



Integrated Cloud Applications & Platform Services

## Java SE: Programming II

Student Guide - Volume II

D102474GC10

Edition 1.0 | May 2019 | D105723

Learn more from Oracle University at [education.oracle.com](https://education.oracle.com)



## Authors

Kenny Somerville  
Anjana Shenoy  
Nick Ristuccia

## Technical Contributors and Reviewers

Joe Greenwald  
Jeffrey Picchione  
Joe Boulenouar  
Steve Watts  
Pete Iaseau  
Henry Jen  
Nick Ristuccia  
Alex Buckley  
Vasily Strelnikov  
Aurelio García-Ribeyro  
Stuart Marks  
Geertjan Wielenga  
Mike Williams

## Editors

Moushmi Mukherjee  
Raj Kumar

## Graphic Designers

Anne Elizabeth  
Yogita Chawdhary  
Kavya Bellur

## Publishers

Asief Baig  
Jayanthi Keshavamurthy

**Copyright © 2019, 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 Objectives 1-2
- Introductions 1-4
- Audience 1-5
- Prerequisites 1-6
- Course Roadmap 1-7
- Lesson Format 1-12
- Practice Environment 1-13
- How Do You Learn More After the Course? 1-14
- Additional Resources 1-15
- Summary 1-17
- Practice 1: Overview 1-18

### 2 Java OOP Review

- Objectives 2-2
- Java Language Review 2-3
- A Simple Java Class: Employee 2-4
- Encapsulation: Private Data, Public Methods 2-5
- Subclassing 2-6
- Constructors in Subclasses 2-7
- Using super 2-8
- Using Access Control 2-9
- Protected Access Control: Example 2-10
- Inheritance: Accessibility of Overriding Methods 2-11
- Final Methods 2-12
- Final Classes 2-13
- Applying Polymorphism 2-14
- Overriding methods of Object Class 2-16
- Overriding methods of Object Class: `toString` Method 2-17
- Overriding methods of Object Class: `equals` Method 2-18
- Overriding methods of Object Class: `hashCode` Method 2-20
- Casting Object References 2-21
- Upward Casting Rules 2-22
- Downward Casting Rules 2-23
- Methods Using Variable Arguments 2-24

Static Imports	2-26
Nested Classes	2-27
Example: Member Class	2-28
What Are Enums?	2-29
Complex Enums	2-30
Methods in Enums	2-31
Summary	2-33
Practice 2: Overview	2-34
Quiz	2-35

### **3 Exception Handling and Assertions**

Objectives	3-2
Error Handling	3-3
Exception Types	3-4
Exception Handling Techniques in Java	3-6
Exception Handling Techniques: try Block	3-7
Exception Handling Techniques: finally Clause	3-8
Exception Handling Techniques: try-with-resources	3-9
Exception Handling Techniques: try-with-resources Improvements	3-10
Exception Handling Techniques:   operator in a catch block	3-12
Exception Handling Techniques: Declaring Exceptions	3-13
Creating Custom Exceptions	3-14
Assertions	3-15
Assertion Syntax	3-16
Internal Invariants	3-17
Control Flow Invariants	3-18
Class Invariants	3-19
Controlling Runtime Evaluation of Assertions	3-20
Summary	3-21
Practice 3: Overview	3-22
Quiz	3-23

### **4 Java Interfaces**

Objectives	4-2
Java Interfaces	4-3
Java SE 7 Interfaces	4-4
Implementing Java SE 7 Interface Methods	4-5
Example: Implementing abstract Methods	4-6
Example: Duplicating Logic	4-7
Implementing Methods in Interfaces	4-8
Example: Implementing default Methods	4-9

Example: Inheriting default Methods 4-10  
Example: Overriding a default Method 4-11  
What About the Problems of Multiple Inheritance? 4-12  
Inheritance Rules of default Methods 4-13  
Interfaces Don't Replace Abstract Classes 4-16  
What If default Methods Duplicate Logic? 4-17  
Duplication Between default Methods 4-18  
The Problem with This Approach 4-19  
Introducing private Methods in Interfaces 4-20  
Example: Using private Methods to Reduce Duplication Between  
default Methods 4-21  
Types of Methods in Java SE 9 Interfaces 4-22  
Anonymous Inner Classes 4-23  
Anonymous Inner Class: Example 4-24  
Summary 4-25  
Practice 4: Overview 4-26  
Quiz 4-27

## 5 Collections and Generics

Objectives 5-2  
Type-Wrapper Classes 5-3  
Autoboxing and Auto-Unboxing 5-4  
Generic Methods 5-5  
A Generic Method 5-6  
Generic Classes 5-7  
Generic Cache Class 5-8  
Testing the Generic Cache Class 5-9  
Generics with Type Inference Diamond Notation 5-10  
Java SE 9: Diamond Notation with Anonymous Inner Classes 5-11  
Collections 5-12  
Collections Framework in Java 5-13  
Benefits of the Collections Framework 5-14  
Collection Types 5-15  
Key Collections Interfaces 5-16  
ArrayList 5-17  
ArrayList Without Generics 5-18  
Generic ArrayList 5-19  
TreeSet: Implementation of Set 5-20  
Map Interface 5-21  
TreeMap: Implementation of Map 5-22  
Stack with Deque: Example 5-23

Ordering Collections	5-24
Comparable: Example	5-25
Comparable Test: Example	5-26
Comparator Interface	5-27
Comparator: Example	5-28
Comparator Test: Example	5-29
Wildcards	5-30
Wildcards: Upper Bound	5-31
Why Use Generics?	5-32
Java SE 9: Convenience Methods for Collections	5-33
of Convenience Method	5-34
Overloading of Method	5-35
ofEntries Method for Maps	5-36
Features of Convenience Methods	5-37
Summary	5-38
Practice 5: Overview	5-39
Quiz	5-40

## **6 Functional Interfaces and Lambda Expressions**

Objectives	6-2
Problem Statement	6-3
RoboCall Class	6-4
RoboCall Every Person	6-5
RoboCall Use Case: Eligible Drivers	6-6
RoboCall Use Case: Eligible Voters	6-7
RoboCall Use Case: Legal Drinking Age	6-8
Solution: Parameterization of Values	6-9
Solution: Parameterized Methods	6-10
Parameters for Age Range	6-11
Using Parameters for Age Range	6-12
Corrected Use Case	6-13
Parameterized Computation	6-14
How To Pass a Function in Java?	6-15
Prior to Java SE 8: Pass a Function Wrapped in an Object	6-16
Prior to SE 8: Abstract Behavior With an Interface	6-17
Prior to SE 8: Replace Implementation Class with Anonymous Inner Class	6-18
Lambda Solution: Replace Anonymous Inner Class with Lambda Expression	6-19
Rewriting the Use Cases Using Lambda	6-20
What Is a Lambda?	6-21
What is a Functional Interface	6-22
Which of These Interfaces Are Functional Interfaces?	6-23

Lambda Expression	6-24
Using Lambdas	6-25
Which of The Following Are Valid Lambda Expressions?	6-26
Lambda Expression: Type Inference	6-27
To Create a Lambda Expression	6-28
Examples of Lambdas	6-29
Statement Lambdas: Lambda with Body	6-30
Examples: Lambda Expression	6-31
Lambda Parameters	6-32
Local-Variable Syntax for Lambda Parameters	6-33
Functional Interfaces: Predicate	6-35
Using Functional Interfaces	6-36
Quiz	6-37
Summary	6-39
Practice 6: Overview	6-40

## **7 Collections, Streams, and Filters**

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

## **8 Lambda Built-in Functional Interfaces**

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

Consumer	8-8
Consumer: Example	8-9
Function	8-10
Function: Example	8-11
Supplier	8-12
Supplier: Example	8-13
Primitive Interface	8-14
Return a Primitive Type	8-15
Return a Primitive Type: Example	8-16
Process a Primitive Type	8-17
Process Primitive Type: Example	8-18
Binary Types	8-19
Binary Type: Example	8-20
Unary Operator	8-21
UnaryOperator: Example	8-22
Wildcard Generics Review	8-23
A Closer Look at Consumer	8-24
Interface List<E> use of forEach method	8-25
Example of Range of Valid Parameters	8-26
Plant Class Example	8-27
TropicalFruit Class Example	8-29
Use of Generic Expressions and Wildcards	8-30
Consumer andThen method	8-31
Using Consumer.andThen method	8-32
Summary	8-34
Practice 8: Overview	8-35

## 9 Lambda Operations

Objectives	9-2
Streams API	9-3
Types of Operations	9-4
Extracting Data with Map	9-5
Taking a Peek	9-6
Search Methods: Overview	9-7
Search Methods	9-8
Optional Class	9-9
Short-Circuiting Example	9-10
Stream Data Methods	9-11
Performing Calculations	9-12
Sorting	9-13
Comparator Updates	9-14

Saving Data from a Stream	9-15
Collectors Class	9-16
Quick Streams with Stream.of	9-17
Flatten Data with flatMap	9-18
flatMap in Action	9-19
Summary	9-20
Practice 9: Overview	9-21

## 10 The Module System

Objectives	10-2
Module System	10-3
Module System: Advantages	10-4
Java Modular Applications	10-5
What Is a Module?	10-6
A Modular Java Application	10-7
Issues with Access Across Nonmodular JARs	10-8
Dependencies Across Modules	10-9
What Is a Module?	10-11
Module Dependencies with requires	10-12
Module Package Availability with exports	10-13
Module Graph 1	10-14
Module Graph 2	10-15
Transitive Dependencies	10-16
Access to Types via Reflection	10-17
Example Hello World Modular Application Code	10-18
Example Hello World Modular File Structure	10-19
Compiling a Modular Application	10-20
Single Module Compilation Example	10-21
Multi Module Compilation Example	10-22
Creating a Modular JAR	10-23
Running a Modular Application	10-24
The Modular JDK	10-25
Java SE Modules	10-26
The Base Module	10-28
Finding the Right Platform Module	10-29
Illegal Access to JDK Internals in JDK 9	10-30
What Is a Custom Runtime Image?	10-31
Link Time	10-32
Using jlink to Create a Runtime Image	10-33
Example: Using jlink to Create a Runtime Image	10-34
Examining the Generated Image	10-35

Modules Resolved in a Custom Runtime Image	10-36
Advantages of a Custom Runtime Image	10-37
JIMAGE Format	10-38
Running the Application	10-39
Summary	10-41
Practice 10: Overview	10-42

## **11 Migrating to a Modular Application**

Objectives	11-2
Topics	11-3
The League Application	11-4
Run the Application	11-5
The Unnamed Module	11-6
Topics	11-7
Top-down Migration and Automatic Modules	11-8
Automatic Module	11-9
Top-Down Migration	11-10
Creating module-info.java—Determining Dependencies	11-11
Check Dependencies	11-12
Library JAR to Automatic Module	11-13
Typical Application Modularized	11-14
Topics	11-15
Bottom-up Migration	11-16
Bottom-Up Migration	11-17
Modularized Library	11-18
Run Bottom-Up Migrated Application	11-19
Fully Modularized Application	11-20
Module Resolution	11-21
Topics	11-22
More About Libraries	11-23
Run Application with Jackson Libraries	11-24
Open Soccer to Reflection from Jackson Libraries	11-25
Topics	11-26
Split Packages	11-27
Splitting a Java 8 Application into Modules	11-28
Java SE 8 Application Poorly Designed with Split Packages	11-29
Migration of Split Package JARs to Java SE 9	11-30
Addressing Split Packages	11-31
Topics	11-32
Cyclic Dependencies	11-33
Addressing Cyclic Dependency 1	11-34

Addressing Cyclic Dependency 2 11-35  
Top-down or Bottom-up Migration Summary 11-36  
Summary 11-37  
Practice 11: Overview 11-38

## **12 Services in a Modular Application**

Objectives 12-2  
Topics 12-3  
Modules and Services 12-4  
Components of a Service 12-5  
Produce and Consume Services 12-6  
Module Dependencies Without Services 12-7  
Service Relationships 12-8  
Expressing Service Relationships 12-9  
Topics 12-10  
Using the Service Type in competition 12-11  
Choosing a Provider Class 12-12  
Module Dependencies and Services 1 12-14  
Module Dependencies and Services 2 12-15  
Module Dependencies and Services 3 12-16  
Designing a Service Type 12-17  
Topics 12-18  
TeamGameManager Application with Additional Services 12-19  
module-info.java for competition module 12-20  
module-info.java for league and knockout modules 12-21  
module-info.java for soccer and basketball modules 12-22  
Summary 12-23  
Practice 12: Overview 12-24

## **13 Concurrency**

Objectives 13-2  
Task Scheduling 13-3  
Legacy Thread and Runnable 13-4  
Extending Thread 13-5  
Implementing Runnable 13-6  
The java.util.concurrent Package 13-7  
Recommended Threading Classes 13-8  
java.util.concurrent.ExecutorService 13-9  
Example ExecutorService 13-10  
Shutting Down an ExecutorService 13-11  
java.util.concurrent.Callable 13-12

Example Callable Task	13-13
java.util.concurrent.Future	13-14
Example	13-15
Threading Concerns	13-16
Shared Data	13-17
Problems with Shared Data	13-18
Nonshared Data	13-19
Atomic Operations	13-20
Out-of-Order Execution	13-21
The synchronized Keyword	13-22
synchronized Methods	13-23
synchronized Blocks	13-24
Object Monitor Locking	13-25
Threading Performance	13-26
Performance Issue: Examples	13-27
java.util.concurrent Classes and Packages	13-28
The java.util.concurrent.atomic Package	13-29
java.util.concurrent.CyclicBarrier	13-30
Thread-Safe Collections	13-32
CopyOnWriteArrayList: Example	13-33
Summary	13-34
Practice 13: Overview	13-35
Quiz	13-36

## **14 Parallel Streams**

Objectives	14-2
Streams Review	14-3
Old Style Collection Processing	14-4
New Style Collection Processing	14-5
Stream Pipeline: Another Look	14-6
Styles Compared	14-7
Parallel Stream	14-8
Using Parallel Streams: Collection	14-9
Using Parallel Streams: From a Stream	14-10
Pipelines Fine Print	14-11
Embrace Statelessness	14-12
Avoid Statefulness	14-13
Streams Are Deterministic for Most Part	14-14
Some Are Not Deterministic	14-15
Reduction	14-16
Reduction Fine Print	14-17

Reduction: Example	14-18
A Look Under the Hood	14-24
Illustrating Parallel Execution	14-25
Performance	14-36
A Simple Performance Model	14-37
Summary	14-38
Practice 14: Overview	14-39

## **15 Terminal Operations: Collectors**

Objectives	15-2
Agenda	15-3
Streams and Collectors Versus Imperative Code	15-4
Collection	15-5
Predefined Collectors	15-6
A Simple Collector	15-7
A More Complex Collector	15-8
Agenda	15-9
The Three Argument collect Method of Stream	15-10
The collect Method Used with a Sequential Stream	15-11
The collect Method Used with a Parallel Stream	15-12
The collect Method: Collect to an ArrayList Example	15-13
Agenda	15-14
The Single Argument collect Method of Stream	15-15
Using Predefined Collectors From the Collectors Class	15-16
List of Predefined Collectors	15-17
Stand-Alone Collectors	15-18
Stand-Alone Collector:List all Elements	15-19
maxBy() Example	15-20
Adapting Collector: filtering()	15-21
Composing Collectors : groupingBy()	15-22
Composing Collectors: Using Mapping	15-23
toMap() and Duplicate Keys	15-24
Agenda	15-25
groupingBy and partitioningBy Collectors	15-26
Stand-Alone groupingBy: Person Elements By City	15-27
Stream Operations Or Equivalent Collectors?	15-28
Stream Operations Or Equivalent Collectors with groupingBy	15-29
Stream.count(), Collectors.counting and groupingBy	15-30
Composing Collectors: Tallest in Each City	15-31
groupingBy: Additional Processing with entrySet()	15-32
Agenda	15-33

Nested Values	15-34
Data Organization of ComplexSalesTxn	15-35
Displaying Nested Values: Listing Line Items in Each Transaction	15-36
Displaying Nested Values: Listing Line items	15-37
Displaying Nested Values: Grouping LineItem elements	15-38
groupingBy Examples for ComplexSalesTxn	15-39
Group Items by Salesperson	15-40
Agenda	15-42
Complex Custom Collectors	15-43
The collect Method: Using a Custom Collector	15-44
Creating a Custom Collector: Methods to Implement	15-45
Custom Example MyCustomCollector	15-46
Finisher Example MyCustomCollector	15-47
A More Complex Collector	15-48
A More Complex Collector CustomGroupingBy	15-49
Summary	15-51
Practice 15: Overview	15-52

## 16 Creating Custom Streams

Objectives	16-2
Topics	16-3
Performance: Intuition and Measurement	16-4
Parallel Versus Sequential Example	16-5
Spliterator	16-6
Decomposition with trySplit()	16-8
Integration with Streams	16-9
Modifying LongStream Spliterator	16-10
Spliterator TestSpliterator	16-11
Using TestSpliterator	16-12
Topics	16-13
Creating a Custom Spliterator	16-14
Is a Custom Spliterator Needed for a Game Engine?	16-15
A Tic-Tac-Toe Engine Using the map Method of Stream	16-16
A Custom Spliterator Example for a Custom Collection	16-17
Custom N-ary Tree	16-18
Possible Implementation for NaryTreeSpliterator	16-19
Stream<Node> in Use	16-20
Implementing Parallel Processing	16-21
Stream<Node> in Use	16-22
Summary of NTreeAsListSpliterator	16-23

Summary 16-24

Practice 16: Overview 16-25

## 17 Java I/O Fundamentals and File I/O (NIO.2)

Objectives 17-2

Java I/O Basics 17-3

I/O Streams 17-4

I/O Application 17-5

Data Within Streams 17-6

Byte Stream InputStream Methods 17-7

Byte Stream: Example 17-8

Character Stream Methods 17-9

Character Stream: Example 17-10

I/O Stream Chaining 17-11

Chained Streams: Example 17-12

Console I/O 17-13

Writing to Standard Output 17-14

Reading from Standard Input 17-15

Channel I/O 17-16

Persistence 17-17

Serialization and Object Graphs 17-18

Transient Fields and Objects 17-19

Transient: Example 17-20

Serial Version UID 17-21

Serialization: Example 17-22

Writing and Reading an Object Stream 17-23

Serialization Methods 17-24

readObject: Example 17-25

New File I/O API (NIO.2) 17-26

Limitations of java.io.File 17-27

File Systems, Paths, Files 17-28

Relative Path Versus Absolute Path 17-29

Java NIO.2 Concepts 17-30

Path Interface 17-31

Path Interface Features 17-32

Path: Example 17-33

Removing Redundancies from a Path 17-34

Creating a Subpath 17-35

Joining Two Paths 17-36

Symbolic Links 17-37

Working with Links 17-38

File Operations	17-39
BufferedReader File Stream	17-40
NIO File Stream	17-41
Read File into ArrayList	17-42
Managing Metadata	17-43
Summary	17-44
Quiz	17-45
Practice 17: Overview	17-50

## **18 Secure Coding Guidelines**

Objectives	18-2
Java SE Security Overview	18-3
Secure Coding Guidelines	18-5
Vulnerabilities	18-6
Secure Coding Antipatterns	18-7
Antipatterns in Java	18-8
Fundamentals	18-9
Fundamentals: Why Should I Care?	18-16
Denial of Service	18-17
Confidential Information	18-20
Injection and Inclusion	18-24
Accessibility and Extensibility	18-26
Input Validation	18-31
Mutability	18-32
Object Construction	18-35
Serialization and Deserialization	18-38
Summary	18-40
Resources	18-41
Practice 18: Overview	18-42
Quiz	18-43

## **19 Building Database Applications with JDBC**

Objectives	19-2
What Is the JDBC API?	19-3
What Is JDBC Driver?	19-4
Connecting to a Database	19-5
Obtaining a JDBC Driver	19-6
Register the JDBC driver with the DriverManager	19-7
Constructing a Connection URL	19-8
Establishing a Connection	19-9
Using the JDBC API	19-10

Key JDBC API Components	19-11
Writing Queries and Getting Results	19-12
Using a ResultSet Object	19-13
CRUD Operations Using JDBC API: Retrieve	19-14
CRUD Operations Using JDBC: Retrieve	19-15
CRUD Operations Using JDBC API: Create	19-16
CRUD Operations Using JDBC API: Update	19-17
CRUD Operations Using JDBC API: Delete	19-18
SQLException Class	19-19
Closing JDBC Objects	19-20
try-with-resources Construct	19-21
Using PreparedStatement	19-22
Using PreparedStatement: Setting Parameters	19-23
Executing PreparedStatement	19-24
PreparedStatement:Using a Loop to Set Values	19-25
Using CallableStatement	19-26
Summary	19-27
Practice 19: Overview	19-28
Quiz	19-29

## 20 Localization

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

Format Styles 20-22  
Summary 20-23  
Practice 20: Overview 20-24  
Quiz 20-25

## A Annotations

Objectives A-2  
Topics A-3  
Scenario A-4  
@FunctionallInterface Annotation A-5  
Annotation Characteristics A-6  
@Override Annotation A-7  
@Deprecated Annotation A-8  
@Deprecated Annotation Recommendations and Options A-9  
@SuppressWarnings Annotation A-10  
@SafeVarargs Annotation A-11  
Topics A-12  
Examples from the Deprecated Annotation A-13  
@Documented Meta Annotation Effects A-14  
Topics A-15  
Scenario A-16  
Solution: Write a Custom Annotation A-17  
Applying a Custom Annotation A-18  
Reading Annotation Elements Through Reflection A-19  
Other Details A-20  
Inheriting an Annotation A-21  
Reading an Inheriting an Annotation Through Reflection A-22  
Repeating an Annotation A-23  
Repeating Annotation Example A-24  
Reading a Repeatable Annotation Through Reflection A-25  
Topics A-26  
Frameworks A-27  
@NonNull Type Annotation and Pluggable Type Systems A-28  
Summary A-29

## B Security Survey

Objectives B-2  
About This Lesson B-3  
Topics B-4  
Denial of Service (DoS) Attack B-5  
Sockets B-6

Other DoS Examples	B-7
Topics	B-8
Enemies Target Confidential Information	B-9
Purge Sensitive Information from Exceptions	B-10
Do Not Log Highly Sensitive Information	B-11
Topics	B-12
Validate Inputs	B-13
Numbers That Make Programs Go Awry	B-14
Directory Traversal Attacks with ..	B-15
SQL Injection Through Dynamic SQL	B-16
Safer SQL	B-17
XML Inclusion	B-18
The Problem with XML Entities	B-19
Failure to Verify Bytecode	B-20
Topics	B-21
Isolate Unrelated Code	B-22
Stronger Encapsulation with Modules	B-23
Stronger Encapsulation Against Reflection	B-24
Limit Extensibility	B-25
Beware of Superclass Changes	B-26
Problem: Vulnerable Object Fields	B-27
Solution: Create Copies of Mutable or Subclassable Input Values	B-28
File Security Bug Reports	B-29
Topics	B-30
Serialization and Deserialization	B-31
Problem: Fields Are Accessible After Serialization	B-32
Solution 3a: Implement writeObject and readObject Methods	B-33
Solution 3b: Implement writeObject with PutField, and readObject with GetField	B-34
Solution 4: Implement a Serialization Proxy Pattern	B-35
Deserialize Cautiously	B-36
Summary	B-37

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Services in a Modular Application

12



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



ORACLE®

## Objectives

After completing this lesson, you should be able to describe:

- How services are supported in Java SE 9
- The distinction between a service supplying a concrete object versus a proxy object
- How services help address cyclic dependencies



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Topics

- Service-based design
- ServiceLoader class
- TeamGameManager example



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Modules and Services

Module system supports services to achieve a much looser kind of coupling than can be achieved with the `requires` and `exports` directives.

