 9-10
Function: Example 9-11
Supplier 9-12
Supplier: Example 9-13
Primitive Interface 9-14
Return a Primitive Type 9-15
Return a Primitive Type: Example 9-16
Process a Primitive Type 9-17
Process Primitive Type: Example 9-18
Binary Types 9-19
Binary Type: Example 9-20
Unary Operator 9-21
UnaryOperator: Example 9-22
Wildcard Generics Review 9-23
Summary 9-24
Practice Overview 9-25

10 Lambda Operations

Objectives 10-2
Streams API 10-3
Types of Operations 10-4
Extracting Data with Map 10-5
Taking a Peek 10-6
Search Methods: Overview 10-7
Search Methods 10-8
Optional Class 10-9
Lazy Operations 10-10
Stream Data Methods 10-11
Performing Calculations 10-12
Sorting 10-13
Comparator Updates 10-14
Saving Data from a Stream 10-15
Collectors Class 10-16
Quick Streams with Stream.of 10-17
Flatten Data with flatMap 10-18
Summary 10-19
Practice Overview 10-20

11 Exceptions and Assertions

Objectives 11-2
Error Handling 11-3
Exception Handling in Java 11-4
try-catch Statement 11-5
Exception Objects 11-6
Exception Categories 11-7
Handling Exceptions 11-8
finally Clause 11-9
try-with-resources Statement 11-10
Catching Multiple Exceptions 11-11
Declaring Exceptions 11-12
Handling Declared Exceptions 11-13
Throwing Exceptions 11-14
Custom Exceptions 11-15
Assertions 11-16
Assertion Syntax 11-17
Internal Invariants 11-18
Control Flow Invariants 11-19
Class Invariants 11-20
Controlling Runtime Evaluation of Assertions 11-21
Summary 11-22
Practice 11-1 Overview: Catching Exceptions 11-23
Practice 11-2 Overview: Extending Exception and Using throw and throws 11-24
Quiz 11-25

12 Java Date/Time API

Objectives 12-2
Why Is Date and Time Important? 12-3
Previous Java Date and Time 12-4
Java Date and Time API: Goals 12-5
Working with Local Date and Time 12-6
Working with LocalDate 12-7
LocalDate: Example 12-8
Working with LocalTime 12-9
LocalTime: Example 12-10
Working with LocalDateTime 12-11
LocalTimeDate: Example 12-12
Working with Time Zones 12-13
Daylight Savings Time Rules 12-14
Modeling Time Zones 12-15

Creating ZonedDateTime Objects 12-16
Working with ZonedDateTime Gaps/Overlaps 12-17
ZoneRules 12-18
Working Across Time Zones 12-19
Date and Time Methods 12-20
Date and Time Amounts 12-21
Period 12-22
Duration 12-23
Calculating Between Days 12-24
Making Dates Pretty 12-25
Using Fluent Notation 12-26
Summary 12-27
Practices 12-28

13 Java I/O Fundamentals

Objectives 13-2
Java I/O Basics 13-3
I/O Streams 13-4
I/O Application 13-5
Data Within Streams 13-6
Byte Stream InputStream Methods 13-7
Byte Stream OutputStream Methods 13-8
Byte Stream: Example 13-9
Character Stream Reader Methods 13-10
Character Stream Writer Methods 13-11
Character Stream: Example 13-12
I/O Stream Chaining 13-13
Chained Streams: Example 13-14
Console I/O 13-15
Writing to Standard Output 13-16
Reading from Standard Input 13-17
Channel I/O 13-18
Persistence 13-19
Serialization and Object Graphs 13-20
Transient Fields and Objects 13-21
Transient: Example 13-22
Serial Version UID 13-23
Serialization: Example 13-24
Writing and Reading an Object Stream 13-25
Serialization Methods 13-26
readObject: Example 13-27

Summary	13-28
Practice 13-1 Overview: Writing a Simple Console I/O Application	13-29
Practice 13-2 Overview: Serializing and Deserializing a ShoppingCart	13-30
Quiz	13-31

14 Java File I/O (NIO.2)

Objectives	14-2
New File I/O API (NIO.2)	14-3
Limitations of java.io.File	14-4
File Systems, Paths, Files	14-5
Relative Path Versus Absolute Path	14-6
Java NIO.2 Concepts	14-7
Path Interface	14-8
Path Interface Features	14-9
Path: Example	14-10
Removing Redundancies from a Path	14-11
Creating a Subpath	14-12
Joining Two Paths	14-13
Symbolic Links	14-14
Working with Links	14-15
File Operations	14-16
Checking a File or Directory	14-17
Creating Files and Directories	14-19
Deleting a File or Directory	14-20
Copying a File or Directory	14-21
Moving a File or Directory	14-22
List the Contents of a Directory	14-23
Walk the Directory Structure	14-24
BufferedReader File Stream	14-25
NIO File Stream	14-26
Read File into ArrayList	14-27
Managing Metadata	14-28
Symbolic Links	14-29
Summary	14-30
Practice Overview	14-31
Quiz	14-33

15 Concurrency

Objectives	15-2
Task Scheduling	15-3
Legacy Thread and Runnable	15-4

Extending Thread	15-5
Implementing Runnable	15-6
The java.util.concurrent Package	15-7
Recommended Threading Classes	15-8
java.util.concurrent.ExecutorService	15-9
Example ExecutorService	15-10
Shutting Down an ExecutorService	15-11
java.util.concurrent.Callable	15-12
Example Callable Task	15-13
java.util.concurrent.Future	15-14
Example	15-15
Threading Concerns	15-16
Shared Data	15-17
Problems with Shared Data	15-18
Nonshared Data	15-19
Atomic Operations	15-20
Out-of-Order Execution	15-21
The synchronized Keyword	15-22
synchronized Methods	15-23
synchronized Blocks	15-24
Object Monitor Locking	15-25
Threading Performance	15-26
Performance Issue: Examples	15-27
java.util.concurrent Classes and Packages	15-28
The java.util.concurrent.atomic Package	15-29
java.util.concurrent.CyclicBarrier	15-30
Thread-Safe Collections	15-32
CopyOnWriteArrayList: Example	15-33
Summary	15-34
Practice 15-1 Overview: Using the java.util.concurrent Package	15-35
Quiz	15-36

16 The Fork-Join Framework

Objectives	16-2
Parallelism	16-3
Without Parallelism	16-4
Naive Parallelism	16-5
The Need for the Fork-Join Framework	16-6
Work-Stealing	16-7
A Single-Threaded Example	16-8
java.util.concurrent.ForkJoinTask<V>	16-9

RecursiveTask Example	16-10
compute Structure	16-11
compute Example (Below Threshold)	16-12
compute Example (Above Threshold)	16-13
ForkJoinPool Example	16-14
Fork-Join Framework Recommendations	16-15
Summary	16-16
Practice 16-1 Overview: Using the Fork-Join Framework	16-17
Quiz	16-18

17 Parallel Streams

Objectives	17-2
Streams Review	17-3
Old Style Collection Processing	17-4
New Style Collection Processing	17-5
Stream Pipeline: Another Look	17-6
Styles Compared	17-7
Parallel Stream	17-8
Using Parallel Streams: Collection	17-9
Using Parallel Streams: From a Stream	17-10
Pipelines Fine Print	17-11
Embrace Statelessness	17-12
Avoid Statefulness	17-13
Streams Are Deterministic for Most Part	17-14
Some Are Not Deterministic	17-15
Reduction	17-16
Reduction Fine Print	17-17
Reduction: Example	17-18
A Look Under the Hood	17-24
Illustrating Parallel Execution	17-25
Performance	17-36
A Simple Performance Model	17-37
Summary	17-38
Practice	17-39

18 Building Database Applications with JDBC

Objectives	18-2
Using the JDBC API	18-3
Using a Vendor's Driver Class	18-4
Key JDBC API Components	18-5
Writing Queries and Getting Results	18-6

Using a ResultSet Object	18-7
CRUD Operations Using JDBC API: Retrieve	18-8
CRUD Operations Using JDBC: Retrieve	18-9
CRUD Operations Using JDBC API: Create	18-10
CRUD Operations Using JDBC API: Update	18-11
CRUD Operations Using JDBC API: Delete	18-12
SQLException Class	18-13
Closing JDBC Objects	18-14
try-with-resources Construct	18-15
Using PreparedStatement	18-16
Using PreparedStatement: Setting Parameters	18-17
Executing PreparedStatement	18-18
PreparedStatement: Using a Loop to Set Values	18-19
Using CallableStatement	18-20
Summary	18-21
Practice 18-1 Overview: Working with the Derby Database and JDBC	18-22
Quiz	18-23

19 Localization

Objectives	19-2
Why Localize?	19-3
A Sample Application	19-4
Locale	19-5
Properties	19-6
Loading and Using a Properties File	19-7
Loading Properties from the Command Line	19-8
Resource Bundle	19-9
Resource Bundle File	19-10
Sample Resource Bundle Files	19-11
Initializing the Sample Application	19-12
Sample Application: Main Loop	19-13
The printMenu Method	19-14
Changing the Locale	19-15
Sample Interface with French	19-16
Format Date and Currency	19-17
Displaying Currency	19-18
Formatting Currency with NumberFormat	19-19
Displaying Dates	19-20
Displaying Dates with DateTimeFormatter	19-21
Format Styles	19-22
Summary	19-23

Practice 19-1 Overview: Creating a Localized Date Application 19-24
Quiz 19-25

20 Oracle Cloud

Agenda 20-2
What is Cloud? 20-3
What is Cloud Computing? 20-4
History – Cloud Evolution 20-5
Components of Cloud Computing 20-6
Characteristics of Cloud 20-7
Cloud Deployment Models 20-8
Cloud Service Models 20-9
Industry Shifting from On-Premises to the Cloud 20-13
Oracle IaaS Overview 20-15
Oracle PaaS Overview 20-16
Oracle SaaS Overview 20-17
Summary 20-18

21 Oracle Application Container Cloud Service Overview

Objectives 21-2
Oracle Application Container Cloud Service 21-3
Oracle Application Container Cloud 21-4
Polyglot Platform 21-5
Open Platform 21-6
Container-based Application Platform as a Service 21-7
Elastic Scaling 21-8
Profiling 21-9
Manageable 21-10
Deploy—Application Archive (Zip) 21-12
Application Deployment 21-13
Application Container Cloud Architecture 21-14
Load Balancer 21-15
Oracle Developer Cloud Service 21-16
Developer Cloud Service – Easy Adoption/Integration 21-17
Application Container Cloud Service Advantages 21-19
Summary 21-20

11

Exceptions and Assertions

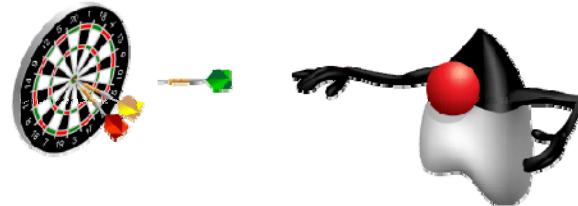
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-catch, and `finally` clauses
- Autoclose resources with a `try-with-resources` statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants by using assertions



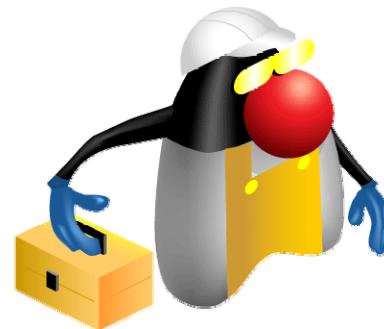
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Error Handling

Applications sometimes encounter errors while executing. Reliable applications should handle errors as gracefully as possible. Errors:

- Should be an exception and not the expected behavior
- Must be handled to create reliable applications
- Can occur as the result of application bugs
- Can occur because of factors beyond the control of the application
 - Databases becoming unreachable
 - Hard drives failing



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Returning a Failure Result

Some programming languages use the return value of a method to indicate whether or not a method completed successfully. For instance, in the C example `int x = printf("hi");`, a negative value for `x` would indicate a failure. Many of C's standard library functions return a negative value upon failure. The problem is that this example could also be written as `printf("hi");` where the return value is ignored. In Java, you also have the same concern; any return value can be ignored.

When a method you write in the Java language fails to execute successfully, consider using the exception-generating and handling features available in the language instead of using return values.

Exception Handling in Java

When you are using Java libraries that rely on external resources, the compiler will require you to “handle or declare” the exceptions that might occur.

- Handling an exception means that you must add in a code block to handle the error.
- Declaring an exception means that you declare that a method may fail to execute successfully.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Handle or Declare Rule

To use many libraries, you require knowledge of exception handling. They include:

- File IO (NIO: `java.nio`)
- Database access (JDBC: `java.sql`)

Handling an exception means that you use a `try-catch` statement to transfer control to an exception-handling block when an exception occurs. Declaring an exception means to add a `throws` clause to a method declaration, indicating that the method may fail to execute in a specific way. In other words, handling means it is your problem to deal with and declaring means that it is someone else's problem to deal with.

try-catch Statement

The try-catch statement is used to handle exceptions.

```
try {  
    System.out.println("About to open a file");  
    InputStream in =  
        new FileInputStream("missingfile.txt");  
    System.out.println("File open");  
} catch (Exception e) {  
    System.out.println("Something went wrong!");  
}
```

This line is skipped if the previous line failed to open the file.

This line runs only if something went wrong in the try block.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The catch Clause

When an exception occurs inside of a try block, execution will transfer to the attached catch block. Any lines inside the try block that appear after the exception are skipped and are not executed. The catch clause should be used to:

- Retry the operation
- Try an alternate operation
- Gracefully exit or return

Avoid having an empty catch block. Silently swallowing an exception is a bad practice.

Exception Objects

A catch clause is passed as a reference to a `java.lang.Exception` object.

The `java.lang.Throwable` class is the parent class for `Exception` and it outlines several methods that you may use.

```
try{
    //...
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Logging Exceptions

When things go wrong in your application, you will often want to record what happened. Java developers have a choice of several logging libraries including Apache's Log4j and the built-in `java.util` logging framework. Although these logging libraries are beyond the scope of this course, you may notice that IDEs such as NetBeans recommend that you should remove any calls to `printStackTrace()`. This is because production-quality applications should use a logging library instead of printing debug messages to the screen.

Using `getMessage()` and `printStackTrace()`

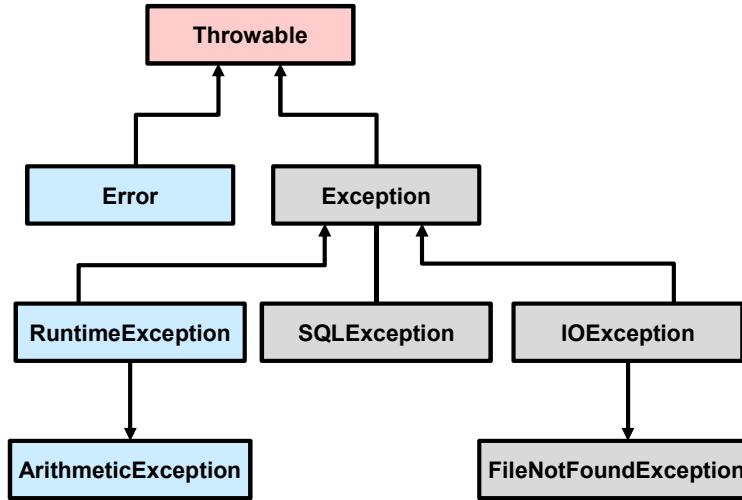
- `printStackTrace()`: When debugging, stack traces are very useful, because they tell you exactly where the exception happened and what the sequence of method calls is up to the point where the exception was thrown. So a stack trace helps to track down the cause of the exception.
- `getMessage()`: When you only want to know what the error message is and do not want the full stack trace, you can get the message of the exception.

Users of your application should not deal with a stack trace full of technical information, instead they should just with an error message. Therefore, it is preferable to use `getMessage()` rather than `printStackTrace()`.

Exception Categories

The `java.lang.Throwable` class forms the basis of the hierarchy of exception classes. There are two main categories of exceptions:

- Checked exceptions, which must be “handled or declared”
- Unchecked exceptions, which are not typically “handled or declared”



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Dealing with Exceptions

When an `Exception` object is generated and passed to a `catch` clause, it is instantiated from a class that represents the specific type of problem that occurred. These exception-related classes can be divided into two categories: checked and unchecked.

Unchecked Exceptions

`java.lang.RuntimeException` and `java.lang.Error` and their subclasses are categorized as unchecked exceptions. These types of exceptions should not normally occur during the execution of your application. You can use a `try-catch` statement to help discover the source of these exceptions. However, when an application is ready for production use, there should be a little code remaining that deals with `RuntimeException` and its subclasses. The `Error` subclasses represent errors that are beyond your ability to correct, such as the JVM running out of memory. Common `RuntimeExceptions` that you may have to troubleshoot include:

- `ArrayIndexOutOfBoundsException`: Accessing an array element that does not exist
- `NullPointerException`: Using a reference that does not point to an object
- `ArithmaticException`: Dividing by zero

Handling Exceptions

You should always catch the most specific type of exception.
Multiple catch blocks can be associated with a single try.

```
try {  
    System.out.println("About to open a file");  
    InputStream in = new FileInputStream("missingfile.txt");  
    System.out.println("File open");  
    int data = in.read();  
    in.close();  
} catch (FileNotFoundException e) {  
    System.out.println(e.getClass().getName());  
    System.out.println("Quitting");  
} catch (IOException e) {  
    System.out.println(e.getClass().getName());  
    System.out.println("Quitting");  
}
```

Order is important. You must catch the most specific exceptions first (that is, child classes before parent classes).



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Checked Exceptions

Every class that is a subclass of `Exception` except `RuntimeException` and its subclasses falls into the category of checked exceptions. You must “handle or declare” these exceptions with a `try` or `throws` statement. The HTML documentation for the Java API (Javadoc) will describe which checked exceptions can be generated by a method or constructor and why.

Catching the most specific type of exception enables you to write `catch` blocks that are targeted at handling very specific types of errors. You should avoid catching the base type of `Exception`, because it is difficult to create a general purpose `catch` block that can deal with every possible error.

Note: Exceptions thrown by the Java Persistence API (JPA) extend `RuntimeException`, and as such they are categorized as unchecked exceptions. These exceptions may need to be “handled or declared” in production-ready code, even though you are not required to do so by the compiler.

finally Clause

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally { A finally clause runs regardless of whether or not an Exception was generated.
    try {
        if(in != null) in.close(); You always want to close open resources.
    } catch(IOException e) {
        System.out.println("Failed to close file");
    }
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Closing Resources

When you open resources, such as files or database connections, you should always close them when they are no longer needed. Attempting to close resources inside the `try` block can be problematic because you can end up skipping the close operation. A `finally` block always runs regardless of whether or not an error occurred during the execution of the `try` block. If control jumps to a `catch` block, the `finally` block executes after the `catch` block.

Sometimes the operation that you want to perform in your `finally` block may itself cause an Exception to be generated. In that case, you may be required to nest a `try-catch` inside of a `finally` block. You may also nest a `try-catch` inside of `try` and `catch` blocks.

try-with-resources Statement

- The `try-with-resources` statement is a `try` statement that declares one or more resources.
- Any class that implements `java.lang.AutoCloseable` can be used as a resource.

```
System.out.println("About to open a file");
try (InputStream in =
      new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Closeable Resources

The `try-with-resources` statement can eliminate the need for a lengthy `finally` block. Resources opened by using the `try-with-resources` statement are always closed. If a resource should be autoclosed, its reference must be declared within the `try` statement's parenthesis.

Multiple resources can be opened if they are separated by semicolons. If you open multiple resources, they should be closed in the opposite order in which you opened them.

Catching Multiple Exceptions

Using the multi-catch clause, a single catch block can handle more than one type of exception.

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new ObjectInputStream(is)) {
    cart = (ShoppingCart) in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Multiple exception types
are separated with a
vertical bar.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Benefits of Multi-catch

Sometimes you want to perform the same action regardless of the exception being generated. The new multi-catch clause reduces the amount of code you must write by eliminating the need for multiple catch clauses with the same behaviors.

Another benefit of the multi-catch clause is that it makes it less likely that you will attempt to catch a generic exception. Catching `Exception` prevents you from noticing other types of exceptions that might be generated by code that you add later to a `try` block.

The type alternatives that are separated with vertical bars cannot have an inheritance relationship. You may not list both `FileNotFoundException` and `IOException` in a multi-catch clause.

File I/O and object serialization are covered in the lesson titled “Java I/O Fundamentals.”

Declaring Exceptions

You may declare that a method throws an exception instead of handling it.

```
public static int readByteFromFile() throws IOException {  
    try (InputStream in = new FileInputStream("a.txt")) {  
        System.out.println("File open");  
        return in.read();  
    }  
}
```

Notice the lack of `catch` clauses. The `try-with-resources` statement is being used only to close resources.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the `throws` clause, a method may declare that it throws one or more exceptions during execution.

If an exception is generated while executing the method, the method stops executing and the exception is thrown to the caller.

Overridden methods may declare the same exceptions, fewer exceptions, or more specific exceptions, but not additional or more generic exceptions.

A method may declare multiple exceptions with a comma-separated list.

```
public static int readByteFromFile() throws FileNotFoundException,  
IOException {  
    try (InputStream in = new FileInputStream("a.txt")) {  
        System.out.println("File open");  
        return in.read();  
    }  
}
```

Technically, you do not need to declare `FileNotFoundException` because it is a subclass of `IOException`, but it is a good practice to do so.

Handling Declared Exceptions

The exceptions that methods may throw must still be handled. Declaring an exception just makes it someone else's job to handle them.

```
public static void main(String[] args) {  
    try {  
        int data = readByteFromFile();  
    } catch (IOException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

Method that declared an exception



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Handling Exceptions

Your application should always handle its exceptions. Adding a `throws` clause to a method only delays the handling of the exception. In fact, an exception can be thrown repeatedly up the call stack. A standard Java SE application must handle any exceptions before they are thrown out of the `main` method; otherwise, you risk having your program terminate abnormally. It is possible to declare that `main` throws an exception, but unless you are designing programs to terminate in a nongraceful fashion, you should avoid doing so.

Throwing Exceptions

The `throw` statement is used to throw an instance of exception.

```
1 import java.io.FileNotFoundException;
2 class DemoThrowsException {
3     public void readFile(String file) throws
4         FileNotFoundException {
5         boolean found = findFile(file);
6         if (!found)
7             throw new FileNotFoundException("Missing file");
8         else {
9             //code to read file
10        }
11    }
12    boolean findFile(String file) {
13        //code to return true if file can be located
14    }
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The code snippet in the slide demonstrates creating a method that throws a checked exception

Line 3: The `throws` statement indicates that this method can throw `FileNotFoundException`.

Line 7: If the file cannot be found, the code creates and throws an object of `FileNotFoundException` by using the `throw` statement.

A method chooses to throw an exception as opposed to handling it itself. It is a contract between the calling method and the called method.

In this example, the method `readFile` does not handle `FileNotFoundException` itself because its responsibilities do not include how to locate a file.

Custom Exceptions

You can create custom exception classes by extending `Exception` or one of its subclasses.

```
class InvalidPasswordException extends Exception {  
  
    InvalidPasswordException() {  
    }  
    InvalidPasswordException(String message) {  
        super(message);  
    }  
    InvalidPasswordException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Custom exceptions are never thrown by standard Java class libraries. To take advantage of a custom exception class, you must throw it yourself. For example:

```
throw new InvalidPasswordException();
```

A custom exception class may override methods or add new functionality. The rules of inheritance are the same, even though the parent class type is an exception.

Because exceptions capture information about a problem that has occurred, you may need to add fields and methods depending on the type of information that needs to be captured. If a string can capture all the necessary information, you can use the `getMessage()` method that all `Exception` classes inherit from `Throwable`. Any `Exception` constructor that receives a string will store it to be returned by `getMessage()`.

Assertions

- Use assertions to document and verify the assumptions and internal logic of a single method:
 - Internal invariants
 - Control flow invariants
 - Class invariants
- Inappropriate uses of assertions
 - Do not use assertions to check the parameters of a public method.
 - Do not use methods that can cause side effects in the assertion check.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Why Use Assertions

You can use assertions to add code to your applications, which would ensure that the application is executing as expected. Using assertions, you test for various conditions failing; if they do, you terminate the application and display debugging-related information. Assertions should not be used if the checks to be performed should always be executed because assertion checking may be disabled.

Assertion Syntax

There are two forms of the `assert` statement:

- **`assert booleanExpression;`**
 - This statement tests the boolean expression.
 - It does nothing if the boolean expression evaluates to `true`.
 - If the boolean expression evaluates to `false`, this statement throws an `AssertionError`.
- **`assert booleanExpression : expression;`**
 - This form acts just like `assert booleanExpression;`.
 - In addition, if the boolean expression evaluates to `false`, the second argument is converted to a string and is used as descriptive text in the `AssertionError` message.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `assert` Statement

`AssertionError` is a subclass of `Error` and, therefore, falls in the category of unchecked exceptions.

Internal Invariants

```
public class Invariant {  
  
    static void checkNum(int num) {  
        int x = num;  
        if (x > 0) {  
            System.out.print("number is positive" + x);  
  
        } else if (x == 0) {  
            System.out.print("number is zero" + x);  
        } else {  
            assert (x > 0);  
        }  
    }  
    public static void main(String args[]) {  
  
        checkNum(-4);  
    }  
}
```

Internal Invariant



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An invariant is something that should always be true. An internal invariant is a “fact” that you believe to be true at a certain point in the program.

In the code snippet in the slide, the `assert` statement determines whether the number is less than zero and, if so, it throws an `AssertionError`.

Control Flow Invariants

```
1 switch (suit) {  
2     case Suit.CLUBS: // ...  
3         break;  
4     case Suit.DIAMONDS: // ...  
5         break;  
6     case Suit.HEARTS: // ...  
7         break;  
8     case Suit.SPADES: // ...  
9         break;  
10    default:  
11        assert false : "Unknown playing card suit";  
12    break;  
13 }
```

Control Flow Invariant



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Assertion can be used in a `switch` statement with no `default` `case`, when the programmer is sure that one of the `switch` `cases` will be executed every time he or she can omit the `default` `case` as in the example in the slide.

To test this assumption, the programmer can add an `assert` statement in the `default` `case`. By using the `assert` statement, you can check the assumption about the applications flow of control. Assertion can be placed at any location where the control will not be reached.

Class Invariants

```
public class PersonClassInvariant {  
    String name;  
    String ssn;  
    int age;  
  
    private void checkAge()  
    {  
        assert age >= 18 && age < 150;  
    }  
  
    public void changeName(String fname)  
    {  
        checkAge();  
        name=fname;  
    }  
}
```

Class Invariant



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A class invariant is one that an object must satisfy in order to be a valid member of a class.

Controlling Runtime Evaluation of Assertions

- If assertion checking is disabled, the code runs as fast as it would if the check were not there.
- Assertion checks are disabled by default. Enable assertions with either of the following commands:

```
java -enableassertions MyProgram
```

```
java -ea MyProgram
```

- Assertion checking can be controlled on class, package, and package hierarchy basis. See:
<http://download.oracle.com/javase/7/docs/technotes/guides/language/assert.html>



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Enabling Assertions in Netbeans

1. In Netbeans, right-click the project and select **Properties**.
2. In the window that appears, select **Run**.
3. Enter **-enableassertions** in VM Options.

Summary

In this lesson, you should have learned how to:

- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-`catch`, and `finally` clauses
- Autoclose resources with a `try-with-resources` statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants by using assertions



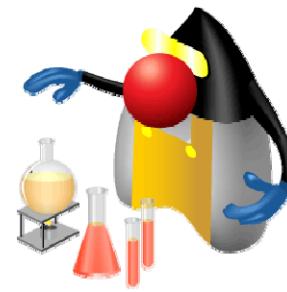
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 11-1 Overview: Catching Exceptions

This practice covers the following topics:

- Adding try-catch statements to a class
- Handling exceptions



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this practice, you write code to deal with both checked and unchecked exceptions.

Practice 11-2 Overview: Extending Exception and Using throw and throws

This practice covers the following topics:

- Extending the Exception class
- Throwing exceptions using throw and throws



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

A `NullPointerException` must be caught by using a try-catch statement.

- a. True
- b. False



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which of the following types are all checked exceptions (instanceof) ?

- a. Error
- b. Throwable
- c. RuntimeException
- d. Exception



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which keyword would you use to add a clause to a method stating that the method might produce an exception?

- a. throw
- b. thrown
- c. throws
- d. assert



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Assertions should be used to perform user-input validation.

- a. True
- b. False



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

12

Java Date/Time API

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Create and manage date-based events
- Create and manage time-based events
- Combine date and time into a single object
- Work with dates and times across time zones
- Manage changes resulting from daylight savings
- Define and create timestamps, periods, and durations
- Apply formatting to local and zoned dates and times



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Why Is Date and Time Important?

In the development of applications, programmers often need to represent time and use it to perform calculations:

- The current date and time (locally)
- A date and/or time in the future or past
- The difference between two dates/time in seconds, minutes, hours, days, months, years
- The time or date in another country (time zone)
- The correct time after daylight savings time is applied
- The number of days in the month of February (leap years)
- A time duration (hours, mins, secs) or a period (years, months, days)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Current time and date are used to calculate events in the future, and as timestamps.
- Calculating a time or date offset is important when determining what the time and date are when n hours or n days are added to a date.
- Determining time and date in other countries is often a critical factor in determining when meetings happen, or what the local time is when a plane lands.
- Leap years are incredibly tricky to manage.

Previous Java Date and Time

Disadvantages of `java.util.Date` (`Calendar`, `TimeZone` & `DateFormat`):

- Does not support fluent API approach
- Instances are mutable – not compatible with lambda
- Not thread-safe
- Weakly typed calendars
- One size fits all



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `java.time` API is a major evolution of the previous APIs that supported the `java.util` API's `Date`, `Calendar`, `TimeZone`, and `DateFormat`.

Java Date and Time API: Goals

- The classes and methods should be straightforward.
- The API should support a fluent API approach.
- Instances of time/date objects should be immutable. (This is important for lambda operations.)
- Use ISO standards to define date and time.
- Time and date operations should be thread-safe.
- The API should support strong typing, which makes it much easier to develop good code first. (The compiler is your friend!)
- `toString` will always return a human-readable format.
- Allow developers to extend the API easily.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The ISO Calendar is also known as the Gregorian calendar in JDK code. The ISO calendar system applies the current rules for leap years both forward and backward in time.

Working with Local Date and Time

The `java.time` API defines two classes for working with local dates and times (without a time zone):

- `LocalDate`:
 - Does not include time
 - A year-month-day representation
 - `toString` – ISO 8601 format (YYYY-MM-DD)
- `LocalTime`:
 - Does not include date
 - Stores hours:minutes:seconds.nanoseconds
 - `toString` – (HH:mm:ss.SSSS)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ISO-8601 defines the international format of dates as the year first, followed by the month, day, hour, minutes, and seconds. The definition is based on the relative importance of each unit of time.

Working with LocalDate

`LocalDate` is a class that holds an event date: a birth date, anniversary, meeting date, and so on.

- A date is a label for a day.
- `LocalDate` uses the ISO calendar by default.
- `LocalDate` does not include time, so it is portable across time zones.
- You can answer the following questions about dates with `LocalDate`:
 - Is it in the future or past?
 - Is it in a leap year?
 - What day of the week is it?
 - What is the day a month from now?
 - What is the date next Tuesday?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

`java.util.Date` includes a time, and developers would often use midnight to represent just a date. But some time zones do not have a midnight depending upon where they are in day light savings time.

LocalDate: Example

```

import java.time.LocalDate;
import static java.time.temporal.TemporalAdjusters.*;
import static java.time.DayOfWeek.*;
import static java.lang.System.out;

public class LocalDateExample {

    public static void main(String[] args) {
        LocalDate now, bDate, nowPlusMonth, nextTues;
        now = LocalDate.now();
        out.println("Now: " + now);
        bDate = LocalDate.of(1995, 5, 23); // Java's Birthday
        out.println("Java's Bday: " + bDate);
        out.println("Is Java's Bday in the past? " + bDate.isBefore(now));
        out.println("Is Java's Bday in a leap year? " + bDate.isLeapYear());
        out.println("Java's Bday day of the week: " + bDate.getDayOfWeek());
        nowPlusMonth = now.plusMonths(1);
        out.println("The date a month from now: " + nowPlusMonth);
        nextTues = now.with(next(TUESDAY));
        out.println("Next Tuesday's date: " + nextTues);
    }
}

```

next method

TUESDAY

LocalDate objects are immutable – methods return a new instance.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- **TemporalAdjusters** is a final class of static utility methods used to modify temporal objects (such as `LocalDate`).
- **DayOfWeek** is an enum of days of the week, that is, `FRIDAY`, `TUESDAY`
- **Sample output:**

```

Now: 2014-02-14
Java's Bday: 1995-05-23
Is Java's Bday in the past? true
Is Java's Bday in a leap year? false
Java's Bday day of the week: TUESDAY
The date a month from now: 2014-03-14
Next Tuesday's date: 2014-02-18

```

Working with LocalTime

LocalTime stores the time within a day.

- Measured from midnight
- Based on a 24-hour clock (13:30 is 1:30 PM.)
- Questions you can answer about time with LocalTime
 - When is my lunch time?
 - Is lunch time in the future or past?
 - What is the time 1 hour 15 minutes from now?
 - How many minutes until lunch time?
 - How many hours until bedtime?
 - How do I keep track of just the hours and minutes?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

LocalTime: Example

```
import java.time.LocalTime;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalTimeExample {
    public static void main(String[] args) {
        LocalTime now, nowPlus, nowHrsMins, lunch, bedtime;
        now = LocalTime.now();
        out.println("The time now is: " + now);
        nowPlus = now.plusHours(1).plusMinutes(15);
        out.println("What time is it 1 hour 15 minutes from now? " + nowPlus);
        nowHrsMins = now.truncatedTo(MINUTES);
        out.println("Truncate the current time to minutes: " + nowHrsMins);
        out.println("It is the " + now.toSecondOfDay()/60 + "th minute");
        lunch = LocalTime.of(12, 30);
        out.println("Is lunch in my future? " + lunch.isAfter(now));
        long minsToLunch = now.until(lunch, MINUTES);
        out.println("Minutes till lunch: " + minsToLunch);
        bedtime = LocalTime.of(21, 0);
        long hrsToBedtime = now.until(bedtime, HOURS);
        out.println("How many hours until bedtime? " + hrsToBedtime);
    }
}
```

HOURS, MINUTES



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ChronoUnit is an enum that holds time units, including HALF_DAYS, HOURS, YEARS, and WEEKS.

Sample output:

```
The time now is: 11:21:26.302
What time is it 1 hour 15 minutes from now? 12:36:26.302
Truncate the current time to minutes: 11:21
It is the 681th minute
Is lunch in my future? true
Minutes till lunch: 68
How many hours until bedtime? 9
```

Working with `LocalDateTime`

`LocalDateTime` is a combination of `LocalDate` and `LocalTime`.

- `LocalDateTime` is useful for narrowing events.
- You can answer the following questions with `LocalDateTime`:
 - When is the meeting with corporate?
 - When does my flight leave?
 - When does the course start?
 - If I move the meeting to Friday, what is the date?
 - If the course starts at 9 AM on Monday and ends at 5 PM on Friday, how many hours am I in class?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

LocalTimeDate: Example

```
import java.time.*;  
import static java.time.Month.*;  
import static java.time.temporal.ChronoUnit.*;  
import static java.lang.System.out;  
  
public class LocalDateTimeExample {  
    public static void main(String[] args) {  
        LocalDateTime meeting, flight, courseStart, courseEnd;  
        meeting = LocalDateTime.of(2014, MARCH, 21, 13, 30);  
        out.println("Meeting is on: " + meeting);  
        LocalDate flightDate = LocalDate.of(2014, MARCH, 31);  
        LocalTime flightTime = LocalTime.of(21, 45);  
        flight = LocalDateTime.of(flightDate, flightTime);  
        out.println("Flight leaves: " + flight);  
        courseStart = LocalDateTime.of(2014, MARCH, 24, 9, 00);  
        courseEnd = courseStart.plusDays(4).plusHours(8);  
        out.println("Course starts: " + courseStart);  
        out.println("Course ends: " + courseEnd);  
        long courseHrs = (courseEnd.getHour() - courseStart.getHour()) *  
                         (courseStart.until(courseEnd, DAYS) + 1);  
        out.println("Course is: " + courseHrs + " hours long.");  
    }  
}
```

LocalDateTime,
LocalDate, LocalTime

MARCH

Combine LocalDate
and LocalTime
objects.

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

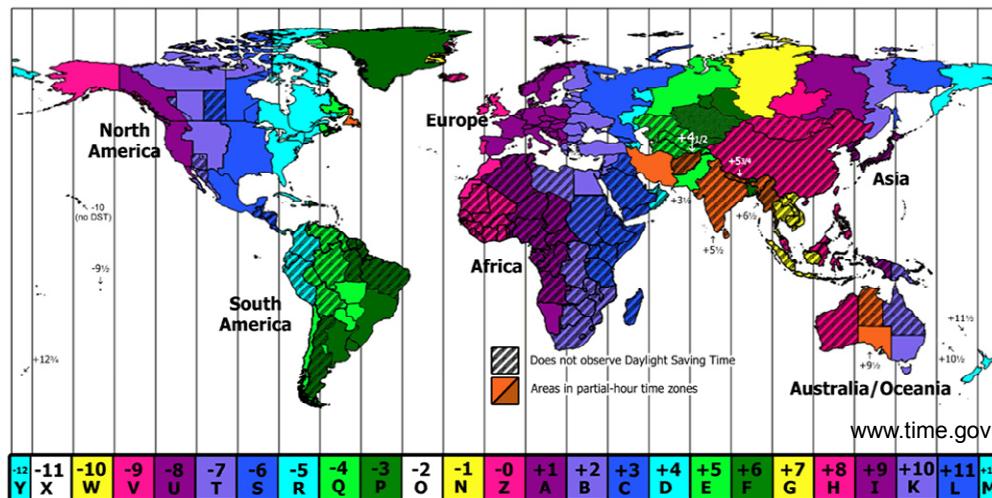
Sample output:

```
Meeting is on: 2014-03-21T13:30  
Flight leaves: 2014-03-31T21:45  
Course starts: 2014-03-24T09:00  
Course ends: 2014-03-28T17:00  
Course is: 40 hours long.
```

Working with Time Zones

Time zones are geographic, but the time in a specific location is defined by the government in that location.

- When a country (and sometimes a state) observes changes (for daylight savings) varies.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Time zones are relative to the Coordinated Universal Time (UTC).
- Time rules are based on the offset from UTC:
 - New York is UTC – 5 hours during standard time.
 - New York is UTC – 4 hours during daylight savings.
- Daylight savings can change from year to year, and sometimes even in a single year.
 - USA DST started: 3/14/2010, 3/13/2011, 3/12/2012, 3/10/2013, 3/9/2014, and so on
 - Arizona does not recognize DST.
 - Egypt had two DST periods in 2010, and no DST changes since 2011.

Daylight Savings Time Rules

Time changes result in a local hour gap/overlap:

Sunday, March 9, 2014 (New York)	Local time	UTC Offset
	1:59:58 AM	UTC-5h EST
	1:59:59 AM	UTC-5h EST
Starting DST causes a one hour gap.	2:00:00 -> 3:00:00	UTC-4h EDT
	3:00:01 AM	UTC-4h EDT

Sunday, November 2, 2014 (New York)	Local time	UTC Offset
	1:59:58 AM	UTC-4h EST
	1:59:59 AM	UTC-4h EST
Ending DST causes a one hour overlap.	2:00:00 -> 1:00:00	UTC-5h EDT
	1:00:01 AM	UTC-5h EDT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Greenwich Mean Time (GMT) was previously used as the world standard time based on the average day observed at the Greenwich Observatory (in the suburbs of London).

Note that some time changes occur at midnight, which means that you cannot use midnight in a date, because some countries do not have a midnight during changes to DST.

Modeling Time Zones

- `ZoneId`: Is a specific location or offset relative to UTC

```
ZoneId nyTZ = ZoneId.of("America/New_York");
ZoneId EST = ZoneId.of("US/Eastern");
ZoneId Romeo = ZoneId.of("Europe/London");
```

- `ZoneOffset`: Extends `ZoneId`; specifies the actual time difference from UTC

```
ZoneOffset USEast = ZoneOffset.of("-5");
ZoneOffset Nepal = ZoneOffset.ofHoursMinutes(5, 45);
ZoneId EST = ZoneId.ofOffset("UTC", USEast);
```

- `ZoneRules`: Is the class used to determine offsets



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The relationship between these objects is that a `ZoneId` is a specific time zone that falls within a `ZoneOffset`. `ZoneRules` are used with a `ZoneId` to determine changes in the `ZoneOffset` based on the specific date and daylight savings time changes.

Creating ZonedDateTime Objects

- Stores LocalDateTime, ZoneId, and ZoneOffset

```
ZoneId USEast = ZoneId.of("America/New_York");
LocalDate date = LocalDate.of(2014, MARCH, 23);
LocalTime time = LocalTime.of(9, 30);
LocalDateTime dateTime = LocalDateTime.of(date, time);
ZonedDateTime courseStart = ZonedDateTime.of(date, time, USEast);
ZonedDateTime hereNow = ZonedDateTime.now(USEast).truncatedTo(MINUTES);
System.out.println("Here now: " + hereNow);
System.out.println("Course start: " + courseStart);
ZonedDateTime newCourseStart = courseStart.plusDays(2).minusMinutes(30);
System.out.println("New Course Start: " + newCourseStart);
```

```
Here now: 2014-02-19 T 17:00 -05:00 [America/New_York]
Course start: 2014-03-23 T 09:30 -04:00 [America/New_York]
New Course Start: 2014-03-25 T 09:00 -04:00 [America/New York]
```

Space added to make the fields more clear



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Working with ZonedDateTime Gaps/Overlaps

Given a meeting date the day before daylight savings (2AM on March 9th), what happens if the meeting is moved out by a day?

```
// DST Begins March 9th, 2014
LocalDate meetDate = LocalDate.of(2014, MARCH, 8);
LocalTime meetTime = LocalTime.of(16, 00);
ZonedDateTime meeting = ZonedDateTime.of(meetDate, meetTime, USEast);
System.out.println("meeting time:      " + meeting);
ZonedDateTime newMeeting = meeting.plusDays(1);
System.out.println("new meeting time: " + newMeeting)
```

```
meeting time:      2014-03-08 16:00 -05:00[America/New_York]
new meeting time: 2014-03-09 16:00 -04:00[America/New_York]
```

- The local time is not changed, and the offset is managed correctly.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Likewise, the overlaps are also handled in the same way:

```
// DST Ends November 2nd, 2014
meetDate = LocalDate.of(2104, NOVEMBER, 1);
meeting = ZonedDateTime.of(meetDate, meetTime, USEast);
System.out.println("meeting time:      " + meeting);
newMeeting = meeting.plusDays(1);
System.out.println("new meeting time: " + newMeeting);
```

ZoneRules

- Each time zone (`ZoneId`) has a set of rules that are part of the JDK.
- Date or times that land on time changes can be determined by using the rules.

```
// Ask the rules if there was a gap or overlap
ZoneId USEast = ZoneId.of("America/New_York");
LocalDateTime lateNight = LocalDateTime.of(2014, MARCH, 9, 2, 30);
ZoneOffsetTransition zot = USEast.getRules().getTransition(lateNight);
if (zot != null) {
    if (zot.isGap()) System.out.println("gap");
    if (zot.isOverlap()) System.out.println("overlap");
}
```

- Given the code above, what will print?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

From Javadocs: “The Java virtual machine has a default provider that provides zone rules for the time-zones defined by IANA Time Zone Database (TZDB). If the system property `java.time.zone.DefaultZoneRulesProvider` is defined, then it is taken to be the fully-qualified name of a concrete `ZoneRulesProvider` class to be loaded as the default provider, using the system class loader. If this system property is not defined, a system-default provider will be loaded to serve as the default provider.”

Note: In the JDK, the default provider is a class, `TzdbZoneRulesProvider`. This class reads the time zone database located in the `jdk1.8.0/jre/lib/tzdb.dat` file.

Working Across Time Zones

The `OffsetDateTime` class stores a `LocalDateTime` and `ZoneOffset`.

- This is useful for determining `ZonedDateTime`s across time zones.

```
LocalDateTime meeting = LocalDateTime.of(2014, JUNE, 13, 12, 30);
ZoneId SanFran = ZoneId.of("America/Los_Angeles");
ZonedDateTime staffCall = ZonedDateTime.of(meeting, SanFran);
OffsetDateTime = staffCall.toOffsetDateTime();
```

- The offset is used to calculate date/time using zone rules:

```
ZoneId London = ZoneId.of("Europe/London");
OffsetDateTime staffCallOffset = staffCall.toOffsetDateTime();
ZonedDateTime staffCallUK = staffCallOffset.atZoneSameInstant(London);
System.out.println("Staff call (Pacific) is at: " + staffCall);
System.out.println("Staff call (UK) is at: " + staffCallLondon);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The easiest way to get an `OffsetDateTime` is from a `ZonedDateTime` by using the `toOffsetDateTime` method.

Output:

```
Staff call (Pacific) is at: 2014-06-13T12:30-07:00 [America/Los_Angeles]
Staff call (UK) is at: 2014-06-13T20:30+01:00 [Europe/London]
```

Date and Time Methods

Prefix	Example	Use
now	today = LocalDate.now()	Creates an instance using the system clock
of	meet = LocalTime.of(13, 30)	Creates an instance by using the parameters passed
get	today.get(DAY_OF_WEEK)	Returns part of the state of the target
with	meet.withHour(12)	Returns a copy of the target object with one element changed
plus, minus	nextWeek.plusDays(7) sooner.minusMinutes(30)	Returns a copy of the object with the amount added or subtracted
to	meet.toSecondOfDay()	Converts this object to another type. Here returns int seconds.
at	today.atTime(13, 30)	Combines this object with another; returns a LocalDateTime object
until	today.until	Calculates the amount of time until another date in terms of the unit
isBefore, isAfter	today.isBefore(lastWeek)	Compares this object with another on the timeline
isLeapYear	today.isLeapYear()	Checks if this object is a leap year



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The list shown in the slide is not a complete list of the methods supported.

Date and Time Amounts

- Instant – Stores an instant in time on the time-line
 - Useful for: timestamps, e.g. login events
 - Stored as seconds (`long`) and nanoseconds (`int`)
 - Methods used to compare before and after

```
Instant now = Instant.now();
Thread.sleep(0,1); // long milliseconds, int nanoseconds
Instant later = Instant.now();
System.out.println("now is before later? " + now.isBefore(later));
System.out.println("Now:    " + now);
System.out.println("Later: " + later);
```

```
now is before later? true
Now: 2014-02-21 T 16:11:34.788 Z
Later: 2014-02-21 T 16:11:34.789 Z
```

toString includes
nanoseconds to three digits



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Instants are stored in a long using positive values for dates after the EPOCH, 1970-01-01T00:00:00Z, and negative values before. Instants are always recorded using UTC. Instant is the closest equivalent to `java.util.Date`.

An instant relies on the Java Time-Scale to maintain accuracy. See javadoc for the complete details.

Nanosecond values are stored in a range between 0 and 999,999,999.

Period

Period is a class that holds a date-based amount.

- Years, months, and days based on the ISO-8601 calendar
- Plus and minus work with a conceptual day, thus preserving daylight savings changes

```
Period oneDay = Period.ofDays(1);
System.out.println("Period of one day: " + oneDay);
LocalDateTime beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
ZonedDateTime newYorkTime =
    ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After: " + newYorkTime.plus(oneDayYear));
```

The time is preserved, because
only "days" are added.

```
Period of one day: P1D
Before: 2014-03-08 T 12:00 -05:00 [America/New_York]
After: 2014-03-09 T 12:00 -04:00 [America/New_York]
```

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Duration

Duration is a class that stores a time-based amount.

- Time is measured in actual seconds and nanoseconds.
- Days are treated as 24 hours, and daylight savings is ignored.

```
Duration one24hourDay = Duration.ofDays(1);
System.out.println("Duration of one day: " + one24hourDay);
beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
newYorkTime = ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After: " + newYorkTime.plus(one24hourDay));
```

The time is not preserved because
24 hours are added.

```
Duration of one day: PT24H
Before: 2014-03-08 T 12:00 -05:00 [America/New_York]
After: 2014-03-09 T 13:00 -04:00 [America/New_York]
```

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Calculating Between Days

TemporalUnit is an interface representing a unit of time.

- Implemented by the enum class ChronoUnit

```
import static java.time.temporal.ChronoUnit.*;  
  
LocalDate christmas = LocalDate.of(2014, DECEMBER, 25);  
LocalDate today = LocalDate.now();  
long days = DAYS.between(today, christmas);  
System.out.println("There are " + days + " shopping days til Christmas");
```

- Period also provides a between method

```
Period tilXMas = Period.between(today, christmas);  
System.out.println("There are " + tilXMas.getMonths() +  
                  " months and " + tilXMas.getDays() +  
                  " days til Christmas");
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

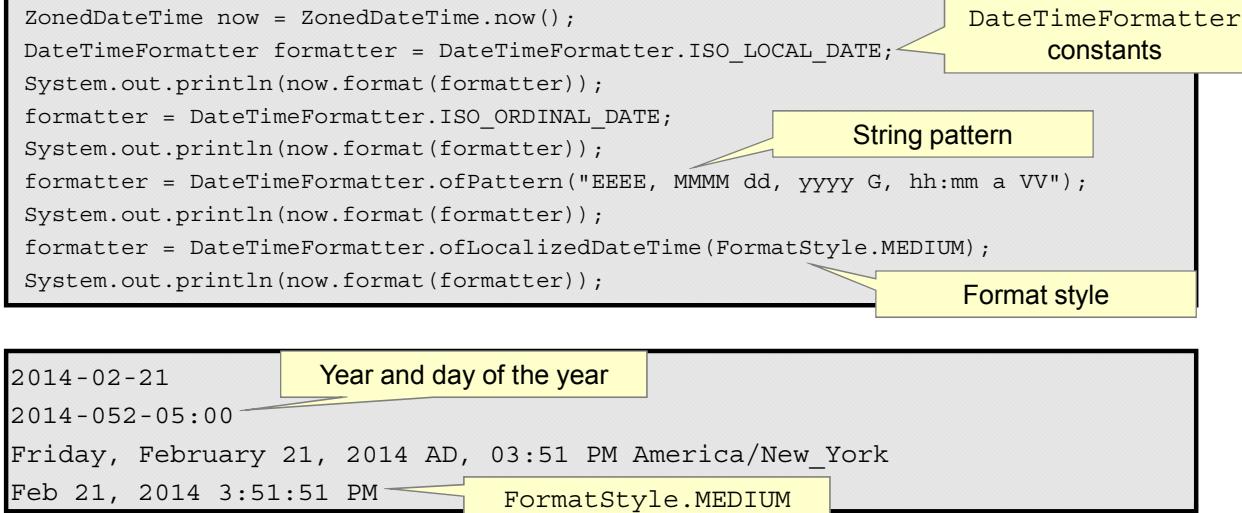
Output:

```
There are 307 shopping days til Christmas  
There are 10 months and 4 days til Christmas
```

Making Dates Pretty

`DateTimeFormatter` produces formatted date/times

- Using predefined constants, patterns letters, or a localized style



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Use four characters for the full representation of a field – for example, E represents the day of the week. One E is used for Tue and four (EEEE) represent Tuesday.

`FormatStyle` is an enum with SHORT, MEDIUM, LONG, and FULL.

Using Fluent Notation

One of the goals of JSR-310 was to make the API fluent.

- Examples:

```
// Not very readable - is this June 11 or November 6th?  
LocalDate myBday = LocalDate.of(1970, 6, 11);  
  
// A fluent approach  
myBday = Year.of(1970).atMonth(JUNE).atDay(11);  
  
// Schedule a meeting fluently  
LocalDateTime meeting = LocalDate.of(2014, MARCH, 25).atTime(12, 30);  
  
// Schedule that meeting using the London timezone  
ZonedDateTime meetingUK = meeting.atZone(ZoneId.of("Europe/London"));  
  
// What time is it in San Francisco for that meeting?  
ZonedDateTime earlyMeeting =  
    meetingUK.withZoneSameInstant(ZoneId.of("America/Los_Angeles"));
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have learned how to:

- Create and manage date-based events
- Create and manage time-based events
- Combine date and time into a single object
- Work with dates and times across time zones
- Manage changes resulting from daylight savings
- Define and create timestamps, periods and durations
- Apply formatting to local and zoned dates and times



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practices

- Practice 12-1: Working with Local Dates and Times
- Practice 12-2: Working with Dates and Times Across Time Zones
- Practice 12-3: Formatting Dates



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

13

Java I/O Fundamentals

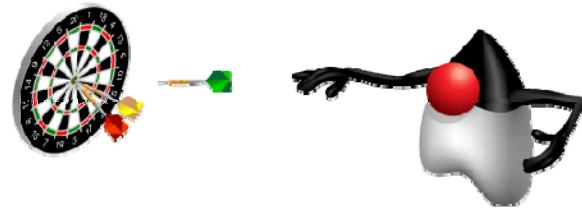
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use I/O streams to read and write files
- Read and write objects by using serialization



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java I/O Basics

The Java programming language provides a comprehensive set of libraries to perform input/output (I/O) functions.

- Java defines an I/O channel as a stream.
- An I/O stream represents an input source or an output destination.
- An I/O stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- I/O streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

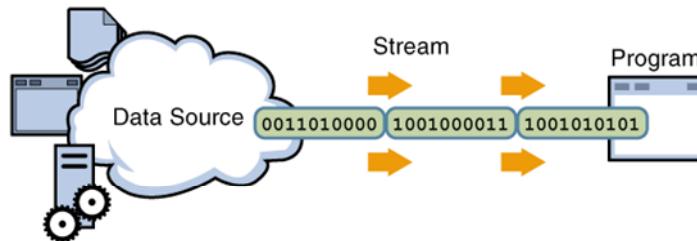


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

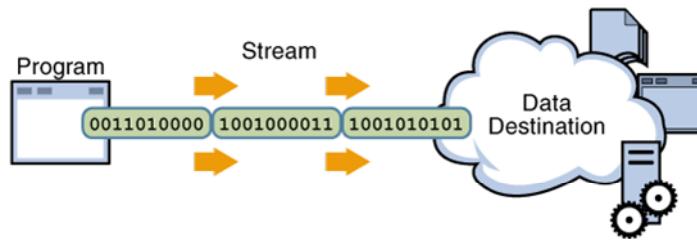
Some I/O streams simply pass on data; others manipulate and transform the data in useful ways.

I/O Streams

- A program uses an input stream to read data from a source, one item at a time.



- A program uses an output stream to write data to a destination (sink), one item at time.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

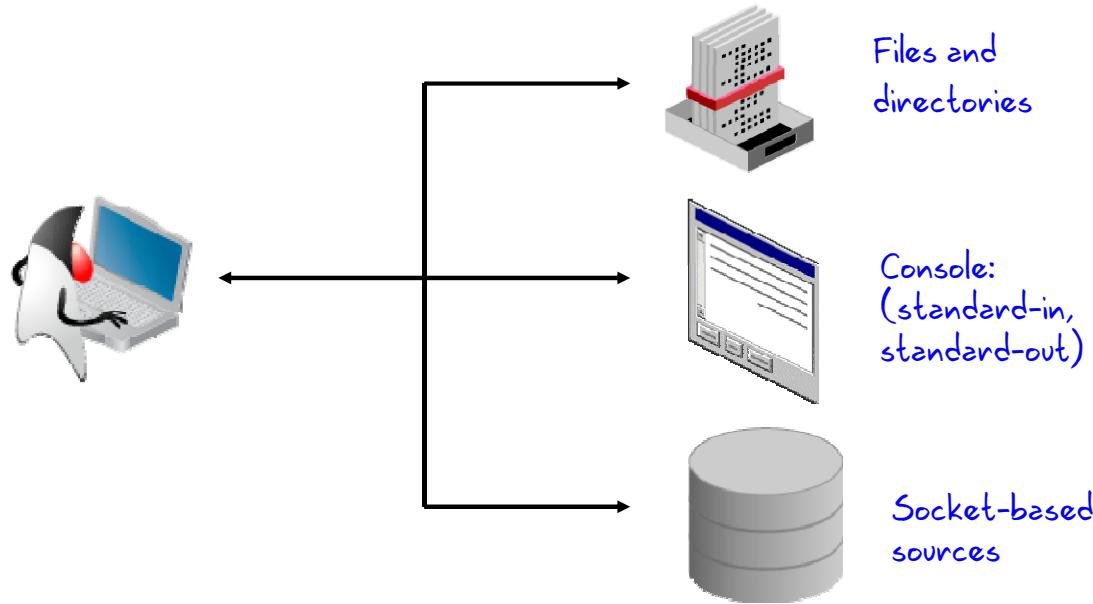
No matter how they work internally, all streams present the same simple model to programs that use them. A stream is a sequential flow of data. A stream can come from a source or can be generated to a sink.

- A source stream initiates the flow of data, also called an input stream.
- A sink stream terminates the flow of data, also called an output stream.

Sources and sinks are both node streams. Types of node streams are files, memory, and pipes between threads or processes.

I/O Application

Typically, a developer uses input and output in three ways:



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An application developer typically uses I/O streams to read and write files, to read information from and write information to some output device, such as the keyboard (standard in) and the console (standard out). Finally, an application may need to use a socket to communicate with another application on a remote system.

Data Within Streams

- Java technology supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
 - Normally, the term *stream* refers to a byte stream.
 - The terms *reader* and *writer* refer to character streams.

Stream	Byte Streams	Character Streams
Source streams	InputStream	Reader
Sink streams	OutputStream	Writer



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java technology supports two types of data in streams: raw bytes and Unicode characters. Typically, the term *stream* refers to byte streams and the terms *reader* and *writer* refer to character streams.

More specifically, byte input streams are implemented by subclasses of the `InputStream` class and byte output streams are implemented by subclasses of the `OutputStream` class. Character input streams are implemented by subclasses of the `Reader` class and character output streams are implemented by subclasses of the `Writer` class.

Byte streams are best applied to reading and writing of raw bytes (such as image files, audio files, and objects). Specific subclasses provide methods to provide specific support for each of these stream types.

Character streams are designed for reading characters (such as in files and other character-based streams).

Byte Stream InputStream Methods

- The three basic read methods are:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close();           // Close an open stream
int available();       // Number of bytes available
long skip(long n);    // Discard n bytes from stream
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

InputStream Methods

The `read()` method returns an `int`, which contains either a byte read from the stream, or a `-1`, which indicates the end-of-file condition. The other two `read` methods read the stream into a byte array and return the number of bytes read. The two `int` arguments in the third method indicate a subrange in the target array that needs to be filled.

Note: For efficiency, always read data in the largest practical block, or use buffered streams.

When you have finished with a stream, close it. If you have a stack of streams, use filter streams to close the stream at the top of the stack. This operation also closes the lower streams.

`InputStream` implements `AutoCloseable`, which means that if you use an `InputStream` (or one of its subclasses) in a `try-with-resources` block, the stream is automatically closed at the end of the `try`.

The `available` method reports the number of bytes that are immediately available to be read from the stream. An actual read operation following this call might return more bytes.

The `skip` method discards the specified number of bytes from the stream.

Byte Stream OutputStream Methods

- The three basic write methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close(); // Automatically closed in try-with-resources
void flush(); // Force a write to the stream
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

OutputStream Methods

As with input, always try to write data in the largest practical block.