- Modules can produce and consume services, rather than just expose all public classes in selected packages.
- Additional services can be created as modules and added to the module path at any time.
- Optional dependencies can be easily encoded because the consumer module is responsible for determining what should be done if an implementation is not found.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In Java, one should program to an interface, not an implementation. In JDK 9, the module system introduces “services” to make this concept explicit; modules declare which interfaces they program to, and the module system automatically discovers implementations.

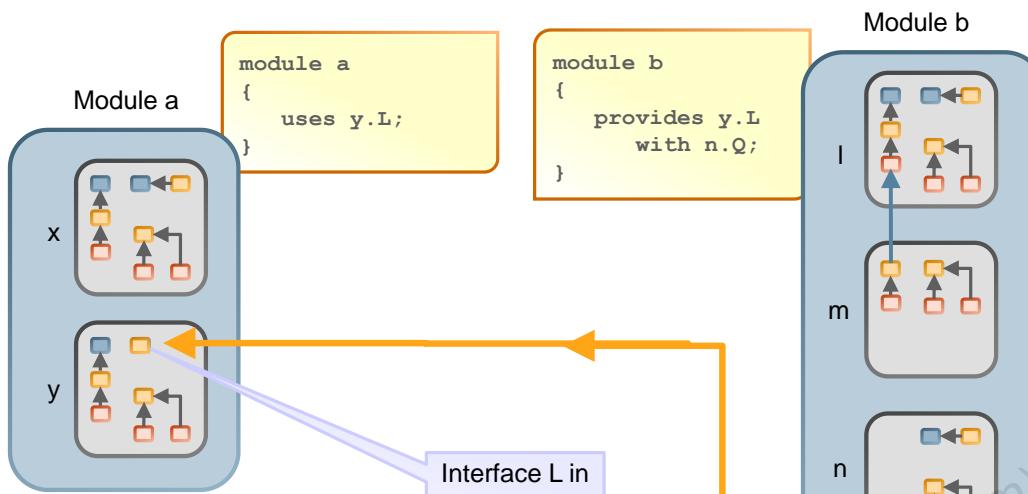
## Components of a Service

- A service consists of:
  - An interface or abstract class
  - One or more implementation classes
  - Code using the `ServiceLoader` class to find all the implementation classes and decide which one (if any) should be used.
- There are two directives to define a service in the `module-info.java` file.
  - The `uses` directive specifies an interface or an abstract class that defines a service that this module would like to consume.
  - The `provides...`with directive specifies a class that a module provides as a service implementation.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Produce and Consume Services

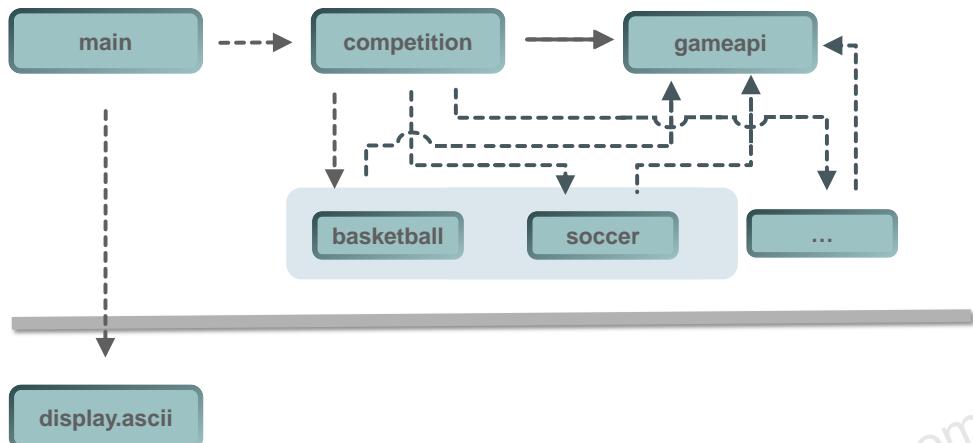


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A `uses` module directive specifies a service used by this module—making module a a service consumer. A service is an object of a class that implements the interface or extends the abstract class specified in the `uses` directive.

A `provides...with` module directive specifies that a module provides a service implementation—making the module a service provider. The `provides` part of the directive specifies an interface or abstract class listed in a module's `uses` directive and the `with` part of the directive specifies the name of the service provider class that implements the interface or extends the abstract class.

## Module Dependencies Without Services



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

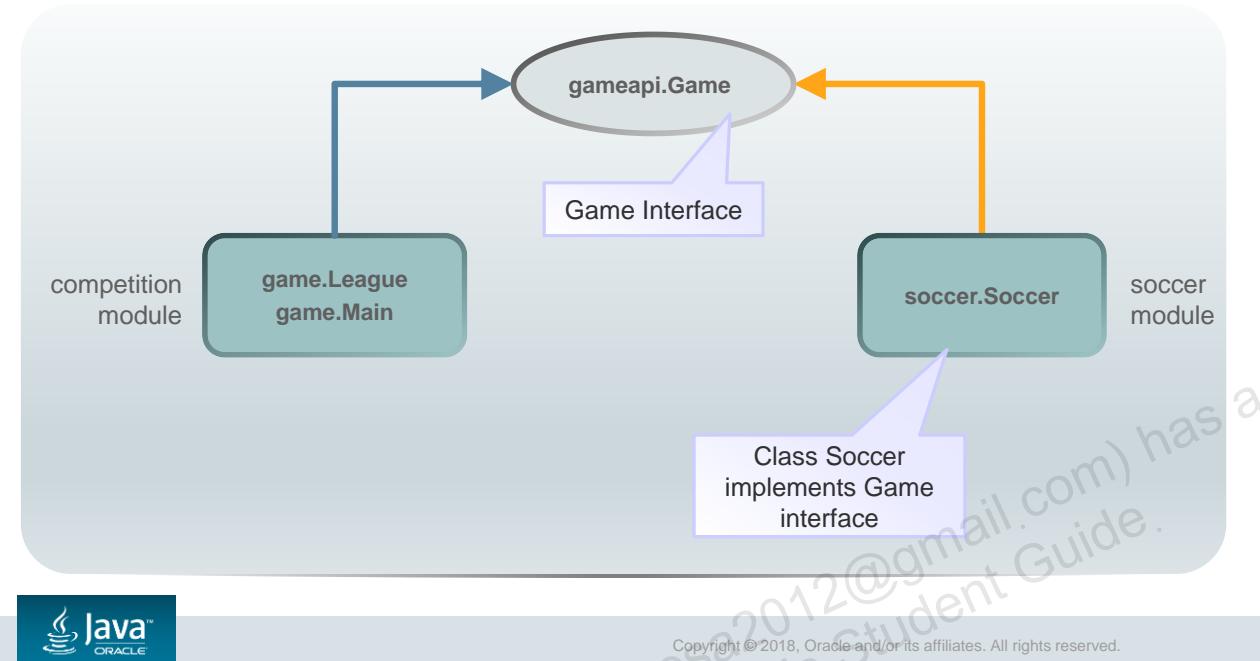
The graphic in the slide represents an application that runs a league and provides for future extension in the form of new games to be supported. Currently basketball and soccer are supported. To provide support for another game, perform the following steps:

- Create a new module, for example, the `lacrosse` module, with a dependency on `gameapi`.
- In the `lacrosse` module, implement the interfaces that are required by the `competition` module to run a league of lacrosse games.
- Add functionality to the `lacrosse` module to represent the game of lacrosse.
- Use the `exports` keyword to ensure that the necessary implementations are available to the `competition` module.
- Rewrite the `Factory` class in `competition` so that it can create the needed objects for the game of lacrosse.

There are a lot of dependencies! It's not enough to create the new module; existing modules also need to be modified to accommodate the new module. Also, notice that if you decide to no longer support a particular game (for example, basketball), it's not enough to simply not include it in the module path. You will also need to modify the `Factory` class in the `competition` module.

This is a very tightly coupled architecture.

## Service Relationships

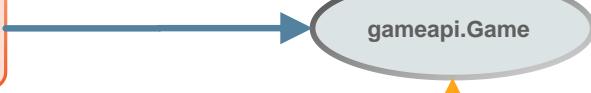


Here's how it works. A module requests an implementation of an interface, which is provided by another module (or, in some cases, even by the same module). In the example shown in the slide, the Game interface implemented by the Soccer class can be implemented by any number of other classes, like, for example, the Basketball class.

## Expressing Service Relationships

Consumer module

```
module competition {  
    uses gameapi.Game;  
}
```



gameapi.Game

Provider module

```
module soccer {  
    provides gameapi.Game with soccer.Soccer;  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A module requesting the implementation of an interface uses the `uses` directive in its `module-info.java` file to register this. This module is called the consumer module. A module implementing this interface uses the `provides` directive to declare this.

## Topics

- Service-based design
- `ServiceLoader` class
- TeamGameManager example



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Using the Service Type in competition

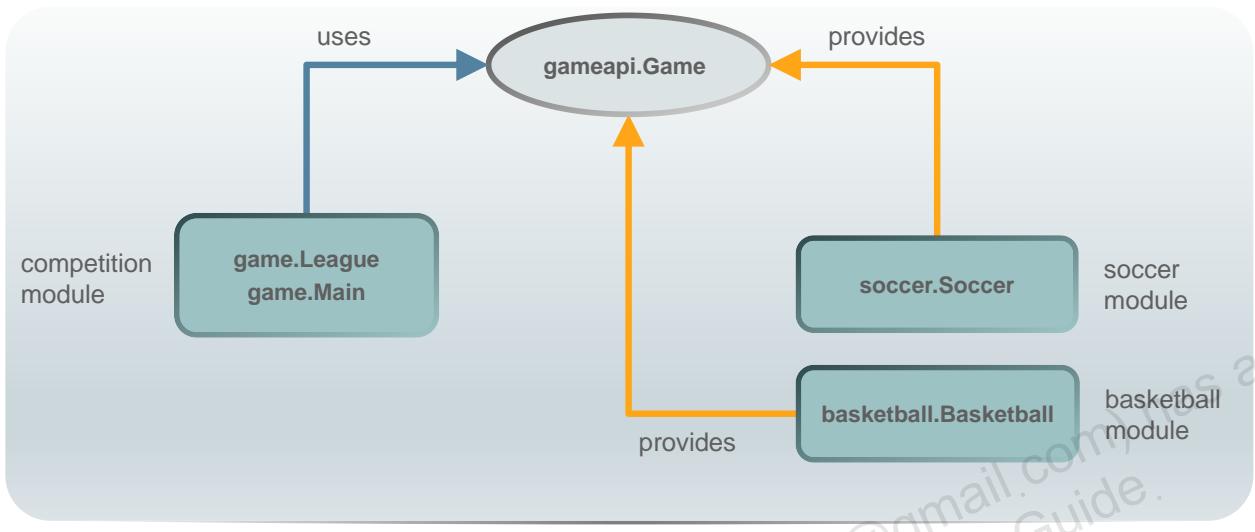
```
ServiceLoader<Game> game = ServiceLoader.load(Game.class);
ArrayList<Game> providers = new ArrayList<>();
for (Game currGame : game) {
    providers.add(currGame);
}
return providers;
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The slide shows the basic code for getting the providers (the implementations). The `ServiceLoader` class is used to get the providers. Its key method is static and called `load`. This method creates a new `ServiceLoader` for a given service type. `ServiceLoader` implements `Iterable`, and so you can use the `for` clause as shown to load and instantiate each of the available providers.

## Choosing a Provider Class



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The module system finds all the implementations of the interface in the module path and, using the `ServiceLoader` class, makes them available to the consumer module. The provider modules do not know of each other, and the selection of which implementation is the best one given the need must be determined by the consumer module. But the design of the implementation may help with this. For example, the Game interface may declare a `getGameType` method that must be implemented by the Soccer, Basketball, or other class that implements it. This `getGameType` method can then be used to determine if the providers support the type of game desired.

## Choosing a Provider Class

```
ServiceLoader<Game> game = ServiceLoader.load(Game.class);
for (Game currGame : game) {
    if (currGame.getType().equals("soccer")) return currGame;
}
throw new RuntimeException("No suitable service provider found.");
```



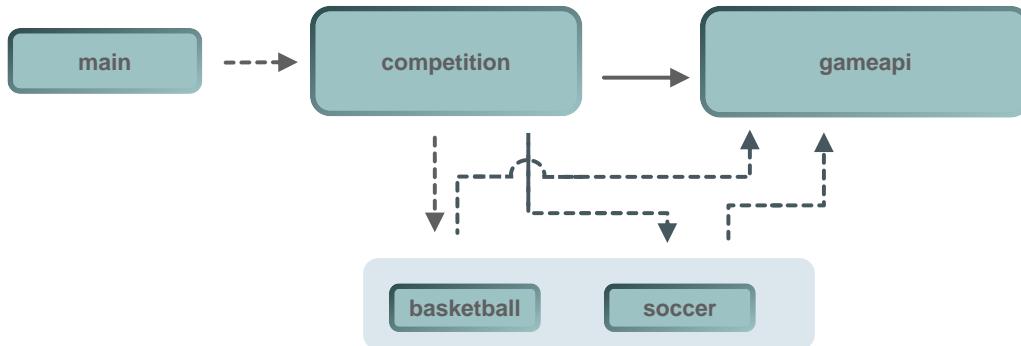
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The code in the slide uses the `getType` method to return a provider that identifies itself as “soccer.” More than one provider could do this, so there could be multiple soccer providers, in which case if the consumer does not simply wish to consume the first one returned, it must specify and test other criteria to make the decision.

Note that the Game interface will have specified the `getType` method for this purpose, and any implementing class must implement it.

In the example in the slide, if the `for` loop completes without a provider being found, a `RuntimeException` will be thrown. The decision to do this is again entirely up to the consumer. In some cases, a default provider may be loaded at this point, or in other cases, services might be used to model optional dependencies, so no provider would need to be loaded.

## Module Dependencies and Services 1

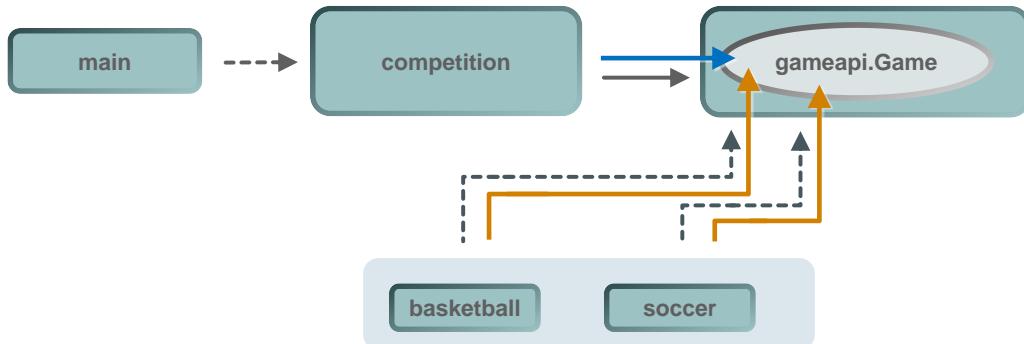


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



This slide and the following two slides show how services can replace hard-coded dependencies, thus changing the architecture from tightly coupled to loosely coupled. The graphic on this slide shows the application using hard-coded dependencies where the interfaces implemented in the soccer and basketball modules must be in a module other than the competition module so that a cyclic dependency does not occur.

## Module Dependencies and Services 2

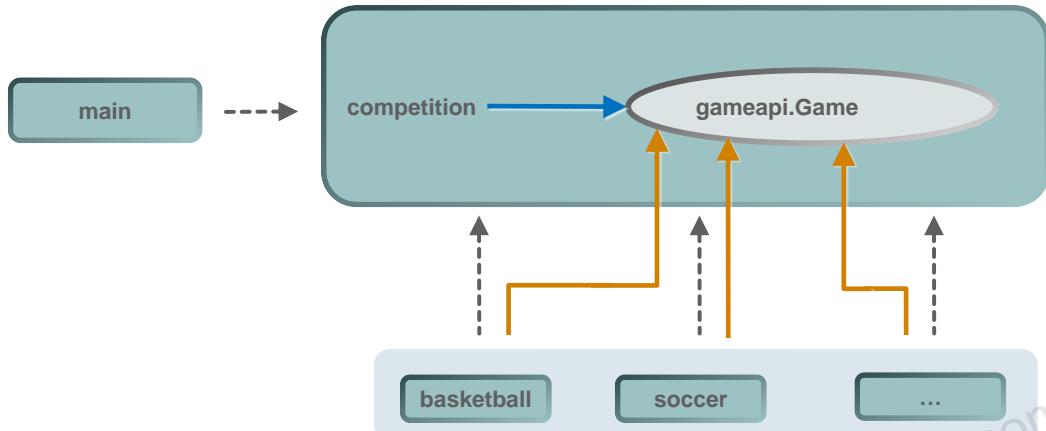


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The graphic in the slide shows basketball and soccer functionality being provided as services. The following are some important points to note:

- The competition module does not declare a dependency on any provider, only on the module that contains the Game interface for which it requests an implementation.
- Each provider does have a dependency on the module that contains the Game interface because that is the interface it implements.

## Module Dependencies and Services 3



The graphic in this slide shows how using services can help to address cyclic dependencies.

Notice that the Game interface has been moved back into the competition. A cyclic dependency is when module A requires module B and vice versa. It is not a cyclic dependency if module A uses an implementation that lives in module B while module B requires module A. Therefore, the interfaces can be in any module, and in this simple example application, it makes sense to put the Game interface in the competition module.