Byte Stream: Example

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.io.FileNotFoundException; import java.io.IOException;
3
4 public class ByteStreamCopyTest {
5     public static void main(String[] args) {
6         byte[] b = new byte[128];
7         // Example use of InputStream methods
8         try (FileInputStream fis = new FileInputStream (args[0]);
9              FileOutputStream fos = new FileOutputStream (args[1])) {
10             System.out.println ("Bytes available: " + fis.available());
11             int count = 0; int read = 0;
12             while ((read = fis.read(b)) != -1) {
13                 fos.write(b);
14                 count += read;
15             }
16             System.out.println ("Wrote: " + count);
17         } catch (FileNotFoundException f) {
18             System.out.println ("File not found: " + f);
19         } catch (IOException e) {
20             System.out.println ("IOException: " + e);
21         }
22     }
23 }
```

Note that you must keep track of how many bytes are read into the byte array each time.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This example copies one file to another by using a byte array. Note that `FileInputStream` and `FileOutputStream` are meant for streams of raw bytes, such as image files.

Note: The `available()` method, according to Javadocs, reports "an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking."

Character Stream Reader Methods

- The three basic read methods are:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Other methods include:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reader Methods

The first method returns an int, which contains either a Unicode character read from the stream, or a -1, which indicates the end-of-file condition. The other two methods read into a character array and return the number of bytes read. The two int arguments in the third method indicate a subrange in the target array that needs to be filled.

Character Stream Writer Methods

- The basic write methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Writer Methods

These methods are analogous to the OutputStream methods.

Character Stream: Example

```
1 import java.io.FileReader; import java.io.FileWriter;
2 import java.io.IOException; import java.io.FileNotFoundException;
3
4 public class CharStreamCopyTest {
5     public static void main(String[] args) {
6         char[] c = new char[128];
7         // Example use of InputStream methods
8         try (FileReader fr = new FileReader(args[0]));
9             FileWriter fw = new FileWriter(args[1])) {
10             int count = 0;
11             int read = 0;
12             while ((read = fr.read(c)) != -1) {
13                 fw.write(c);
14                 count += read;
15             }
16             System.out.println("Wrote: " + count + " characters.");
17         } catch (FileNotFoundException f) {
18             System.out.println("File " + args[0] + " not found.");
19         } catch (IOException e) {
20             System.out.println("IOException: " + e);
21         }
22     }
23 }
```

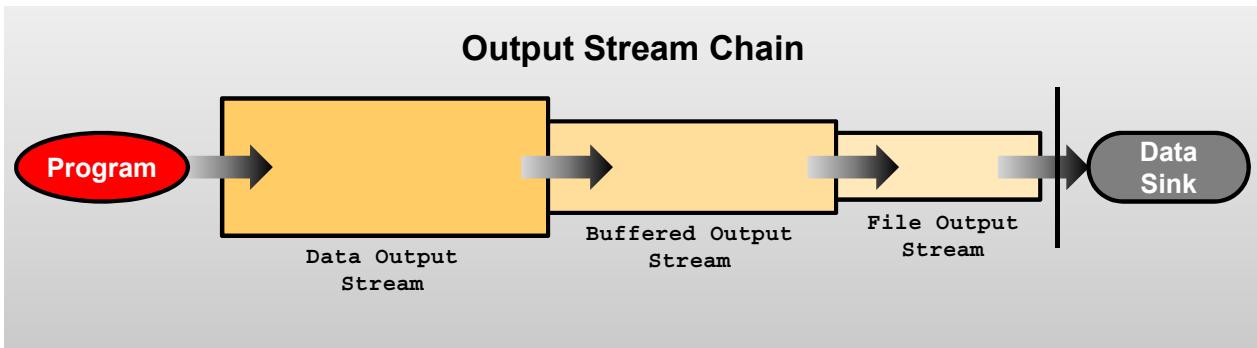
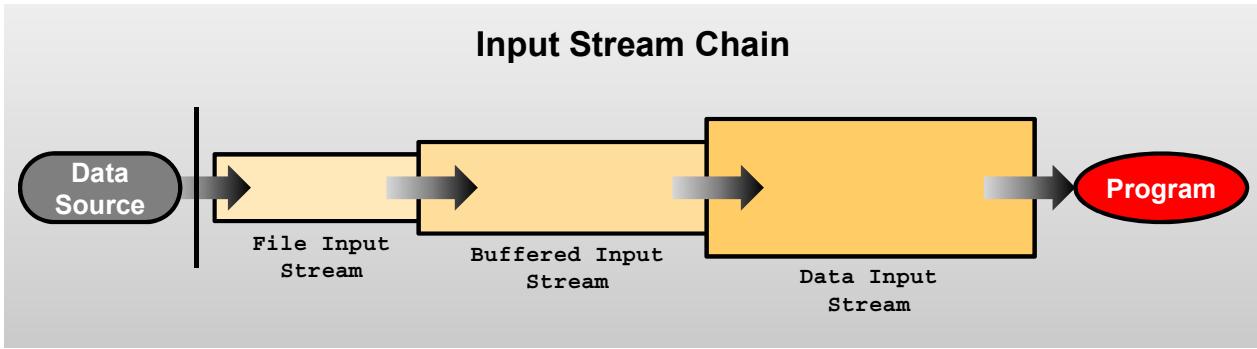
Now, rather than a byte array, this version uses a character array.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Similar to the byte stream example, this example copies one file to another by using a character array instead of a byte array. `FileReader` and `FileWriter` are classes designed to read and write character streams, such as text files.

I/O Stream Chaining



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A program rarely uses a single stream object. Instead, it chains a series of streams together to process the data. The first graphic in the slide demonstrates an example of input stream; in this case, a file stream is buffered for efficiency and then converted into data (Java primitives) items. The second graphic demonstrates an example of output stream; in this case, data is written, then buffered, and finally written to a file.

C chained Streams: Example

```
1 import java.io.BufferedReader; import java.io.BufferedWriter;
2 import java.io.FileReader; import java.io.FileWriter;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class BufferedStreamCopyTest {
6     public static void main(String[] args) {
7         try (BufferedReader bufInput
8              = new BufferedReader(new FileReader(args[0])));
9             BufferedWriter bufOutput
10            = new BufferedWriter(new FileWriter(args[1]))) {
11             String line = "";
12             while ((line = bufInput.readLine()) != null) {
13                 bufOutput.write(line);
14                 bufOutput.newLine();
15             }
16         } catch (FileNotFoundException f) {
17             System.out.println("File not found: " + f);
18         } catch (IOException e) {
19             System.out.println("Exception: " + e);
20         }
21     }
22}
```

A FileReader chained to a
BufferedFileReader: This allows you
to use a method that reads a String.

The character buffer replaced
by a String. Note that
readLine() uses the newline
character as a terminator.
Therefore, you must add that
back to the output file.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide shows the copy application one more time. This version illustrates the use of a BufferedReader chained to the BufferedReader that you saw before.

The flow of this program is the same as before. Instead of reading a character buffer, this program reads a line at a time using the line variable to hold the String returned by the readLine method, which provides greater efficiency. The reason is that each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. A BufferedReader reads characters from the stream into a buffer. (The size of the buffer can be set, but the default value is generally sufficient.)

Console I/O

The `System` class in the `java.lang` package has three static instance fields: `out`, `in`, and `err`.

- The `System.out` field is a static instance of a `PrintStream` object that enables you to write to *standard output*.
- The `System.in` field is a static instance of an `InputStream` object that enables you to read from *standard input*.
- The `System.err` field is a static instance of a `PrintStream` object that enables you to write to *standard error*.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Console I/O Using `System`

- `System.out` is the “standard” output stream. This stream is already open and ready to accept output data. Typically, this stream corresponds to display output or another output destination specified by the host environment or user.
- `System.in` is the “standard” input stream. This stream is already open and ready to supply input data. Typically, this stream corresponds to keyboard input or another input source specified by the host environment or user.
- `System.err` is the “standard” error output stream. This stream is already open and ready to accept output data.
Typically, this stream corresponds to display output or another output destination specified by the host environment or user. By convention, this output stream is used to display error messages or other information that should come to the immediate attention of a user even if the principal output stream, the value of the variable `out`, has been redirected to a file or other destination that is typically not continuously monitored.

Writing to Standard Output

- The `println` and `print` methods are part of the `java.io.PrintStream` class.
- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reading from Standard Input

```
7 public class KeyboardInput {  
8  
9     public static void main(String[] args) {  
10         String s = "";  
11         try (BufferedReader in = new BufferedReader(new  
12             InputStreamReader(System.in))) {  
13             System.out.print("Type xyz to exit: ");  
14             s = in.readLine();  
15             while (s != null) {  
16                 System.out.println("Read: " + s.trim());  
17                 if (s.equals("xyz")) {  
18                     System.exit(0);  
19                 }  
20                 System.out.print("Type xyz to exit: ");  
21                 s = in.readLine();  
22             } catch (IOException e) { // Catch any IO exceptions.  
23                 System.out.println("Exception: " + e);  
24             }  
25         }  
26     }
```

Chain a buffered reader to an input stream that takes the console input.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `try-with-resources` statement on line 6 opens `BufferedReader`, which is chained to an `InputStreamReader`, which is chained to the static standard console input `System.in`.

If the string read is equal to “xyz,” the program exits. The purpose of the `trim()` method on the `String` returned by `in.readLine` is to remove any whitespace characters.

Note: A null string is returned if an end of stream is reached (the result of a user pressing `Ctrl + C` in Windows, for example), thus the test for null on line 13.

Channel I/O

Introduced in JDK 1.4, a channel reads bytes and characters in blocks, rather than one byte or character at a time.

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.nio.channels.FileChannel; import java.nio.ByteBuffer;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class ByteChannelCopyTest {
6     public static void main(String[] args) {
7         try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
8              FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
9             ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
10            fcIn.read(buff);
11            buff.position(0);
12            fcOut.write(buff);
13        } catch (FileNotFoundException f) {
14            System.out.println("File not found: " + f);
15        } catch (IOException e) {
16            System.out.println("IOException: " + e);
17        }
18    }
19 }
```

Create a buffer sized the same as
the file size, and then read and write
the file in a single operation.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, a file can be read in its entirety into a buffer, and then written out in a single operation.

Channel I/O was introduced in the `java.nio` package in JDK 1.4.

Persistence

Saving data to some type of permanent storage is called persistence. An object that is persistent-capable can be stored on disk (or any other storage device), or sent to another machine to be stored there.

- A non-persisted object exists only as long as the Java Virtual Machine is running.
- Java serialization is the standard mechanism for saving an object as a sequence of bytes that can later be rebuilt into a copy of the object.
- To serialize an object of a specific class, the class must implement the `java.io.Serializable` interface.

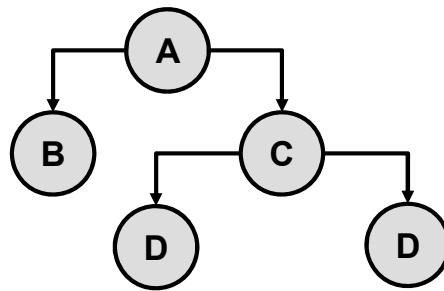


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `java.io.Serializable` interface defines no methods, and serves only as a marker to indicate that the class should be considered for serialization.

Serialization and Object Graphs

- When an object is serialized, only the fields of the object are preserved.
- When a field references an object, the fields of the referenced object are also serialized, if that object's class is also serializable.
- The tree of an object's fields constitutes the *object graph*.



ORACLE®

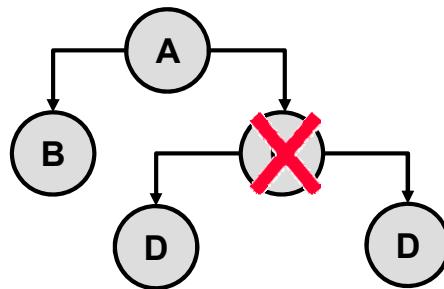
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Object Graphs

Serialization traverses the object graph and writes that data to the file (or other output stream) for each node of the graph.

Transient Fields and Objects

- Some object classes are not serializable because they represent transient operating system–specific information.
- If the object graph contains a non-serializable reference, a `NotSerializableException` is thrown and the serialization operation fails.
- Fields that should not be serialized or that do not need to be serialized can be marked with the keyword `transient`.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Transient

If a field containing an object reference is encountered that is not marked as serializable (`implement java.io.Serializable`), a `NotSerializableException` is thrown and the entire serialization operation fails. To serialize a graph containing fields that reference objects that are not serializable, those fields must be marked using the keyword `transient`.

Transient: Example

```
public class Portfolio implements Serializable {  
    public transient FileInputStream inputFile;  
    public static int BASE = 100; // static fields are not serialized.  
    private transient int totalValue = 10;  
    protected Stock[] stocks; // Serialization includes all of the members of the stocks array.  
}
```

- The field access modifier has no effect on the data field being serialized.
- The values stored in static fields are not serialized.
- When an object is deserialized, the values of static fields are set to the values declared in the class. The value of non-static transient fields is set to the default value for the type.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When an object is deserialized, the values of static and transient fields are set to the values defined in the class declaration. The values of non-static fields are set to the default value of their type. So in the example shown in the slide, the value of BASE will be 100, per the class declaration. The value of non-static transient fields, inputFile and totalValue, are set to their default values, null and 0, respectively.

Serial Version UID

- During serialization, a version number, serialVersionUID, is used to associate the serialized output with the class used in the serialization process.
- After deserialization, the serialVersionUID is checked to verify that the classes loaded are compatible with the object being deserialized.
- If the receiver of a serialized object has loaded classes for that object with different serialVersionUID, deserialization will result in an `InvalidClassException`.
- A serializable class can declare its own serialVersionUID by explicitly declaring a field named `serialVersionUID` as a static final and of type long:

```
private static long serialVersionUID = 42L;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

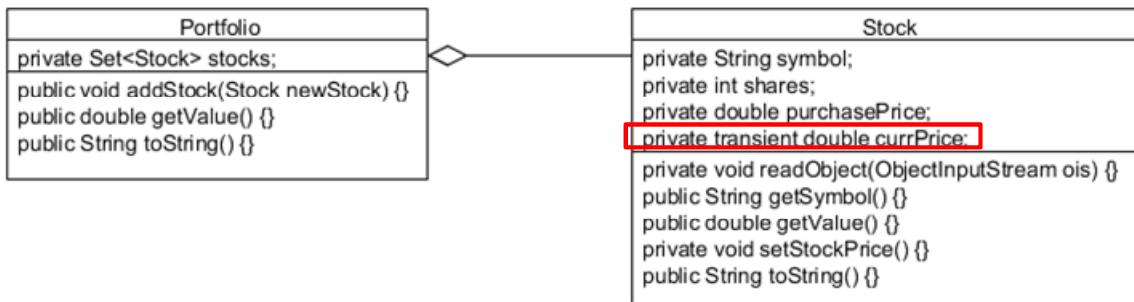
Note: The documentation for `java.io.Serializable` states the following:

"If a serializable class does not explicitly declare a serialVersionUID, then the serialization run time will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different Java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members. Array classes cannot declare an explicit serialVersionUID, so they always have the default computed value, but the requirement for matching serialVersionUID values is waived for array classes."

Serialization: Example

In this example, a Portfolio is made up of a set of Stocks.

- During serialization, the current price is not serialized, and is, therefore, marked transient.
- However, the current value of the stock should be set to the current market price after deserialization.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Writing and Reading an Object Stream

```

1 public static void main(String[] args) {
2     Stock s1 = new Stock("ORCL", 100, 32.50);
3     Stock s2 = new Stock("APPL", 100, 245);
4     Stock s3 = new Stock("GOOG", 100, 54.67);
5     Portfolio p = new Portfolio(s1, s2, s3);
6     try (FileOutputStream fos = new FileOutputStream(args[0]));
7         ObjectOutputStream out = new ObjectOutputStream(fos)) {
8         out.writeObject(p);           <-----> Portfolio is the root
9     } catch (IOException i) {      object.
10        System.out.println("Exception writing out Portfolio: " + i);
11    }
12    try (InputStream fis = new FileInputStream(args[0]));
13        ObjectInputStream in = new ObjectInputStream(fis)) {
14        Portfolio newP = (Portfolio)in.readObject(); <-----> The readObject method
15    } catch (ClassNotFoundException | IOException i) {      restores the object from
16        System.out.println("Exception reading in Portfolio: " + i);  the file stream.
17    }

```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The SerializeStock class.

- **Line 6 – 8:** A `FileOutputStream` is chained to an `ObjectOutputStream`. This allows the raw bytes generated by the `ObjectOutputStream` to be written to a file through the `writeObject` method. This method walks the object's graph and writes the data contained in the non-transient and non-static fields as raw bytes.
- **Line 12 – 14:** To restore an object from a file, a `FileInputStream` is chained to an `ObjectInputStream`. The raw bytes read by the `readObject` method restore an `Object` containing the non-static and non-transient data fields. This `Object` must be cast to expected type.

Serialization Methods

An object being serialized (and deserialized) can control the serialization of its own fields.

```
public class MyClass implements Serializable {  
    // Fields  
    private void writeObject(ObjectOutputStream oos) throws IOException {  
        oos.defaultWriteObject();  
        // Write/save additional fields  
        oos.writeObject(new java.util.Date());  
    }  
}
```

defaultWriteObject is called to perform the serialization of this class' fields.

- For example, in this class, the current time is written into the object graph.
- During deserialization, a similar method is invoked:

```
private void readObject(ObjectInputStream ois) throws  
ClassNotFoundException, IOException {}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `writeObject` method is invoked on the object being serialized. If the object does not contain this method, the `defaultWriteObject` method is invoked instead.

- This method must also be called once and only once from the object's `writeObject` method.

During deserialization, the `readObject` method is invoked on the object being deserialized (if present in the class file of the object). The signature of the method is important.

```
private void readObject(ObjectInputStream ois) throws  
    ClassNotFoundException, IOException {  
    ois.defaultReadObject();  
    // Print the date this object was serialized  
    System.out.println ("Restored from date: " +  
        (java.util.Date)ois.readObject());  
}
```

readObject: Example

```
1 public class Stock implements Serializable {  
2     private static final long serialVersionUID = 100L;  
3     private String symbol;  
4     private int shares;  
5     private double purchasePrice;  
6     private transient double currPrice;  
7  
8     public Stock(String symbol, int shares, double purchasePrice) {  
9         this.symbol = symbol;  
10        this.shares = shares;  
11        this.purchasePrice = purchasePrice;  
12        setStockPrice();  
13    }  
14  
15    // This method is called post-serialization  
16    private void readObject(ObjectInputStream ois)  
17        throws IOException, ClassNotFoundException {  
18        ois.defaultReadObject();  
19        // perform other initialization  
20        setStockPrice();  
21    }  
22 }
```

Stock currPrice is set by the setStockPrice method during creation of the Stock object, but the constructor is not called during deserialization.

Stock currPrice is set after the other fields are serialized.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the Stock class, the `readObject` method is provided to ensure that the stock's `currPrice` is set (by the `setStockPrice` method) after deserialization of the Stock object.

Note: The signature of the `readObject` method is critical for this method to be called during deserialization.

Summary

In this lesson, you should have learned how to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use streams to read and write files
- Write and read objects by using serialization



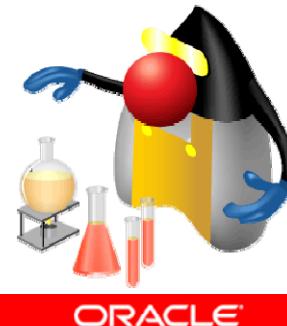
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 13-1 Overview: Writing a Simple Console I/O Application

This practice covers the following topics:

- Writing a main class that accepts a file name as an argument
- Using `System console` I/O to read a search string
- Using stream chaining to use the appropriate method to search for the string in the file and report the number of occurrences
- Continuing to read from the console until an exit sequence is entered



ORACLE

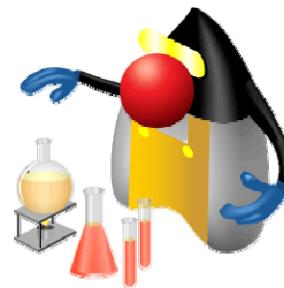
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this practice, you will write the code necessary to read a file name as an application argument, and use the `System console` to read from standard input until a termination character is typed in.

Practice 13-2 Overview: Serializing and Deserializing a ShoppingCart

This practice covers the following topics:

- Creating an application that serializes a ShoppingCart object that is composed of an ArrayList of Item objects
- Using the transient keyword to prevent the serialization of the ShoppingCart total. This will allow items to vary their cost.
- Using the writeObject method to store today's date on the serialized stream
- Using the readObject method to recalculate the total cost of the cart after deserialization and print the date that the object was serialized



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

The purpose of chaining streams together is to:

- a. Allow the streams to add functionality
- b. Change the direction of the stream
- c. Modify the access of the stream
- d. Meet the requirements of JDK 7



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

To prevent the serialization of operating system–specific fields, you should mark the field:

- a. private
- b. static
- c. transient
- d. final



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Given the following fragments:

```
public MyClass implements Serializable {  
    private String name;  
    private static int id = 10;  
    private transient String keyword;  
    public MyClass(String name, String keyword) {  
        this.name = name; this.keyword = keyword;  
    }  
}
```

```
MyClass mc = new MyClass ("Zim", "xyzzy");
```

Assuming no other changes to the data, what is the value of `name` and `keyword` fields after deserialization of the `mc` object instance?

- a. Zim, ""
- b. Zim, null
- c. Zim, xyzzy
- d. "", null



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and CAS TRAINING, S.L. use only

14

Java File I/O (NIO.2)

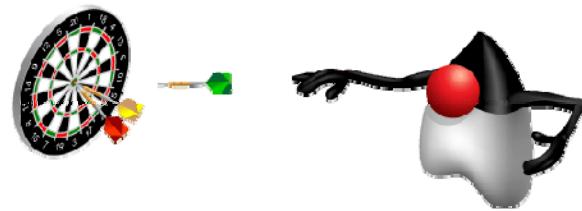
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

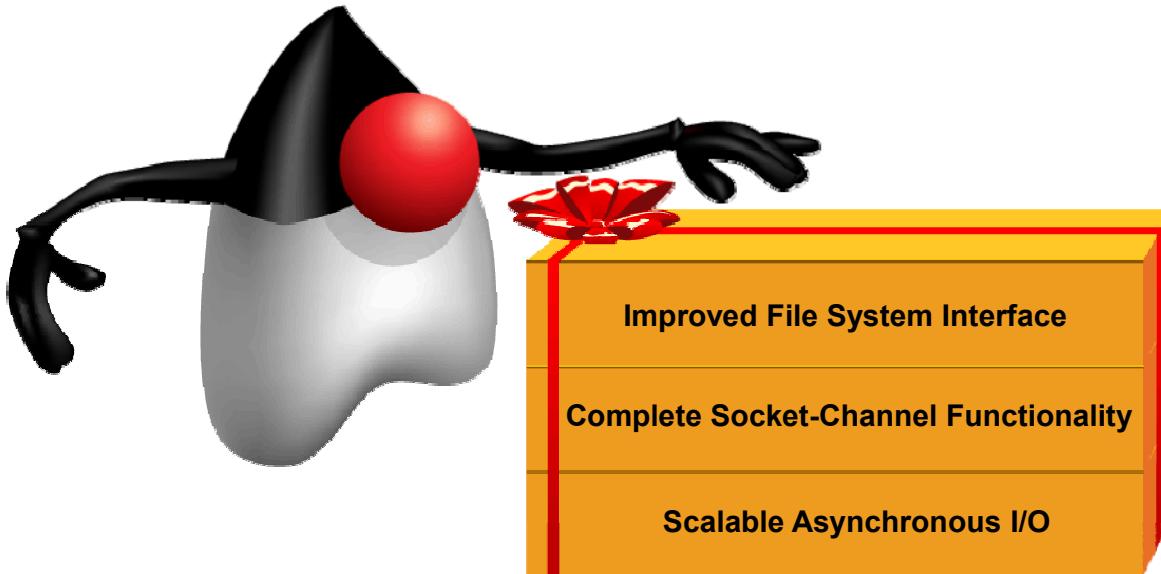
- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use Stream API with NIO2



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

New File I/O API (NIO.2)



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

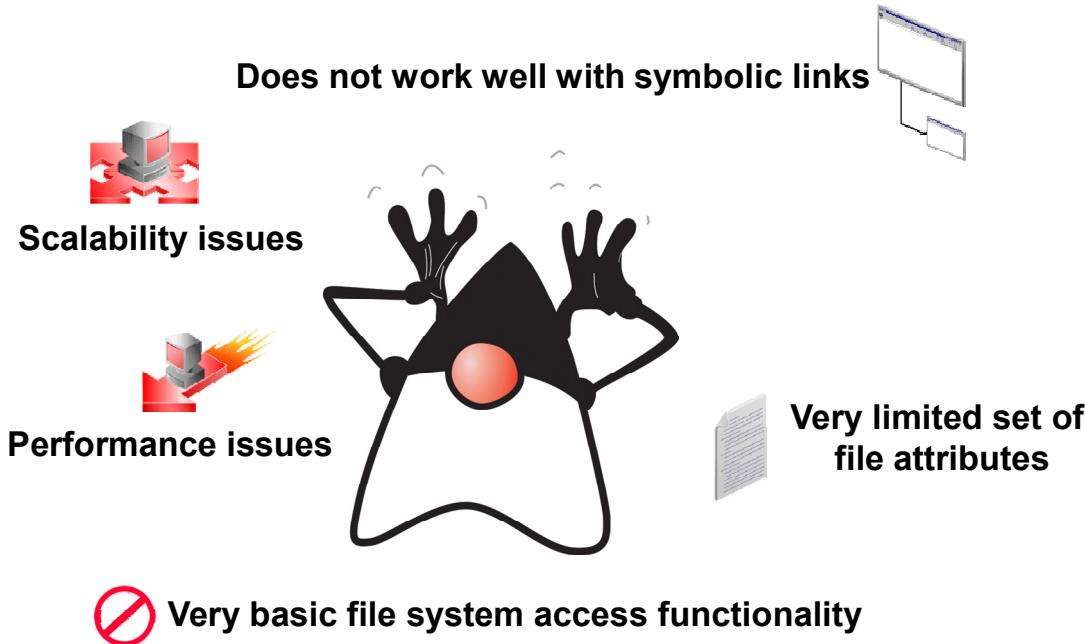
NIO API in JSR 51 established the basis for NIO in Java, focusing on buffers, channels, and charsets. JSR 51 delivered the first piece of the scalable socket I/Os into the platform, providing a non-blocking, multiplexed I/O API, thus allowing the development of highly scalable servers without having to resort to native code.

For many developers, the most significant goal of JSR 203 is to address issues with `java.io.File` by developing a new file system interface.

The new API:

- Works more consistently across platforms
- Makes it easier to write programs that gracefully handle the failure of file system operations
- Provides more efficient access to a larger set of file attributes
- Allows developers of sophisticated applications to take advantage of platform-specific features when absolutely necessary
- Allows support for non-native file systems, to be “plugged in” to the platform

Limitations of `java.io.File`



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

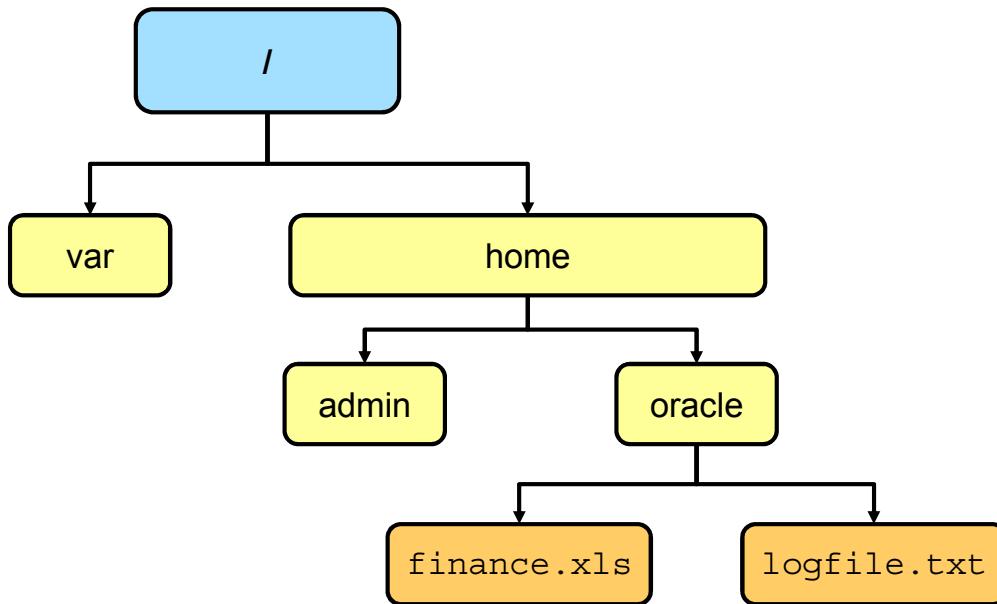
The Java I/O File API (`java.io.File`) presented challenges for developers.

- Many methods did not throw exceptions when they failed, so it was impossible to obtain a useful error message.
- Several operations were missing (file copy, move, and so on).
- The rename method did not work consistently across platforms.
- There was no real support for symbolic links.
- More support for metadata was desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata was inefficient—every call for metadata resulted in a system call, which made the operations very inefficient.
- Many of the File methods did not scale. Requesting a large directory listing on a server could result in a hang.
- It was not possible to write reliable code that could recursively walk a file tree and respond appropriately if there were circular symbolic links.

Further, the overall I/O was not written to be extended. Developers had requested the ability to develop their own file system implementations. For example, by keeping a pseudofile system in memory, or by formatting files as zip files.

File Systems, Paths, Files

In NIO.2, both files and directories are represented by a path, which is the relative or absolute location of the file or directory.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

File Systems

Prior to the NIO.2 implementation in JDK 7, files were represented by the `java.io.File` class.

In NIO.2, instances of `java.nio.file.Path` objects are used to represent the relative or absolute location of a file or directory.

File systems are hierarchical (tree) structures. File systems can have one or more root directories. For example, typical Windows machines have at least two disk root nodes: C:\ and D:\.

Note that file systems may also have different characteristics for path separators, as shown in the slide.

Relative Path Versus Absolute Path

- A path is either *relative* or *absolute*.
- An absolute path always contains the root element and the complete directory list required to locate the file.
- Example:

```
...  
/home/peter/statusReport  
...
```

- A relative path must be combined with another path in order to access a file.
- Example:

```
...  
clarence/foo  
...
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A path can either be relative or absolute. An absolute path always contains the root element and the complete directory list required to locate the file. For example, `/home/peter/statusReport` is an absolute path. All the information needed to locate the file is contained in the path string.

A relative path must be combined with another path in order to access a file. For example, `clarence/foo` is a relative path. Without more information, a program cannot reliably locate the `clarence/foo` directory in the file system.

Java NIO.2 Concepts

Prior to JDK 7, the `java.io.File` class was the entry point for all file and directory operations. With NIO.2, there is a new package and classes:

- `java.nio.file.Path`: Locates a file or a directory by using a system-dependent path
- `java.nio.file.Files`: Using a `Path`, performs operations on files and directories
- `java.nio.file.FileSystem`: Provides an interface to a file system and a factory for creating a `Path` and other objects that access a file system
- All the methods that access the file system throw `IOException` or a subclass.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java NIO.2

A significant difference between NIO.2 and `java.io.File` is the architecture of access to the file system. With the `java.io.File` class, the methods used to manipulate path information are in the same class with methods used to read and write files and directories.

In NIO.2, the two concerns are separated. Paths are created and manipulated using the `Path` interface, while operations on files and directories is the responsibility of the `Files` class, which operates only on `Path` objects.

Finally, unlike `java.io.File`, `Files` class methods that operate directly on the file system, throw an `IOException` (or a subclass). Subclasses provide details on what the cause of the exception was.

Path Interface

- The `java.nio.file.Path` interface provides the entry point for the NIO.2 file and directory manipulation.