Services are easily extensible as more provider modules can be added at any time, and any or all service modules can be removed without competition being modified.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Designing a Service Type

```
Interface Game {  
    String getType();  
    Team getHomeTeam();  
    Team getAwayTeam();  
    void playGame();  
    ...  
}
```

```
Interface GameProvider {  
    Game getGame(Team homeTeam, Team awayTeam, LocalDateTime date);  
    Team getTeam(String TeamName, Player[] players);  
    Player getPlayer(String playerName);  
    String getType();  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The service type itself can either return the concrete class to be used in the code or a proxy that will itself return a concrete class. In the examples shown in the slide, the implementation of the Game interface might be the actual class to be used in the code of the competition module. However, the Game implementation needs a Team type, and the Team implementation in turn needs a Player type, so implementations of all three types, Game, Team, and Player, would need to be available as providers. Also, each implementation class would need a no-argument constructor and probably also another method to populate the data in the class, given that the constructor is no-argument.

Another approach is to use a proxy provider. In this case, the proxy is not the actual class to be used in the competition module. Instead, it provides the classes that are to be used. This means that only one provider is required for the three classes that make up a type of game (Game, Team, Player), and the implementations of these three classes do not need a no-argument constructor. Instead, the necessary parameters can be passed directly to the constructor.

## Topics

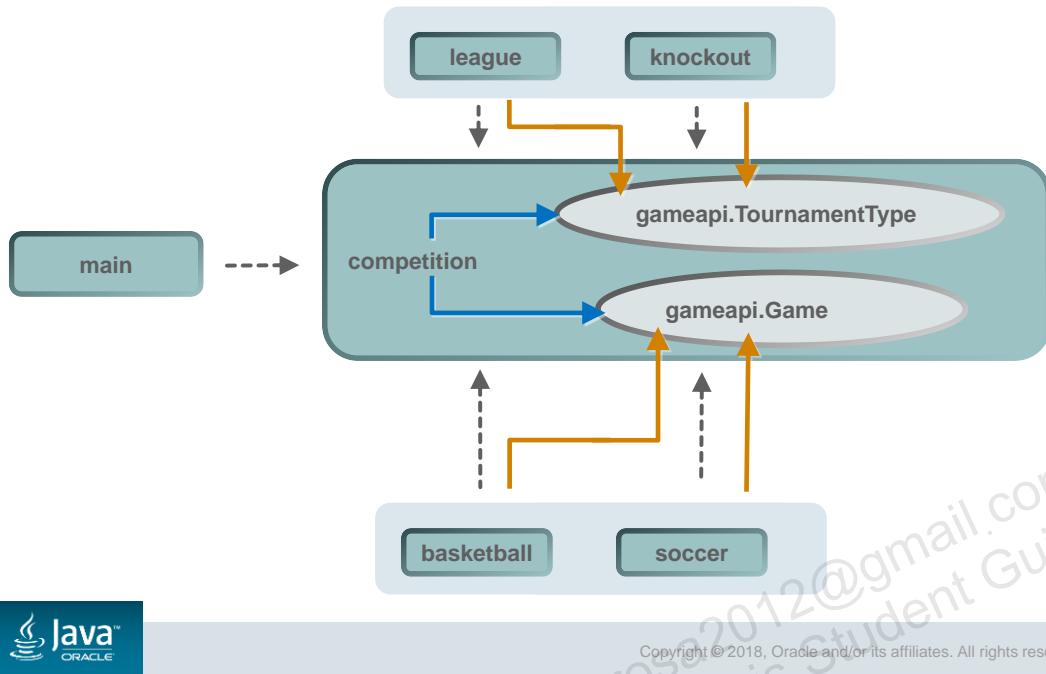
- Service-based design
- ServiceLoader class
- TeamGameManager example



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

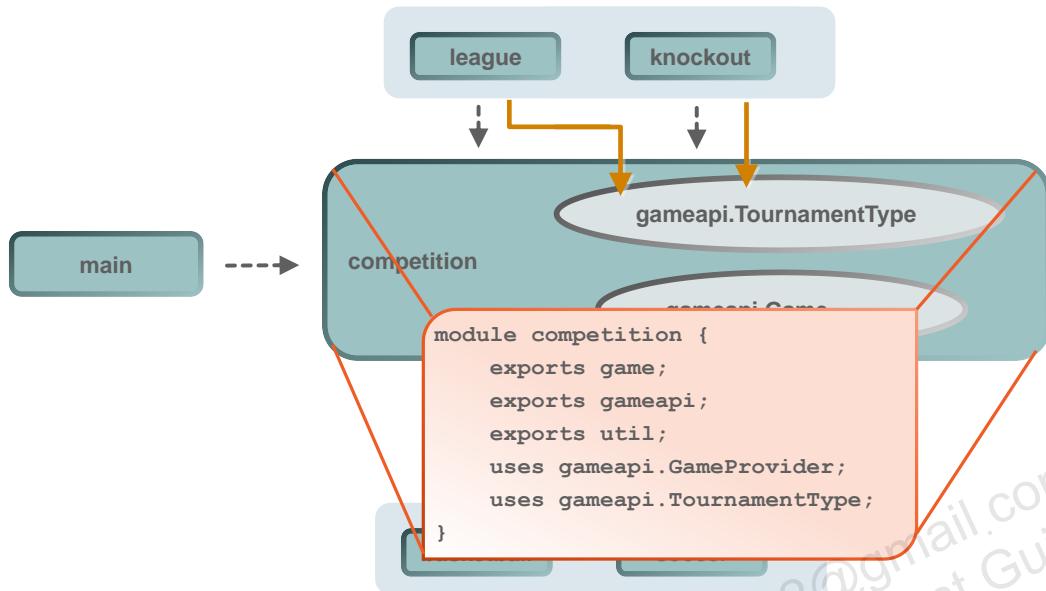


## TeamGameManager Application with Additional Services



The slide shows a dependency diagram for the TeamGameManager application. The application has also been extended so that the type of competition is now also provided as a service. The two examples in the slide are league and knockout, but there are many other types possible.

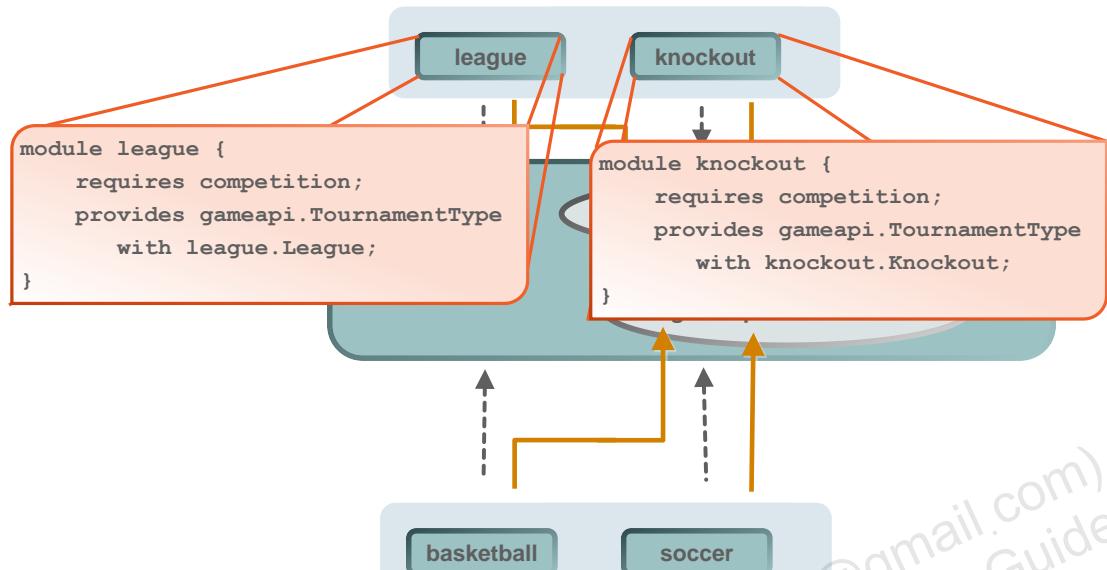
## module-info.java for competition module



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



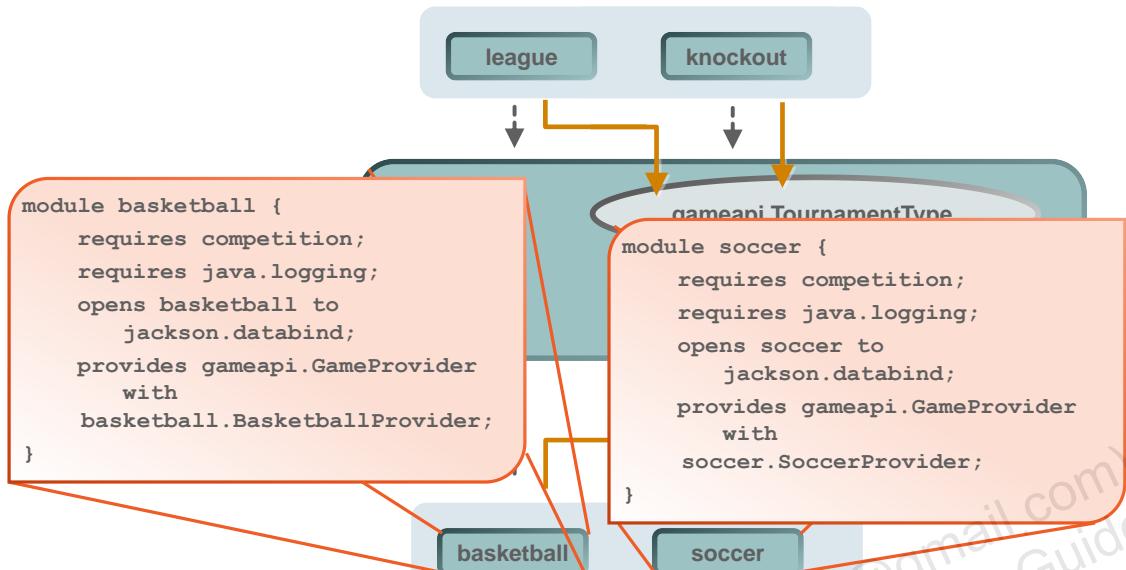
## module-info.java for league and knockout modules



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## module-info.java for soccer and basketball modules



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Summary

In this lesson, you should have learned to describe:

- How services are supported in Java SE 9
- The distinction between a service supplying a concrete object versus a proxy object
- How services can help address cyclic dependencies



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



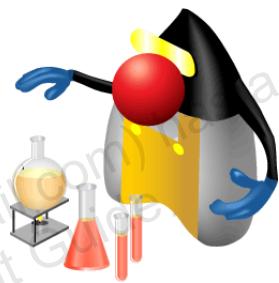
## Practice 12: Overview

This practice covers the following topics:

- Practice 12-1: Creating services
- Practice 12-2: More services



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.





Adolfo De+la+Rosa (adolfoldelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Concurrency



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfo.delarosa2020@gmail.com) has a  
non-transferable license to use this Student Guide.

## 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 the classes in the `java.concurrent.atomic` package to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections



Copyright © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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

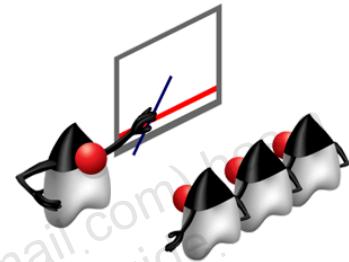


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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 © 2018, 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 overworking 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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();           Gets the result of the Callable's
} catch (ExecutionException|InterruptedException ex) {           call method (blocks if needed).
}
}                                                               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).

If the Callable  
threw an Exception



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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 © 2018, 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 © 2018, 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



Copyright © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 the classes in the `java.concurrent.atomic` package to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Practice 13: Overview

This practice covers the following topics:

- 13-1: Using the `java.util.concurrent` package
- 13-2: Creating a Network Client using the `java.util.concurrent` package



Copyright © 2018, 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfolalarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Parallel Streams



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfo.delarosa2020@gmail.com) has a  
non-transferable license to use this Student Guide.

## 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## 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: $%,.2f %n",  
result);  
29     }
```



Copyright © 2018, 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 © 2018, 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 © 2018, 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 -> 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);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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.

Collectors are covered in much more detail in the lesson titled “Terminal Operations: Collectors.”

## 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 © 2018, 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 that 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 © 2018, Oracle and/or its affiliates. All rights reserved.

A deterministic algorithm is an algorithm that, 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 in which 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 © 2018, 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) \rightarrow a + b)`
  - `(0, (sum, element) \rightarrow sum + element)`



Copyright © 2018, 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.

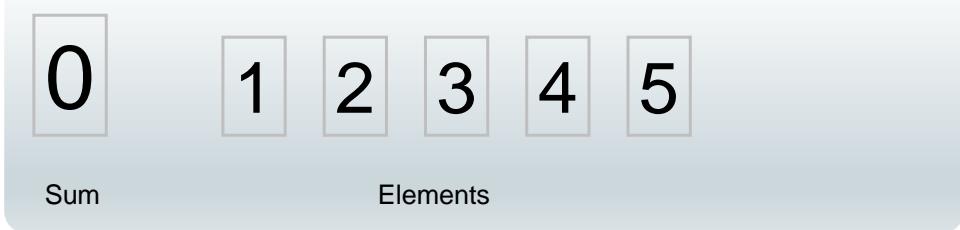


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

As the above text 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);
```

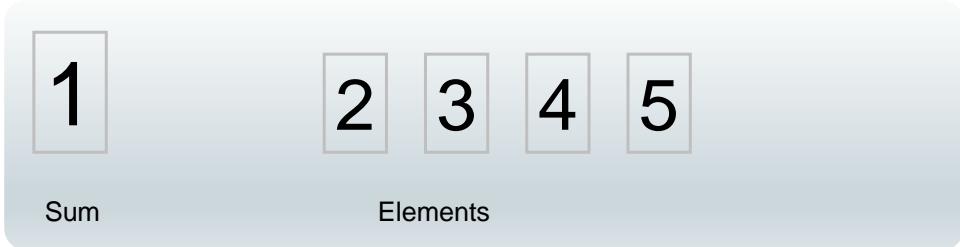


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

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);
```



Copyright © 2018, 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 © 2018, 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



Copyright © 2018, 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



Copyright © 2018, 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

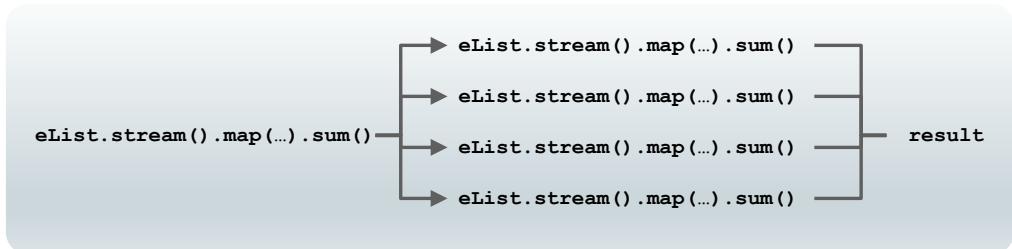
Elements



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## A Look Under the Hood

- Pipeline decomposed into subpipelines
  - Each subpipeline produces a subresult.
  - Subresults are combined into final result.



Copyright © 2018, 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);
```

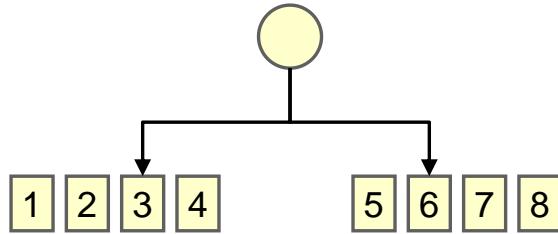


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

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

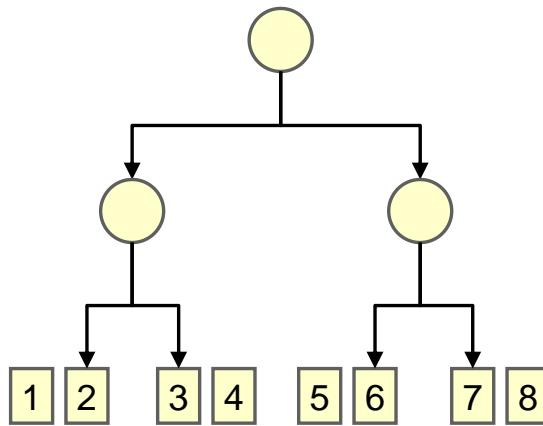
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

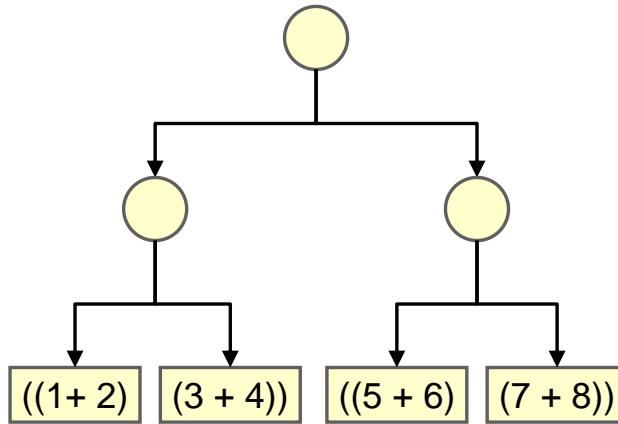
## Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()
19         .reduce(0, (sum, element) -> sum + element);
```



## Illustrating Parallel Execution

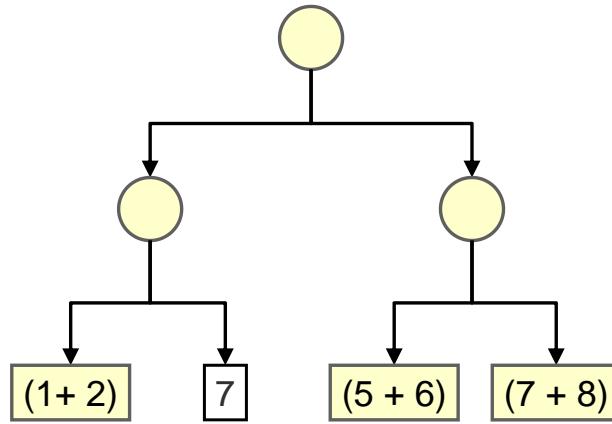
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Illustrating Parallel Execution

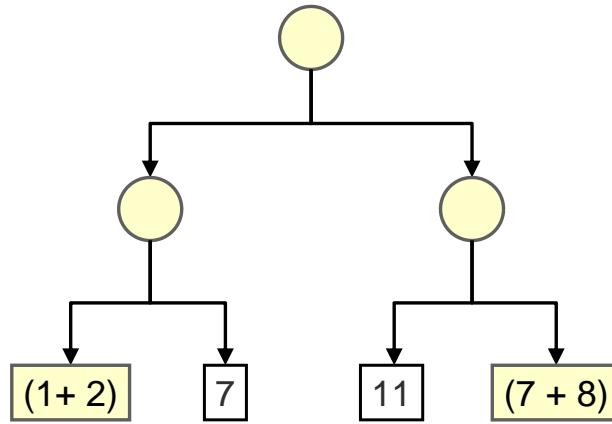
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Illustrating Parallel Execution

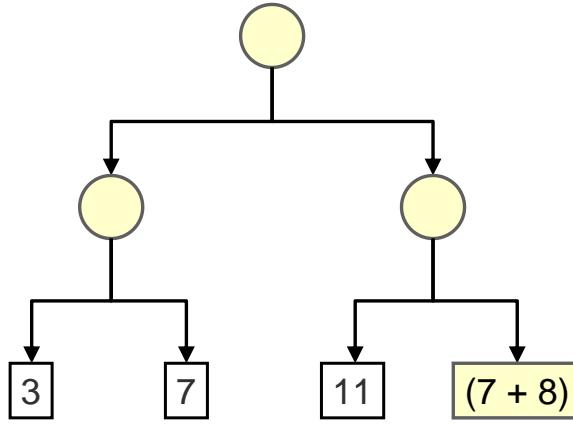
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Illustrating Parallel Execution

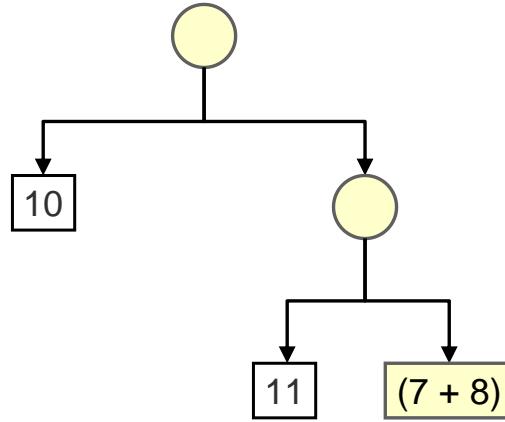
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Illustrating Parallel Execution

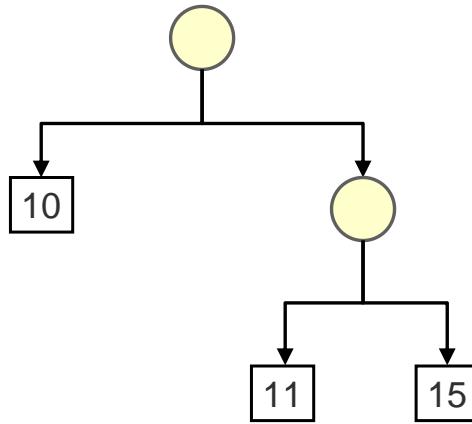
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Illustrating Parallel Execution

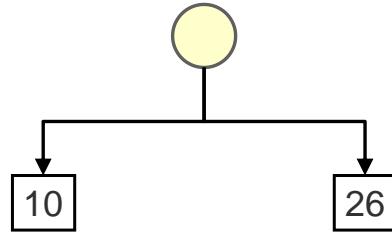
```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Copyright © 2018, 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## Performance

- Do not assume parallel is always faster.
  - Parallel is 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 are provided for performance.
  - Boxing/Unboxing negatively impacts performance.



Copyright © 2018, 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 ~= Cost of the pipeline

- Larger N\*Q -> 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.



## Practice 14: Overview

This practice covers the following topics:

- Practice 14-1: Calculating total sales without a pipeline
- Practice 14-2: Calculating sales totals using Parallel Streams
- Practice 14-3: Calculating sales totals using Parallel Streams and Reduce



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfoldelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Terminal Operations: Collectors



ORACLE®



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adolfo De+la+Rosa (adolfodelarosa2022@gmail.com) has a  
non-transferable license to use this Student Guide.

## Objectives

After completing this lesson, you should be able to use Collectors to:

- Create a collection
- Group elements into a collection
- Perform summarizing on a collection
- Create a custom collector in code
- Create a custom collector by combining collectors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Agenda

- Introduction to collectors
- Three argument collect method of Stream
- Single argument collect method of Stream
- Grouping by collectors
- Nested values
- Complex custom collectors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Streams and Collectors Versus Imperative Code

Functional style code has advantages over imperative coding:

- Concise
  - Defines aggregate operations
- Readable
- Flexible and extensible
  - Composition possible
- Parallel ready
- Dealing with data in the aggregate
  - Collectors are important here



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Collection

- Reduction is the most important kind of terminal operation.
  - Combines a sequence or collection of an arbitrary value into one value.
  - `forEach()` is useful but should generally not be used to create results.
- Reduction uses either:
  - The `reduce` method of `Stream`: used with immutable types
  - The `collect` method of `Stream`: used with mutable types
- There are many predefined collectors that can be used with the `collect` method to create complex queries that generate a collection.
- Custom collectors can be created by:
  - Providing functions (often using lambda):
    - To the three argument `Stream.collect()` method
    - To the `Collectors.of()` method
  - By implementing the `Collector` interface directly



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Reduction combines a sequence or collection of an arbitrary value into one value. A collector facilitates a particular kind of reduction, which is one that reduces stream elements into a container (such as an array or Collection) using mutation. Collectors provide mutable reduction.

Immutable reduction using the `reduce` method of `Stream` is covered in the lesson “Parallel Streams.”

## Predefined Collectors

- Most examples in this lesson use a very simple Person class that contains

```
public class Person {  
    public enum City {  
        Belfast,  
        Tulsa,  
        Athens,  
        London; }  
    private City city;  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    // Not shown: A constructor that populates all fields,  
    // and getter and setter methods.
```

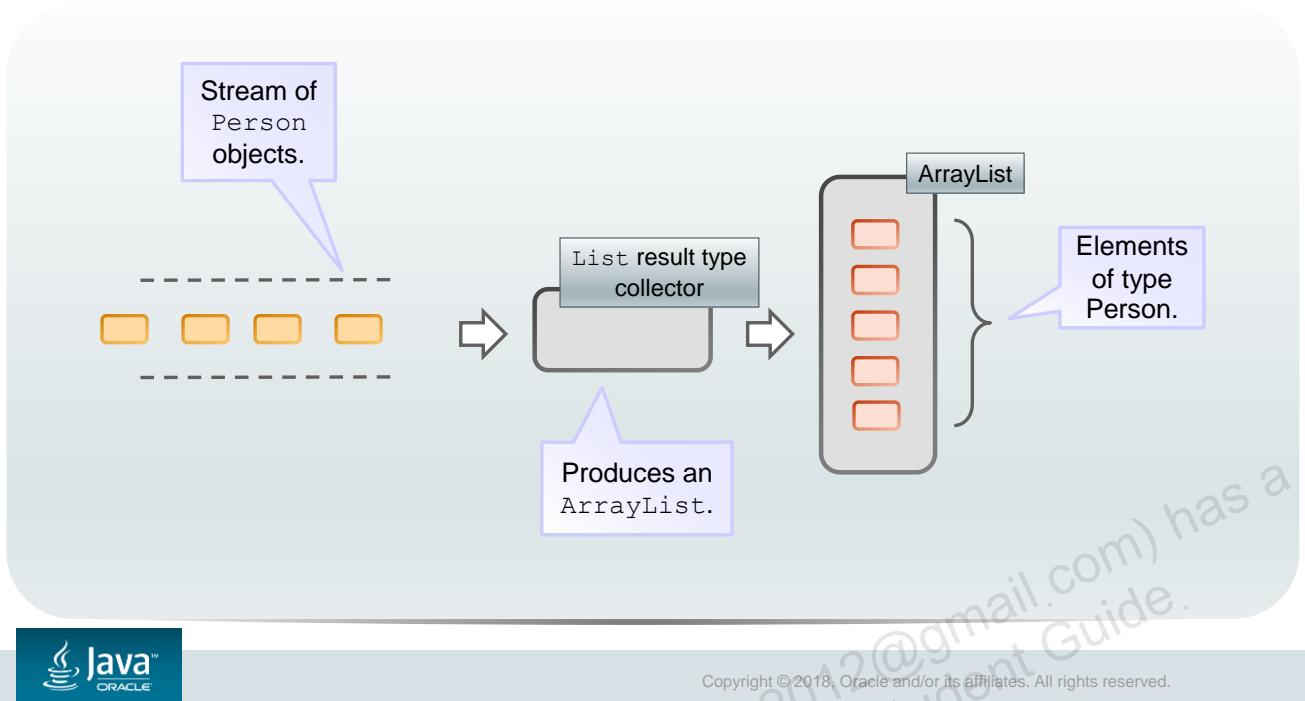


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The data used in most of the examples can be created like this:

```
static List<Person> people =  
    List.of(new Person(City.Tulsa, "Joe", "Bloggs", 42),  
            new Person(City.Athens, "Amy", "Laverda", 21),  
            new Person(City.London, "Bill", "Gordon", 33),  
            new Person(City.Athens, "Eric", "Vincent", 33),  
            new Person(City.Tulsa, "Eric", "Dunmore", 29));
```

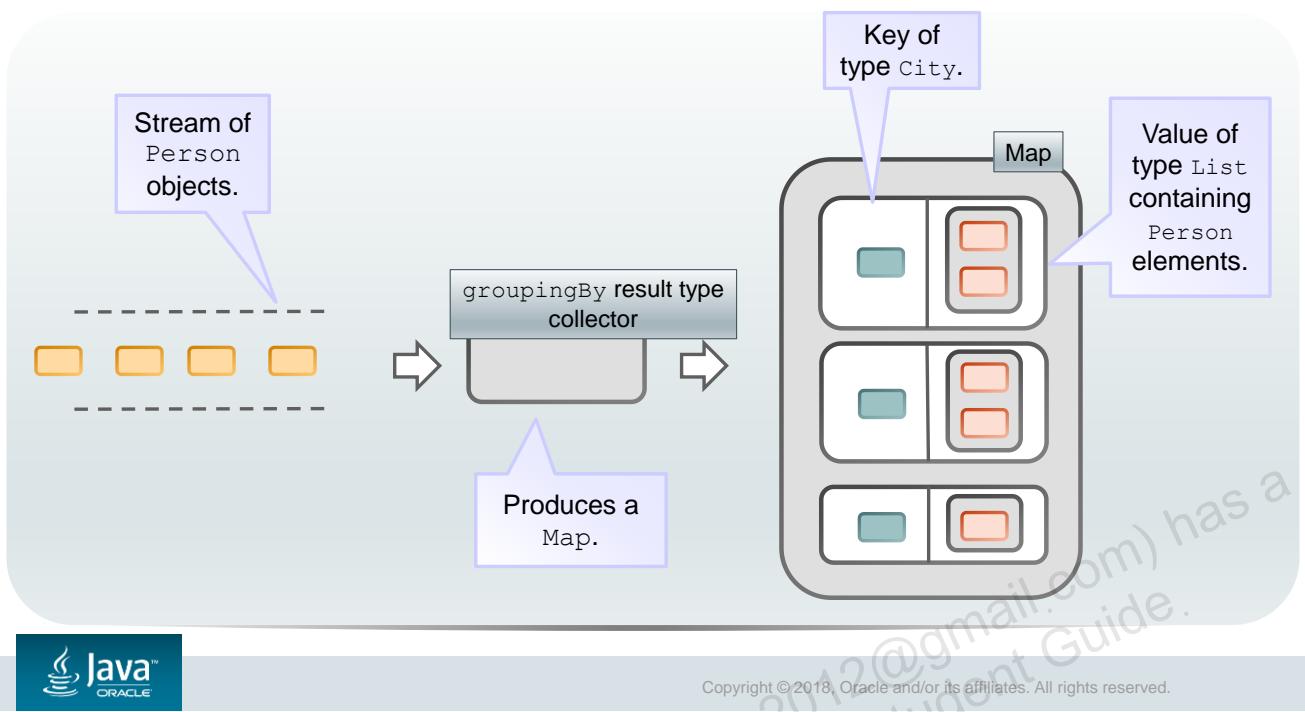
## A Simple Collector



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The diagram shows the basic operation of a simple collector.

## A More Complex Collector



The diagram shows a more complex manipulation of the stream of data. The data elements are classified by a particular data item, in this case by the city the person lives in. The `groupingBy` collector is the easiest way to do this, but in later slides, you see that you can use the `toMap` collector or even write your own custom collector to achieve similar results.

There are a number of ways to achieve this type of processing and a number of approaches are covered later in the lesson.

# Agenda

- Introduction to collectors
- Three argument collect method of Stream
- Single argument collect method of Stream
- Grouping by collectors
- Nested values
- Complex custom collectors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## The Three Argument collect Method of Stream

```
<R> R collect(Supplier<R> supplier,  
BiConsumer<R, ? super T> accumulator,  
BiConsumer<R, R> combiner)
```

- `supplier` - a Supplier that creates a new mutable result container (or containers)
- `accumulator` - a BiConsumer that must fold an element into a result container
- `combiner` - a BiConsumer that merges two partial result containers

Types:

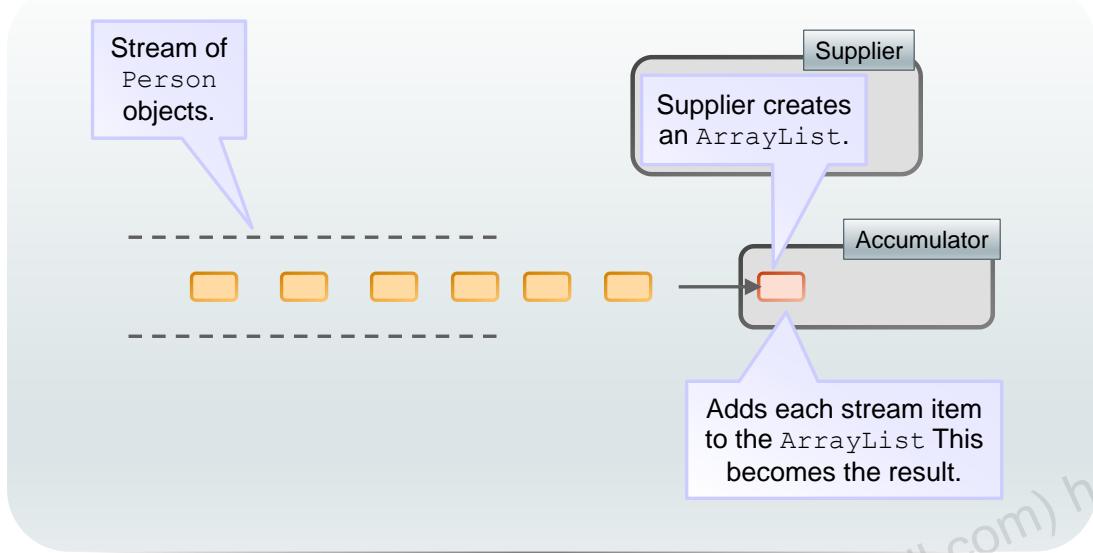
- `R` – The result type
- `T` – The type of the stream elements



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

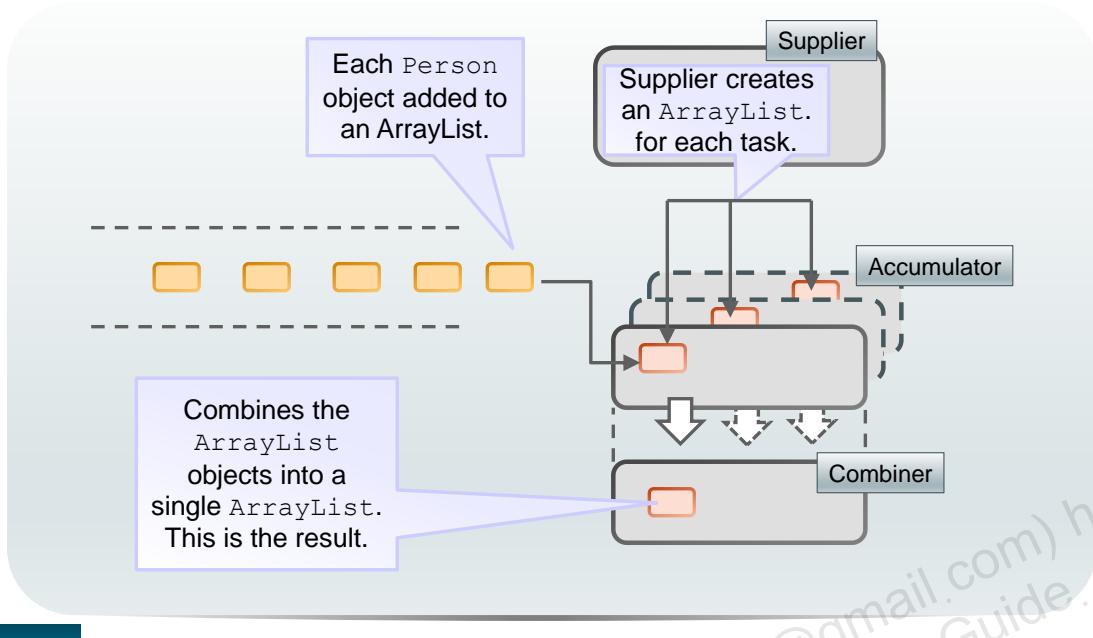
Result container in this context is whatever type has been chosen to be created to store the accumulating data. Typically it will be a collection, but it could be a simple primitive if, for example, all that is required is a simple count.

## The collect Method Used with a Sequential Stream



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## The collect Method Used with a Parallel Stream



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## The collect Method: Collect to an ArrayList Example

Assuming a Stream of Person elements:

```
List<Person> myPpl = people.stream()
    .collect( ArrayList::new ,
              ArrayList::add ,
              ArrayList::addAll );
System.out.println(myPpl);
[Joe, Amy, Bill, Eric, Eric]
```

Supplier: creates an ArrayList as the new result type.

Accumulator: adds Person elements to the result ArrayList.

Combiner: if parallel stream, takes two ArrayList objects and combines them.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Agenda

- Introduction to collectors
- Three argument collect method of Stream
- **Single argument collect method of Stream**
- Grouping by collectors
- Nested values
- Complex custom collectors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## The Single Argument collect Method of Stream

```
collect(Collector<? super T,A,R> collector)
```

- **Collector** - a Collector object that uses the types shown and transforms the stream into a single object (often but not necessarily a collection)
  - Many predefined Collectors available in the Collectors class
    - For example, `Collectors.toList()` is a static factory method that creates a collector that reduces the stream to an `ArrayList`.
  - The types are:
    - `? super T` is the type of the input elements to the Collector.
    - `A` is the mutable accumulation type of the reduction operation (often hidden as an implementation detail).
    - `R` is the result type of the reduction operation.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

`? super T` means that the code within the Collector will operate on elements of the stream or any superclass of elements of the stream. For example, with a stream of `Person` elements, the Collector can work with `Person` or a superclass of `Person`. For example, a Collector that returns an `ArrayList<Object>` elements would work with `Stream<Person>`.

Note that when using a particular Collector class in this way, the accumulation type is not necessarily the same as the result type. The reason for this will be covered in more details later in this lesson.

## Using Predefined Collectors From the Collectors Class

Implementations of `Collector` that implement various useful reduction operations.

- Available from static factory methods of the `Collectors` class.

Can be used in two ways:

- Standalone
  - Accumulate to collections
    - Some factory methods allow the collection type to be chosen, e.g. `HashSet` or `TreeSet`
  - Accumulate to a Map using a classifier function to determine the keys
- Composing
  - Collectors can be used to adapt the functionality of another Collector:
    - For example, the filtering Collector can be used to adapt another “downstream” Collector
  - Composing useful for collectors that partition or group elements to do further processing “downstream”



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## List of Predefined Collectors

Standalone  
collectors

- averaging
- counting
- groupingBy\***
- maxBy
- minBy
- partitioningBy\***
- reducing
- summarizing
- summing
- toCollection
- toList
- toSet
- toMap

Collectors that adapt other  
“downstream” collectors.

- collectingAndThen
- filtering
- flatMapping
- mapping
- groupingBy**
- partitioningBy**

Factory methods  
available for creating  
both standalone and  
composing collectors.

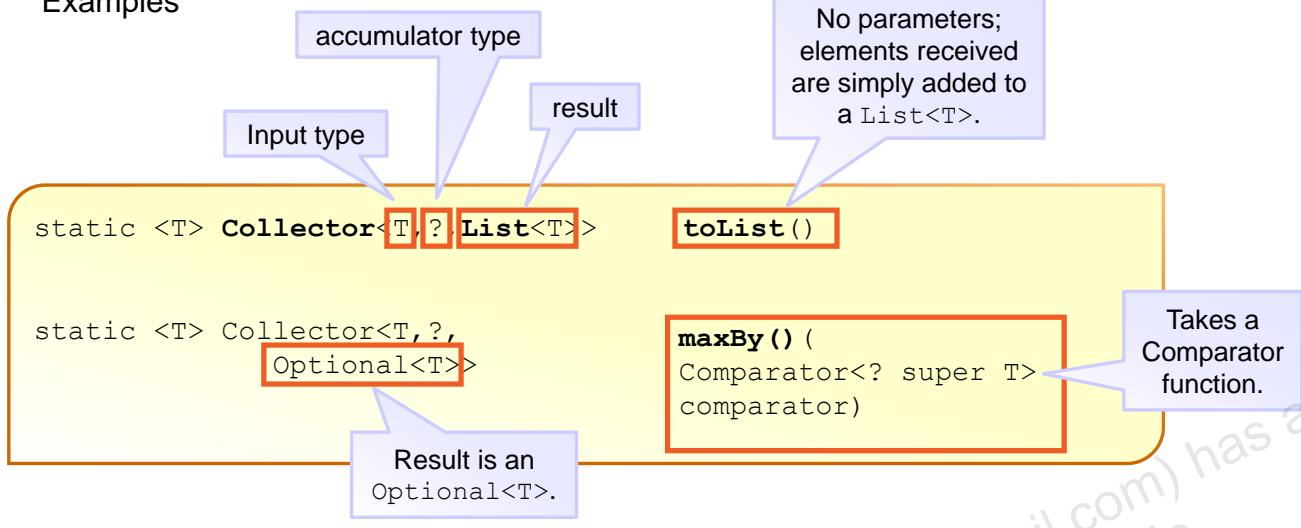
Summing, summarizing,  
and averaging collectors  
have multiple factory  
methods for the four  
types of streams (object,  
double, long, int).



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Stand-Alone Collectors

### Examples



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Stand-Alone Collector: List all Elements

```
static List<Person> people =
    List.of(new Person(City.Tulsa, "Joe", "Bloggs", 42),
            new Person(City.Athens, "Amy", "Laverda", 21),
            new Person(City.London, "Bill", "Gordon", 33),
            new Person(City.Athens, "Eric", "Vincent", 33),
            new Person(City.Tulsa, "Eric", "Dunmore", 29));
```

Creates List  
of Person  
objects

```
List<Person> myPpl = people.stream()
    .collect(toList());
```

Using static import of  
Collectors.toList  
( ) for readability.

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

```
[Joe, Amy, Bill, Eric, Eric]
```

Uses toString  
method of Person  
for output.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

```
public class Person {
    public String toString() {
        return firstName;
    }
    public Person() {}
    public Person(City city, String name, int age) {
        this.city = city;
        private String firstName;
        private String lastName;
        this.age = age;
    }
    public enum City {
        Belfast,
        Tulsa,
        Athens,
        London;
    }
    private City city;
    private String name;
    private int age;

//... (lines omitted - setters and getters available for all fields)...
}
```

## maxBy() Example

```
people.stream()
    .collect(maxBy(Comparator.comparing(Person::getAge)))
    .ifPresentOrElse(m -> System.out.println(m +
        ", " + m.getAge() + ", is oldest"),
        () -> System.out.println("No Person of max age found"));
```

```
Joe, 42, is oldest
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Adapting Collector: filtering()

### Examples

```
static <T,A,R>
Collector<T,?,R>
    .filtering(
        Predicate<? Super T>
            predicate,
        Collector<? Super T,A,R>
            downstream)
```

Parameters necessary:  
filters each input element to the downstream collector.

Predicate filter.

Downstream collector.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The collector being adapted is referred to as the “downstream” collector.

## Composing Collectors : groupingBy()

### Examples

Downstream collector modified into one that produces a Map. Note the type of the value parameter for the Map.

```
static <T, K, A, D>  
Collector<T, ?, Map<K, D>>
```

```
groupingBy(  
    Function<? super T,  
    ? Extends K>  
    classifier,  
    Collector<? Super  
    T, A, D> downstream)
```

Note the downstream collector and note the type of its result, D.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Composing Collectors: Using Mapping

mapping() adapts the downstream collector.

```
List<String> myPpl = people.stream()
    .collect(mapping(Person::getName) , toList()));
System.out.println(myPpl);
[Joe Bloggs, Amy Laverda, Bill Gordon, Eric Vincent, Eric Dunmore]
```

Mapping Person to a String.

Adding each element to a List.

```
List<String> PplCities = people.stream()
    .collect(mapping(p -> p.getCity() + ":" + p.getName()) ,
        toList()));

System.out.println(PplCities);
[Tulsa:Joe Bloggs, Athens:Amy Laverda, London:Bill Gordon,
 Athens:Eric Vincent, Tulsa:Eric Dunmore]
```

Result is a List of type String.

Note how some cities are listed multiple times.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

An example of using mapping with another downstream collector. The joining() collector adds each element to a String and separates each with a delimiter.

```
String stringOfPpl = people.stream()

    .collect(mapping(Person::getName ,joining("/")));

System.out.println(stringOfPpl);
Joe Bloggs/Amy Laverda/Bill Gordon/Eric Vincent/Eric Dunmore
```

## toMap() and Duplicate Keys

```
Map<City, String> pplCitiesMap = people.stream()
    .collect(toMap c -> c.getCity(), p -> p.getName(),
              (a, b) -> a + ":" + b));
System.out.println(pplCitiesMap);
{Tulsa=Joe Bloggs:Eric Dunmore, Athens=Amy Laverda:Eric Vincent,
 London=Bill Gordon}
```

toMap takes two functions for accumulating keys and values.

toMap can also take a third parameter (BinaryOperator) to handle duplicate keys.

The result is useful, but a `List` value would be better.

How can this be achieved?

- By writing a custom collector as shown earlier
- By using the `groupingBy()` collector

Result is a Map of String objects, where each String lists all the people in a particular city.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

If you do not create a merging BinaryOperator, and there is a duplicate key, an `IllegalStateException` will be thrown.

This is a very common need, and while it is good to know about how to use the `toMap` collector and resolve the duplicates, there's another approach that is superior in most cases, using `groupingBy()`.