```
FileSystem fs = FileSystems.getDefault();  
Path p1 = fs.getPath ("/home/oracle/labs/resources/myFile.txt");
```

- To obtain a `Path` object, obtain an instance of the default file system, and then invoke the `getPath` method:

```
Path p1 = Paths.get ("/home/oracle/labs/resources/myFile.txt");  
Path p2 = Paths.get ("/home/oracle", "labs", "resources", "myFile.txt");
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Path Interface Features

The `Path` interface defines the methods used to locate a file or a directory in a file system. These methods include:

- To access the components of a path:
 - `getFileName`, `getParent`, `getRoot`, `getNameCount`
- To operate on a path:
 - `normalize`, `toUri`, `toAbsolutePath`, `subpath`,
`resolve`, `relativize`
- To compare paths:
 - `startsWith`, `endsWith`, `equals`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Path Objects Are Like String Objects

It is best to think of `Path` objects in the same way you think of `String` objects. `Path` objects can be created from a single text string, or a set of components:

- A *root component*, that identifies the file system hierarchy
- A *name element*, farthest from the root element, that defines the file or directory the path points to
- Additional elements may be present as well, separated by a special character or delimiter that identify directory names that are part of the hierarchy.

`Path` objects are immutable. Once created, operations on `Path` objects return new `Path` objects.

Path: Example

```
1 public class PathTest
2     public static void main(String[] args) {
3         Path p1 = Paths.get(args[0]);
4         System.out.format("getFileName: %s%n", p1.getFileName());
5         System.out.format("getParent: %s%n", p1.getParent());
6         System.out.format("getNameCount: %d%n", p1.getNameCount());
7         System.out.format("getRoot: %s%n", p1.getRoot());
8         System.out.format("isAbsolute: %b%n", p1.isAbsolute());
9         System.out.format("toAbsolutePath: %s%n", p1.toAbsolutePath());
10        System.out.format("toURI: %s%n", p1.toUri());
11    }
12 }
```

```
java PathTest /home/oracle/file1.txt
getFileName: file1.txt
getParent: /home/oracle
getNameCount: 3
getRoot: /
isAbsolute: true
toAbsolutePath: /home/oracle/file1.txt
toURI: file:///home/oracle/file1.txt
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Unlike the `java.io.File` class, files and directories are represented by instances of `Path` objects in a *system-dependent* way.

The `Path` interface provides several methods for reporting information about the path:

- `Path getFileName`: The end point of this `Path`, returned as a `Path` object
- `Path getParent`: The parent path or null. Everything in `Path` up to the file name (file or directory)
- `int getNameCount`: The number of name elements that make up this path
- `Path getRoot`: The root component of this `Path`
- `boolean isAbsolute`: true if this path contains a system-dependent root element.
Note: Because this example is being run on a Windows machine, the *system-dependent* root element contains a drive letter and colon. On a UNIX-based OS, `isAbsolute` returns true for any path that begins with a slash.
- `Path toAbsolutePath`: Returns a path representing the absolute path of this path
- `java.net.URI toUri`: Returns an absolute URI

Note: A `Path` object can be created for any path. The actual file or directory need not exist.

Removing Redundancies from a Path

- Many file systems use “.” notation to denote the current directory and “..” to denote the parent directory.
- The following examples both include redundancies:

```
/home/./clarence/foo  
/home/peter/../clarence/foo
```

- The `normalize` method removes any redundant elements, which includes any “.” or “directory/..” occurrences.
- Example:

```
Path p = Paths.get("/home/peter/../clarence/foo");  
Path normalizedPath = p.normalize();
```

```
/home/clarence/foo
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Many file systems use “.” notation to denote the current directory and “..” to denote the parent directory. You might have a situation where a Path contains redundant directory information. Perhaps a server is configured to save its log files in the “/dir/logs/.” directory, and you want to delete the trailing “/.” notation from the path.

The `normalize` method removes any redundant elements, which includes any “.” or “directory/..” occurrences. The slide examples would be normalized to `/home/clarence/foo`.

It is important to note that `normalize` does not check the file system when it cleans up a path. It is a purely syntactic operation. In the second example, if `peter` were a symbolic link, removing `peter/..` might result in a path that no longer locates the intended file.

Creating a Subpath

- A portion of a path can be obtained by creating a subpath using the `subpath` method:

```
Path subpath(int beginIndex, int endIndex);
```

- The element returned by `endIndex` is one less than the `endIndex` value.
- Example:

home= 0
oracle = 1
Temp = 2

```
Path p1 = Paths.get ("/home/oracle/Temp/foo/bar");  
Path p2 = p1.subpath (1, 3);
```

oracle/Temp

Include the element at index 2.

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The element name closest to the root has index 0.

The element farthest from the root has index `count - 1`.

Note: The returned `Path` object has the name elements that begin at `beginIndex` and extend to the element at index `endIndex - 1`.

Joining Two Paths

- The `resolve` method is used to combine two paths.
- Example:

```
Path p1 = Paths.get("/home/clarence/foo");
p1.resolve("bar");      // Returns /home/clarence/foo/bar
```

- Passing an absolute path to the `resolve` method returns the passed-in path.

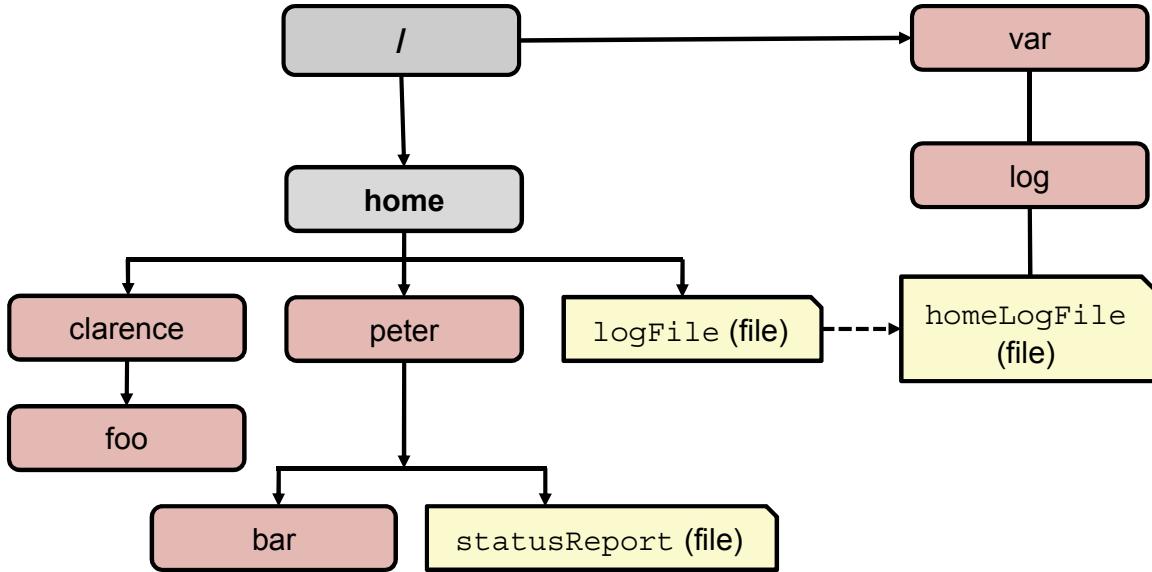
```
Paths.get("foo").resolve("/home/clarence"); // Returns /home/clarence
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `resolve` method is used to combine paths. It accepts a partial path, which is a path that does not include a root element, and that partial path is appended to the original path.

Symbolic Links



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

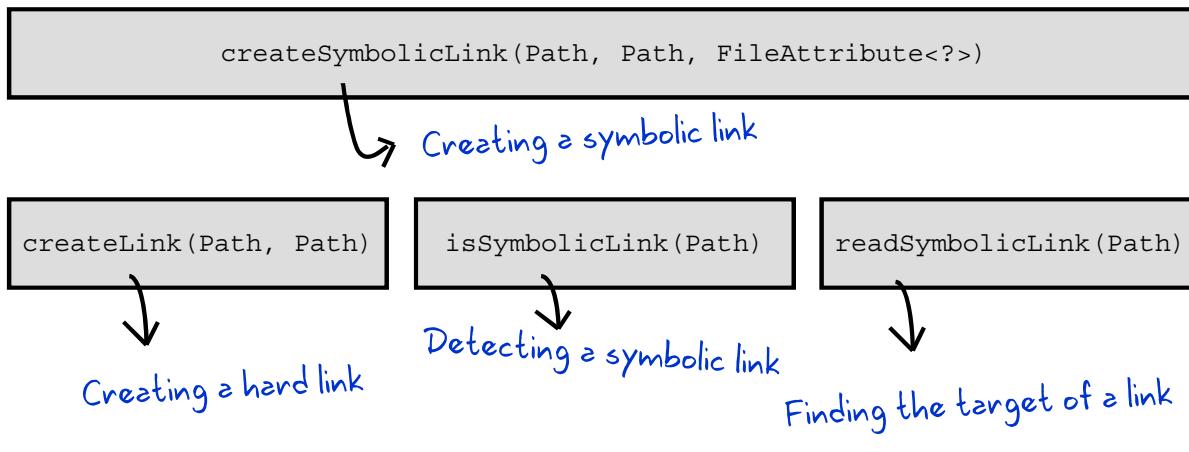
File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a “symlink” or a “soft link.”

A symbolic link is a special file that serves as a reference to another file. A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

In the slide’s diagram, `logFile` appears to the user to be a regular file, but it is actually a symbolic link to `/var/log/homeLogFile`. `homeLogFile` is the target of the link.

Working with Links

- Path interface is “link aware.”
- Every Path method either:
 - Detects what to do when a symbolic link is encountered, or
 - Provides an option enabling you to configure the behavior when a symbolic link is encountered



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

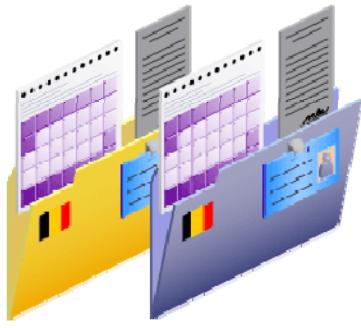
The `java.nio.file` package and the `Path` interface in particular are “link aware.” Every `Path` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

Some file systems also support hard links. Hard links are more restrictive than symbolic links, as follows:

- The target of the link must exist.
- Hard links are generally not allowed on directories.
- Hard links are not allowed to cross partitions or volumes. Therefore, they cannot exist across file systems.
- A hard link looks, and behaves, like a regular file, so they can be hard to find.
- A hard link is, for all intents and purposes, the same entity as the original file. They have the same file permissions, time stamps, and so on. All attributes are identical.

Because of these restrictions, hard links are not used as often as symbolic links, but the `Path` methods work seamlessly with hard links.

File Operations



Checking a File or Directory

Deleting a File or Directory

Copying a File or Directory

Moving a File or Directory

Managing Metadata

Reading, Writing, and Creating Files

Random Access Files

Creating and Reading Directories

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `java.nio.file.Files` class is the primary entry point for operations on `Path` objects.

Static methods in this class read, write, and manipulate files and directories represented by `Path` objects.

The `Files` class is also link aware—methods detect symbolic links in `Path` objects and automatically manage links or provide options for dealing with links.

Checking a File or Directory

A Path object represents the concept of a file or a directory location. Before you can access a file or directory, you should first access the file system to determine whether it exists using the following Files methods:

- `exists(Path p, LinkOption... option)`
Tests to see whether a file exists. By default, symbolic links are followed.
- `notExists(Path p, LinkOption... option)`
Tests to see whether a file does not exist. By default, symbolic links are followed.
- Example:

```
Path p = Paths.get(args[0]);  
System.out.format("Path %s exists: %b%n", p,  
                  Files.exists(p, LinkOption.NOFOLLOW_LINKS));
```

Optional argument

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Recall that Path objects may point to files or directories that do not exist. The `exists()` and `notExists()` methods are used to determine whether the Path points to a legitimate file or directory, and the particulars of that file or directory.

When testing for the existence of a file, there are three outcomes possible:

- The file is verified to exist.
- The file is verified to not exist.
- The file's status is unknown. This result can occur when the program does not have access to the file.

Note: `!Files.exists(path)` is not equivalent to `Files.notExists(path)`. If both `exists` and `notExists` return false, the existence of the file or directory cannot be determined. For example, in Windows, it is possible to achieve this by requesting the status of an offline drive, such as a CD-ROM drive.

Checking a File or Directory

To verify that a file can be accessed, the `Files` class provides the following boolean methods.

- `isReadable(Path)`
- `isWritable(Path)`
- `isExecutable(Path)`

Note that these tests are not atomic with respect to other file system operations. Therefore, the results of these tests may not be reliable once the methods complete.

- The `isSameFile(Path, Path)` method tests to see whether two paths point to the same file. This is particularly useful in file systems that support symbolic links.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The result of any of these tests is immediately outdated once the operation completes. According to the documentation: “Note that result of this method is immediately outdated. There is no guarantee that a subsequent attempt to open the file for writing will succeed (or even that it will access the same file). Care should be taken when using this method in security-sensitive applications.”

Creating Files and Directories

Files and directories can be created using one of the following methods:

```
Files.createFile (Path dir);  
Files.createDirectory (Path dir);
```

- The `createDirectories` method can be used to create directories that do not exist, from top to bottom:

```
Files.createDirectories(Paths.get ("/home/oracle/Temp/foo/bar/example"));
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `Files` class also has methods to create temporary files and directories, hard links, and symbolic links.

Deleting a File or Directory

You can delete files, directories, or links. The `Files` class provides two methods:

- `delete(Path)`
- `deleteIfExists(Path)`

```
//...
Files.delete(path);
//...
```

Throws a `NoSuchFileException`,
`DirectoryNotEmptyException`, or
`IOException`

```
//...
Files.deleteIfExists(Path)
//...
```

No exception thrown

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `delete(Path)` method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist, a `NoSuchFileException` is thrown.

The `deleteIfExists(Path)` method also deletes the file, but if the file does not exist, no exception is thrown. Failing silently is useful when you have multiple threads deleting files and you do not want to throw an exception just because one thread did so first.

Copying a File or Directory

- You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method.
- When directories are copied, the files inside the directory are not copied.

StandardCopyOption parameters

```
//...
copy(Path, Path, CopyOption...)
//...
```

REPLACE_EXISTING
COPY_ATTRIBUTES
NOFOLLOW_LINKS

- Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.copy(source, target, REPLACE_EXISTING, NOFOLLOW_LINKS);
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method. The copy fails if the target file exists, unless the `REPLACE_EXISTING` option is specified.

Directories can be copied. However, files inside the directory are not copied, so the new directory is empty even when the original directory contains files.

When copying a symbolic link, the target of the link is copied. If you want to copy the link itself, and not the contents of the link, specify either the `NOFOLLOW_LINKS` or `REPLACE_EXISTING` option.

The following `StandardCopyOption` and `LinkOption` enums are supported:

- **REPLACE_EXISTING**: Performs the copy even when the target file already exists. If the target is a symbolic link, the link itself is copied (and not the target of the link). If the target is a non-empty directory, the copy fails with the `FileAlreadyExistsException` exception.
- **COPY_ATTRIBUTES**: Copies the file attributes associated with the file to the target file. The exact file attributes supported are file system- and platform-dependent, but last-modified-time is supported across platforms and is copied to the target file.
- **NOFOLLOW_LINKS**: Indicates that symbolic links should not be followed. If the file to be copied is a symbolic link, the link is copied (and not the target of the link).

Moving a File or Directory

- You can move a file or directory by using the `move (Path, Path, CopyOption...)` method.
- Moving a directory will not move the contents of the directory.

StandardCopyOption parameters

```
//...
move(Path, Path, CopyOption...)
//...
```

**REPLACE_EXISTING
ATOMIC_MOVE**

- Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.move(source, target, REPLACE_EXISTING);
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Guidelines for moves:

- If the target path is a directory and that directory is empty, the move succeeds if `REPLACE_EXISTING` is set.
- If the target directory does not exist, the move succeeds. Essentially, this is a rename of the directory.
- If the target directory exists and is not empty, a `DirectoryNotEmptyException` is thrown.
- If the source is a file and the target is a directory that exists, and `REPLACE_EXISTING` is set, the move will rename the file to the intended directory name.

To move a directory containing files to another directory, essentially you need to recursively copy the contents of the directory, and then delete the old directory.

You can also perform the move as an atomic file operation using `ATOMIC_MOVE`.

- If the file system does not support an atomic move, an exception is thrown. With an `ATOMIC_MOVE` you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

List the Contents of a Directory

To get a list of the files in the current directory, use the `Files.list()` method.

```
public class FileList {  
  
    public static void main(String[] args) {  
  
        try(Stream<Path> files = Files.list(Paths.get("."))) {  
  
            files  
                .forEach(line -> System.out.println(line));  
  
        } catch (IOException e) {  
            System.out.println("Message: " + e.getMessage());  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Walk the Directory Structure

The `Files.walk()` method walks a directory structure.

```
public class AllFileWalk {  
  
    public static void main(String[] args) {  
  
        try(Stream<Path> files = Files.walk(Paths.get("."))) {  
  
            files  
                .forEach(line -> System.out.println(line));  
  
        } catch (Exception e) {  
            System.out.println("Message: " + e.getMessage());  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

BufferedReader File Stream

The new `lines()` method converts a `BufferedReader` into a stream.

```
public class BufferedReader {
    public static void main(String[] args) {
        try(BufferedReader bReader =
            new BufferedReader(new FileReader("tempest.txt"))){

            bReader.lines()
                .forEach(line ->
                    System.out.println("Line: " + line));

            } catch (IOException e){
                System.out.println("Message: " + e.getMessage());
            }
        }
    }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

NIO File Stream

The `lines()` method can be called using NIO classes

```
public class ReadNio {  
  
    public static void main(String[] args) {  
  
        try(Stream<String> lines =  
            Files.lines(Paths.get("tempest.txt"))){  
  
            lines.forEach(line ->  
                System.out.println("Line: " + line));  
  
        } catch (IOException e){  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Read File into ArrayList

Use `readAllLines()` to load a file into an `ArrayList`.

```
public class ReadAllNio {  
    public static void main(String[] args) {  
        Path file = Paths.get("tempest.txt");  
        List<String> fileArr;  
  
        try{  
  
            fileArr = Files.readAllLines(file);  
  
            fileArr.stream()  
                .filter(line -> line.contains("PROSPERO"))  
                .forEach(line -> System.out.println(line));  
  
        } catch (IOException e){  
            System.out.println("Message: " + e.getMessage());  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Managing Metadata

Method	Explanation
size	Returns the size of the specified file in bytes
isDirectory	Returns true if the specified Path locates a file that is a directory
isRegularFile	Returns true if the specified Path locates a file that is a regular file
isSymbolicLink	Returns true if the specified Path locates a file that is a symbolic link
isHidden	Returns true if the specified Path locates a file that is considered hidden by the file system
getLastModifiedTime	Returns or sets the specified file's last modified time
setLastModifiedTime	
getAttribute	Returns or sets the value of a file attribute
setAttribute	

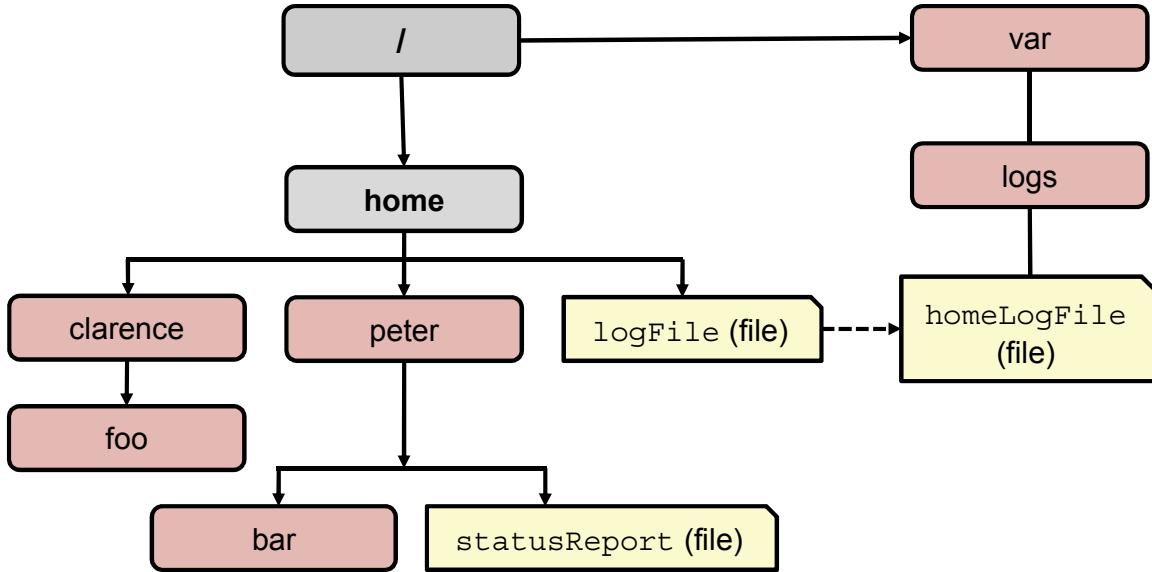


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If a program needs multiple file attributes around the same time, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance. For this reason, the `Files` class provides two `readAttributes` methods to fetch a file's attributes in one bulk operation.

- `readAttributes(Path, String, LinkOption...)`
- `readAttributes(Path, Class<A>, LinkOption...)`

Symbolic Links



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a “symlink” or a “soft link.”

A symbolic link is a special file that serves as a reference to another file. A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

In the slide’s diagram, `logFile` appears to the user to be a regular file, but it is actually a symbolic link to `dir/logs/homeLogFile`. `homeLogFile` is the target of the link.

Summary

In this lesson, you should have learned how to:

- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use Stream API with NIO2



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice Overview

Practice 14-1: Working with Files

In this practice, read text files using new features in Java 8 and the lines method.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice Overview

Practice 14-2: Working with Directories

In this practice, list directories and files using new features found in Java 8.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Given any starting directory path, which `FileVisitor` method(s) would you use to delete a file tree?

- a. `preVisitDirectory()`
- b. `postVisitDirectory()`
- c. `visitFile()`
- d. `visitDirectory()`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Given a Path object with the following path:

/export/home/duke/./peter/.documents

What Path method would remove the redundant elements?

- a. normalize
- b. relativize
- c. resolve
- d. toAbsolutePath



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Given the following fragment:

```
Path p1 = Paths.get("/export/home/peter");
Path p2 = Paths.get("/export/home/peter2");
Files.move(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

If the peter2 directory does not exist, and the peter directory is populated with subfolders and files, what is the result?

- a. DirectoryNotEmptyException
- b. NotDirectoryException
- c. Directory peter2 is created.
- d. Directory peter is copied to peter2.
- e. Directory peter2 is created and populated with files and directories from peter.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Given this fragment:

```
Path source = Paths.get(args[0]);  
Path target = Paths.get(args[1]);  
Files.copy(source, target);
```

Assuming source and target are not directories, how can you prevent this copy operation from generating `FileAlreadyExistsException`?

- a. Delete the target file before the copy.
- b. Use the move method instead.
- c. Use the `copyExisting` method instead.
- d. Add the `REPLACE_EXISTING` option to the method.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

To copy, move, or open a file or directory using NIO.2, you must first create an instance of:

- a. Path
- b. Files
- c. FileSystem
- d. Channel



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and CAS TRAINING, S.L. use only

15

Concurrency

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe operating system task scheduling
- Create worker threads using `Runnable` and `Callable`
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and `concurrent atomic` to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications. There are two types of tasks that can be scheduled for execution:

- **Processes:** A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.
- **Thread:** A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Preemptive Multitasking

Modern computers often have more tasks to execute than CPUs. Each task is given an amount of time (called a time slice) during which it can execute on a CPU. A time slice is usually measured in milliseconds. When the time slice has elapsed, the task is forcefully removed from the CPU and another task is given a chance to execute.

Legacy Thread and Runnable

Prior to Java 5, the `Thread` class was used to create and start threads. Code to be executed by a thread is placed in a class, which does either of the following:

- Extends the `Thread` class
 - Simpler code
- Implements the `Runnable` interface
 - More flexible
 - `extends` is still free.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Extending Thread

Extend `java.lang.Thread` and override the `run` method:

```
public class ExampleThread extends Thread {  
    @Override  
    public void run() {  
        for(int i = 0; i < 10; i++) {  
            System.out.println("i:" + i);  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `run` Method

The code to be executed in a new thread of execution should be placed in a `run` method. You should avoid calling the `run` method directly. Calling the `run` method does not start a new thread and the effect would be no different than calling any other method.

Implementing Runnable

Implement `java.lang.Runnable` and implement the `run` method:

```
public class ExampleRunnable implements Runnable {  
    private final String name;  
  
    public ExampleRunnable(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(name + ":" + i);  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `run` Method

Just as when extending `Thread`, calling the `run` method does not start a new thread. The benefit of implementing `Runnable` is that you may still extend a class of your choosing.

The `java.util.concurrent` Package

Java 5 introduced the `java.util.concurrent` package, which contains classes that are useful in concurrent programming. Features include:

- Concurrent collections
- Synchronization and locking alternatives
- Thread pools
 - Fixed and dynamic thread count pools available
 - Parallel divide and conquer (Fork-Join) new in Java 7



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Recommended Threading Classes

Traditional Thread related APIs are difficult to code properly.

Recommended concurrency classes include:

- `java.util.concurrent.ExecutorService`, a higher level mechanism used to execute tasks
 - It may create and reuse Thread objects for you.
 - It allows you to submit work and check on the results in the future.
- The Fork-Join framework, a specialized work-stealing `ExecutorService` new in Java 7



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

java.util.concurrent.ExecutorService

An ExecutorService is used to execute tasks.

- It eliminates the need to manually create and manage threads.
- Tasks **might** be executed in parallel depending on the ExecutorService implementation.
- Tasks can be:
 - java.lang.Runnable
 - java.util.concurrent.Callable
- Implementing instances can be obtained with Executors.

```
ExecutorService es = Executors.newCachedThreadPool();
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Behavior of an ExecutorService

A cached thread pool ExecutorService:

- Creates new threads as needed
- Reuses its threads (Its threads do not die after finishing their task.)
- Terminates threads that have been idle for 60 seconds

Other types of ExecutorService implementations are available:

```
int cpuCount = Runtime.getRuntime().availableProcessors();
ExecutorService es = Executors.newFixedThreadPool(cpuCount);
```

A fixed thread pool ExecutorService:

- Contains a fixed number of threads
- Reuses its threads (Its threads do not die after finishing their task.)
- Queues up work until a thread is available
- Could be used to avoid over working a system with CPU-intensive tasks

Example ExecutorService

This example illustrates using an ExecutorService to execute Runnable tasks:

```
package com.example;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        es.execute(new ExampleRunnable("one"));
        es.execute(new ExampleRunnable("two"));
        es.shutdown(); Execute this Runnable task sometime in the future
    }
}
```

Shut down the executor



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Shutting Down an ExecutorService

Shutting down an ExecutorService is important because its threads are nondaemon threads and will keep your JVM from shutting down.

```
es.shutdown();  
  
try {  
    es.awaitTermination(5, TimeUnit.SECONDS);  
} catch (InterruptedException ex) {  
    System.out.println("Stopped waiting early");  
}
```

Stop accepting new Callables.
If you want to wait for the Callables to finish



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

java.util.concurrent.Callable

The Callable interface:

- Defines a task submitted to an ExecutorService
- Is similar in nature to Runnable, but can:
 - Return a result using generics
 - Throw a checked exception

```
package java.util.concurrent;  
public interface Callable<V> {  
    V call() throws Exception;  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example Callable Task

```
public class ExampleCallable implements Callable {  
  
    private final String name;  
    private final int len;  
    private int sum = 0;  
  
    public ExampleCallable(String name, int len) {  
        this.name = name;  
        this.len = len;  
    }  
  
    @Override  
    public String call() throws Exception {  
        for (int i = 0; i < len; i++) {  
            System.out.println(name + ":" + i);  
            sum += i;  
        }  
        return "sum: " + sum;  
    }  
}
```

Return a String from this task: the sum of the series



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

java.util.concurrent.Future

The Future interface is used to obtain the results from a Callable's V call() method.

```
Future<V> future = es.submit(callable);  
//submit many callables  
try {  
    V result = future.get();  
} catch (ExecutionException|InterruptedException ex) {  
}  
  
If the Callable threw  
an Exception
```

ExecutorService controls when the work is done.

Gets the result of the Callable's call method (blocks if needed).



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Waiting on a Future

Because the call to Future.get() will block, you must do one of the following:

- Submit all your work to the ExecutorService before calling any Future.get() methods.
- Be prepared to wait for that Future to obtain the result.
- Use a non-blocking method such as Future.isDone() before calling Future.get() or use Future.get(long timeout, TimeUnit unit), which will throw a TimeoutException if the result is not available within a given duration.

Example

```
public static void main(String[] args) {  
  
    ExecutorService es = Executors.newFixedThreadPool(4);  
    Future<String> f1 = es.submit(new ExampleCallable("one",10));  
    Future<String> f2 = es.submit(new ExampleCallable("two",20));  
  
    try {  
        es.shutdown();  
        es.awaitTermination(5, TimeUnit.SECONDS);  
        String result1 = f1.get();  
        System.out.println("Result of one: " + result1);  
        String result2 = f2.get();  
        System.out.println("Result of two: " + result2);  
    } catch (ExecutionException | InterruptedException ex) {  
        System.out.println("Exception: " + ex);  
    }  
}
```

Wait 5 seconds for the tasks to complete

Get the results of tasks f1 and f2



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Threading Concerns

- Thread Safety
 - Classes should continue to behave correctly when accessed from multiple threads.
- Performance: Deadlock and livelock
 - Threads typically interact with other threads. As more threads are introduced into an application, the possibility exists that threads will reach a point where they cannot continue.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Thread safety is really about the ability of a class to perform the same way when it is accessed by one thread, or multiple threads. Fundamentally, a class performs actions and holds data. Using this definition of thread safety – a class is thread safe if the actions the class performs and the data stored are consistent when used accessed by multiple threads.

Deadlock is a situation where thread A is blocked waiting for a condition set by thread B, but thread B is also blocked waiting for a condition set by thread A.

Livelock is a condition where a thread is not blocked, but cannot move forward because an operation it continually retries fails. Livelock is related to another condition, starvation, where a thread attempts to access a resource that it can never access – likely because other higher priority threads are continually accessing the resource.

Shared Data

Static and instance fields are potentially shared by threads.

```
public class SharedValue {  
    private int i;  
    // Return a unique value  
    public int getNext() {  
        return i++;  
    }  
}
```

Potentially shared variable



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Problems with Shared Data

Shared data must be accessed cautiously. Instance and static fields:

- Are created in an area of memory known as heap space
- Can potentially be shared by any thread
- Might be changed concurrently by multiple threads
 - There are no compiler or IDE warnings.
 - “Safely” accessing shared fields is your responsibility.

Two threads accessing an instance of the `SharedValue` class might produce the following:

```
i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...
```

Zero produced twice

Out of sequence

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Debugging Threads

Debugging threads can be difficult because the frequency and duration of time each thread is allocated can vary for many reasons including:

- Thread scheduling is handled by an operating system and operating systems may use different scheduling algorithms
- Machines have different counts and speeds of CPUs
- Other applications may be placing load on the system

This is one of those cases where an application may seem to function perfectly while in development, but strange problems might manifest after it is in production because of scheduling variations. It is your responsibility to safeguard access to shared variables.

Nonshared Data

Some variable types are never shared. The following types are always thread-safe:

- Local variables
- Method parameters
- Exception handler parameters
- Immutable data



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Shared Thread-Safe Data

Any shared data that is immutable, such as `String` objects or final fields, are thread-safe because they can only be read and not written.

Atomic Operations

Atomic operations function as a single operation. A single statement in the Java language is not always atomic.

- `i++;`
 - Creates a temporary copy of the value in `i`
 - Increments the temporary copy
 - Writes the new value back to `i`
- `l = 0xffff_ffff_ffff_ffff;`
 - 64-bit variables might be accessed using two separate 32-bit operations.

What inconsistencies might two threads incrementing the same field encounter?

What if that field is long?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Inconsistent Behavior

One possible problem with two threads incrementing the same field is that a lost update might occur. Imagine if both threads read a value of 41 from a field, increment the value by one, and then write their results back to the field. Both threads will have done an increment but the resulting value is only 42. Depending on how the Java Virtual Machine is implemented and the type of physical CPU being used, you may never or rarely see this behavior. However, you must always assume that it could happen.

If you have a long value of `0x0000_0000_ffff_ffff` and increment it by 1, the result should be `0x0000_0001_0000_0000`. However, because it is legal for a 64-bit field to be accessed using two separate 32-bit writes, there could temporarily be a value of `0x0000_0001_ffff_ffff` or even `0x0000_0000_0000_0000` depending on which bits are modified first. If a second thread was allowed to read a 64-bit field while it was being modified by another thread, an incorrect value could be retrieved.

Out-of-Order Execution

- Operations performed in one thread may not appear to execute in order if you observe the results from another thread.
 - Code optimization may result in out-of-order operation.
 - Threads operate on cached copies of shared variables.
- To ensure consistent behavior in your threads, you must synchronize their actions.
 - You need a way to state that an action happens before another.
 - You need a way to flush changes to shared variables back to main memory.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Synchronizing Actions

Every thread has a *working memory* in which it keeps its own *working copy* of variables that it must use or assign. As the thread executes a program, it operates on these working copies. There are several actions that will synchronize a thread's *working memory* with main memory:

- A volatile read or write of a variable (the `volatile` keyword)
- Locking or unlocking a monitor (the `synchronized` keyword)
- The first and last action of a thread
- Actions that start a thread or detect that a thread has terminated

The synchronized Keyword

The **synchronized keyword** is used to create thread-safe code blocks. A synchronized code block:

- Causes a thread to write all of its changes to main memory when the end of the block is reached
- Is used to group blocks of code for exclusive execution
 - Threads block until they can get exclusive access
 - Solves the atomic problem



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Synchronized code blocks are used to ensure that data that is not thread-safe will not be accessed concurrently by multiple threads.

synchronized Methods

```
3 public class SynchronizedCounter {  
4     private static int i = 0;  
5  
6     public synchronized void increment() {  
7         i++;  
8     }  
9  
10    public synchronized void decrement() {  
11        i--;  
12    }  
13  
14    public synchronized int getValue() {  
15        return i;  
16    }  
17 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Synchronized Method Behavior

In the example in the slide, you can call only one method at a time in a `SynchronizedCounter` object because all its methods are synchronized. In this example, the synchronization is per `SynchronizedCounter`. Two `SynchronizedCounter` instances could be used concurrently.

If the methods were not synchronized, calling `decrement` while `getValue` is accessed might result in unpredictable behavior.

synchronized Blocks

```
18 public void run(){  
19     for (int i = 0; i < countSize; i++){  
20         synchronized(this){  
21             count.increment();  
22             System.out.println(threadName  
23                     + " Current Count: " + count.getValue());  
24         }  
25     }  
26 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Synchronization Bottlenecks

Synchronization in multithreaded applications ensures reliable behavior. Because synchronized blocks and methods are used to restrict a section of code to a single thread, you are potentially creating performance bottlenecks. synchronized blocks can be used in place of synchronized methods to reduce the number of lines that are exclusive to a single thread.

Use synchronization as little as possible for performance, but as much as needed to guarantee reliability.

Object Monitor Locking

Each object in Java is associated with a monitor, which a thread can lock or unlock.

- synchronized methods use the monitor for the this object.
- static synchronized methods use the classes' monitor.
- synchronized blocks must specify which object's monitor to lock or unlock.

```
synchronized (this) { }
```

- synchronized blocks can be nested.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Nested synchronized Blocks

A thread can lock multiple monitors simultaneously by using nested synchronized blocks.

Threading Performance

To execute a program as quickly as possible, you must avoid performance bottlenecks. Some of these bottlenecks are:

- Resource Contention: Two or more tasks waiting for exclusive use of a resource
- Blocking I/O operations: Doing nothing while waiting for disk or network data transfers
- Underutilization of CPUs: A single-threaded application uses only a single CPU



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Multithreaded Servers

Even if you do not write code to create new threads of execution, your code might be run in a multithreaded environment. You must be aware of how threads work and how to write thread-safe code. When creating code to run inside of another piece of software (such as a middleware or application server), you must read the product's documentation to discover whether threads will be created automatically. For instance, in a Java EE application server, there is a component called a Servlet that is used to handle HTTP requests. Servlets must always be thread-safe because the server starts a new thread for each HTTP request.

Performance Issue: Examples

- **Deadlock** results when two or more threads are blocked forever, waiting for each other.

```
synchronized(obj1) {  
    synchronized(obj2) {  
        }  
    }  
}
```

Thread 1 pauses after locking
obj1's monitor.

```
synchronized(obj2) {  
    synchronized(obj1) {  
        }  
    }  
}
```

Thread 2 pauses after locking
obj2's monitor.

- **Starvation and Livelock**



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by “greedy” threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock

A thread often acts in response to the action of another thread. If the other thread’s action is also a response to the action of another thread, *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked; they are simply too busy responding to each other to resume work.

java.util.concurrent Classes and Packages

The `java.util.concurrent` package contains a number of classes that help with your concurrent applications. Here are just a few examples.

- `java.util.concurrent.atomic` package
 - Lock free thread-safe variables
- `CyclicBarrier`
 - A class that blocks until a specified number of threads are waiting for the thread to complete.
- Concurrency collections



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The use of synchronized code blocks can result in performance bottlenecks. Several components of the `java.util.concurrent` package provide alternatives to using synchronized code blocks.

The `java.util.concurrent.atomic` Package

The `java.util.concurrent.atomic` package contains classes that support lock-free thread-safe programming on single variables.

```
7  public static void main(String[] args) {  
8      AtomicInteger ai = new AtomicInteger(5);  
9      System.out.println("New value: "  
10         + ai.incrementAndGet());  
11     System.out.println("New value: "  
12         + ai.getAndIncrement());  
13     System.out.println("New value"  
14         + ai.getAndIncrement());  
15 }  
16 }
```

An atomic operation increments value to 6 and returns the value.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There is no need to use the `synchronized` keyword with atomic variables. Methods exist to increment a value before or after the value is returned.

The output is:

```
New value: 6  
New value: 6  
New value: 7
```

java.util.concurrent.CyclicBarrier

The CyclicBarrier is an example of the synchronizer category of classes provided by java.util.concurrent.

```
10 final CyclicBarrier barrier = new CyclicBarrier(2);  
// lines omitted  
24     public void run() {  
25         try {  
26             System.out.println("before await - "  
27                     + threadCount.incrementAndGet());  
28             barrier.await();  
29             System.out.println("after await - "  
30                     + threadCount.get());  
31         } catch (BrokenBarrierException|InterruptedException  
ex) {  
32         }  
33     }
```

Two threads must await before they can unblock.
May not be reached



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

CyclicBarrier Behavior

In this example, if only one thread calls `await()` on the barrier, that thread may block forever. After a second thread calls `await()`, any additional call to `await()` will again block until the required number of threads is reached. A `CyclicBarrier` contains a method, `await(long timeout, TimeUnit unit)`, which will block for a specified duration and throw a `TimeoutException` if that duration is reached.

Synchronizers

A framework of classes in the `java.util.concurrent` package that provide mechanics for atomically managing synchronization state, blocking and unblocking threads, and queuing. The `CyclicBarrier` class is an example.

java.util.concurrent.CyclicBarrier

- If line 18 is uncommented, the program will exit

```
9 public class CyclicBarrierExample implements Runnable{  
10     final CyclicBarrier barrier = new CyclicBarrier(2);  
11     AtomicInteger threadCount = new AtomicInteger(0);  
12  
13  
14     public static void main(String[] args) {  
15         ExecutorService es = Executors.newFixedThreadPool(4);  
16  
17         CyclicBarrierExample ex = new CyclicBarrierExample();  
18         es.submit(ex);  
19         //es.submit(ex);  
20  
21         es.shutdown();  
22     }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If the main method runs as shown, the application will just wait. If line 18 is uncommented, then the program will exit normally.

Thread-Safe Collections

The `java.util` collections are not thread-safe. To use collections in a thread-safe fashion:

- Use synchronized code blocks for all access to a collection if writes are performed
- Create a synchronized wrapper using library methods, such as
`java.util.Collections.synchronizedList(List<T>)`
- Use the `java.util.concurrent` collections

Note: Just because a Collection is made thread-safe, this does not make its elements thread-safe.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Concurrent Collections

The `ConcurrentLinkedQueue` class supplies an efficient, scalable, thread-safe, nonblocking FIFO queue. Five implementations in `java.util.concurrent` support the extended `BlockingQueue` interface, which defines blocking versions of `put` and `take`: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, and `DelayQueue`.

Besides queues, this package supplies Collection implementations designed for use in multithreaded contexts: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`. When many threads are expected to access a given collection, a `ConcurrentHashMap` is normally preferable to a synchronized `HashMap`, and a `ConcurrentSkipListMap` is normally preferable to a synchronized `TreeMap`. A `CopyOnWriteArrayList` is preferable to a synchronized `ArrayList` when the expected number of reads and traversals greatly outnumber the number of updates to a list.

CopyOnWriteArrayList: Example

```
7 public class ArrayListTest implements Runnable{  
8     private CopyOnWriteArrayList<String> wordList =  
9         new CopyOnWriteArrayList<>();  
10  
11    public static void main(String[] args) {  
12        ExecutorService es = Executors.newCachedThreadPool();  
13        ArrayListTest test = new ArrayListTest();  
14  
15        es.submit(test); es.submit(test); es.shutdown();  
16  
17        // Print code here  
18        public void run(){  
19            wordList.add("A");  
20            wordList.add("B");  
21            wordList.add("C");  
22        }  
23    }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A **CopyOnWriteArrayList** is a **thread safe** **ArrayList** implementation found in the **java.util.concurrent** library.

Note: The three `es` statements were combined onto one line so the source code would fit in the slide.

Summary

In this lesson, you should have learned how to:

- Describe operating system task scheduling
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and `concurrent atomic` to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections



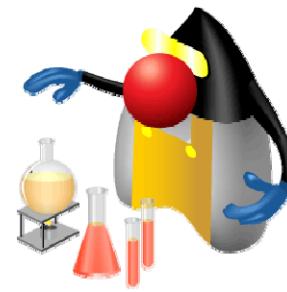
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 15-1 Overview: Using the `java.util.concurrent` Package

This practice covers the following topics:

- Using a cached thread pool (`ExecutorService`)
- Implementing `Callable`
- Receiving `Callable` results with a `Future`



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this practice, you create a multithreaded network client.

Quiz

An ExecutorService will always attempt to use all of the available CPUs in a system.

- a. True
- b. False



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Variables are thread-safe if they are:

- a. local
- b. static
- c. final
- d. private



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and CAS TRAINING, S.L. use only

16

The Fork-Join Framework

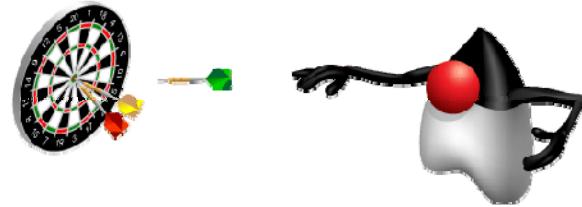
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Apply the Fork-Join framework



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallelism

Modern systems contain multiple CPUs. Taking advantage of the processing power in a system requires you to execute tasks in parallel on multiple CPUs.

- Divide and conquer: A task should be divided into subtasks. You should attempt to identify those subtasks that can be executed in parallel.
- Some problems can be difficult to execute as parallel tasks.
- Some problems are easier. Servers that support multiple clients can use a separate task to handle each client.
- Be aware of your hardware. Scheduling too many parallel tasks can negatively impact performance.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

CPU Count

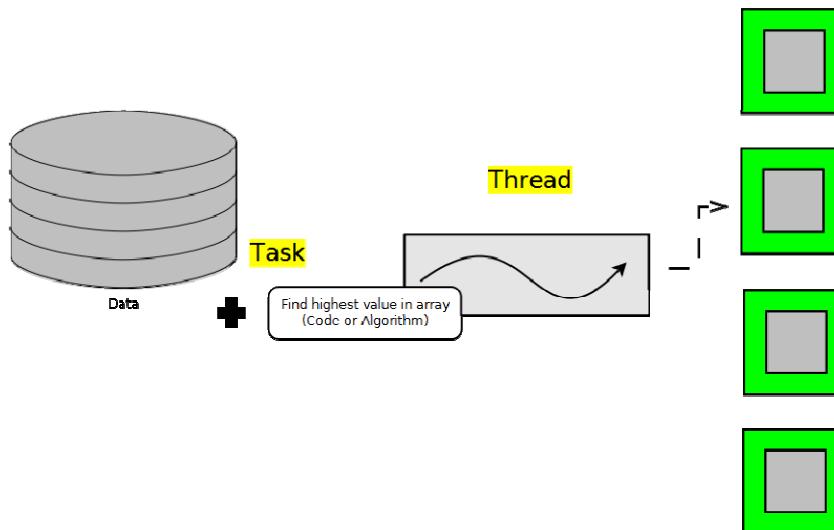
If your tasks are compute-intensive as opposed to I/O intensive, the number of parallel tasks should not greatly outnumber the number of processors in your system. You can detect the number of processors easily in Java:

```
int count = Runtime.getRuntime().availableProcessors();
```

Without Parallelism

Modern systems contain multiple CPUs. If you do not leverage threads in some way, only a portion of your system's processing power will be utilized.

CPU(s)



ORACLE®

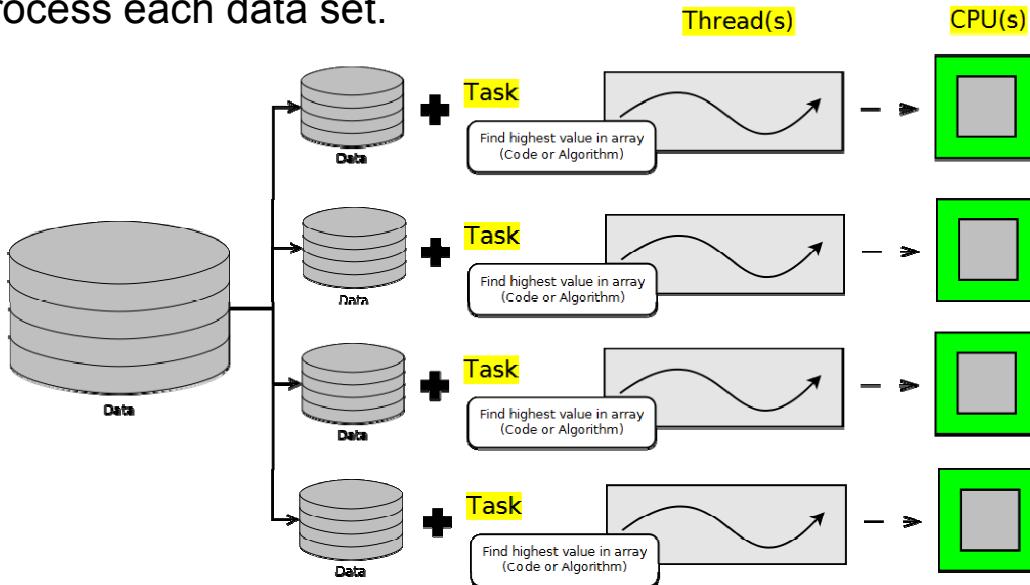
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Setting the Stage

If you have a large amount of data to process but only one thread to process that data, a single CPU will be used. In the slide's graphic, a large set of data (an array, possibly) is being processed. The array processing could be a simple task, such as finding the highest value in the array. In a four CPU system, there would be three CPUs sitting idle while the array was being processed.

Naive Parallelism

A simple parallel solution breaks the data to be processed into multiple sets: one data set for each CPU and one thread to process each data set.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

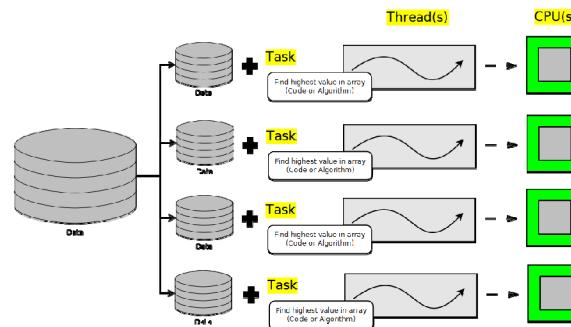
Splitting the Data

In the slide's graphic, a large set of data (an array, possibly) is split into four subsets of data, one subset for each CPU. A thread per CPU is created to process the data. After processing the subsets of data, the results will have to be combined in a meaningful way. There are several ways to subdivide the large dataset to be processed. It would be overly memory-intensive to create a new array per thread that contains a copy of a portion of the original array. Each array can share a reference to the single large array but access only a subset in a non-blocking thread-safe way.

The Need for the Fork-Join Framework

Splitting datasets into equal sized subsets for each thread to process has a couple of problems. Ideally all CPUs should be fully utilized until the task is finished, but:

- CPUs may run at different speeds
- Non-Java tasks require CPU time and may reduce the time available for a Java thread to spend executing on a CPU
- The data being analyzed may require varying amounts of time to process



ORACLE®

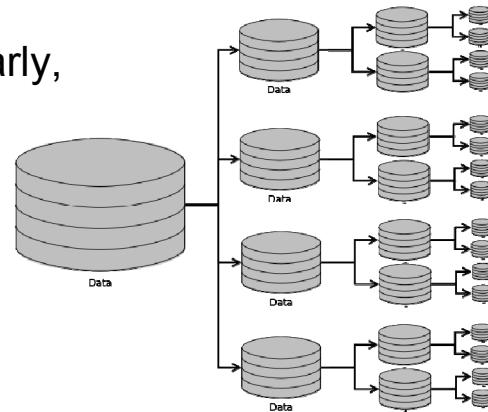
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Work-Stealing

To keep multiple threads busy:

- Divide the data to be processed into a large number of subsets
- Assign the data subsets to a thread's processing queue
- Each thread will have many subsets queued

If a thread finishes all its subsets early, it can “steal” subsets from another thread.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Work Granularity

By subdividing the data to be processed until there are more subsets than threads, we are facilitating “work-stealing.” In work-stealing, a thread that has run out of work can steal work (a data subset) from the processing queue of another thread. You must determine the optimal size of the work to add to each thread's processing queue. Overly subdividing the data to be processed can cause unnecessary overhead, while insufficiently subdividing the data can result in underutilization of CPUs.

A Single-Threaded Example

```
int[] data = new int[1024 * 1024 * 256]; //1G  
  
for (int i = 0; i < data.length; i++) {  
    data[i] = ThreadLocalRandom.current().nextInt();  
}  
  
int max = Integer.MIN_VALUE;  
for (int value : data) {  
    if (value > max) {  
        max = value;  
    }  
}  
  
System.out.println("Max value found: " + max);
```

A very large dataset

Fill up the array with values.

Sequentially search the array for the largest value.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Potential

In this example, there are two separate tasks that could be executed in parallel. Initializing the array with random values and searching the array for the largest possible value could both be done in parallel.

java.util.concurrent.ForkJoinTask<V>

A ForkJoinTask object represents a task to be executed.

- A task contains the code and data to be processed. Similar to a Runnable or Callable.
- A huge number of tasks are created and processed by a small number of threads in a Fork-Join pool.
 - A ForkJoinTask typically creates more ForkJoinTask instances until the data to process has been subdivided adequately.
- Developers typically use the following subclasses:
 - RecursiveAction: When a task does not need to return a result
 - RecursiveTask: When a task needs to return a result



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

RecursiveTask Example

```
public class FindMaxTask extends RecursiveTask<Integer> {  
    private final int threshold;  
    private final int[] myArray;  
    private int start;  
    private int end;  
  
    public FindMaxTask(int[] myArray, int start, int end,  
int threshold) {  
        // copy parameters to fields  
    }  
    protected Integer compute() {  
        // shown later  
    }  
}
```

Result type of the task

The data to process

Where the work is done.
Notice the generic return type.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

compute Structure

```
protected Integer compute() {  
    if DATA_SMALL_ENOUGH {  
        PROCESS_DATA  
        return RESULT;  
    } else {  
        SPLIT_DATA_INTO_LEFT_AND_RIGHT_PARTS  
        TASK t1 = new TASK(LEFT_DATA);  
        t1.fork(); ————— Asynchronously execute  
        TASK t2 = new TASK(RIGHT_DATA);  
        return COMBINE(t2.compute(), t1.join());  
    }  
}
```

Process in current thread

Asynchronously execute

Block until done



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

compute Example (Below Threshold)

```
protected Integer compute() {  
    if (end - start < threshold) {  
        int max = Integer.MIN_VALUE;  
        for (int i = start; i <= end; i++) {  
            int n = myArray[i];  
            if (n > max) {  
                max = n;  
            }  
        }  
        return max;  
    } else {  
        // split data and create tasks  
    }  
}
```

The range within the array

You decide the threshold.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

compute Example (Above Threshold)

```
protected Integer compute() {  
    if (end - start < threshold) {  
        // find max  
    } else {  
        int midway = (end - start) / 2 + start;  
        FindMaxTask a1 = new FindMaxTask(myArray, start, midway, threshold);  
        a1.fork();  
        FindMaxTask a2 = new FindMaxTask(myArray, midway + 1, end, threshold);  
        return Math.max(a2.compute(), a1.join());  
    }  
}
```

Task for left half of data

Task for right half of data



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Memory Management

Notice that the same array is passed to every task but with different start and end values. If the subset of values to be processed were copied into a new array each time a task was created, memory usage would quickly skyrocket.

ForkJoinPool Example

A ForkJoinPool is used to execute a ForkJoinTask. It creates a thread for each CPU in the system by default.

```
ForkJoinPool pool = new ForkJoinPool();
FindMaxTask task =
    new FindMaxTask(data, 0, data.length-1, data.length/16);
Integer result = pool.invoke(task);
```

The task's compute method is automatically called .



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Fork-Join Framework Recommendations

- Avoid I/O or blocking operations.
 - Only one thread per CPU is created by default. Blocking operations would keep you from utilizing all CPU resources.
- Know your hardware.
 - A Fork-Join solution will perform slower on a one-CPU system than a standard sequential solution.
 - Some CPUs increase in speed when only using a single core, potentially offsetting any performance gain provided by Fork-Join.
- Know your problem.
 - Many problems have additional overhead if executed in parallel (parallel sorting, for example).



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Sorting

When using Fork-Join to sort an array in parallel, you end up sorting many small arrays and then having to combine the small sorted arrays into larger sorted arrays. For an example see the sample application provided with the JDK in C:\Program Files\Java\jdk1.7.0\sample\forkjoin\mergesort.

Summary

In this lesson, you should have learned how to:

- Apply the Fork-Join framework



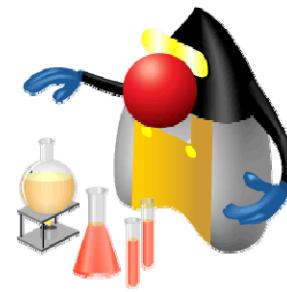
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 16-1 Overview: Using the Fork-Join Framework

This practice covers the following topics:

- Extending RecursiveAction
- Creating and using a ForkJoinPool



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this practice, you create a multithreaded network client.

Quiz

Applying the Fork-Join framework will always result in a performance benefit.

- a. True
- b. False



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

17

Parallel Streams

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Streams Review

- Pipeline
 - Multiple streams passing data along
 - Operations can be Lazy
 - Intermediate, Terminal, and Short-Circuit Terminal Operations
- Stream characteristics
 - Immutable
 - Once elements are consumed they are no longer available from the stream.
 - Can be sequential (default) or **parallel**



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Old Style Collection Processing

```
15     double sum = 0;
16
17     for (Employee e:eList) {
18         if (e.getState().equals("CO") &&
19             e.getRole().equals(Role.EXECUTIVE)) {
20             e.printSummary();
21             sum += e.getSalary();
22         }
23     }
24
25     System.out.printf("Total CO Executive Pay:
$%,9.2f %n", sum);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are a couple of key points that can be made about the above code.

- All elements in the collections must be iterated through every time.
- The code is more about "how" information is obtained and less about "what" the code is trying to accomplish.
- A mutator must be added to the loop to calculate the total.
- There is no easy way to parallelize this code.

New Style Collection Processing

```
15     double result = eList.stream()
16         .filter(e -> e.getState().equals("CO"))
17         .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18         .peek(e -> e.printSummary())
19         .mapToDouble(e -> e.getSalary())
20         .sum();
21
22     System.out.printf("Total CO Executive Pay: $%,9.2f
%n", result);
```

- What are the advantages?
 - Code reads like a problem.
 - Acts on the data set
 - Operations can be lazy.
 - Operations can be serial or parallel.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are also some key points worth pointing out for this piece of code as well.

- The code reads much more like a problem statement.
- No mutator is needed to get the final result.
- Using this approach provides more opportunity for lazy optimizations.
- This code can easily be parallelized.

Stream Pipeline: Another Look

```
13     public static void main(String[] args) {  
14  
15         List<Employee> eList = Employee.createShortList();  
16  
17         Stream<Employee> s1 = eList.stream();  
18  
19         Stream<Employee> s2 = s1.filter(  
20             e -> e.getState().equals("CO"));  
21  
22         Stream<Employee> s3 = s2.filter(  
23             e -> e.getRole().equals(Role.EXECUTIVE));  
24         Stream<Employee> s4 = s3.peek(e -> e.printSummary());  
25         DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());  
26         double result = s5.sum();  
27  
28         System.out.printf("Total CO Executive Pay: $%,9.2f %n",  
result);  
29     }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

So far all the examples have used lambda expressions and stream pipelines to perform the tasks. In the above example, the `Stream` class is used with regular Java statements to perform the same steps as those found in a pipeline. Even though the approach is possible, a stream pipeline seems like a much better solution.

Styles Compared

Imperative Programming

- Code deals with individual data items.
- Focused on how
- Code does not read like a problem.
- Steps mashed together
- Leaks extraneous details
- Inherently sequential

Streams

- Code deals with data set.
- Focused on what
- Code reads like a problem.
- Well-factored
- No "garbage variables" (Temp variables leaked into scope)
- Code can be sequential or parallel.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Stream

- May provide better performance
 - Many chips and cores per machine
 - GPUs
- Map/Reduce in the small
- Fork/join is great, but too low level
 - A lot of boilerplate code
 - Stream uses fork/join under the hood
- Many factors affect performance
 - Data size, decomposition, packing, number of cores
- Unfortunately, not a magic bullet
 - Parallel is not always faster



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Making a stream run in parallel is pretty easy. Just call the `parallelStream` or `parallel` method in the stream. With that call, when the stream executes it uses all the processing cores available to the current JVM to perform the task.

The fork/join framework is used to break the work into smaller tasks, execute each task, and then recombine the results. But as you will see, much less code is needed to do this with streams than would be necessary if fork/join was coded by hand.

Remember, parallel is not always faster. For certain types of tasks, serial processing will produce better results.