## Agenda

- Introduction to collectors
- Three argument collect method of Stream
- Single argument collect method of Stream
- Grouping by collectors
- Nested values
- Complex custom collectors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## groupingBy and partitioningBy Collectors

Both groupingBy and partitioningBy create a Map

- partitioningBy
  - Creates a map with two keys, true and false
  - The standalone version creates a value (type List) for each key
  - Uses a predicate to determine whether the element should go in the true or false List
- groupingBy
  - Uses a Function to create a set of keys for the Map
  - The standalone version creates a value (type List) for each key
  - Based on the function, elements with a particular key are added to the value for that key
- partitioningBy and groupingBy can be standalone or can use a downstream collector.
  - The standalone collector collects each group of values into a List.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

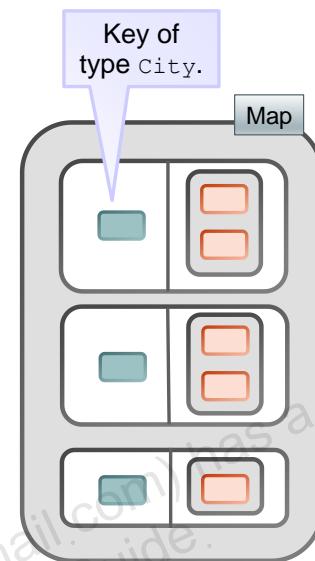
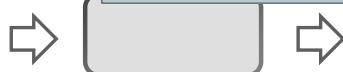
## Stand-Alone groupingBy: Person Elements By City

```
Map<City, List<Person>> PplCitiesMap =  
    people.stream()  
    .collect(groupingBy(Person::getCity));  
  
System.out.println(PplCitiesMap);  
{Tulsa=[Joe, Eric], Athens=[Amy, Eric],  
 London=[Bill]}
```

Stream of  
Person  
objects.



Map<List<Person>>  
result type collector



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



This is a stand-alone groupingBy collector, so by default its result type is Map<City, List<Person>>.

## Stream Operations Or Equivalent Collectors?

Many stream methods are available as collectors. For example:

- Stream.map() and Collectors.mapping()
- Stream.filter() and Collectors.filtering()

```
people.stream()
    .map(p -> p.getName() + ":" + p.getAge())
    .collect(toList())
    .forEach((v) -> System.out.println(v));
```

```
people.stream()
    .collect(mapping(p -> p.getName() + ":" +
                    p.getAge(), toList()))
    .forEach((v) -> System.out.println(v));
```

The code  
fragments  
produce the  
same output.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Stream Operations Or Equivalent Collectors with groupingBy

Using Stream.map() or Collectors.mapping() with groupingBy.

```
Map<City, List<String>> peopleByCity =  
    people.stream()  
        .collect(groupingBy(Person::getCity, mapping(p -> p.getName() +  
            ":" + p.getAge(), toList())));  
    System.out.println(peopleByCity);  
  
{Tulsa=[Joe Harley:42, Eric Greeves:29], London=[Bill Honda:34],  
 Athens=[Amy Beemer:21, Eric Vincent:33]}
```

This works because  
the mapping takes  
place downstream of  
the groupingBy.

```
people.stream()  
    .map(p -> p.getName() + ":" + p.getAge())  
    .collect(groupingBy(Person::getCity));
```

This cannot work  
because the classifier  
cannot access City to  
perform grouping.



## Stream.count(), Collectors.counting and groupingBy

groupingBy affects what is summarized.

```
Long numPpl = people.stream()  
    .count();  
System.out.println(numPpl);
```

5

Number of elements  
in the entire stream.

```
Map<City, Long> numPplByCity =  
people.stream()  
    .collect(groupingBy(Person::getCity, counting()));  
System.out.println(numPplByCity);  
  
{Tulsa=2, London=1, Athens=2}
```

Number of  
elements in  
each group.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Composing Collectors: Tallest in Each City

Collectors have the equivalent of many Stream methods but operating on groups.

```
Map<City, Optional<Person>> oldestByCity = people.stream()
    .collect(groupingBy(Person::getCity,
        maxBy(comparing(Person::getAge))));
```

The forEach method is useful for printing.

```
oldestByCity.forEach((k, v) ->
    { System.out.print(k + ":");
        System.out.println(v.isPresent() ?
            v.get().getFirstName() + " age " + v.get().getAge()
            :"No oldest person!");});
```

Print the key.

Print the value, but as v is an Optional, you can use the isPresent method in a ternary expression to determine what to print. See notes.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A more elegant way to deal with the Optional is to use the map and orElse methods of Stream.

```
oldestByCity.forEach((k, v) ->
    { System.out.print(k + ":");
        System.out.println(
            v.map(person -> person.getFirstName() + " age " + person.getAge())
            .orElse("No oldest person!"));});
```

## groupingBy: Additional Processing with entrySet()

The code below produces a Map of cities showing the population of each.  
But what if a list of cities with population > than 1 is what is required?

```
Map<City, Long> populousCities = people.stream()
    .collect(groupingBy(Person::getCity, counting()));
```

Use entrySet method to create a new Stream.

```
Set<City> populousCities = people.stream()
    .collect(groupingBy(Person::getCity, counting()))
    .entrySet().stream() //Set<Entry<City, Long>>
    .filter(e -> e.getValue() > 1)
    .map(Map.Entry::getKey)
    .collect(toSet());
```

This line creates a new Stream of type shown in the comment. Note that this is also a new pipeline.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

It may seem that it is possible to use a Collector that adapts the first example above instead.

In looking in the documentation, the only candidate is collectingAndThen.

To use collectingAndThen, you could do the following, but the example above is more succinct and readable. Note that either way, there are two separate stream pipelines being used, so the compiler will not be able to combine them for optimization purposes.

```
Map<City, Long> populousCities = people.stream().parallel()
    .collect(collectingAndThen(groupingBy(Person::getCity, counting()), f -> {
        // Need to iterate through map to search for cities with > 1 pop
        // Will need to use entrySet to do this just as in the other version!
        f.entrySet().removeIf(d -> d.getValue() < 2);
        return f;
    }));
}
```

## Agenda

- Introduction to collectors
- Three argument collect method of Stream
- Single argument collect method of Stream
- Grouping by collectors
- **Nested values**
- Complex custom collectors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Nested Values

The remaining examples use a more complex type, ComplexSalesTxn.

- More fields than Person.
- One of the fields is a `List` type.

Fields of `ComplexSalesTxn` class

```
private long txnId;  
private String salesPerson;  
private Buyer buyer;  
private List<LineItem> lineItems;  
private LocalDate txnDate;  
private String city;  
private State state;  
private String code;
```

Fields of `LineItem` class

```
private String name;  
private int quantity;  
private int unitPrice;;
```

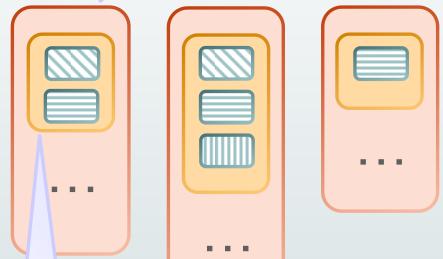
A `ComplexSalesTxn` may have one or many `LineItem` elements.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Data Organization of ComplexSalesTxn

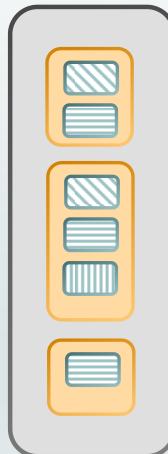
A ComplexSalesTxn element.



A List of LineItem elements.

Other fields of ComplexSalesTxn elements.

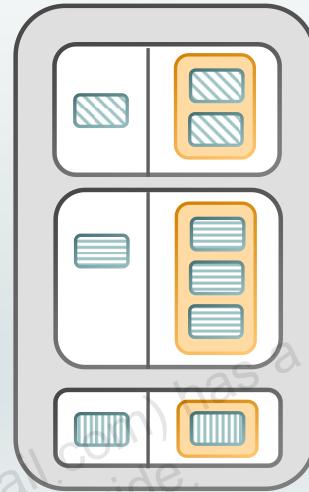
mapping,  
toList



flatMapping,  
toList



flatMapping,  
groupingBy



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Note how flatMapping ensures the individual Lists that group the LineItem elements in each transaction do not exist in the output in the flatMapping/toList collector. In the flatMapping/groupingBy collector, the individual LineItem elements are once more grouped in a List, but this time they're grouped by whatever is the groupingBy classifier, and not by transaction.

## Displaying Nested Values: Listing Line Items in Each Transaction

Use mapping with `toList` to list `LineItem` elements for each transaction.

```
List<List<LineItem>> lineItemsInTransaction = tList.stream()
    .collect(mapping(ComplexSalesTxn::getLineItems, toList()));

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

LineItem  
element.

Result is a List of LineItem elements within a List.

```
[[Widget, Widget Pro II], [Widget Pro], [Widget Pro II, Widget], ... ]
```

List of LineItem  
elements.

List of List of LineItem elements.

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Displaying Nested Values: Listing Line items

Use flatMapping with `toList` to list all the `LineItem` elements.

```
List<LineItem> lineItems = tList.stream()
    .collect(flatMapping(t -> t.getLineItems().stream(), toList()));

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

LineItem  
element.

Result is a List of LineItem elements.  
This can be grouped by product type.

[Widget, Widget Pro II, Widget Pro, Widget Pro II, Widget, ... ]

List of LineItem elements.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Now the List just contains each `LineItem` element without any reference to the transaction it came from.

Notice that in the code above or the code on the previous slide, instead of using the mapping or `flatMapping` adaptor collectors, you could use `map` or `flatMap` operations on the stream. For example, the following code would generate the same List as the code in the slide.

```
List<LineItem> lineItems = tList.stream()
    .flatMap(t -> t.getLineItems().stream())
    .collect(toList());

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

## Displaying Nested Values: Grouping LineItem elements

Use mapping with `toList` to list LineItem elements.

```
Map<String, List<Double>> valueOfEachLineItem = tList.stream()
    .collect(Collectors.flatMapping(t -> t.getLineItems().stream(),
        groupingBy(LineItem::getName,
            mapping(o -> o.getQuantity() * o.getUnitPrice(),
                toList()))));
System.out.println(valueOfEachLineItem);
```

Result is a Map organized by product types. The amount for product type is shown.

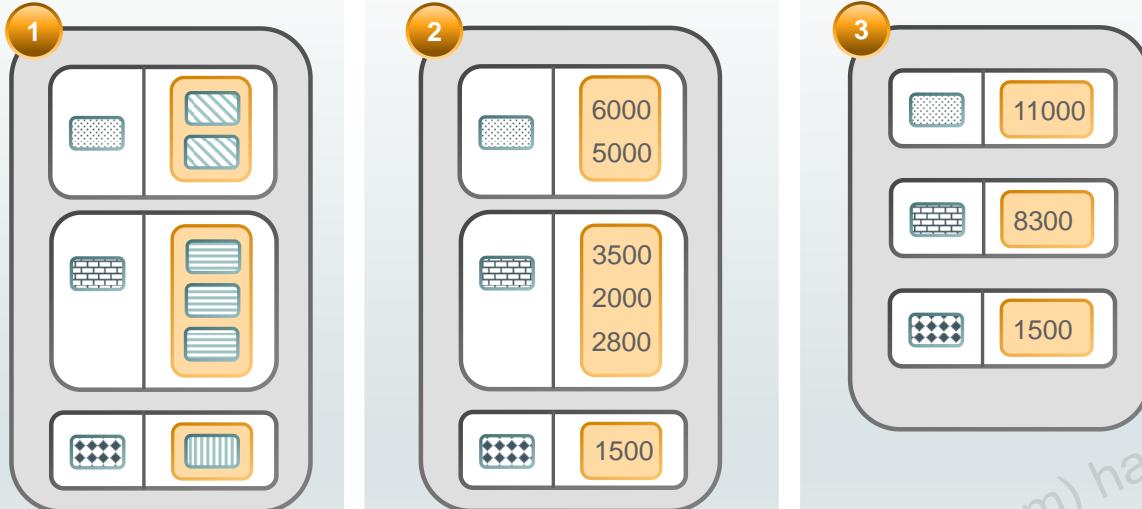
```
{Widget=[6000.0, 6000.0, 36000.0, 10200.0, 16500.0, 6000.0, 11100.0],
 Widget Pro=[20000.0, ...]}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

LineItem elements now grouped by product type of the LineItem element. This allows you to find information like how many of each product have been ordered and which product type generates the most revenue.

## groupingBy Examples for ComplexSalesTxn



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

1. The first example groups the LineItem elements by the type of product.

```
Map<String, List<LineItem>> groupLineItemsByItemName = tList.stream()
    .collect(flatMapping(t -> t.getLineItems().stream(),
        groupingBy(LineItem::getName, toList())));
System.out.println(groupLineItemsByItemName);
{Widget=[Widget, Widget, ...], Widget Pro=[Widget Pro, ...], Widget Pro II=[Widget Pro II, Widget Pro II, ...]}
```

2. The second uses mapping to map from the LineItem type to a Double that displays the value for each product type.

```
Map<String, List<Double>> valueOfEachLineItem = tList.stream()
    .collect(flatMapping(t -> t.getLineItems().stream(),
        groupingBy(LineItem::getName,
            mapping(o -> o.getQuantity() * o.getUnitPrice(), toList()))));
System.out.println(valueOfEachLineItem);
{Widget=[6000.0, 6000.0, 36000.0, ...], Widget Pro=[20000.0, ...], Widget Pro II=[45000.0, 52500.0, ...]}
```

3. The third sums the individual LineItem elements for each product.

```
Map<String, Double> valueOfSalesByProduct2 = tList.stream()
    .collect(flatMapping(t -> t.getLineItems().stream(),
        groupingBy(LineItem::getName,
            summingDouble(o -> o.getUnitPrice() * o.getQuantity()))));
System.out.println(valueOfSalesByProduct2);
{Widget=91800.0, Widget Pro=69500.0, Widget Pro II=435000.0}
```

## Group Items by Salesperson

Use flatMapping and summingDouble to total sales for each salesperson.

```
Map<String, Double> salesPerSalesPerson = tList.stream()
    .collect(groupingBy(ComplexSalesTxn::getSalesPerson,
        flatMapping(t -> t.getLineItems().stream(),
            summingDouble(o -> o.getQuantity() * o.getUnitPrice()))));
    System.out.println(salesPerSalesPerson);
{Samuel Adams=87600, John Smith=116000, Rob Doe=58500 ...}
```

It's a Map so no ordering.

```
salesPerSalesPerson.entrySet().stream()
    .sorted(comparing(Entry::getValue))
    .forEach(System.out::println);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

For a reversed sort, use reversed. Note that you now cannot use method reference and the type is required also.

```
salesPerSalesPerson.entrySet().stream()
    .sorted(Comparator.comparing((Entry<String, Double> a) ->
        a.getValue()).reversed())
    .forEach(System.out::println);
```

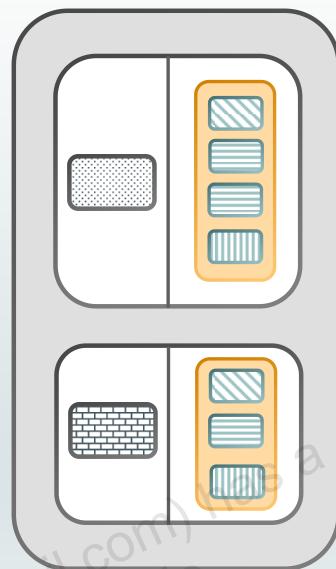
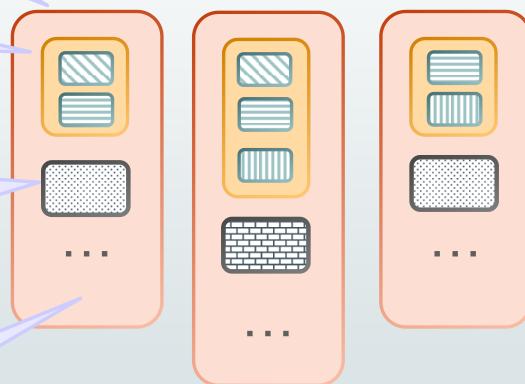
## Group Items by Salesperson

A ComplexSalesTxn element.

A List of LineItem elements.

A field on the ComplexSalesTxn element for salesperson.

Other fields on the ComplexSalesTxn element.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Agenda

- Introduction to collectors
- Three argument collect method of Stream
- Single argument collect method of Stream
- Grouping by collectors
- Nested values
- Complex custom collectors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Complex Custom Collectors

To create more complex custom collectors:

- Create a new custom collector with the functionality required.
  - Sometimes necessary, but can be complex to code.
- Combine the predefined collectors in order to achieve the functionality required.
  - Usually the best approach as the predefined collectors are designed to be combined.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## The collect Method: Using a Custom Collector

```
collect(Collector<? super T,A,R> collector)
```

- The other `collect` method takes a `Collector` as a parameter.
  - Many predefined collectors available in the `Collectors` class

```
List<Person> myPpl = people.stream()
    .collect(new MyCustomCollector());
System.out.println(myPpl);
[Joe, Amy, Bill, Eric, Eric]
```

Uses a custom collector to create an ArrayList Of Person elements. Also possible to use a predefined collector.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Creating a Custom Collector: Methods to Implement

```
Supplier<A> supplier()
BiConsumer<A,T> accumulator()
BinaryOperator<A> combiner()
Function<A,R> finisher()
```

### Types:

- T – the type of the stream elements
- A – the mutable accumulation type
- R – the result type

Unlike the method

`collect(Supplier<R> s, BiConsumer<R,? super T> a, BiConsumer<R,R> c)`  
the result type of a `Collector` is not necessarily the same as the accumulation type.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Custom Example MyCustomCollector

```
public class MyCustomCollector implements  
Collector<Person, List, List> {  
    public Supplier supplier() { return ArrayList::new; }  
    public BiConsumer<List, Person> accumulator() {  
        return List::add;  
    }  
    public BinaryOperator<List> combiner() {  
        return (l1, l2) -> { l1.addAll(l2);  
        return l1;  
    }  
    public Function<List, List> finisher() {  
        return Function.identity();  
    }  
    public Set<String> characteristics() { return Set.of(); }  
}
```

Unlike collect method that specifies combiner as an argument, here the combiner is a BinaryOperator.

The finisher operation allows result type different than combiner type. Here the finisher specifies using the combiner type as result type.

The characteristics method here is returning no characteristics for the collector.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Collector characteristics can be:

- CONCURRENT – Indicates that this collector is concurrent, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.
- IDENTITY\_FINISH – Indicates that the finisher function is the identity function and can be elided.
- UNORDERED – Indicates that the collection operation does not commit to preserving the encounter order of input elements.

## Finisher Example MyCustomCollector

```
public class MyCustomCollector
    implements Collector<Person, List, Person[]> {
    ... < supplier, accumulator, and combiner as previous slide > ...
    public Function<List, Person[]> finisher() {
        return l -> (Person[]) l.toArray(new Person[0]);
    }
    public Set characteristics() {return
        Collections.singleton(Characteristics.UNORDERED); }
}
```

Result type different than accumulator type.

Converting the ArrayList to a Person[].

Indicates that the collection operation does not guarantee preserving the order of the elements.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In the example, the collector has a different return type than the type used in the accumulator. An ArrayList is much more convenient to work with than an array, so it may be a better choice for the accumulator, but if a Person array is required, this could be provided by the finisher.

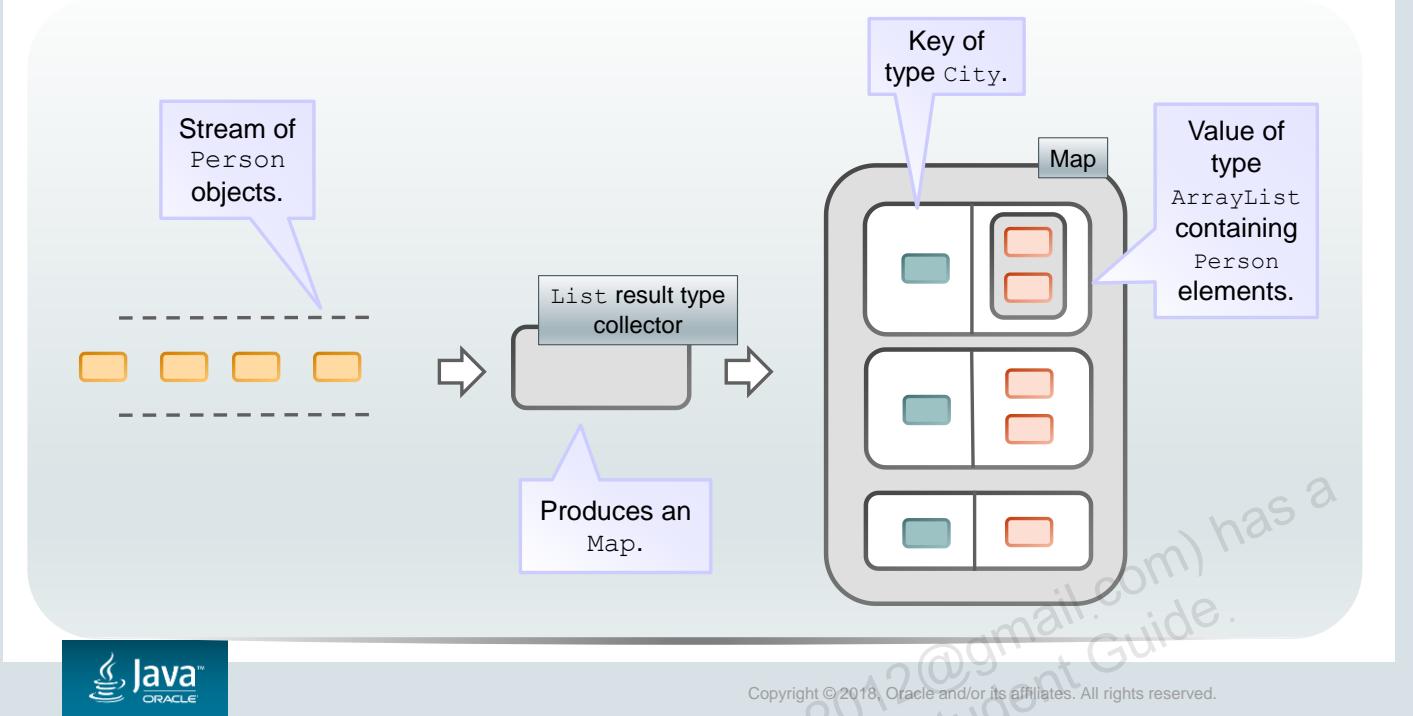
This is a very simple example and is intended to show the functionality of the collector's methods rather than a realistic typing decision.

Note that the Collector.of method can also be used to create a Collector. Making the Collector a standalone class makes sense if it's likely to be needed in more than one application, but Collector.of may be convenient if the Collector is not so general purpose. Note that it's still more reusable than `collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R,R> combiner)` as you can create a reference to reuse, and it also gives you the opportunity to create a finisher and to set Characteristics.

For example:

```
Collector<Person, List, Person[]> myCustomCollector
    = Collector.of(
        ArrayList::new,           // Supplier
        List::add,                // Accumulator
        (l1, l2) -> {           // Combiner - not called if
sequential stream
            l1.addAll(l2);
            return l1;
        },
        l -> (Person[]) l.toArray(new Person[0]) // 
    );
    Collector.of() is overloaded so the finisher does not have to be included.
)
```

## A More Complex Collector



For coding a collector that groups Person elements within a Map requires coding the supplier, accumulator, combiner. The code is shown on the next page. It is quite a bit more involved than the simple collector for producing a List of elements.

In the next topic of this lesson, you look at another way to do this by combining predefined collectors. It is interesting to compare the complexity of coding a collector for this functionality versus combining collectors to achieve the same result.

## A More Complex Collector CustomGroupingBy

```
Map<City, List<Person>> myPpl = people.stream()
    .collect(toCustomGroupingBy());

System.out.println("\nResult: " +
    myPpl.getClass().getCanonicalName() + " : " + myPpl + "\n");

Result Ppl4: java.util.HashMap : {Athens=[Amy, Eric], Tulsa=[Joe,
Eric], London=[Bill]}
```

- The functionality of the custom grouping collector is similar but less flexible than the supplied grouping by collectors.
- It illustrates why it is almost always better to build the collector you need by combining a number of the collectors from the `Collectors` class.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

```

public class CustomGroupingBy implements Collector<Person, Map<City,
    List<Person>>, Map<City, List<Person>>> {
    public static CustomGroupingBy toCustomGroupingBy() {
        return new CustomGroupingBy();
    }
    @Override
    public Supplier supplier() {
        return HashMap::new;
    }
    @Override
    public BiConsumer<Map<City, List<Person>>, Person> accumulator() {
        return (Map<City, List<Person>> h, Person p) -> {
            h.merge(p.getCity(),
                new ArrayList<>(List.of(p)),
                (List<Person> a, List<Person> b) -> {a.add(p); return a;});
        };
    }
    @Override
    public BinaryOperator<Map<City, List<Person>>> combiner() {
        return (Map<City, List<Person>> h1, Map<City, List<Person>> h2) -> {
            h2.forEach((City c, List<Person> l) -> {
                h1.merge(c, l,
                    (List<Person> a, List<Person> b) -> {
                        b.forEach((Person y) -> a.add(y));
                        return a;
                    });
            });
            return h1;
        };
    }
    @Override
    public Function<Map<City, List<Person>>, Map<City, List<Person>>>
        finisher() {
        return Function.identity();
    }
    @Override
    public Set characteristics() {
        return Collections.singleton(Characteristics.IDENTITY_FINISH);
    }
}

```

## Summary

In this lesson, you should have learned how to:

- Create a collection
- Group elements into a collection
- Perform summarizing on a collection
- Compose collectors into a collection
- Create a custom collector



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Practice 15: Overview

This practice covers the following topics:

- Practice 15-1: Review: A Comparison of Iterative Approach, Streams, and Collectors
- Practice 15-2: Using Collectors for Grouping



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.





Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Creating Custom Streams

16



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



ORACLE®

Adolfo De+la+Rosa (adolfodelarosa2020@gmail.com) has a  
non-transferable license to use this Student Guide.

## Objectives

After completing this lesson, you should be able to:

- Create a custom stream based on a custom data type
- Create a custom Spliterator



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Topics

- Performance and Spliterators
- Custom Spliterators

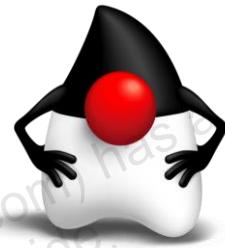


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Performance: Intuition and Measurement

- For small data sets, sequential is usually faster.
- Watch out for boxing.
- Simpler pipelines are easier to intuitively assess.
- Complex pipelines are more difficult to assess.
  - A stream source derived from an Iterator
  - Pipeline containing a limit operation
  - Complex reduction using classification (groupingBy)
- Measure!
  - Use a benchmark tool



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Parallel Versus Sequential Example

```
long begin = 100_000_000;
long window = 100_000;
getNumPrimes(begin, begin + window);

static void getNumPrimes(long begin, long end) {
    LongStream newStream = LongStream.range(begin, end)
        .filter(i -> isPrime(i));
    System.out.println("Num of primes = " + newStream.count());
}

static boolean isPrime(long n) {
    return LongStream.rangeClosed(2, (long) sqrt(n))
        .noneMatch(divisor -> n % divisor == 0);
}
```

Use  
.parallel()  
?      Yes  
No



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

This is a good example of where testing really helps as it is quite difficult to foresee. The isPrime method tends to split into many tasks. Also the short-circuiting operation, noneMatch, while supported in parallel operations, works less optimally than for a single task. Alternatively, the getNumPrimes method has a simple task of calling isPrime. Parallelizing here will have the effect of running a number of isPrime methods in parallel, a clean separation.

## Spliterator

Parallel analogue of `Iterator`, but better, plus decomposition for parallel processing.

- Traverses and partitions elements from the source
- Adds a split operation
- More efficient access
  - Does not require a method call to determine if there are remaining elements
- Reports characteristics of source (used by clients of the `Spliterator`):
  - `SIZED`, `DISTINCT`, `ORDERED`, `SORTED`
  - `CONCURRENT`, `IMMUTABLE`, `NONNULL`, `SUBSIZED`
- JDK collections come with good spliterator implementations
  - Can write your own for custom data sources



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

See `Spliterator` documentation for a summary of the characteristics.

# Spliterator

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    void forEachRemaining(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
    Comparator getComparator();
    ...
}
```

If a remaining element exists, performs the given action on it, returning true; else returns false.

Return a new Spliterator and adjust this Spliterator so they will not cover same elements. Return null if no further splits should be made.

If this Spliterator has SORTED as a characteristic, return the Comparator.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Size is an estimate unless characteristics say otherwise!

Note tryAdvance(). The elements are pushed into the Consumer.

This Consumer instance will be doing the filtering, mapping, and summing (for example). That's how the code gets delivered to where the iteration occurs, and that's how performance is enhanced and parallelism achieved.

## Decomposition with `trySplit()`

```
ArrayList l = ...;

Spliterator s = lspliterator();

Spliterator s1 = s.trySplit();

Spliterator s2 = s.trySplit();

Spliterator s3 = s1.trySplit();
```



## Integration with Streams

- A collection has a Spliterator.
  - `Collection.spliterator()`
  - JDK collections classes provide optimal implementations.
- A collection-based stream is constructed from the collection's `Spliterator`.

```
public interface Collection<E> extends Iterable<E> {  
  
    default Stream<E> stream() {  
        return StreamSupport.stream(spliterator(), false);  
    }  
    ...  
}
```

Resultant  
Stream  
will not be  
parallel.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

If `true` is used instead of `false`, the resultant stream may be parallel, but it is not guaranteed. For some collection types, it is very difficult to split the collection and would require so much work that it would not provide any performance improvement.

A `LinkedList` is a good example of this, as it must be traversed sequentially in order to get to a possible split point.

## Modifying LongStream Spliterator

```
static void getNumPrimes(long begin, long end ) {  
    LongStream newStream =  
        StreamSupport.longStream(  
            new TestSpliterator(range(begin, end).spliterator()),  
            true)  
            .filter(i -> isPrime(i));  
    System.out.println("Num of primes = " + newStream.count());  
}  
  
static boolean isPrime(long n) {  
    return LongStream.range(2, (long) sqrt(n))  
        .noneMatch(divisor -> n % divisor == 0);  
}
```

Create a Spliterator by modifying the existing one for a LongStream. Note true to make parallel.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Spliterator TestSpliterator

```
public class TestSpliterator implements Spliterator.OfLong{
    private Spliterator.OfLong split;
    public static AtomicInteger splitNum = new AtomicInteger(1);
    public MyTestSpliterator(Spliterator.OfLong split) {
        this.split = split;
    }
    public boolean tryAdvance(LongConsumer action) {
        return split.tryAdvance(action);
    }
    public Spliterator.OfLong trySplit() {
        splitNum.incrementAndGet();
        return split.trySplit();
    }
    //... Remaining methods not shown ...
}
```

To find out how many times trySplit called.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Wrapping the standard spliterator allows you to experiment a little by choosing to use the standard method implementations for most methods, but add some extra code to a critical method. Generally the API spliterators are very good, and it's unlikely they can be improved, but you may learn about how the spliterator is functioning, which will help you review your approach.

It's worth remembering that *within* a spliterator, code is executed sequentially, even if the spliterator is producing a parallel stream. The notion is "thread confinement". Each spliterator is executed by a single thread; the parallelism comes from multiple threads operating on multiple spliterators. Thus, methods can modify instance state of this spliterator without worrying about locking. However, as `splitNum` operates on static state of `TestSpliterator`, which is shared across multiple threads, it uses an `AtomicInteger` instead of a plain `int`.

## Using TestSpliterator

```
long begin = 100_000_000;
long window = 100_000;
getNumPrimes(begin, begin + window);
System.out.println("Number splits: " +
    TestSpliterator.splitNum);
```

Number splits: 5



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

While the number of splits chosen by the JDK is not determinable, you can write your own custom Spliterator and in your code determine, say, the maximum number of splits that will be allowed.

## Topics

- Performance and Spliterators
- Custom Spliterators



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Creating a Custom Spliterator

The standard Spliterators for Collections in the JDK are very good.

- It's not necessary to write a custom Spliterator for collections in the JDK.
- It may be useful to write a custom Spliterator for a Collection you have created.
  - Even then:
    - Determine if it is really needed
    - Determine if the collection is really needed
    - Determine if parallel support is needed

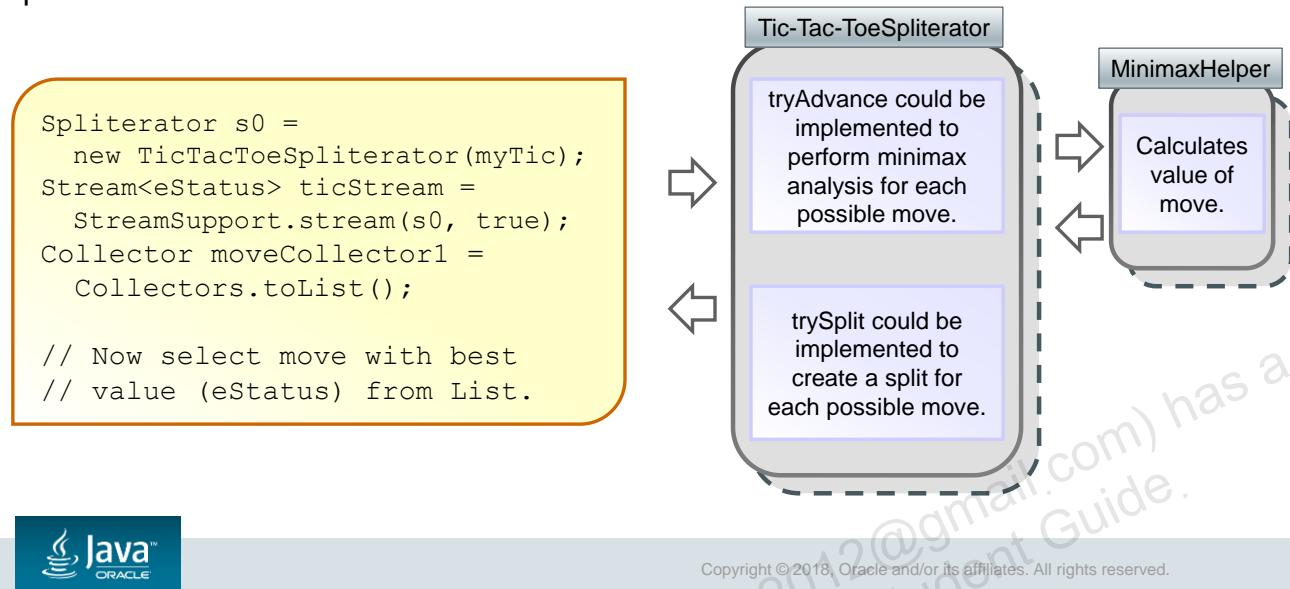


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



# Is a Custom Spliterator Needed for a Game Engine?

Writing a game engine like chess or even tic-tac-toe may seem to require a custom spliterator..



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In a partially completed tic-tac-toe game like the following (assume the game engine is playing 'X'):

```
O X - -  
- - X O  
- - - -  
O X - O
```

`myTic` is of type `TicTacToe` and contains an array that represents this move. It also contains methods to determine what is a legal move and whether a move wins by creating a line.

In the example position, the positions marked with a '-' represent a possible move. Therefore the Spliterator must:

1. Try all possible responses for each move, all possible responses to that, and so on until the end of the game.
2. Determine the score (win, loss, draw) of each of these possible moves by calling a `MinimaxHelper` class.
3. Return a stream based on the values determined by `MinimaxHelper`.

For example, in the position shown the results for each possible move (reading left to right, top to bottom) are: loss, tie, tie, tie, tie, tie, tie, tie, loss.

(Code for this implementation is found in the example folder for this lesson.)

## A Tic-Tac-Toe Engine Using the `map` Method of Stream

But the tic-tac-toe engine can be implemented just as easily by using the `map` method. Remember  $N * Q$  performance model—here  $N$  is small ( $< 15$ ) while  $Q$  is very large.

```
List<TicTacToe> candidateMoves = new ArrayList<>();
// Code to populate the List with TicTacToe objects that represent
// all possible moves

Stream<TicTacToe> ticStream = candidateMoves.parallelStream();
List<eStatus> possMoves = ticStream
    .map(n -> MinimaxHelper.getMoveValue(n, computerPlayer))
    .collect(Collectors.toList());

// Choose move with best value (eStatus)
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In this example, there is no custom Spliterator. As the stream encounters each element, the map function calls MinimaxHelper to determine the value of that move.

(Code for this implementation is found in the example folder for this lesson.)

The  $N * Q$  performance model is covered in the lesson titled “Parallel Streams.” Here it is again.

$N$  = Size of the source data set

$Q$  = Cost per element through the pipeline

$N * Q \approx$  Cost of the pipeline

As stated in the lesson titled “Parallel Streams,” generally a data set should contain more than 10,000 items before showing a difference in performance. In the tic-tac-toe example given, the data set is never more than the number of free moves. In a  $4 \times 4$  tic-tac-toe game, that’s less than 16. But because  $Q$  is so very great, given that it recursively investigates so many board states, there is still a significant advantage to running in parallel.

It’s also interesting that even then improvements to the minimax algorithm can have a huge effect. Sometimes jumping on parallelizing might not be the best way to increase performance even when it is a possible approach.

## A Custom Spliterator Example for a Custom Collection

If a custom binary or n-ary tree is required, stream support may also be necessary.

The next example illustrates:

- Two approaches for a Spliterator for an n-ary Tree
- Support for parallelizing the Spliterator
- Traversal only for preorder traversal



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



The Java API includes tree data types as:

- TreeMap and TreeSet, which use binary trees internally
- Nashorn trees
- Swing trees

It does not include any simple tree data types that can be traversed in the standard inorder, preorder, or postorder order. If you need this functionality, you may need to create a custom tree type.

Some external libraries provide tree types; for example, TreeTraverser is provided by the Guava library.

The Tree used in the following slides and in the practices illustrates the creation of a suitable Spliterator to traverse the tree. Note that if the Spliterator supports parallel streams, the order may not be retained.

# Custom N-ary Tree

```
Owner
|- Head of Installation Dept
|   |- Manager
|   |- Fitter 1
|   |- Fitter 2
|   '- Fitter 3
|- Estimator
|   '- Scheduler
'- Custom Fitter
|- Head of Sales Dept
|   |- North and East Sales
|   '- South Sales
|- Head of Manufacturing Dept
|   |- Head of Design
|   |- Manufacture Supervisor
|   |   |- Machine Operator 1
|   |   |- Machine Operator 2
|   |   '- Machine Operator 3
'- Repairs and Upkeep
```

N-ary tree shown as an outline for convenience.

Preorder traversal will be in order of elements listed.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Possible Implementation for NaryTreeSpliterator

```

public class NTreeAsListSpliterator implements Spliterator<Node>
    private List treeList; private int current, end;
    NTreeAsListSpliterator(Tree t) {
        this.treeList = t.getPreOrderList(t.root, new ArrayList());
        current = 0; end = treeList.size();
    }
    public boolean tryAdvance(Consumer action) {
        if (current < end) {
            action.accept(treeList.get(current));
            current++;
            return true;
        } else return false;
    } // ... See notes for details of other methods
}

```

Tree is stored in preorder order in a List. This makes it easy to work with.  
preorder is a method on BinaryTree.

The tryAdvance method just needs to keep track of an index into the List.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

This is a very simple approach to traversing the tree. If the tree is used to populate a List in preorder order, then the Spliterator can simply work with the List making traversal and splitting quite straightforward. Part of the constructor for the Spliterator could specify preorder—indeed the traversal could be specified with a lambda expression!

The remaining three methods can be implemented very simply as here:

```

@Override
public Spliterator<Node> trySplit() {
    return null;
}

```

```

@Override
public long estimateSize() {
    return end - current;
}

```

```

@Override
public int characteristics() {
    return 0;
}

```

## Stream<Node> in Use

Note the use of Node to set the stream type.

```
Spliterator<Node> s0 = new BinaryTreeSpliterator(theTree);
```

```
StreamSupport.stream(s0, false)  
    .forEach(System.out::println);
```

Assuming an N-ary tree called theTree.

Sequential, so no parallel support required (trySplit method could just return null.)

```
Owner  
Head of Installation Dept  
Manager  
Fitter 1  
Fitter 2  
Fitter 3  
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Implementing Parallel Processing

```
NTreeAsListSpliterator(List treeList,
    int splitPoint, int end) {
    this.treeList = treeList;
    current = splitPoint;
    this.end = end;
}

public Spliterator<Node> trySplit() {
    int splitPoint = (end - current + 1)/2 + current;
    System.out.println(">> Splitting on " + splitPoint);
    NTreeAsListSpliterator newSplit =
        new NTreeAsListSpliterator(treeList, splitPoint, end);
    this.end = splitPoint;
    return newSplit;
}
```

Another constructor is required to create a Spliterator to cover just part of the Tree. Note it's only necessary to set indexes for the List.

trySplit creates a new Spliterator on the same List but changes the indexes.

Including the indexes for the current Spliterator (this).

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Stream<Node> in Use

```
StreamSupport.stream(s0, true)  
    .filter(b -> b.isLeaf())  
    .forEach(System.out::println);
```

Set for parallel.