Using Parallel Streams: Collection

- Call from a Collection

```
15     double result = eList.parallelStream()
16         .filter(e -> e.getState().equals("CO"))
17         .filter(e ->
18             e.getRole().equals(Role.EXECUTIVE))
19         .peek(e -> e.printSummary())
20         .mapToDouble(e -> e.getSalary())
21         .sum();
22
23     System.out.printf("Total CO Executive Pay:
$%,.2f %n", result);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This is an example of using the `parallelStream` method to make the stream pipeline parallel.

Using Parallel Streams: From a Stream

```
27     result = eList.stream()
28         .filter(e -> e.getState().equals("CO"))
29         .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30         .peek(e -> e.printSummary())
31         .mapToDouble(e -> e.getSalary())
32         .parallel()
33         .sum();
34
35     System.out.printf("Total CO Executive Pay: $%,9.2f
%n", result);
```

- Specify with `.parallel` or `.sequential` (**default is sequential**)
- Choice applies to entire pipeline.
 - Last call wins
- Once again, the API doc is your friend.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This example uses the `parallel` method to make the stream pipeline parallel. Both the `sequential` and `parallel` methods may be called in a pipeline. **Whichever method is called last**, will be applied to the stream.

Pipelines Fine Print

- Stream pipelines are like Builders.
 - Add a bunch of intermediate operations, and then execute
 - Cannot "branch" or "reuse" pipeline
- Do not modify the source during a query.
- Operation parameters must be stateless.
 - Do not access any state that might change.
 - **This enables correct operation sequentially or in parallel.**
- Best to banish side effects completely.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Your data should be immutable or read-only when used with stream pipelines. No changes to state should take place during a pipeline.

Embrace Statelessness

```
17 List<Employee> newList02 = new ArrayList<>();  
...  
23 newList02 = eList.parallelStream() // Good Parallel  
24         .filter(e -> e.getDept().equals("Eng"))  
25         .collect(Collectors.toList());
```

- Mutate the stateless way
 - The above is preferable.
 - It is designed to parallelize.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If you want to save the results after a pipeline completes, use the `collect` method and `Collectors` class as shown in the example. This method parallelizes well and treats the data in a stateless way.

Avoid Statefulness

```
15      List<Employee> eList =  
Employee.createShortList();  
16      List<Employee> newList01 = new ArrayList<>();  
17      List<Employee> newList02 = new ArrayList<>();  
18  
19      eList.parallelStream() // Not Parallel. Bad.  
20          .filter(e -> e.getDept().equals("Eng"))  
21          .forEach(e -> newList01.add(e));
```

- Temptation is to do the above.
 - **Do not do this. It does not parallelize.**



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stream pipeline results may be nondeterministic or incorrect if the behavioral parameters to the stream operations are stateful. A stateful lambda is one whose result depends on any state which might change during the execution of the stream pipeline.

Note: Do not write code like that shown in this example.

Streams Are Deterministic for Most Part

```
14     List<Employee> eList = Employee.createShortList();  
15  
16     double r1 = eList.stream()  
17         .filter(e -> e.getState().equals("CO"))  
18         .mapToDouble(Employee::getSalary)  
19         .sequential().sum();  
20  
21     double r2 = eList.stream()  
22         .filter(e -> e.getState().equals("CO"))  
23         .mapToDouble(Employee::getSalary)  
24         .parallel().sum();  
25  
26     System.out.println("The same: " + (r1 == r2));
```

- Will the result be the same?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A deterministic algorithm is an algorithm which, given a particular input, will always produce the same output. The `sum` method is a great example as the order in which elements are combined does not matter. The result will be the same irrespective of the order elements are added.

Some Are Not Deterministic

```
14     List<Employee> eList = Employee.createShortList();  
15  
16     Optional<Employee> e1 = eList.stream()  
17         .filter(e -> e.getRole().equals(Role.EXECUTIVE))  
18         .sequential().findAny();  
19  
20     Optional<Employee> e2 = eList.stream()  
21         .filter(e -> e.getRole().equals(Role.EXECUTIVE))  
22         .parallel().findAny();  
23  
24     System.out.println("The same: " +  
25         e1.get().getEmail().equals(e2.get().getEmail()));
```

- Will the result be the same?
 - In this case, maybe not.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The larger the data set, the more likely the two code blocks will produce a different result. The parallel stream does not search the data sequentially. Consequently, it is possible it will find a different element that meets the criteria first.

Reduction

- Reduction
 - An operation that takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
 - Implemented with the `reduce()` method
- Example: `sum` is a reduction with a base value of 0 and a combining function of `+`.
 - $((((0 + a_1) + a_2) + \dots) + a_n)$
 - `.sum()` is equivalent to `reduce(0, (a, b) -> a + b)`
 - `(0, (sum, element) -> sum + element)`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reduction is an operation that takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation. The `sum` method for the `Stream` class is an application of reduction.

Reduction Fine Print

- If the combining function is associative, reduction parallelizes cleanly
 - Associative means the order does not matter.
 - The result is the same irrespective of the order used to combine elements.
- Examples of: sum, min, max, average, count
 - `.count()` is equivalent to `.map(e -> 1).sum()`.
- **Warning:** If you pass a nonassociative function to reduce, you will get the wrong answer. The function must be associative.

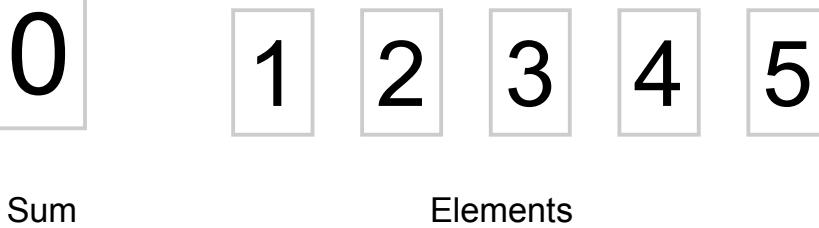


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As the text above points out, a reduction can only be performed on an associative function. In effect, a function where order does not matter. If the function is not associative, you will get the wrong result.

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Note that the integer value of 0 is passed into the reduce method. This is called the *identity* value. It represents the starting value for the reduce function and the default return value if there are no members in the reduction.

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

1

Sum

2

3

4

5

Elements

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

3

Sum

3

4

5

Elements



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

6

Sum

4

5

Elements

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

10

Sum

5

Elements

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

15

Sum

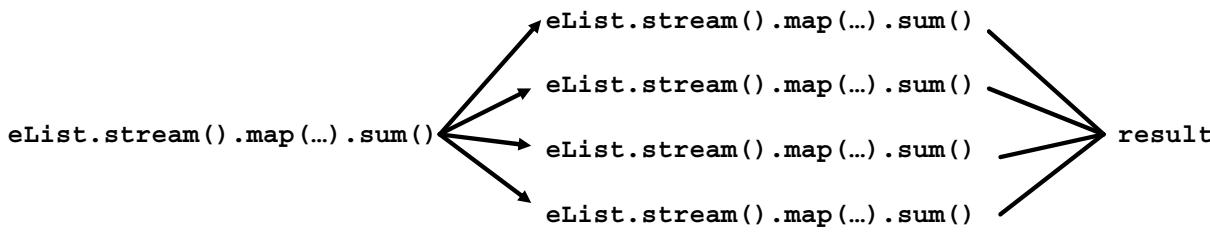
Elements

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Look Under the Hood

- Pipeline decomposed into subpipelines.
 - Each subpipeline produces a subresult.
 - Subresults combined into final result.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This picture shows how the sum is first decomposed into smaller steps. The results are then combined to produce a result.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```

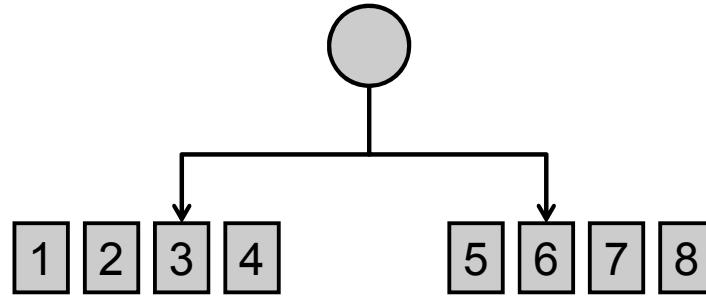


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the steps that follow, the data set above is summed. The steps of decomposition and then combination are shown in detail. Note that for this operation, the order of operations does not matter.

Illustrating Parallel Execution

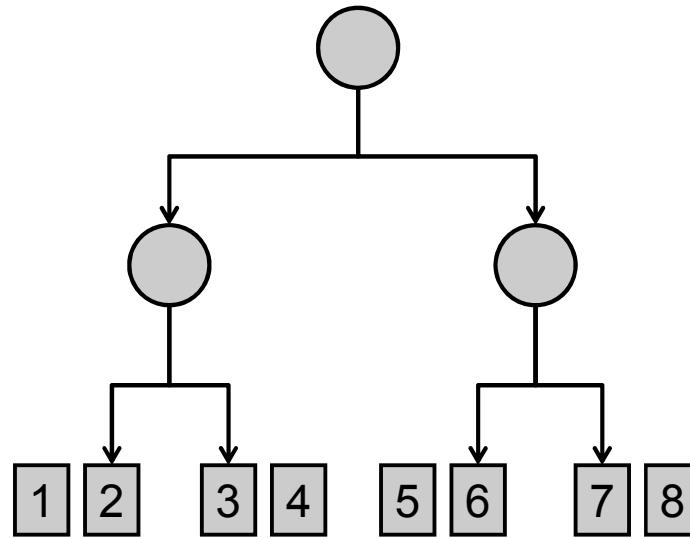
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```

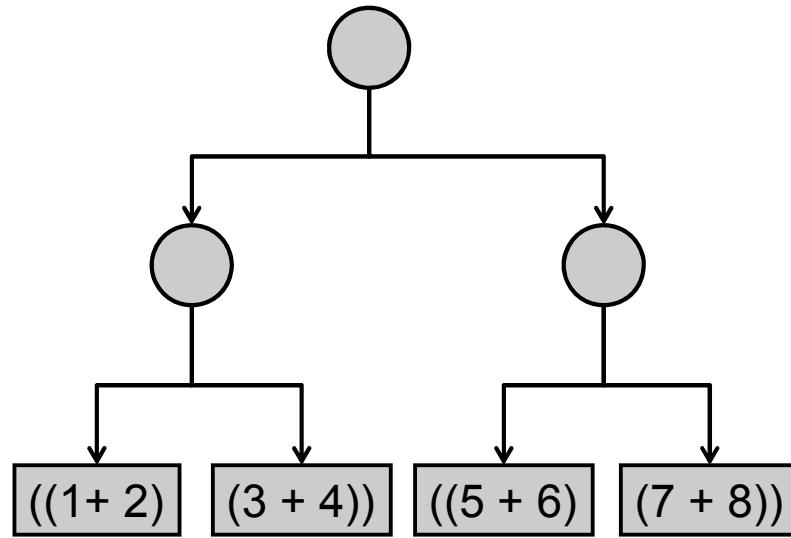


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```

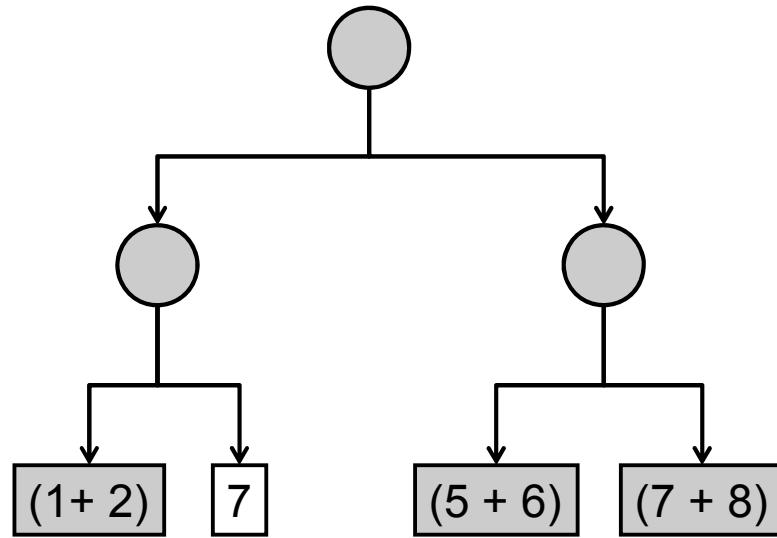


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```

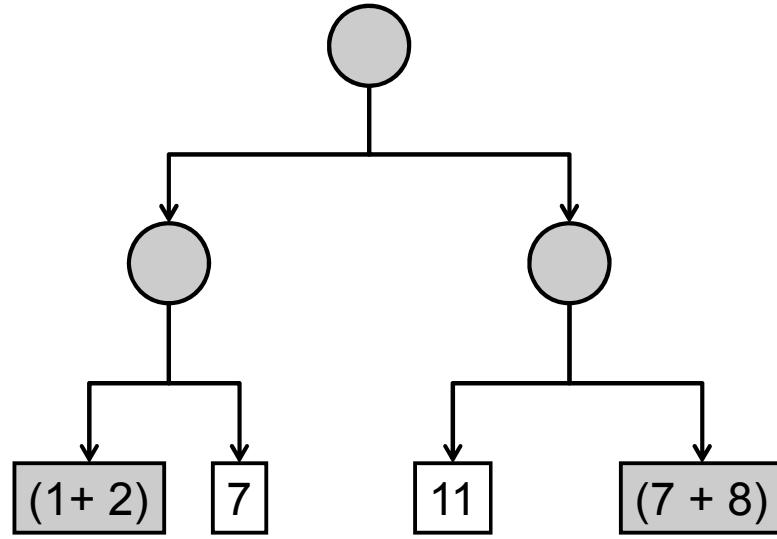


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```

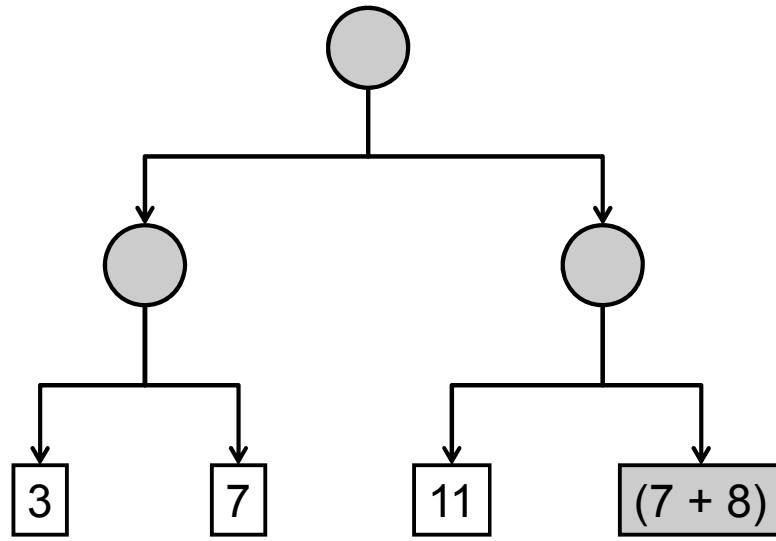


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```

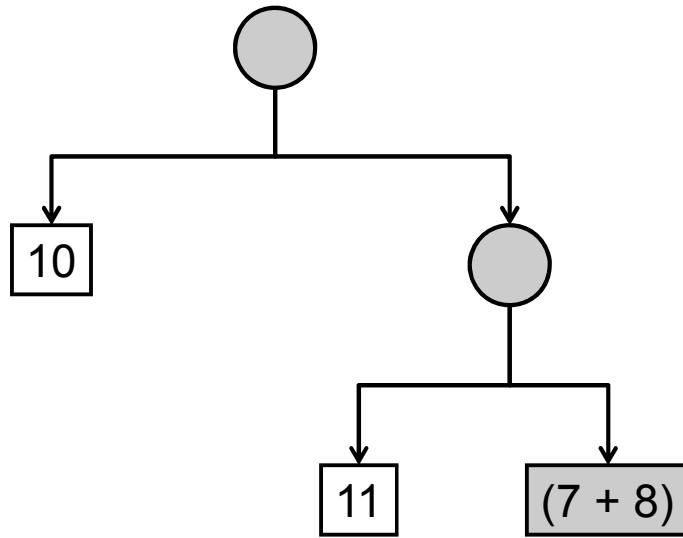


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

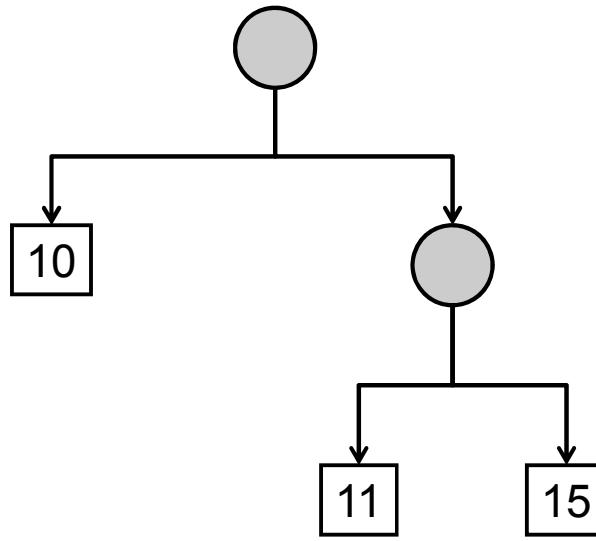
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```

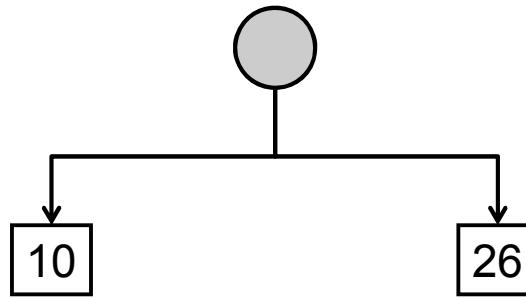


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```

36



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Performance

- Do not assume parallel is always faster.
 - Parallel not always the right solution.
 - Sometimes parallel is slower than sequential.
- Qualitative considerations
 - Does the stream source decompose well?
 - Do terminal operations have a cheap or expensive merge operation?
 - What are stream characteristics?
 - Filters change size for example.
- Primitive streams provided for performance
 - Boxing/Unboxing negatively impacts performance.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As with any code, test and verify that a particular approach works as intended. As stated previously, associative functions decompose well and make good candidates for parallel processing. But operations that do not meet this criteria may perform better when processed sequentially.

A Simple Performance Model

N = Size of the source data set

Q = Cost per element through the pipeline

$N * Q \approx$ Cost of the pipeline

- Larger $N * Q \rightarrow$ Higher chance of good parallel performance
- Easier to know N than Q
- You can reason qualitatively about Q
 - Simple pipeline example
 - $N > 10K, Q=1$
 - Reduction using sum
 - Complex pipelines might
 - Contain filters
 - Contain limit operation
 - Complex reduction using `groupingBy()`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As the slide points out, the larger the data set, the more likely parallel processing is going to show an improvement in performance. Some other observations:

- A system needs to have at least four cores available to the JVM before you will see any substantial difference in performance.
- As a general guideline, a data set should contain more than 10,000 items before showing a difference in performance.
- Any operations or complex operations that cause threads to block will have a negative impact on performance.

Summary

In this lesson, you should have learned how to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice

- Practice 17-1: Calculate Total Sales Without a Pipeline
- Practice 17-2: Calculate Sales Totals Using Parallel Streams
- Practice 17-3: Calculate Sales Totals Using Parallel Streams and Reduce



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and CAS TRAINING, S.L. use only

18

Building Database Applications with JDBC



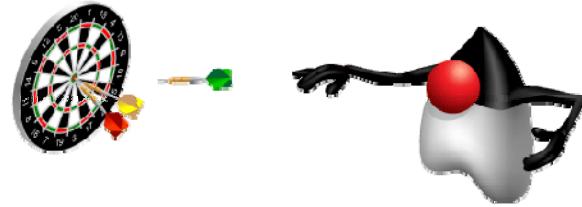
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

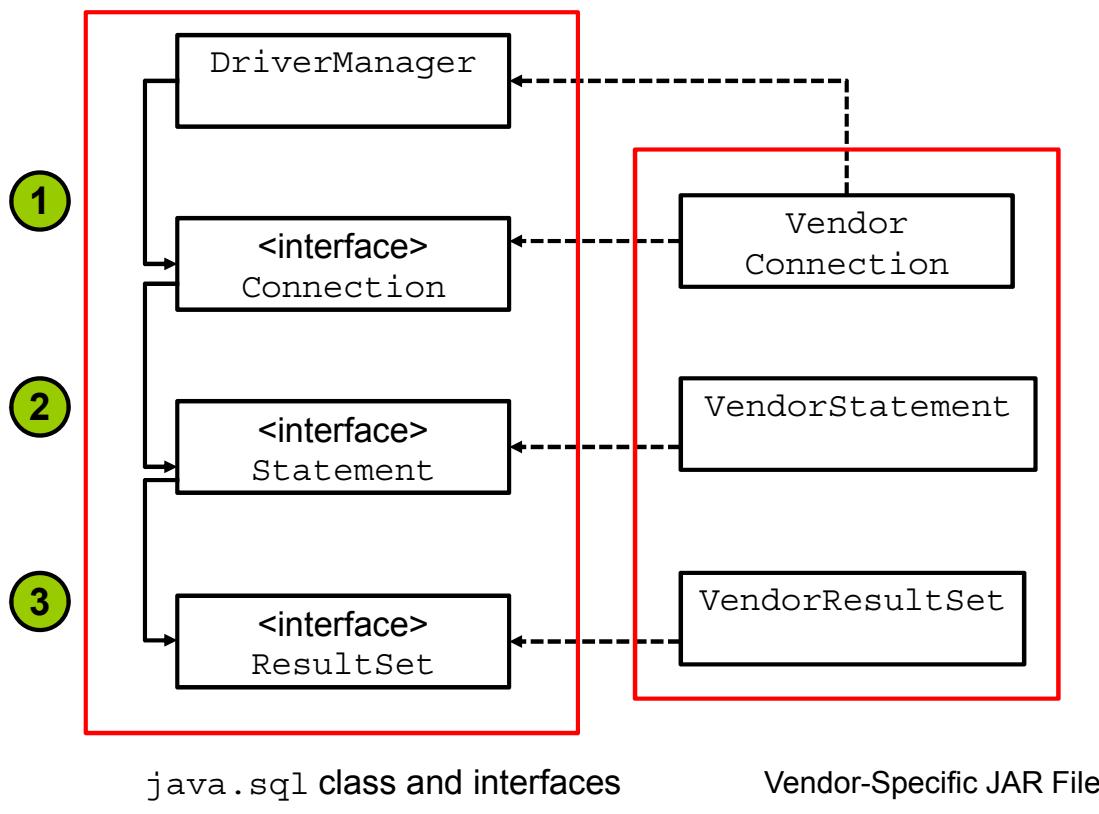
- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the JDBC API



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The JDBC API is made up of some concrete classes, such as `Date`, `Time`, and `SQLException`, and a set of interfaces that are implemented in a driver class that is provided by the database vendor.

Because the implementation is a valid instance of the interface method signature, after the database vendor's Driver classes are loaded, you can access them by following the sequence shown in the slide:

1. Use the `DriverManager` class to obtain a reference to a `Connection` object by using the `getConnection` method. The typical signature of this method is `getConnection (url, name, password)`, where `url` is the JDBC URL, and `name` and `password` are strings that the database accepts for a connection.
2. Use the `Connection` object (implemented by some class that the vendor provided) to obtain a reference to a `Statement` object through the `createStatement` method. The typical signature for this method is `createStatement ()` with no arguments.
3. Use the `Statement` object to obtain an instance of a `ResultSet` through an `executeQuery (query)` method. This method typically accepts a string (`query`), where `query` is a static string.

Using a Vendor's Driver Class

The DriverManager class is used to get an instance of a Connection object by using the JDBC driver named in the JDBC URL:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";  
Connection con = DriverManager.getConnection (url);
```

- The URL syntax for a JDBC driver is:

```
[jdbc:<driver>:[subsubprotocol:] [databaseName] [;attribute=value]
```

- Each vendor can implement its own subprotocol.
- The URL syntax for an Oracle Thin driver is:

```
jdbc:oracle:thin:@// [HOST] [:PORT] /SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

DriverManager

Any JDBC 4.0 drivers that are found in the class path are automatically loaded. The `DriverManager.getConnection` method will attempt to load the driver class by looking at the `META-INF/services/java.sql.Driver` file. This file contains the name of the JDBC driver's implementation of `java.sql.Driver`. For example, the `META-INF/services/java.sql.driver` file in `derbyclient.jar` contains `org.apache.derby.jdbc.ClientDriver`.

Drivers prior to JDBC 4.0 must be loaded manually by using:

```
try {  
    java.lang.Class.forName("<fully qualified path of the driver>");  
} catch (ClassNotFoundException c) {  
}
```

Driver classes can also be passed to the interpreter on the command line:

```
java -djdbc.drivers=<fully qualified path to the driver> <class to  
run>
```

Key JDBC API Components

Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:

- `java.sql.Connection`: A connection that represents the session between your Java application and the database

```
Connection con = DriverManager.getConnection(url,  
    username, password);
```

- `java.sql.Statement`: An object used to execute a static SQL statement and return the result

```
Statement stmt = con.createStatement();
```

- `java.sql.ResultSet`: An object representing a database result set

```
String query = "SELECT * FROM Employee";  
ResultSet rs = stmt.executeQuery(query);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Connections, Statements, and ResultSets

The main advantage of the JDBC API is that it provides a flexible and portable way to communicate with a database.

The JDBC driver that is provided by a database vendor implements each of the following Java interfaces. Your Java code can use the interface knowing that the database vendor provided the implementation of each of the methods in the interface:

- **Connection**: Is an interface that provides a session with the database. While the connection object is open, you can access the database, create statements, get results, and manipulate the database. When you close a connection, the access to the database is terminated and the open connection closed.
- **Statement**: Is an interface that provides a class for executing SQL statements and returning the results. The Statement interface is for static SQL queries. There are two other subinterfaces: `PreparedStatement`, which extends `Statement` and `CallableStatement`, which extends `PreparedStatement`.
- **ResultSet**: Is an interface that manages the resulting data returned from a Statement

Note: SQL commands and keywords are not case-sensitive—that is, you can use `SELECT` or `Select`. SQL table and column names (identifiers) can be case-sensitive or not case-sensitive, depending upon the database. SQL identifiers are not case-sensitive in the Derby database (unless delimited). **Java SE 8 Programming 18 - 5**

Writing Queries and Getting Results

To execute SQL queries with JDBC, you must create a SQL query wrapper object, an instance of the `Statement` object.

```
Statement stmt = con.createStatement();
```

- Use the `Statement` instance to execute a SQL query:

```
ResultSet rs = stmt.executeQuery(query);
```

- Note that there are three `Statement` execute methods:

Method	Returns	Used for
<code>executeQuery(sqlString)</code>	<code>ResultSet</code>	<code>SELECT</code> statement
<code>executeUpdate(sqlString)</code>	<code>int (rows affected)</code>	<code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code> , or a <code>DDL</code>
<code>execute(sqlString)</code>	<code>boolean (true if there was a ResultSet)</code>	Any SQL command or commands



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

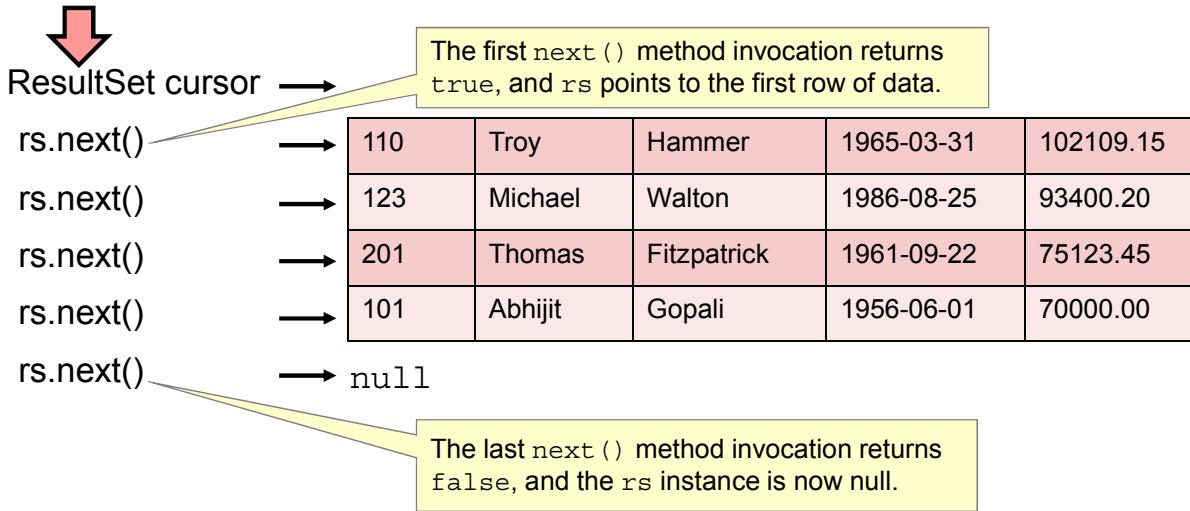
A SQL statement is executed against a database using an instance of a `Statement` object. The `Statement` object is a wrapper object for a query. A `Statement` object is obtained through a `Connection` object—the database connection. So it makes sense that from a `Connection`, you get an object that you can use to write statements to the database.

The `Statement` interface provides three methods for creating SQL queries and returning a result. Which one you use depends upon the type of SQL statement you want to use:

- `executeQuery(sqlString)`: For a `SELECT` statement, returns a `ResultSet` object
- `executeUpdate(sqlString)`: For `INSERT`, `UPDATE`, and `DELETE` statements, returns an `int (number of rows affected)`, or `0` when the statement is a Data Definition Language (DDL) statement, such as `CREATE TABLE`.
- `execute(sqlString)`: For any SQL statement, returns a `boolean` indicating if a `ResultSet` was returned. Multiple SQL statements can be executed with `execute`.

Using a ResultSet Object

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ResultSet Objects

- `ResultSet` maintains a cursor to the returned rows. The cursor is initially pointing before the first row.
- The `ResultSet.next()` method is called to position the cursor in the next row.
- The default `ResultSet` is not updatable and has a cursor that points only forward.
- It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable:

```
Statement stmt
    = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                         ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

Note: Not all databases support scrollable result sets.

`ResultSet` has accessor methods to read the contents of each column returned in a row.
`ResultSet` has a getter method for each type.

CRUD Operations Using JDBC API: Retrieve

```
1 package com.example.text;
2
3 import java.sql.DriverManager;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.Date;
7
8 public class SimpleJDBCTest {
9
10    public static void main(String[] args) {
11        String url = "jdbc:derby://localhost:1527/EmployeeDB";
12        String username = "public";
13        String password = "tiger";
14        String query = "SELECT * FROM Employee";
15        try (Connection con =
16             DriverManager.getConnection (url, username, password);
17             Statement stmt = con.createStatement ());
18             ResultSet rs = stmt.executeQuery (query)) {
```

The hard-coded JDBC URL, username, and password are just for this simple example.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

CRUD (Create, Retrieve, Update, and Delete) operations are equivalent to the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements in SQL.

In the following slide, you see a complete example of a JDBC application, a simple one that reads all the rows from an Employee database and returns the results as strings to the console.

- **Lines 15–16:** Use a `try-with-resources` statement to get an instance of an object that implements the `Connection` interface.
- **Line 17:** Use the `connection` object to get an instance of an object that implements the `Statement` interface from the `Connection` object.
- **Line 18:** Create a `ResultSet` by executing the string `query` using the `Statement` object.

Note: Hard coding the JDBC URL, username, and password makes an application less portable. Instead, consider using `java.io.Console` to read the username and password and/or some type of authentication service.

CRUD Operations Using JDBC: Retrieve

Loop through all of the rows in the ResultSet.

```
19     while (rs.next()) {
20         int empID = rs.getInt("ID");
21         String first = rs.getString("FirstName");
22         String last = rs.getString("LastName");
23         Date birthDate = rs.getDate("BirthDate");
24         float salary = rs.getFloat("Salary");
25         System.out.println("Employee ID: " + empID + "\n"
26             + "Employee Name: " + first + " " + last + "\n"
27             + "Birth Date: " + birthDate + "\n"
28             + "Salary: " + salary);
29     } // end of while
30 } catch (SQLException e) {
31     System.out.println("SQL Exception: " + e);
32 } // end of try-with-resources
33 }
34 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- **Lines 20–24:** Get the results of each of the data fields in each row read from the Employee table.
- **Lines 25–28:** Print the resulting data fields to the system console.
- **Line 30:** SQLException: This class extends Exception thrown by the DriverManager, Statement, and ResultSet methods.
- **Line 32:** This is the closing brace for the try-with-resources statement on line 15.

This example is from the SimpleJDBCEExample project.

Output:

```
run:
Employee ID: 110
Employee Name: Troy Hammer
Birth Date: 1965-03-31
Salary: 102109.15
```

CRUD Operations Using JDBC API: Create

```
1.  public class InsertJDBCExample {  
2.      public static void main(String[] args) {  
3.          // Create the "url"  
4.          // assume database server is running on the localhost  
5.          String url = "jdbc:derby://localhost:1527/EmployeeDB";  
6.          String username = "scott";  
7.          String password = "tiger";  
8.          try (Connection con = DriverManager.getConnection(url, username,  
password))  
9.          {  
10.             Statement stmt = con.createStatement();  
11.             String query = "INSERT INTO Employee VALUES (500, 'Jill',  
'Murray', '1950-09-21', 150000);  
12.             if (stmt.executeUpdate(query) > 0) {  
13.                 System.out.println("A new Employee record is added");  
14.             }  
15.             String query1="select * from Employee";  
16.             ResultSet rs = stmt.executeQuery(query1);  
17.             //code to display the rows  
18.         }  
19.     }
```

Query to insert a row in
the Employee.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This slide demonstrates the insert operation. An employee record is added to the Employee table and the content of the Employee table after the insert operation is displayed in the output console.

Lines 10–13: Create a query to insert an employee record and execute the query.

Lines 15–17: Print the resulting data fields to the system console.

CRUD Operations Using JDBC API: Update

```
1. public class UpdateJDBCExample {  
2.     public static void main(String[] args) {  
3.         // Create the "url"  
4.         // assume database server is running on the localhost  
5.         String url = "jdbc:derby://localhost:1527/EmployeeDB";  
6.         String username = "scott";  
7.         String password = "tiger";  
8.         try (Connection con = DriverManager.getConnection(url, username,  
password)) {  
9.             Statement stmt = con.createStatement();  
10.            query = "Update Employee SET salary= 200000 where id=500";  
11.            if (stmt.executeUpdate(query) > 0) {  
12.                System.out.println("An existing employee record was updated  
successfully!");  
13.            }  
14.            String query1="select * from Employee";  
15.            ResultSet rs = stmt.executeQuery(query1);  
16.            //code to display the records//  
17.        }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This slide demonstrates the update operation. An existing employee record is updated and the content of the Employee table after the update operation is displayed in the output console.

Lines 9–12: Create a query to update an employee record with ID 500 and execute the query.

Lines 14–16: Print the resulting data fields to the system console.

CRUD Operations Using JDBC API: Delete

```
1. public class DeleteJDBCExample {  
2.     public static void main(String[] args) {  
3.         String url = "jdbc:derby://localhost:1527/EmployeeDB";  
4.         String username = "scott";  
5.         String password = "tiger";  
6.         try (Connection con = DriverManager.getConnection(url, username,  
password)) {  
7.             Statement stmt = con.createStatement();  
8.             String query = "DELETE FROM Employee where id=500";  
9.             if (stmt.executeUpdate(query) > 0) {  
10.                 System.out.println("An employee record was deleted successfully");  
11.             }  
12.             String query1="select * from Employee";  
13.             ResultSet rs = stmt.executeQuery(query1);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This slide demonstrates the delete operation. An existing employee record is deleted and the content of the Employee table after the delete operation is displayed in the output console.

Lines 7–10: Create a query to delete an employee record with ID 500 and execute the query.

Lines 12–13: Print the resulting data fields to the system console.

SQLException Class

SQLException can be used to report details about resulting database errors. To report all the exceptions thrown, you can iterate through the SQLExceptions thrown:

```
1 catch(SQLException ex) {  
2     while(ex != null) {  
3         System.out.println("SQLState: " + ex.getSQLState());  
4         System.out.println("Error Code:" + ex.getErrorCode());  
5         System.out.println("Message: " + ex.getMessage());  
6         Throwable t = ex.getCause();  
7         while(t != null) {  
8             System.out.println("Cause:" + t);  
9             t = t.getCause();  
10        }  
11        ex = ex.getNextException();  
12    }  
13 }
```

Vendor-dependent state codes, error codes, and messages



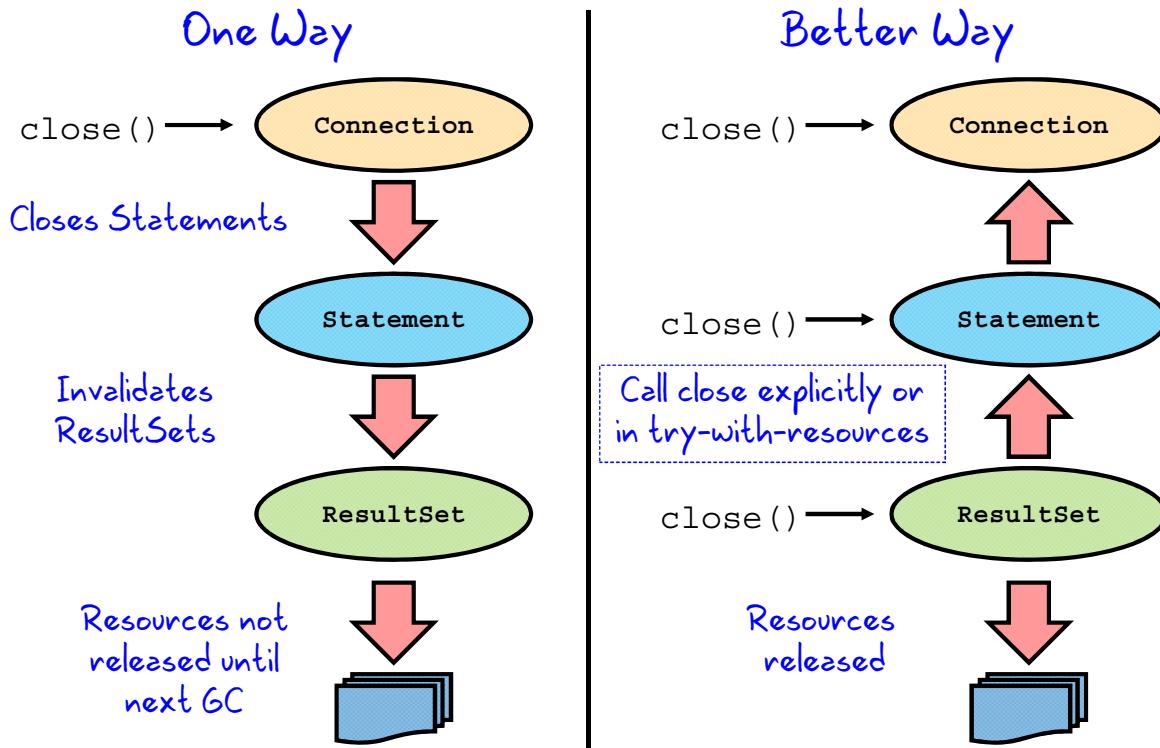
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- A SQLException is thrown from errors that occur in one of the following types of actions: driver methods, methods that access the database, or attempts to get a connection to the database.
- The SQLException class also implements Iterable. Exceptions can be chained together and returned as a single object.
- A SQLException is thrown if the database connection cannot be made due to incorrect username or password information or if the database is offline.
- SQLException can also result by attempting to access a column name that is not part of the SQL query.
- SQLException is also subclassed, providing granularity of the actual exception thrown.

Note: SQLState and SQLerrorCode values are database dependent. For Derby, the SQLState values are defined at:

<http://download.oracle.com/javadb/10.8.1.2/ref/rrefexcept71493.html>

Closing JDBC Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Closing a `Connection` object will automatically close any `Statement` objects created with this `Connection`.
- Closing a `Statement` object will close and invalidate any instances of `ResultSet` created by the `Statement` object.
- Resources held by the `ResultSet` may not be released until garbage is collected. Therefore, it is a good practice to explicitly close `ResultSet` objects when they are no longer needed.
- When the `close()` method on `ResultSet` is executed, external resources are released.
- `ResultSet` objects are also implicitly closed when an associated `Statement` object is re-executed.

In summary, it is a good practice to explicitly close JDBC `Connection`, `Statement`, and `ResultSet` objects when you no longer need them.

Note: A connection with the database can be an expensive operation. It is a good practice to either maintain `Connection` objects for as long as possible, or use a connection pool.

try-with-resources Construct

Given the following try-with-resources statement:

```
try (Connection con =  
     DriverManager.getConnection(url, username, password);  
     Statement stmt = con.createStatement();  
     ResultSet rs = stmt.executeQuery(query)) {
```

- The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`.
 - This interface includes one method: `void close()`.
- The `close()` method is automatically called at the end of the `try` block in the proper order (last declaration to first).
- Multiple closeable resources can be included in the `try` block, separated by semicolons.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

One of the features is the `try-with-resources` statement. This is an enhancement that will automatically close open resources.

With JDBC 4.1, the JDBC API classes including `ResultSet`, `Connection`, and `Statement`, implement `java.lang.AutoCloseable`. The `close()` method of the `ResultSet`, `Statement`, and `Connection` objects will be called in order in this example.

Using PreparedStatement

PreparedStatement is a subclass of Statement that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

Parameter for substitution.
Substitutes value for the first parameter in the prepared statement.

- In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than \$100,000.
- PreparedStatement is useful when you want to execute a SQL statement multiple times.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The PreparedStatement provides two additional benefits:

- Faster execution
- Parameterized SQL Statements

The SQL statement in the example in the slide is precompiled and stored in the PreparedStatement object. This statement can be used efficiently to execute this statement multiple times. This example could be in a loop, looking at different values.

Prepared statements can also be used to prevent SQL injection attacks. For example, where a user is allowed to enter a string and that string is executed as a part of a SQL statement, it enables the user to alter the database in unintended ways (such as granting the user permissions).

Note: PreparedStatement setXXXX methods index parameters from 1, and not 0. The first parameter in a prepared statement is 1, the second parameter is 2, and so on.

Using PreparedStatement: Setting Parameters

In general, there is a **setxxx** method for each type in the Java programming language.

setxxx arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);  
pStmt.setString(2, "Charles");
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Executing PreparedStatement

In general, there is a `setxxx` method for each type in the Java programming language.

`setxxx` arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);  
pStmt.setString(2, "Charles");
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

PreparedStatement : Using a Loop to Set Values

```
PreparedStatement updateEmp;  
String updateString = "update Employee"  
+ "set SALARY= ? where EMP_NAME like ?";  
updateEmp = con.prepareStatement(updateString);  
int[] salary = {1750, 1500, 6000, 1550, 9050};  
String[] names = {"David", "Tom", "Nick",  
"Harry", "Mark"};  
for(int i:names)  
{  
    updateEmp.setInt(1, salary[i]);  
    updateEmp.setString(2, names[i]);  
    updateEmp.executeUpdate();  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

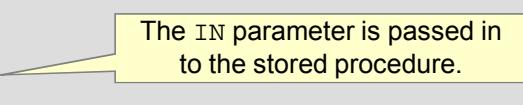
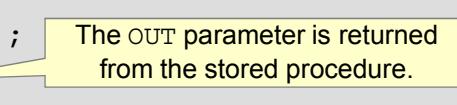
When using a `PreparedStatement` you can make coding easier by using a `for` loop or a `while` loop to set values for input parameters.

The code snippet in the slide demonstrates using a `for` loop to set the values for input parameters.

A `PreparedStatement` object is created and a `for` loop executes five times. Each time through the loop it sets a new value and executes the SQL statement and updates salaries for five different employees.

Using CallableStatement

A CallableStatement allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt  
        = con.prepareCall("{CALL EmplAgeCount (?, ?)}");  
int age = 50;  
cStmt.setInt (1, age);  
  
The IN parameter is passed in  
to the stored procedure.  
ResultSet rs = cStmt.executeQuery();  
cStmt.registerOutParameter(2, Types.INTEGER);  
boolean result = cStmt.execute();  
int count = cStmt.getInt(2);  
  
The OUT parameter is returned  
from the stored procedure.  
System.out.println("There are " + count +  
        " Employees over the age of " + age);
```

- Stored procedures are executed on the database.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stored Procedure

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task. They are used to encapsulate a set of operations or queries to execute on a database server. Stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities.

Calling a Stored Procedure from JDBC

The first step is to create a CallableStatement object. As with Statement and PreparedStatement objects, this is done with an open Connection object. A CallableStatement object contains a call to a stored procedure; it does not contain the stored procedure itself.

Summary

In this lesson, you should have learned how to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API



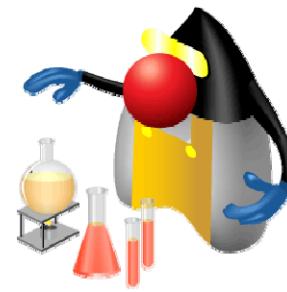
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 18-1 Overview: Working with the Derby Database and JDBC

This practice covers the following topics:

- Starting the JavaDB (Derby) database from within NetBeans IDE
- Populating the database with data (the Employee table)
- Running SQL queries to look at the data
- Compiling and running the sample JDBC application



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this practice, you will start the database from within NetBeans, populate the database with data, run some SQL queries, and compile and run a simple application that returns the rows of the Employee database table.

Quiz

Which Statement method executes a SQL statement and returns the number of rows affected?

- a. `stmt.execute(query);`
- b. `stmt.executeUpdate(query);`
- c. `stmt.executeQuery(query);`
- d. `stmt.query(query);`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

When using a Statement to execute a query that returns only one record, it is not necessary to use the ResultSet's next () method.

- a. True
- b. False



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

19

Localization

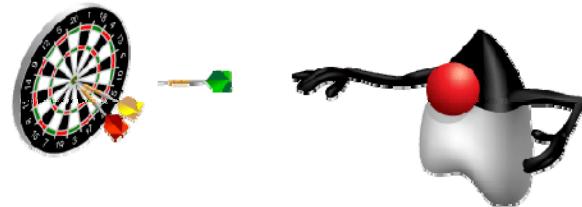
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Create and read a `Properties` file
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle



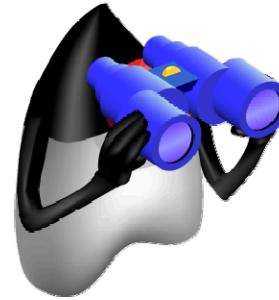
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Why Localize?

The decision to create a version of an application for international use often happens at the start of a development project.

- Region- and language-aware software
- Dates, numbers, and currencies formatted for specific countries
- Ability to plug in country-specific data without changing code



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text.

In addition to language changes, culturally dependent elements, such as dates, numbers, currencies, and so on must be translated.

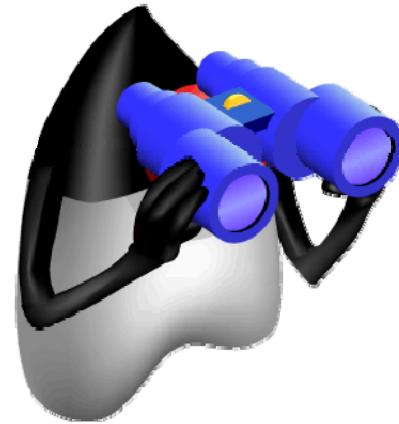
The goal is to design for localization so that no coding changes are required.

A Sample Application

Localize a sample application:

- Text-based user interface
- Localize menus
- Display currency and date localizations

```
==== Localization App ====
1. Set to English
2. Set to French
3. Set to Chinese
4. Set to Russian
5. Show me the date
6. Show me the money!
q. Enter q to quit
Enter a command:
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the remainder of this lesson, this simple text-based user interface will be localized for French, Simplified Chinese, and Russian. Enter the number indicated by the menu and that menu option will be applied to the application. Enter **q** to exit the application.

Locale

A Locale specifies a particular language and country:

- Language
 - An alpha-2 or alpha-3 ISO 639 code
 - “en” for English, “es” for Spanish
 - Always uses lowercase
- Country
 - Uses the ISO 3166 alpha-2 country code or UN M.49 numeric area code
 - “US” for United States, “ES” for Spain
 - Always uses uppercase
- See the Java Tutorials for details of all standards used.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In Java, a locale is specified by using two values: language and country. See the Java Tutorial for standards used:

<http://download.oracle.com/javase/tutorial/i18n/locale/create.html>

Language Samples

- de: German
- en: English
- fr: French
- zh: Chinese

Country Samples

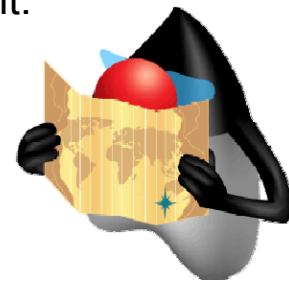
- DE: Germany
- US: United States
- FR: France
- CN: China

Properties

- The `java.util.Properties` class is used to load and save key-value pairs in Java.
- Can be stored in a simple text file:

```
hostName = www.example.com  
userName = user  
password = pass
```

- File name ends in `.properties`.
- File can be anywhere that compiler can find it.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The benefit of a properties file is the ability to set values for your application externally. The properties file is typically read at the start of the application and is used for default values. But the properties file can also be an integral part of a localization scheme, where you store the values of menu labels and text for various languages that your application may support.

The convention for a properties file is `<filename>.properties`, but the file can have any extension you want. The file can be located anywhere that the application can find it.

Loading and Using a Properties File

```
1  public static void main(String[] args) {
2      Properties myProps = new Properties();
3      try {
4          FileInputStream fis = new FileInputStream("ServerInfo.properties");
5          myProps.load(fis);
6      } catch (IOException e) {
7          System.out.println("Error: " + e.getMessage());
8      }
9
10     // Print Values
11     System.out.println("Server: " + myProps.getProperty("hostName"));
12     System.out.println("User: " + myProps.getProperty("userName"));
13     System.out.println("Password: " + myProps.getProperty("password"));
14 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the code fragment, you create a `Properties` object. Then, using a `try` statement, you open a file relative to the source files in your NetBeans project. When it is loaded, the name-value pairs are available for use in your application.

Properties files enable you to easily inject configuration information or other application data into the application.

Loading Properties from the Command Line

- Property information can also be passed on the command line.
- Use the `-D` option to pass key-value pairs:

```
java -Dpropertyname=value -Dpropertyname=value myApp
```

- For example, pass one of the previous values:

```
java -Dusername=user myApp
```

- Get the `Properties` data from the `System` object:

```
String userName = System.getProperty("username");
```

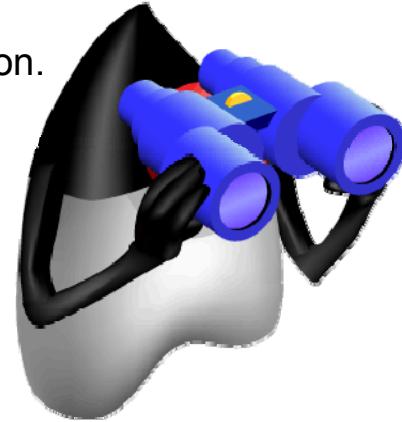


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Property information can also be passed on the command line. The advantage to passing properties from the command line is simplicity. You do not have to open a file and read from it. However, if you have more than a few parameters, a properties file is preferable.

Resource Bundle

- The ResourceBundle class isolates locale-specific data:
 - Returns key/value pairs stored separately
 - Can be a class or a .properties file
- Steps to use:
 - Create bundle files for each locale.
 - Call a specific locale from your application.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Design for localization begins by designing the application so that all the text, sounds, and images can be replaced at run time with the appropriate elements for the region and culture desired. Resource bundles contain key-value pairs that can be hard-coded within a class or located in a .properties file.

Resource Bundle File

- Properties file contains a set of key-value pairs.
 - Each key identifies a specific application component.
 - Special file names use language and country codes.
- Default for sample application:
 - Menu converted into resource bundle

```
MessageBundle.properties
menu1 = Set to English
menu2 = Set to French
menu3 = Set to Chinese
menu4 = Set to Russian
menu5 = Show the Date
menu6 = Show me the money!
menuq = Enter q to quit
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide shows a sample resource bundle file for this application. Each menu option has been converted into a name/value pair. This is the default file for the application. For alternative languages, a special naming convention is used:

MessageBundle_xx_YY.properties

where xx is the language code and YY is the country code.

Sample Resource Bundle Files

Samples for French and Chinese

MessagesBundle_fr_FR.properties

```
menu1 = Régler à l'anglais  
menu2 = Régler au français  
menu3 = Réglez chinoise  
menu4 = Définir pour la Russie  
menu5 = Afficher la date  
menu6 = Montrez-moi l'argent!  
menuq = Saisissez q pour quitter
```

MessagesBundle_zh_CN.properties

```
menu1 = 设置为英语  
menu2 = 设置为法语  
menu3 = 设置为中文  
menu4 = 设置到俄罗斯  
menu5 = 显示日期  
menu6 = 显示我的钱!  
menuq = 输入q退出
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide shows the resource bundle files for French and Chinese. Note that the file names include both language and country. The English menu item text has been replaced with French and Chinese alternatives.

Initializing the Sample Application

```
PrintWriter pw = new PrintWriter(System.out, true);
// More init code here

Locale usLocale = Locale.US;
Locale frLocale = Locale.FRANCE;
Locale zhLocale = new Locale("zh", "CN");
Locale ruLocale = new Locale("ru", "RU");
Locale currentLocale = Locale.getDefault();

ResourceBundle messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);

// more init code here

public static void main(String[] args) {
    SampleApp ui = new SampleApp();
    ui.run();
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

With the resource bundles created, you simply need to load the bundles into the application. The source code in the slide shows the steps. First, create a `Locale` object that specifies the language and country. Then load the resource bundle by specifying the base file name for the bundle and the current `Locale`.

Note that there are a couple of ways to define a `Locale`. The `Locale` class includes default constants for some countries. If a constant is not available, you can use the language code with the country code to define the location. Finally, you can use the `getDefault()` method to get the default location.

Sample Application: Main Loop

```
public void run() {
    String line = "";
    while (!(line.equals("q"))){
        this.printMenu();
        try { line = this.br.readLine(); }
        catch (Exception e){ e.printStackTrace(); }

        switch (line){
            case "1": setEnglish(); break;
            case "2": setFrench(); break;
            case "3": setChinese(); break;
            case "4": setRussian(); break;
            case "5": showDate(); break;
            case "6": showMoney(); break;
        }
    }
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

For this application, a run method contains the main loop. The loop runs until the letter “q” is typed in as input. A string switch is used to perform an operation based on the number entered. A simple call is made to each method to make locale changes and display a formatted output.

The printMenu Method

Instead of text, a resource bundle is used.

- messages is a resource bundle.
- A key is used to retrieve each menu item.
- Language is selected based on the Locale setting.

```
public void printMenu(){  
    pw.println("== Localization App ==");  
    pw.println("1. " + messages.getString("menu1"));  
    pw.println("2. " + messages.getString("menu2"));  
    pw.println("3. " + messages.getString("menu3"));  
    pw.println("4. " + messages.getString("menu4"));  
    pw.println("5. " + messages.getString("menu5"));  
    pw.println("6. " + messages.getString("menu6"));  
    pw.println("q. " + messages.getString("menuq"));  
    System.out.print(messages.getString("menucommand") + " ");  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Instead of printing text, the resource bundle (messages) is called and the current Locale determines what language is presented to the user.

Changing the Locale

To change the Locale:

- Set `currentLocale` to the desired language.
- Reload the bundle by using the current locale.

```
public void setFrench() {  
    currentLocale = frLocale;  
    messages = ResourceBundle.getBundle("MessagesBundle",  
        currentLocale);  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After the menu bundle is updated with the correct locale, the interface text is output by using the currently selected language.

Sample Interface with French

After the French option is selected, the updated user interface looks like the following:

```
==== Localization App ====
1. Réglér à l'anglais
2. Réglér au français
3. Réglez chinoise
4. Définir pour la Russie
5. Afficher la date
6. Montrez-moi l'argent!
q. Saisissez q pour quitter
Entrez une commande:
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The updated user interface is shown in the slide. The first and last lines of the application could be localized as well.

Format Date and Currency

- Numbers can be localized and displayed in their local format.
- Special format classes include:
 - `java.time.format.DateTimeFormatter`
 - `java.text.NumberFormat`
- Create objects using `Locale`.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Changing text is not the only available localization tool. Dates and numbers can also be formatted based on local standards.

Displaying Currency

- Format currency:
 - Get a currency instance from `NumberFormat`.
 - Pass the `Double` to the `format` method.
- Sample currency output:

```
1 000 000 pyō. ru_RU
1 000 000,00 € fr_FR
¥1,000,000.00 zh_CN
£1,000,000.00 en_GB
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Create a `NumberFormat` object by using the selected locale and get a formatted output.

Formatting Currency with NumberFormat

```
1 package com.example.format;
2
3 import java.text.NumberFormat;
4 import java.util.Locale;
5
6 public class NumberTest {
7
8     public static void main(String[] args) {
9
10         Locale loc = Locale.UK;
11         NumberFormat nf = NumberFormat.getCurrencyInstance(loc);
12         double money = 1_000_000.00d;
13
14         System.out.println("Money: " + nf.format(money) + " in
15             Locale: " + loc);
16     }
17 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Set the location and a numeric value to be displayed. Then, set up a `NumberFormat` object with a specified location. Pass the `Double` to the `format` method to print the formatted currency.

Displaying Dates

- Format a date:
 - Get a `DateTimeFormatter` object based on the `Locale`.
 - From the `LocalDateTime` variable, call the `format` method passing the formatter.
- Sample dates:

20 juil. 2011 fr_FR

20.07.2011 ru_RU



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Create a date format object by using the locale and the date is formatted for the selected locale.

Displaying Dates with `DateTimeFormatter`

```
3 import java.time.LocalDateTime;
4 import java.time.format.DateTimeFormatter;
5 import java.time.format.FormatStyle;
6 import java.util.Locale;
7
8 public class DateFormatTest {
9     public static void main(String[] args) {
10
11     LocalDateTime today = LocalDateTime.now();
12     Locale loc = Locale.FRANCE;
13
14     DateTimeFormatter df =
15         DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
16             .withLocale(loc);
17     System.out.println("Date: " + today.format(df)
18                     + " Locale: " + loc.toString());
19 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The setup of the `DateTimeFormatter` is a bit verbose, but fairly clear. A factory is used to specify a style and a locale. Then the formatter is passed to the `LocalDateTime` object's `format` method.

Format Styles

- `DateTimeFormatter` uses the `FormatStyle` enumeration to determine how the data is formatted.
- Enumeration values
 - `SHORT`: Is completely numeric, such as 12.13.52 or 3:30 pm
 - `MEDIUM`: Is longer, such as Jan 12, 1952
 - `LONG`: Is longer, such as January 12, 1952 or 3:30:32 pm
 - `FULL`: Is completely specified date or time, such as Tuesday, April 12, 1952 AD or 3:30:42 pm PST



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `DateTimeFormatter` object uses the `FormatStyle` enumeration to format date, time or date/time.

Note: At the time of this writing, `FULL` and `LONG` can only be used with date or time return values. Only `MEDIUM` or `SHORT` can be used with date/time objects. Using the wrong value may result in a runtime error. We have not yet determined whether this is a feature or a bug.

Summary

In this lesson, you should have learned how to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Create and read a `Properties` file
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle

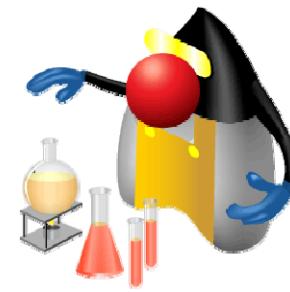


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 19-1 Overview: Creating a Localized Date Application

This practice covers creating a localized application that displays dates in a variety of formats.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which bundle file represents a language of Spanish and a country code of US?

- a. MessagesBundle_ES_US.properties
- b. MessagesBundle_es_es.properties
- c. MessagesBundle_es_US.properties
- d. MessagesBundle_ES_us.properties



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which date format constant provides the most detailed information?

- a. LONG
- b. FULL
- c. MAX
- d. COMPLETE



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.



Oracle Cloud

An Overview

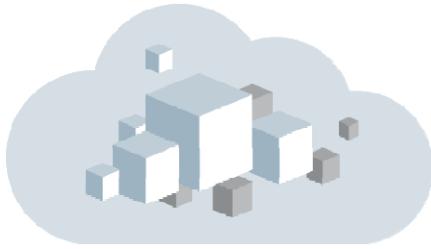
ORACLE®

20



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Agenda



- 1** What is Cloud Computing?
- 2** Cloud Evolution
- 3** Components of Cloud Computing
- 4** Characteristics and Benefits of Cloud
- 5** Cloud Deployment Models
- 6** Cloud Service Models
- 7** Oracle Cloud Services

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

What is Cloud?

The term Cloud refers to a Network or Internet.

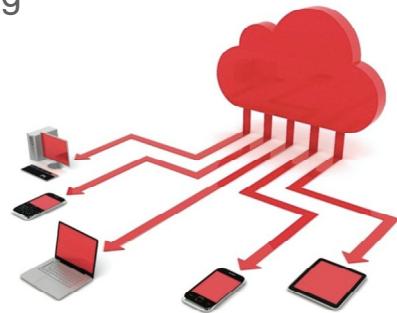
It is a means to access any Software that is available remotely.