Note order is not preorder.

```
>> Splitting on 10  
>> Splitting on 5  
>> Splitting on 15  
>> Splitting on 3  
>> Splitting on 8  
>> Splitting on 13
```

Output continued

```
Head of Design  
Manufacture Supervisor  
North and East Sales  
North and East Sales  
South Sales  
Head of Manufacturing Dept  
>> Splitting on 17
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Summary of NTreeAsListSpliterator

Using a preordered List of the tree nodes is easy to implement, but:

- It requires the entire tree to be traversed and rendered as a `List` before any traversal takes place.
  - This needs to be done for each new traversal order required.

Another approach could be:

- Use an N-ary tree to provide the data for the `Spliterator`
- Have `tryAdvance` move one step along the tree in preorder order
- Create splits by:
  - Creating a new root node for a new tree based on the current tree
  - Changing the current tree to cut off the new tree when traversing

An example of an N-ary Tree Spliterator will be examined in the practice for this lesson.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Summary

In this lesson, you should have learned how to:

- Create a custom Spliterator
- Create a custom stream based on a custom data type

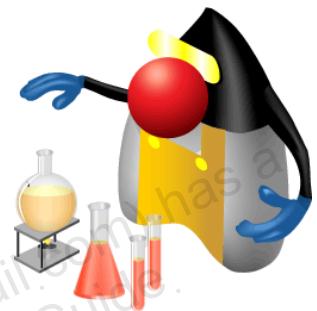


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Practice 16: Overview

This practice covers the following topics:

- Practice 16-1: Examine the PrimeNumbersExample Application
- Practice 16-2: Using JMH (Java Microbench Harness)
- Practice 16-3: Run the Tictactoe Game Engine
- Practice 16-4: Examine a Custom Spliterator



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Java I/O Fundamentals and File I/O (NIO.2)

17



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



ORACLE®

## 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
- 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



Copyright © 2018, 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.

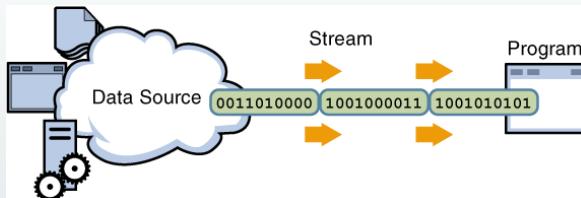


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

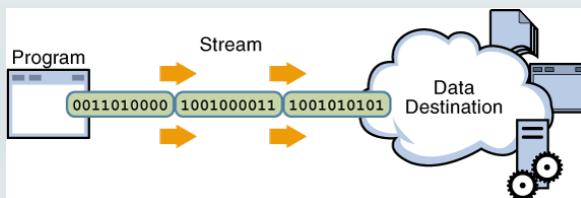
Some I/O streams simply pass on data; others manipulate and transform the data in useful ways.

## 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 a time.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

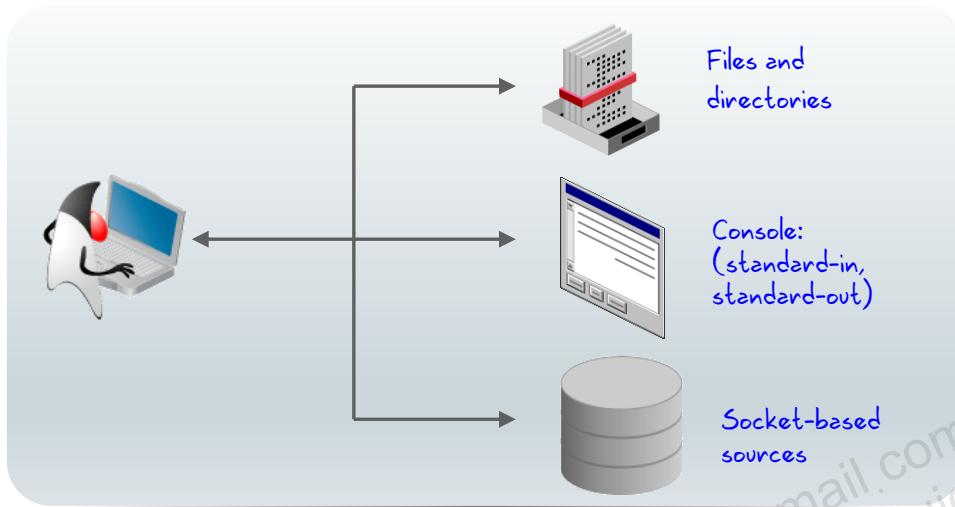
No matter how they work internally, all streams present the same simple model to programs that use them. A stream is a sequential flow of data. A stream can come from a source or can be generated to a sink.

- 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:



Copyright © 2018, 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 © 2018, 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

## 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
```

## 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 © 2018, 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: 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 © 2018, 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 Methods

## 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()
```

## Writer Methods:

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



Copyright © 2018, 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.

## 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 }
```

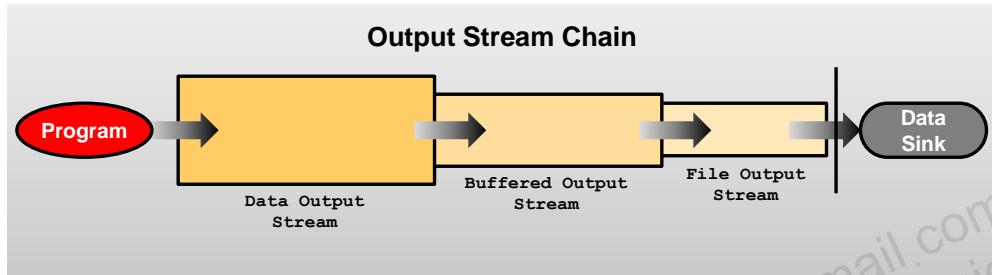
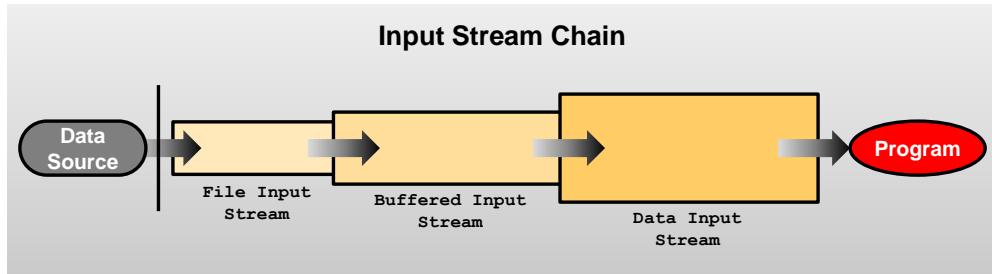
Now, rather than a byte array, this version uses a character array.



Copyright © 2018, 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



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.

## 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 © 2018, Oracle and/or its affiliates. All rights reserved.

# 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 © 2018, 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 © 2018, 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  
InputStreamReader(System.in))) {  
12             System.out.print("Type xyz to exit: ");  
13             s = in.readLine();  
14             while (s != null) {  
15                 System.out.println("Read: " + s.trim());  
16                 if (s.equals("xyz")) {  
17                     System.exit(0);  
18                 }  
19                 System.out.print("Type xyz to exit: ");  
20                 s = in.readLine();  
21             }  
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 © 2018, 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.

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

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



Copyright © 2018, 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 nonpersisted 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.

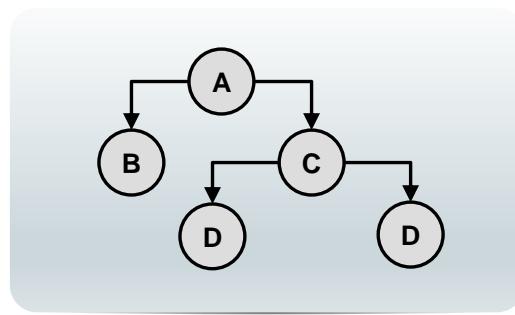


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

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*.



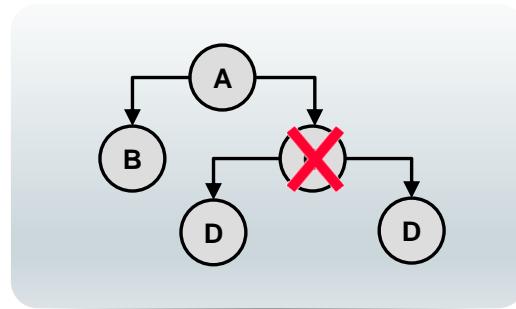
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 nonserializable 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`.



Copyright © 2018, 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 nonstatic transient fields is set to the default value for the type.



Copyright © 2018, 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 nonstatic 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 values of nonstatic 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 © 2018, Oracle and/or its affiliates. All rights reserved.

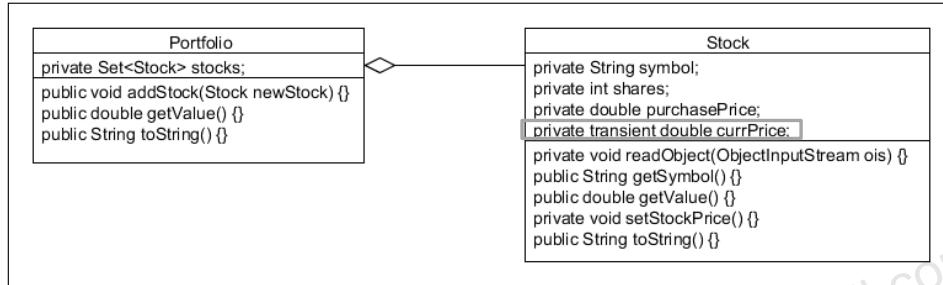
**Note:** The documentation for `java.io.Serializable` states the following:

“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.



Copyright © 2018, 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);           The writeObject method writes the object graph of p to the file stream.
9     } catch (IOException i) {
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 restores the object from the file stream.
15    } catch (ClassNotFoundException | IOException i) {
16        System.out.println("Exception reading in Portfolio: " + i);
17    }

```



Copyright © 2018, 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 nontransient and nonstatic 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 nonstatic and nontransient 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 © 2018, 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(); setStockPrice() is called here  
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(); setStockPrice() is called here  
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 © 2018, 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.

## New File I/O API (NIO.2)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

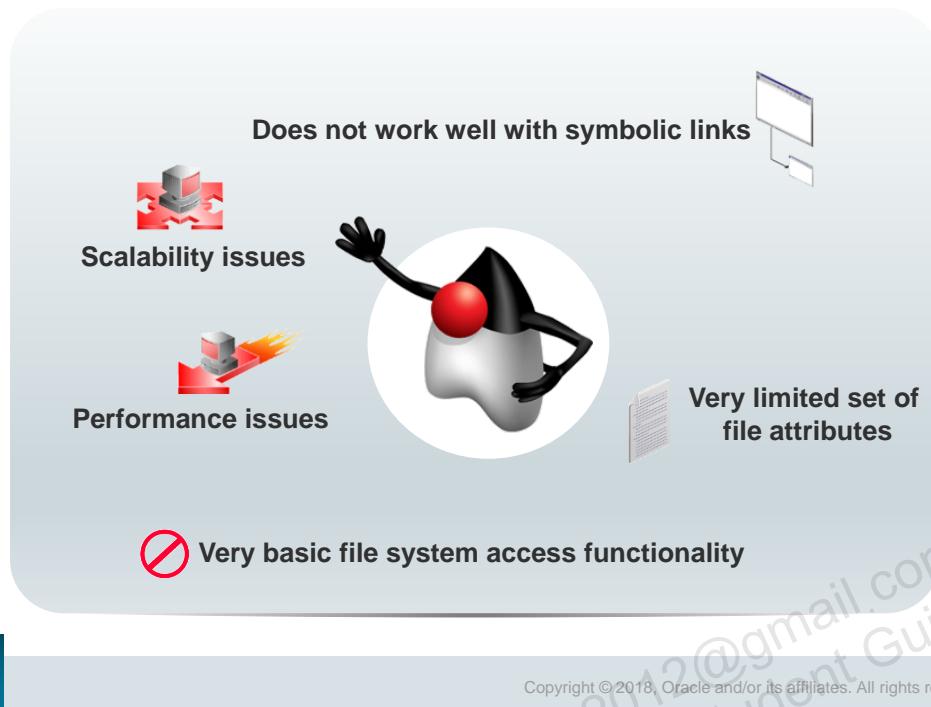
NIO API in JSR 51 established the basis for NIO in Java, focusing on buffers, channels, and charsets. JSR 51 delivered the first piece of the scalable socket I/Os into the platform, providing a nonblocking, multiplexed I/O API, thus allowing the development of highly scalable servers without having to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



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

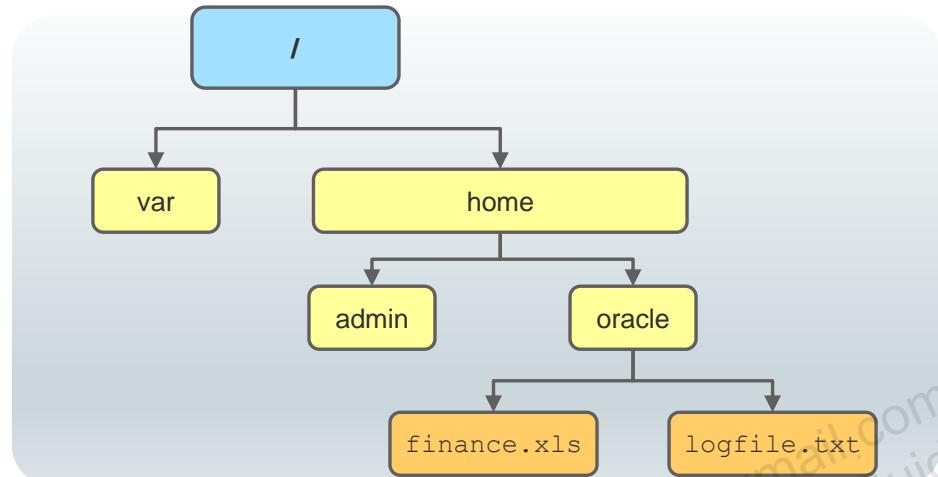
The Java I/O File API (`java.io.File`) presented challenges for developers.

- 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.



Copyright © 2018, 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 © 2018, 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 © 2018, 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 are 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 © 2018, 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 © 2018, 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*, which 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 © 2018, 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 © 2018, 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.



Copyright © 2018, 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.

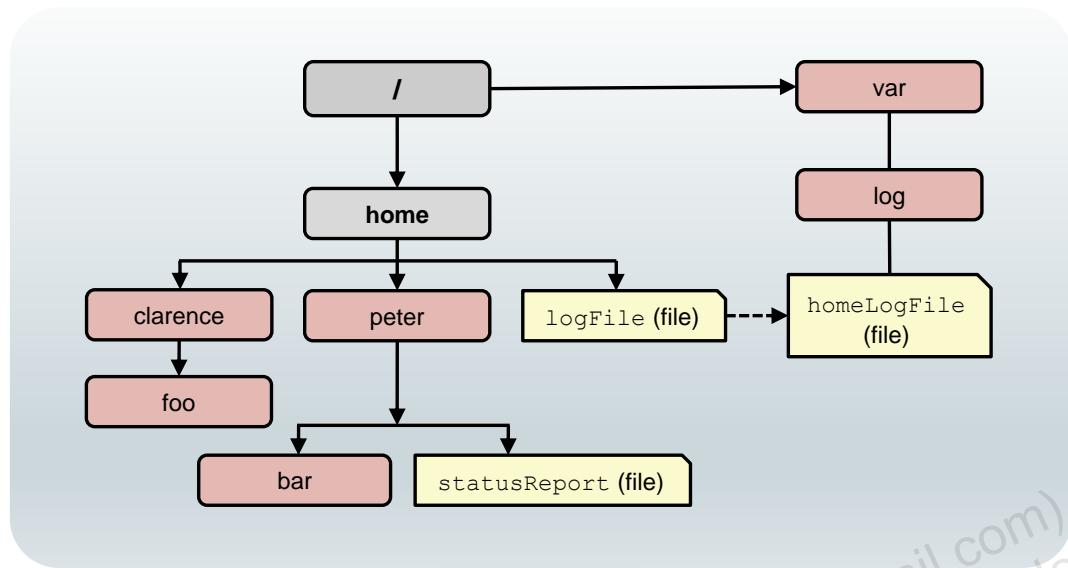
```
Paths.get("foo").resolve("/home/clarence"); // Returns /home/clarence
```



Copyright © 2018, 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.

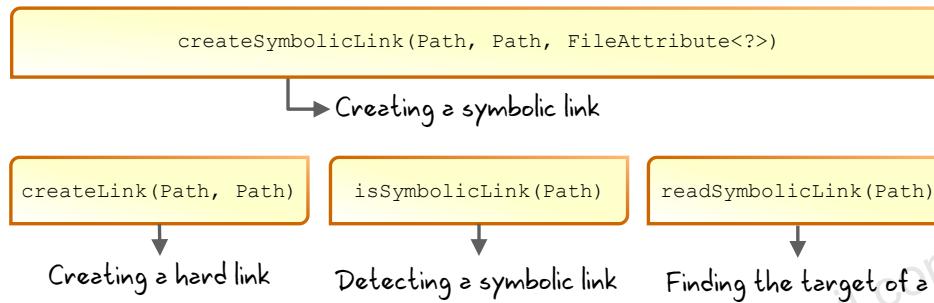
## Symbolic Links



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# 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



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

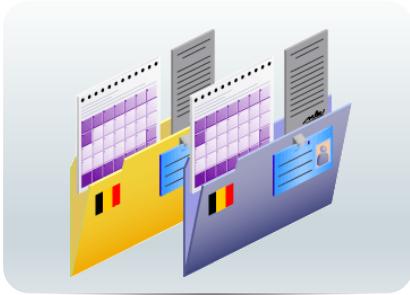
The `java.nio.file` package and the `Path` interface in particular are “link aware.” Every `Path` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

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



Copyright © 2018, 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.

Please refer to the examples provided for this lesson to implement the following operations using `File`:

Checking a File or Directory

Deleting a File

Copying a File

Moving a File

Managing Metadata

## 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 I/O streams to read and write files
- Read and write objects by using serialization
- 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



Copyright © 2018, Oracle and/or its affiliates.

## Quiz



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

- A. private
- B. static
- C. transient
- D. final



Copyright © 2018, 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## Quiz



Given a Path object with the following path:

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

Which Path method would remove the redundant elements?

- A. normalize
- B. relativize
- C. resolve
- D. toAbsolutePath



Copyright © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## Practice 17: Overview

This practice covers the following topics:

- 17-1: Writing a simple console I/O Application
- 17-2: Working with files



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Secure Coding Guidelines



ORACLE®



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adolfo De+la+Rosa (adolfodelarosa2022@gmail.com) has a  
non-transferable license to use this Student Guide.

## Objectives

After completing this lesson, you should be able to:

- Describe the Java SE security overview
- Explain vulnerabilities
- Describe the secure coding guidelines and antipatterns



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Java SE Security Overview

The Java language and virtual machine provide many features to mitigate common programming mistakes.

- The language is type-safe, and the runtime provides automatic memory management and bounds-checking on arrays.
- Java programs and libraries check for illegal state at the earliest opportunity.
- To minimize the likelihood of security vulnerabilities caused by programmer error, Java developers should adhere to recommended coding guidelines.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Java SE Security Overview

- Runtime security
  - Behavior enforced by Java runtime
  - Enables application to be run safely in a restricted environment
- Security APIs
  - Standard APIs for Cryptography, PKI, Authentication, Policy, and secure communication
  - Pluggable implementations
- Security tools
  - `keytool`, `jarsigner`, `policytool`



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Secure Coding Guidelines

Secure coding guidelines provide a more complete set of security-specific coding guidelines targeted at the Java programming language.

- The guideline is organized into nine sections:

Guidelines	
0	Fundamentals
1	Denial of Service
2	Confidential Information
3	Injection and Inclusion
4	Accessibility and Extensibility
5	Input Validation
6	Mutability
7	Object Construction
8	Serialization and deserialization
9	Access control



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Guidelines address the top-level concerns when writing secure code.

**Note:** This lesson doesn't cover the Access control guideline, and you can refer to the details from this tutorial.

Secure Coding Guidelines for Java SE ,

<https://www.oracle.com/technetwork/java/seccodeguide-139067.html>

While sections 0 through 3 are generally applicable across different types of software, most of the guidelines in sections 4 through 9 focus on applications that interact with untrusted code (though some guidelines in these sections are still relevant for other situations). Developers should analyze the interactions that occur across an application's trust boundaries and identify the types of data involved to determine which guidelines are relevant. Performing threat modeling and establishing trust boundaries can help to accomplish this.

These guidelines are intended to help developers build secure software, but they do not focus specifically on software that implements security features. Therefore, topics such as cryptography are not covered in this document (for information on using cryptography with Java). While adding features to software can solve some security-related problems, it should not be relied upon to eliminate security defects.

## Vulnerabilities

- Definition:
  - A flaw or weakness that could be exploited to violate the systems security policy.
- Causes:
  - Design: Faulty assumptions in the application architecture
  - **Implementation: Insecure programming practices (antipatterns)**
  - Composition and setup: Errors in configuration



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Secure Coding Antipatterns

An antipattern is a programming practice that you should avoid.

- May look beneficial in the first but may result in bad consequences
- For example: Implementing speed-optimized methods that don't validate input parameters



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Antipatterns in Java

Common misconceptions lead to a larger attack surface.

- Neglecting to verify valid input formatting
- Granting unnecessary permissions to code
- Misusing mutable public static variables
- Ignoring changes to superclasses
- Assuming exceptions are harmless
- Assuming the value space of integers is unbounded
- Assuming that a constructor exception destroys the object



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Fundamentals

Fundamentals	
0	Prefer to have obviously no flaws rather than no obvious flaws
1	Design APIs to avoid security concerns
2	Avoid duplication
3	Restrict privileges
4	Establish trust boundaries
5	Minimize the number of permission checks
6	Encapsulate methods, fields, and classes to coherent sets of behavior

## Why should I care?

- Security vulnerabilities will never be eliminated.
- Well-designed and tested code is easier to secure.
- Least privilege by design makes the code more secure.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



# Fundamentals

## 0-1: Design APIs to avoid security concerns

- It is better to design APIs with security in mind.
- Trying to retrofit security into an existing API is more difficult and error prone.
- For example, making a class final prevents a malicious subclass from:
  - Adding finalizers
  - Cloning
  - Overriding random methods
  - Calling protected methods



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Fundamentals

### 0-2 : Avoid duplication

- Key characteristic of secure code is to maximize reuse.
  - Duplication of code and data causes many problems. Both code and data tend not to be treated consistently when duplicated; for example, changes may not be applied to all copies.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Fundamentals

### 0-3: Restrict privileges

- Not all coding flaws will be eliminated even in well-reviewed code.
- Whenever possible use the principle of least privileges.
- Reduced privileges means reduced potential impact of exploits
- Apply the Java security mechanisms:
  - Statically by restricting permissions through policy files.
  - Dynamically with `java.security.AccessController.doPrivileged`



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Fundamentals

## 0-4: Establish trust boundaries

- Having simple APIs to distinguish clearly between inside and outside of trust boundaries
  - In order to ensure that a system is protected, it is necessary to establish trust boundaries.
  - Data that crosses these boundaries should be sanitized and validated before use.
  - Trust boundaries are also necessary to allow security audits to be performed efficiently.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Code that ensures integrity of trust boundaries must itself be loaded in such a way that its own integrity is assured.

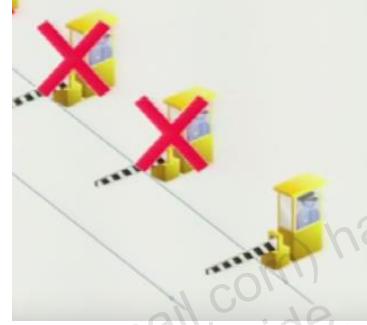
For instance, a web browser is outside of the system for a web server. Equally, a web server is outside of the system for a web browser. Therefore, web browser and server software should not rely upon the behavior of the other for security.

When auditing trust boundaries, there are some questions that should be kept in mind. Are the code and data used sufficiently trusted? Could a library be replaced with a malicious implementation? Is untrusted configuration data being used? Is code calling with lower privileges adequately protected against?

## Fundamentals

### 0-5: Minimize the number of permission checks

- Prefer a point of access
- After initial check, provide clients with (immutable) capability objects.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Fundamentals

## 0-6: Encapsulate

- Group coherent functionality
- Do not expose implementation details.
- Have a simple and stable public API



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Allocate behaviors and provide succinct interfaces. Fields of objects should be private and accessors avoided. The interface of a method, class, package, and module should form a coherent set of behaviors, and no more.

# Fundamentals: Why Should I Care?

## 0-7: Document security-related information

- API documentation should cover security-related information such as required permissions, security-related exceptions, and any preconditions or postconditions that are relevant to security.
- Documenting this information in comments for a tool such as Javadoc can also help to ensure that it is kept up to date.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Allocate behaviors and provide succinct interfaces. Fields of objects should be private and accessors avoided. The interface of a method, class, package, and module should form a coherent set of behaviors, and no more.

## Denial of Service

Input into a system should be checked so that it will not cause excessive resource consumption disproportionate to that used to request the service. Common affected resources are CPU cycles, memory, disk space, and file descriptors.

1- Denial of Service	
1-1	Beware of activities that may use disproportionate resources
1-2	Release resources in all cases
1-3	Resource limit checks should not suffer from integer overflow

### Why should I care?

- Plan against denial-of-service during design phase.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Denial of Service

## 1-1: Beware of activities that may use disproportionate resources

- Examples of attacks include:
  - Large image processing
  - Integer overflows
  - Complex object graphs errors can cause sanity checking of sizes to fail
  - Careless decompression, i.e. "Zip bombs," whereby a short file is very highly compressed
  - "Billion laughs attack" such as that caused by XXE, XML external entity
  - Parsing and processing complex grammars(XPATH,RegEx)
  - Deserialization processing anomalies
  - Logging with inappropriate detail level
  - Parsing corner cases that cause infinite loops



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Requesting a large image size for vector graphics. For instance, SVG and font files.

Integer overflow errors can cause sanity checking of sizes to fail.

An object graph constructed by parsing a text or binary stream may have memory requirements many times that of the original data.

"Zip bombs" whereby a short file is very highly compressed. For instance, ZIPs, GIFs, and gzip-encoded HTTP contents. When decompressing files, it is better to set limits on the decompressed data size rather than relying upon compressed size or meta-data.

"Billion laughs attack" whereby XML entity expansion causes an XML document to grow dramatically during parsing. Set the `XMLConstants.FEATURE_SECURE_PROCESSING` feature to enforce reasonable limits.

Causing many keys to be inserted into a hash table with the same hash code, turning an algorithm of around  $O(n)$  into  $O(n^2)$ .

Regular expressions may exhibit catastrophic backtracking.

XPath expressions may consume arbitrary amounts of processor time.

Java serialization and Java Beans XML deserialization of malicious data may result in unbounded memory or CPU usage.

Detailed logging of unusual behavior may result in excessive output to log files.

Infinite loops can be caused by parsing some corner case data. Ensure that each iteration of a loop makes some progress.

# Denial of Service

## 1-3: Resource limit checks should not suffer from integer overflow

**Antipattern:** Believing the value space of integers is unbounded

- Java language provides bound checking on all arrays.
  - Mitigates the vast majority of integer overflow attacks
- However, all the primitive integer types silently overflow.
  - Potential bypass of Java-level validity checks of native code
  - Causing memory corruption (out of bounds writes)
- Replace suspicious checks

```
private void checkGrowBy(long extra) {
    if (extra < 0 || current > max - extra) {
        throw new IllegalArgumentException();
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

If(current+extra >max) //bad

Some checking can be rearranged so as to avoid overflow. With large values, current + max could overflow to a negative value, which would always be less than max.