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

What is Cloud Computing?

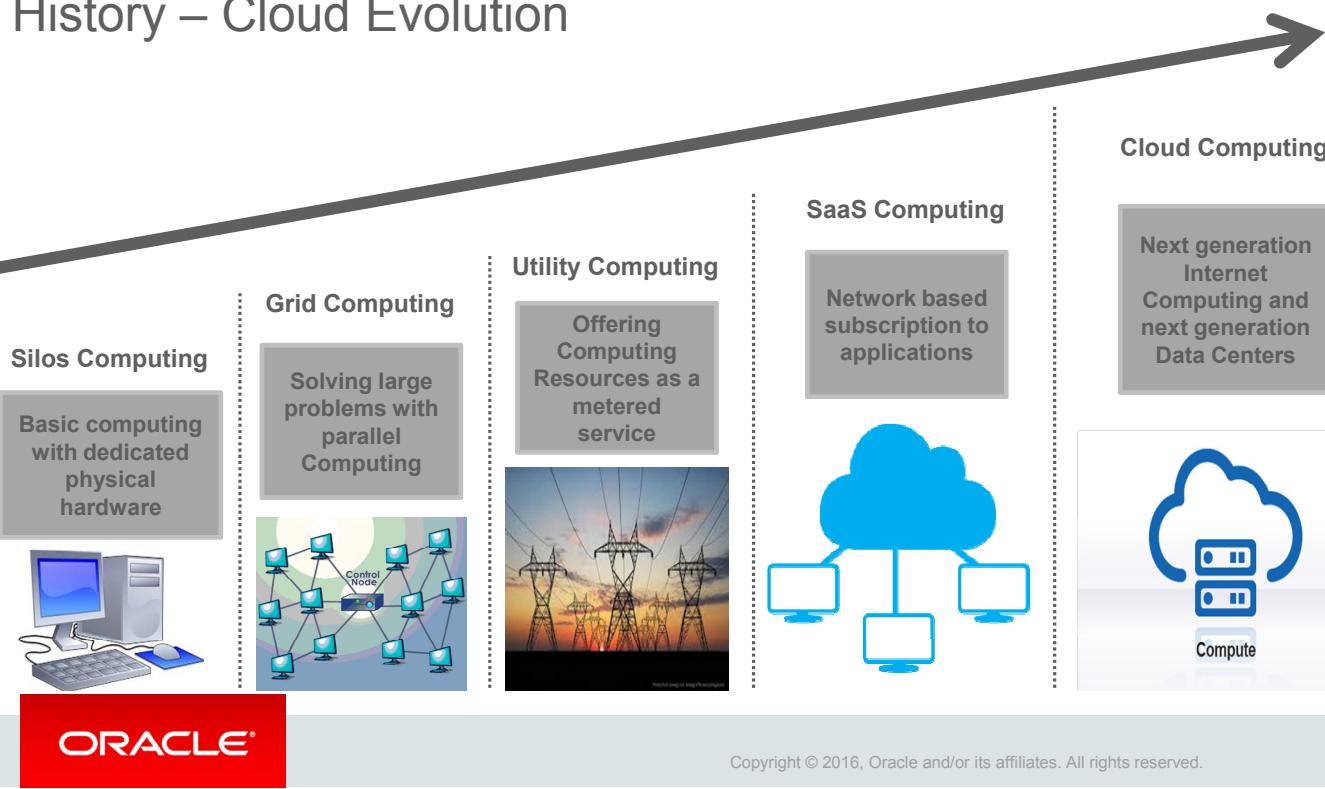
- It is a means to access any Software that is available remotely.
- Refers to the practice of using remote Servers hosted on Internet to store, manage and process data
- When you store your photos online instead of on your home computer, or use webmail or a social networking site, you are using a “cloud computing” service.



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

History – Cloud Evolution



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

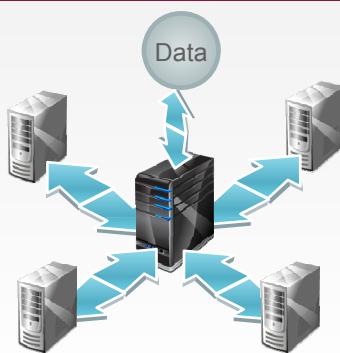
Components of Cloud Computing



Devices that end user interact with cloud. Types of client Thick, Thin (Most popular), Mobile

ORACLE®

Distributed Servers



Often Servers are in geographically different places, but server acts as if they are next to each other

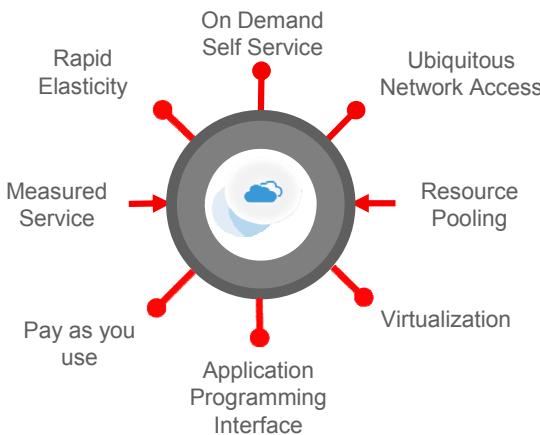
Data Centers



Collection of servers where application is placed and is accessed via Internet

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Characteristics of Cloud



Description

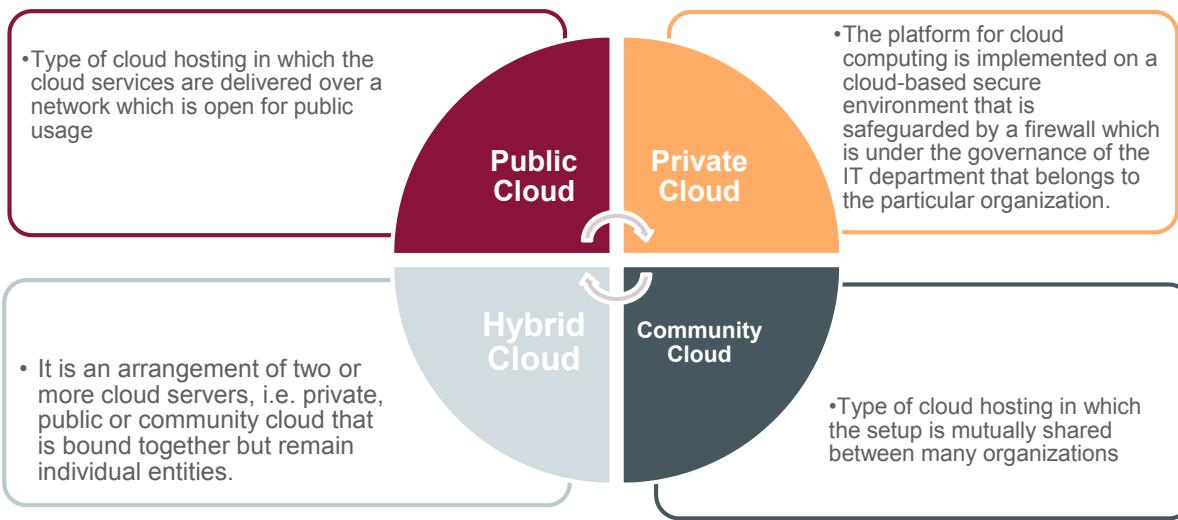
- Allows users to use the service on demand
- Anywhere, Anytime and Any Device
- Draw from a pool of computing resources, usually in remote data centers
- Request and manage own computing resources
- Service is measured and customers are billed accordingly
- Select a configuration of CPU, Memory and storage
- Services can be scaled larger or smaller

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Deployment Models

Deployment models define the type of access to the Cloud.



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models

All three tiers of computing delivered as Service via global network

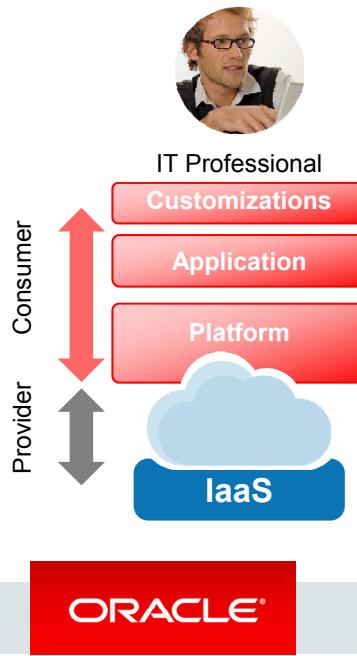
- **Applications:** Software as a Service - SaaS
- **Platform:** Database, Middleware, Analytics, Integration as a Service – Platform as a Service - PaaS
- **Infrastructure:** Storage, Compute, and Network as a service – Infrastructure as a Service - IaaS



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models

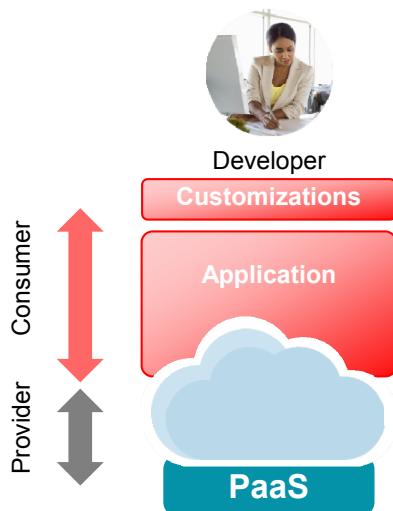


- Provides computer hardware (servers, networking technology, storage and data center space) as a web based service.
- Virtual Machines with pre-installed Operating System
- Target: Administrators
- Ready to Rent

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models

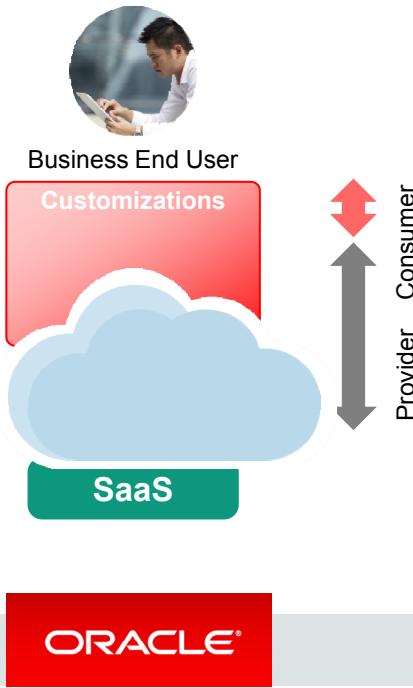


- Provides platform to develop and deploy applications
- Up to Date Software
- Target: Application Developers
- Ready to Use

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models



- Allows usage of the software remotely as a web based service
- Software are automatically Upgraded and Updated
- All Users are running the same version of the Software
- Target: End Users
- Ready to Wear

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Industry Shifting from **On-Premises to the Cloud**

Transition to the Cloud is driven by a desire for:

- **Agility:** Self-service provisioning – deploy a database in minutes
- **Elasticity:** Scale on demand
- **Lower cost:** Reduction in management and total cost – pay for what is used
- **Back to core business:** Focus on core activities
- **More mobility:** Access from any device



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

ORACLE CLOUD

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle IaaS Overview

IaaS

Designed for large enterprises, which allow them to scale up their computing, networking, and storage systems into the cloud, rather than expanding their physical infrastructure.

- Allows large businesses and organizations to run their workloads, replicate their network, and back up their data in the cloud.



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle PaaS Overview

PaaS

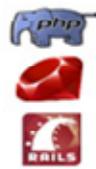
- Develop, deploy, integrate and manage applications on cloud.
- Seamless integration across PaaS and SaaS Applications.



Database Services



Java Services



Web Scripting Services



Mobile Services



Developer Services



Documents Services



Sites Services



Analytics Services

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle SaaS Overview

SaaS

Delivers modern cloud applications that connect business processes across the enterprise.

- Only Cloud integrating ERP, HCM, EPM, SCM
- Seamless co-existence with Oracle's On-Premise Applications



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have :

- Got an overview of Cloud Computing, its Characteristics, History and Technology
- Understood the various components , Deployment Models and Service Models of Cloud Computing
- Understood the Oracle Cloud Services



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.



Oracle Application Container Cloud Service Overview

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

21

Objectives

After completing this lesson, you should be able to:

- Get an overview of Oracle Application Container Cloud
- Understand the unique features of Oracle Application Container Cloud
- Understand how to build, zip, and deploy applications to the cloud



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle Application Container Cloud Service

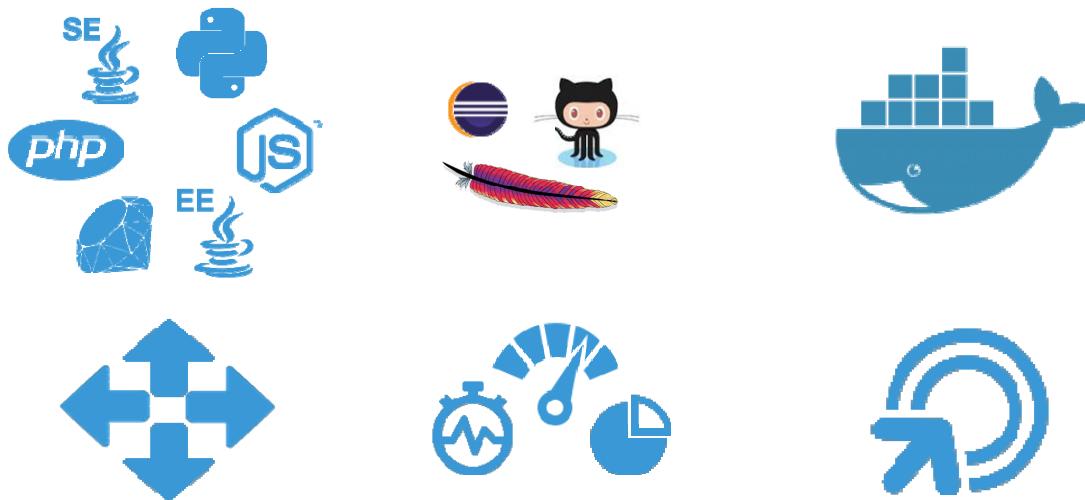


An open highly available
Docker container-based
elastic polyglot cloud
application platform

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

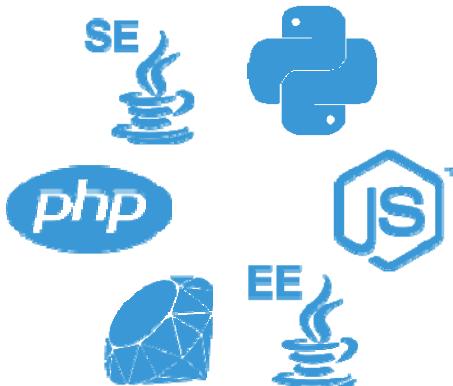
Oracle Application Container Cloud



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Polyglot Platform



ORACLE®

Runtime releases regularly updated to the latest

Deploy applications to a selection of popular language runtimes supported

- Latest release supports Java SE, Java EE Web Apps, Node.js, and PHP

Leverage unique Oracle Java SE features

- Immediate access to platform upgrades, security, platform optimizations
- Continued commercial support for Java SE versions no longer receiving public updates

Node access to Oracle DB with open source database driver

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

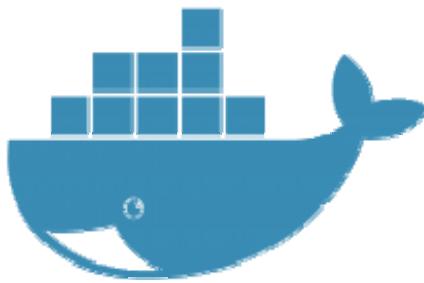
Open Platform



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Use any of the thousands of open source or commercial Java or Node frameworks—no restrictions.

Container-based Application Platform as a Service



Applications run on Oracle Linux in Docker containers

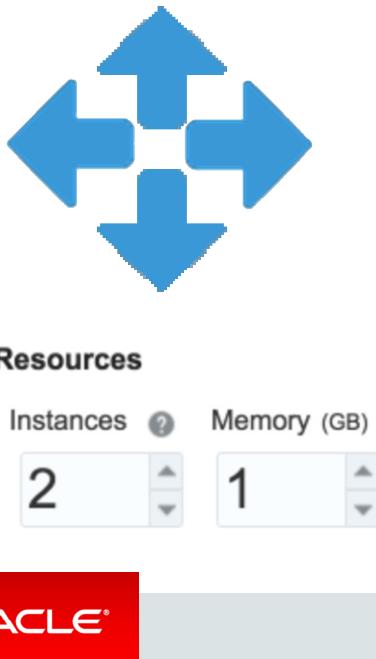
Stateless Applications

- Ephemeral disk
- Permanent storage through database or storage service

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Elastic Scaling



On demand elastic scaling either through the service console or using the service REST API

Scale out / in

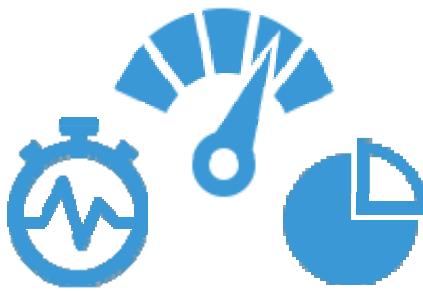
- Add / remove application instances to handle workloads

Scale up / down

- Add / remove RAM to accommodate application memory requirements

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Profiling



Java application can use Java Flight Recorder to monitor application and JVM behavior and analyze in Mission Control

Use Application Performance Monitoring Cloud Service for advanced use cases

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Manageable



New Java and Node releases published in the service console
One-click upgrade to the latest releases—applications are simply restarted to upgrade to new runtime

Updates ①

Current Version: Java SE 8u71

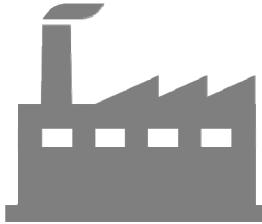
Available Updates

	Runtime: Java SE 8u91 Release Notes	Release Date: Jul 4, 2015 12:00:00 AM UTC	Update
Description: This update contains new features as well as fix for critical issues. Refer to the 'Release Notes' for more details			

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Build Zip Deploy!



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Build

- Use your favourite or corporate standard build system to produce binaries and deployable resources.

Zip

- Zip up all binaries, scripts, html files, images, etc. that make up your application. The structure of the zip is entirely up to the user—we have no opinion on structure.

Deploy

- Deploy the application archive (zip) to the platform and tell us how to start the application. This could be “java –jar”, “java –classpath ... <main>”, “node myapp.js”, or “sh bootmyapp.sh”.

Deploy—Application Archive (Zip)

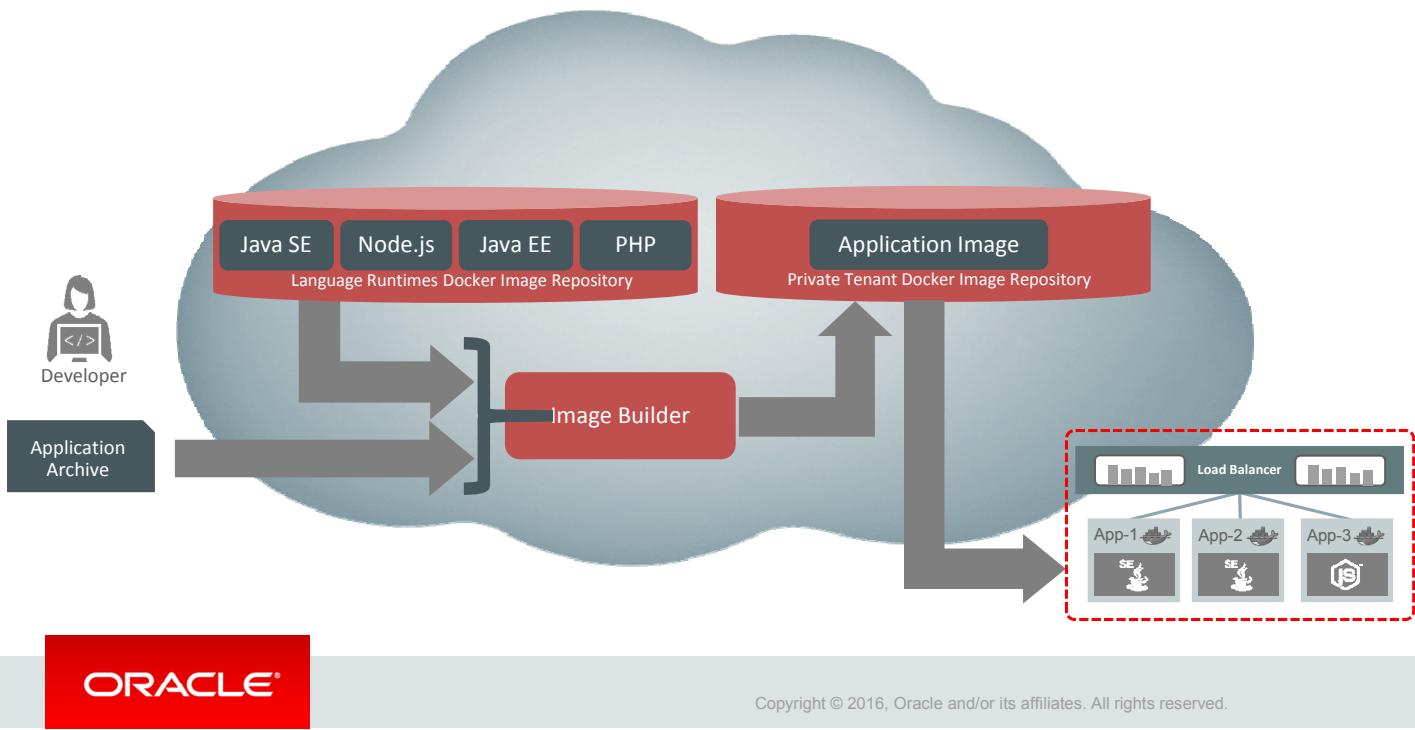
- All application binaries
- All required libraries
- Binaries of any container/embedded container
- Images files
- HTML files

Everything you'd need to run your application on a virgin machine



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

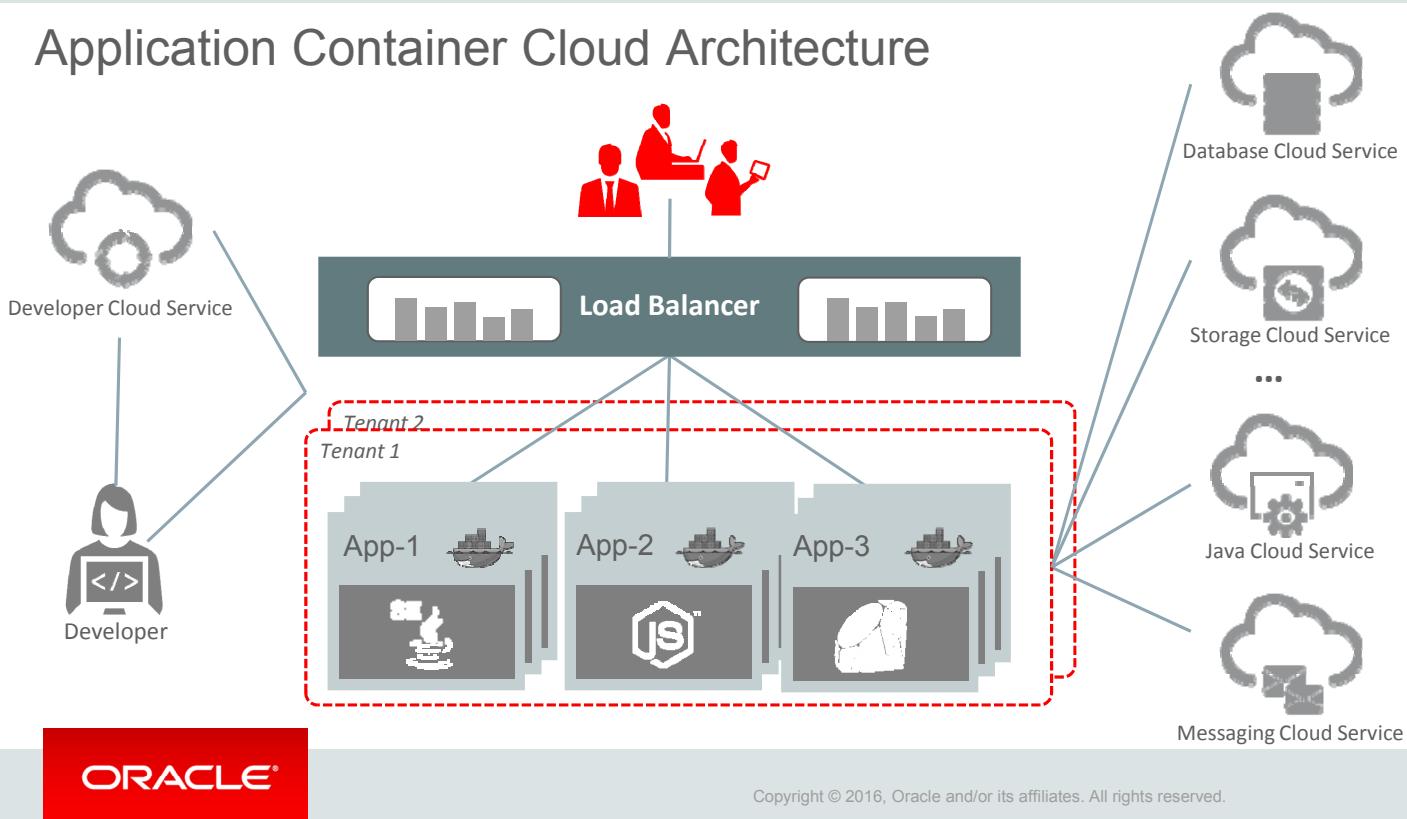
Application Deployment



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Application Container Cloud Architecture



- Tenant Isolation
- Polyglot
- Integrated
- Developer Friendly

Load Balancer



Fully automated—no user management required

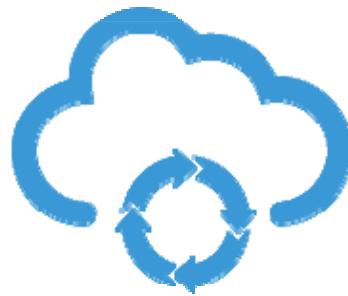
Scale out or in and application instances are automatically registered/unregistered

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Vanity URL support (upcoming) will allow installation of certificates

Oracle Developer Cloud Service



**Source Control
Management**



Issue Tracking



**Hudson Continuous
Integration**



Wiki Collaboration

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Complete, Integrated Development Platform—as a Service

Application Lifecycle Management

Team Management

Entitlement with all Application Container Cloud services

Developer Cloud Service – Easy Adoption/Integration

Pre-integrated development technologies in the cloud

Standards Based

- Git, Maven, Hudson, Ant, Grunt, Gulp, etc.

Built-in IDE Integration

- Eclipse, NetBeans, JDeveloper

Flexible Source Location

- Hosted Git or GitHub

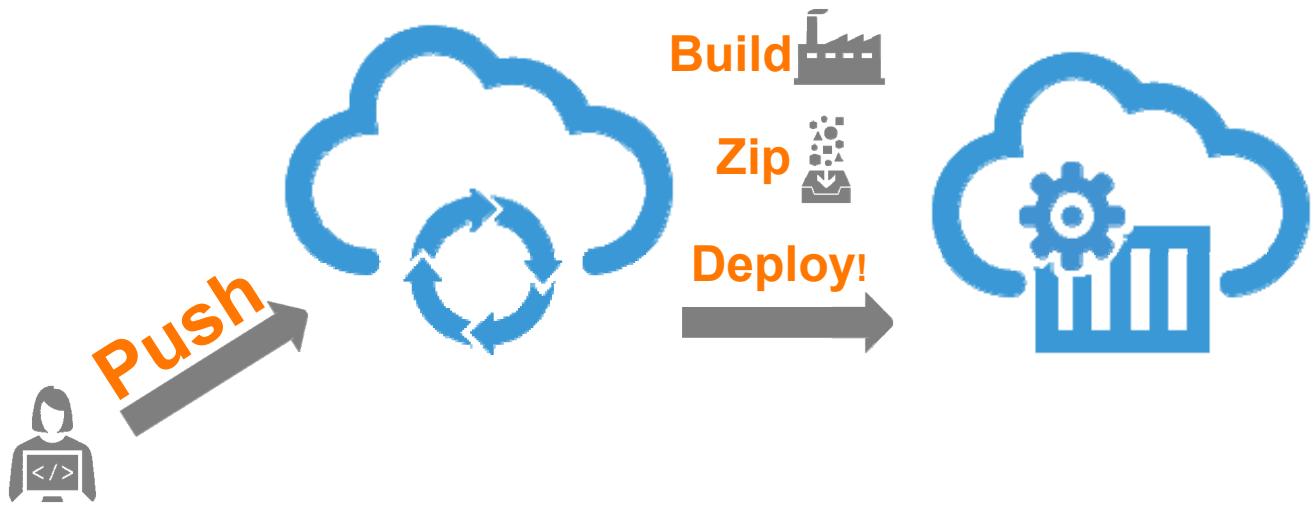
Choice of Deployment Target

- Oracle Cloud or on-premise



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

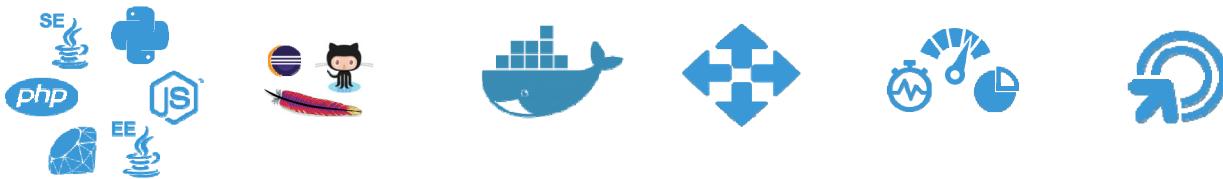


ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Rather than build on-premise, use DevCS to perform continuous build, test, and deployment.

Application Container Cloud Service Advantages



- Integrated **enterprise** ecosystem and services from IaaS to PaaS and SaaS
- Java SE Advanced – completely **unique** and unavailable on any other cloud platform
- Developer Cloud Service – **included** and **integrated**

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have :

- Got an overview of Oracle Application Container Cloud
- Understood the unique features of Oracle Application Container Cloud
- Understood how to build, zip, and deploy applications to the cloud



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.