A peculiarity of two's complement integer arithmetic is that the minimum negative value does not have a matching positive value of the same magnitude. So, `Integer.MIN_VALUE == -Integer.MIN_VALUE`, `Integer.MIN_VALUE == Math.abs(Integer.MIN_VALUE)` and, for integer a, `a < 0` does not imply `-a > 0`. The same edge case occurs for `Long.MIN_VALUE`.

As for Java SE 8, the `java.lang.Math` class also contains methods for various operations (`addExact`, `multiplyExact`, `decrementExact`, etc.) that throw an `ArithmaticException` if the result overflows the given type.

## Confidential Information

- Confidential data should be readable only within a limited context.
- Data that is to be trusted should not be exposed to tampering.
- Privileged code should not be executable through intended interfaces.

### 2- Confidential Information

- |     |   |
|-----|---|
| 2-1 | Purge sensitive information from exceptions                         |
| 2-2 | Do not log highly sensitive information                             |
| 2-3 | Consider purging highly sensitive information from memory after use |

#### Why should I care?

- Prevent malicious information gathering of your configuration settings and passwords.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Confidential Information

## 2-1: Purge sensitive information from exceptions

- Exception objects may convey sensitive information. For example:
  - `java.io.FileInputStream`, to read an underlying configuration file
  - `java.io.FileNotFoundException`, to probe the file system
- Verbose debugging is good but not in production.
  - Consider separating output channels
  - Reuse a good logging framework



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Confidential Information

## 2-2: Don't log highly sensitive information

- Have a security policy in place.
- Encrypt passwords with standard APIs.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Some information, such as Social Security numbers (SSNs) and passwords, is highly sensitive. This information should not be kept for longer than necessary nor where it may be seen, even by administrators. For instance, it should not be sent to log files, and its presence should not be detectable through searches. Some transient data may be kept in mutable data structures, such as char arrays, and cleared immediately after use. Clearing data structures has reduced effectiveness on typical Java runtime systems as objects are moved in memory transparently to the programmer.

This guideline also has implications for implementation and use of lower-level libraries that do not have semantic knowledge of the data they are dealing with. As an example, a low-level string parsing library may log the text it works on. An application may parse an SSN with the library. This creates a situation where the SSNs are available to administrators with access to the log files.

# Confidential Information

## 2-3:Consider purging highly sensitive from memory after use

- Limit exposure time in memory:
  - Delete as soon as possible.
  - Don't depend on garbage collection.
  - Use `char[]` arrays to clear the traces.
  - Keep the information local.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Injection and Inclusion

A very common form of attack involves causing a particular program to interpret data crafted in such a way as to cause an unanticipated change of control. Typically, but not always, this involves text formats.

3- Injection and Inclusion	
3-1	Generate valid formatting
3-2	Avoid dynamic SQL
3-3	XML and HTML generation requires care
3-4	Avoid any untrusted data on the command line
3-5	Restrict XML inclusion
3-6	Care with BMP files
3-7	Disable HTML display in Swing components
3-8	Take care in interpreting untrusted code

Why should I care?

- Protect data integrity:  
Validate data from untrusted sources.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Injection and Inclusion

## 3-1: Generate valid formatting

**Antipattern:** Neglecting to verify valid input formatting.

- Validate input:
  - Check for out-of-bounds values and escape characters.
  - Regular expressions can help validate String inputs.
  - Pass only validated inputs to subcomponents.
  - Re-use well-tested libraries instead of ad-hoc code.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

### Guideline 3-2 / INJECT-2: Avoid dynamic SQL

It is well known that dynamically created SQL statements including untrusted input are subject to command injection. This often takes the form of supplying an input containing a quote character ('') followed by SQL. Avoid dynamic SQL.

For parameterized SQL statements using Java Database Connectivity (JDBC), use `java.sql.PreparedStatement` or `java.sql.CallableStatement` instead of `java.sql.Statement`.

### Guideline 3-3 / INJECT-3: XML and HTML generation requires care

Untrusted data should be properly sanitized before being included in HTML or XML output. Failure to properly sanitize the data can lead to many different security problems, such as Cross-Site Scripting (XSS) and XML Injection vulnerabilities. There are many different ways to sanitize data before including it in output. Characters that are problematic for the specific type of output can be filtered, escaped, or encoded.

### Guideline 3-4 / INJECT-4: Avoid any untrusted data on the command line

When creating new processes, do not place any untrusted data on the command line. Behavior is platform-specific, poorly documented, and frequently surprising. Malicious data may, for instance, cause a single argument to be interpreted as an option (typically a leading—on Unix or / on Windows) or as two separate arguments. Any data that needs to be passed to the new process should be passed either as encoded arguments (e.g., Base64), in a temporary file, or through a inherited channel.

# Accessibility and Extensibility

The task of securing a system is made easier by reducing the "attack surface" of the code.

Accessibility and Extensibility	
4-1	Limit the accessibility of classes, interfaces, methods, and fields
4-2	Limit the accessibility of packages
4-3	Isolate unrelated code
4-4	Limit exposure of ClassLoader instances
4-5	Limit the extensibility of classes and methods
4-6	Understand how a superclass can affect subclass behavior

## Why should I care?

- Reduce the attack surface.
- Assign the least accessibility required for your code.
- Prevent unwanted modifications of your code.

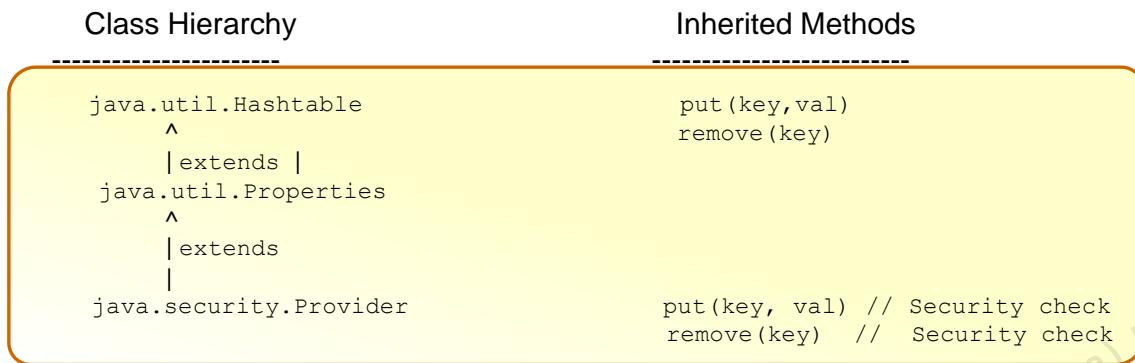


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Accessibility and Extensibility

## 4-6:Understand how a superclass can affect subclass behavior

Consider the following example that occurred in JDK 1.2:



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

# Accessibility and Extensibility

## 4-6: Antipattern-Ignoring changes to superclasses

Attacker bypasses `remove` method and uses inherited `entrySet` method to delete properties.

Class Hierarchy

```
java.util.Hashtable
  ^
  |extends
  |
java.util.Properties
  ^
  |extends
  |
java.security.Provider
```

Inherited Methods

```
put(key, val)
remove(key)
Set entrySet() //Supports removal

put(key, val) // Security check
remove(key) // Security check
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Accessibility and Extensibility

### 4-6:Understand how superclass effects the behavior of a subclass

- Subclasses don't guarantee encapsulation.
  - Subclasses can add new methods.
  - Subclasses may modify behavior of methods that have not been overridden.
- Security checks enforced in subclasses can be bypassed.
  - `Provider.remove` security check is bypassed if attacker calls newly inherited `entrySet` method to perform removal.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Accessibility and Extensibility

### 4-6:Understand how superclass effects the behavior of a subclass

- Avoid inappropriate subclassing
  - Subclass only when the inheritance model is well specified and understood
  - When in doubt, use composition instead of inheritance
- Monitor changes to superclasses
  - Identify behavioral changes to existing inherited methods and override if necessary
  - Identify new methods and override if necessary

```
java.security.Provider          put(key, val) // Security check  
                                remove(key) // Security check  
/*override*/                  Set entrySet() //immutable set
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

# Input Validation

Validating external inputs is an important part of security.

5- Input Validation	
5-1	Validate inputs
5-2	Validate output from untrusted objects as input
5-3	Define wrappers around native methods

## Why should I care?

- Invalid input may trick harmless code into malicious behavior.
- Enables to prevent attackers from modifying the control flow



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

### Guideline 5-1 / INPUT-1: Validate inputs

Input from untrusted sources must be validated before use. Maliciously crafted inputs may cause problems, whether coming through method arguments or external streams. Examples include overflow of integer values and directory traversal attacks by including ".." sequences in filenames. Ease-of-use features should be separated from programmatic interfaces.

### Guideline 5-2 / INPUT-2: Validate output from untrusted objects as input

In general method, arguments should be validated but not return values. However, in the case of an upcall (invoking a method of higher level code) the returned value should be validated. Likewise, an object only reachable as an implementation of an upcall need not validate its inputs.

### Guideline 5-3 / INPUT-3: Define wrappers around native methods

Java code is subject to runtime checks for type, array bounds, and library usage. Native code, on the other hand, is generally not. While pure Java code is effectively immune to traditional buffer overflow attacks, native methods are not. To offer some of these protections during the invocation of native code, do not declare a native method public. Instead, declare it private and expose the functionality through a public Java-based wrapper method. A wrapper can safely perform any necessary input validation prior to the invocation of the native method.

# Mutability

Mutability	
6-1	Prefer immutability for value types
6-2	Create copies of mutable output values
6-3	Create safe copies of mutable and subclassable input values
6-4	Support copy functionality for a mutable class
6-5	Do not trust identity equality when overridable on input reference objects
6-6	Treat passing input to untrusted object as output
6-7	Treat output from untrusted object as input
6-8	Define wrapper methods around modifiable internal state
6-9	Make public static fields final
6-10	Ensure public static final field values are constants
6-11	Do not expose mutable statics
6-12	Do not expose modifiable collections

## Why should I care?

- Runtime security relies on trustworthy objects used by privileged code.
- Don't give chance to attackers to modify those on their behalf.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Mutability

## 6-8: Define wrapper methods around modifiable internal state

If a state that is **internal** to a class must be publicly accessible and modifiable, declare a private field and enable access to it via public wrapper methods.

If the state is only intended to be accessed by **subclasses**, declare a private field and enable access via protected wrapper methods. Wrapper methods allow input validation to occur prior to the setting of a new value.

```
public final class WrappedState {  
    // private immutable object  
    private String state;  
  
    // wrapper method  
    public String getState() {  
        return state;  
    }  
    // wrapper method  
    public void setState(final String newState) {  
        this.state = requireValidation(newState);  
    }  
  
    private static String requireValidation(final String state) {  
        if (...) {  
            throw new IllegalArgumentException("...");  
        }  
        return state;  
    }  
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

# Mutability

## 6-9: Make public static fields as final

### Antipattern: Misusing mutable public static variables

- Always declare public static fields as final.
  - Callers can trivially access and modify public nonfinal static fields
  - Neither accesses nor modifications can be guarded against, and newly set values cannot be validated
- Treat public statics primarily as constants.
  - Consider using enums(type-safe and implicitly static final).

```
public class Files {  
    public static final String separator = "/";  
    public static final String pathSeparator = ":";  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Sensitive static code can be modified by untrusted code.

Static variables are global across a Java runtime environment.

Treat public statics primarily as constants-

Consider using enums(type-safe and implicitly static final).

# Object Construction

7- Object Construction	
7-1	Avoid exposing constructors of sensitive classes
7-2	Prevent the unauthorized construction of sensitive classes
7-3	Defend against partially initialized instances of nonfinal classes
7-4	Prevent constructors from calling methods that can be overridden

Why should I care?

- Stay in charge of creation of critical object instances.
- Don't let attackers control critical classes.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Guideline 7-1 / OBJECT-1: Avoid exposing constructors of sensitive classes

Construction of classes can be more carefully controlled if constructors are not exposed. Define static factory methods instead of public constructors.

## Guideline 7-2 / OBJECT-2: Prevent the unauthorized construction of sensitive classes

Where an existing API exposes a security-sensitive constructor, limit the ability to create instances.

# Object Construction

## 7-3: Defend against partially initialized instances of nonfinal classes

- A constructor exception doesn't always destroy the object.
  - Attackers override finalize method to get partially initialized ClassLoader instance.

```
public class ClassLoader {  
    public ClassLoader() {  
        SecurityCheck();  
        init();  
    }  
}
```

- Treat public statics primarily as constants-
  - Consider using enums(type-safe and implicitly static final).



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Sensitive static code can be modified by untrusted code.

Static variables are global across a Java runtime environment.

Treat public statics primarily as constants-

Consider using enums(type-safe and implicitly static final).

# Object Construction

## 7-3: Defend against partially initialized instances of nonfinal classes

- Declare class as final if possible.
  - If **finalize** method can be overridden, ensure partially uninitialized instances are usable.
- Don't set fields until all checks have completed.
  - Use an **initialized** flag.

```
public class ClassLoader {  
    private boolean initialized=false;  
  
    public ClassLoader() {  
        SecurityCheck();  
        init();  
        initialized=true; //check flag in all relevant methods  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Sensitive static code can be modified by untrusted code.

Static variables are global across a Java runtime environment.

Treat public statics primarily as constants-

Consider using enums(type-safe and implicitly static final).

## Guideline 7-4 / OBJECT-4: Prevent constructors from calling methods that can be overridden

Constructors that call overridable methods give attackers a reference to this (the object being constructed) before the object has been fully initialized.

## Guideline 7-5 / OBJECT-5: Defend against cloning of nonfinal classes

A nonfinal class may be subclassed by a class that also implements `java.lang.Cloneable`. The result is that the base class can be unexpectedly cloned, although only for instances created by an adversary. The clone will be a shallow copy. The twins will share referenced objects but have different fields and separate intrinsic locks.

# Serialization and Deserialization

## 8-Serialization and Deserialization

- |     |   |
|-----|---|
| 8-1 | Avoid serialization for security-sensitive classes  |
| 8-2 | Guard sensitive data during serialization   |
| 8-3 | View deserialization the same as object construction  |
| 8-4 | Duplicate the SecurityManager checks enforced in a class during serialization and deserialization |
| 8-5 | Filter untrusted serial data  |

### Why should I care?

- Careless deserialization from untrusted sources allows attackers to create unwanted instances of critical classes.
- Expect side effects with serialization.
- Wherever possible use XML/DTD.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Serialization and Deserialization

## 8-3: View deserialization the same as object construction

**Antipattern:** Believing deserialization is unrelated to constructors

Deserialization creates a new instance of a class without invoking any constructor on that class. Therefore, deserialization should be designed to behave like normal construction.

```
public final class ByteString implements java.io.Serializable {  
  
    private static final long serialVersionUID = 1L;  
    private byte[] data;  
    public ByteString(byte[] data) {  
        this.data = data.clone(); // Make copy before assignment.  
    }  
  
    private void readObject(java.io.ObjectInputStream in)  
        throws java.io.IOException, ClassNotFoundException  
{  
    java.io.ObjectInputStreadm.GetField fields = in.readFields();  
    this.data = ((byte[])fields.get("data")).clone();  
}  
    ...  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- Default deserialization and `ObjectInputStream.defaultReadObject` can assign arbitrary objects to nontransient fields and does not necessarily return.
- Use `ObjectInputStream.readFields` instead to insert copying before assignment to fields.
- Or, if possible, don't make sensitive classes serializable

## Summary

In this lesson, you should have learned how to:

- Describe the Java SE security overview
- Explain vulnerabilities
- Describe the secure coding guidelines and antipatterns



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Resources

First follow the guidelines from the source:

[Secure Coding Guidelines for Java SE](#)

Additionally:

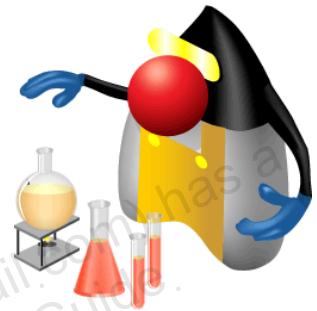
[securecoding.cert.org](#)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Practice 18: Overview

- There are no practices for this lesson.



## Quiz



You can reduce the risk by running vulnerable native code by doing the following:

- A. Defining wrappers around native methods
- B. Using strict input validation
- C. Using a stricter Java security policy



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Quiz



Deserialization of untrusted data should be avoided whenever possible because:

- A. It is unrelated to constructors
- B. It is the same as object construction
- C. It doesn't matter



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Quiz



Which phase should you plan against denial-of-service?

- A. Design
- B. Development
- C. Testing
- D. Deployment



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Quiz



Always declare public static fields as final.

- A. True
- B. False



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Building Database Applications with JDBC



ORACLE®



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adolfo De+la+Rosa (adolfo.delarosa2020@gmail.com) has a  
non-transferable license to use this Student Guide.

## 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



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## What Is the JDBC API?

- The JDBC API provides a standard database-independent interface to interact with any database.
- Typically, you use the JDBC API to connect to a database, query the data, and update the data.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## What Is JDBC Driver?

- The collection of the implementation classes that is supplied by a vendor to interact with a specific database is called a JDBC driver.
- There are different kinds of JDBC drivers that exist for different databases (or for the same database). They differ in the way they are implemented.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You can use three types of JDBC drivers in your Java programs to connect to a RDBMS:

- JDBC Native API Driver
- JDBC-Net Driver
- JDBC Driver:
  - Is also known as a direct-to-database pure Java driver. It is written in Java.
  - To include the driver JAR/ZIP files with your application.
  - All major RDBMS vendors supply this type of JDBC driver.

## Connecting to a Database

- Here are the steps that you need to follow to connect to a database.
  1. Obtain the JDBC driver and add it to the CLASSPATH environment variable on your machine.
  2. Register the JDBC driver with the DriverManager.
  3. Construct a connection URL.
  4. Use the `getConnection()` static method of `DriverManager` to establish a connection.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Obtaining a JDBC Driver

- You need to have the JDBC driver for your database to connect to the database using JDBC.
  - You can get a JDBC driver from the vendor of your database.
  - Typically, a JDBC driver is bundled in one or more JAR/ZIP files.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Register the JDBC driver with the DriverManager

You can call the `registerDriver(java.sql.Driver driver)` static method of the `DriverManager` class with an object of a JDBC driver class to register the JDBC driver.

```
// Register the Oracle JDBC driver with DriverManager  
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```



Copyright © 2018, 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 -d jdbc.drivers=<fully qualified path to the driver> <class to run>
```

## Constructing a Connection URL

- A database connection is established using a connection URL.
  - The format of a connection URL:

```
<protocol>:<sub-protocol>:<data-source-details>
```

- There are three parts of a connection URL.
  - <protocol>: always set to jdbc.
  - <sub-protocol>: is vendor-specific.
  - <data-source-details>: is RDBMS specific that is used to locate the database

- For example:

- Connection URL that uses Oracle's thin JDBC driver to connect to Oracle database:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```

- Connection URL that uses Oracle's thin JDBC driver to connect to Apache DB

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Establishing a Connection

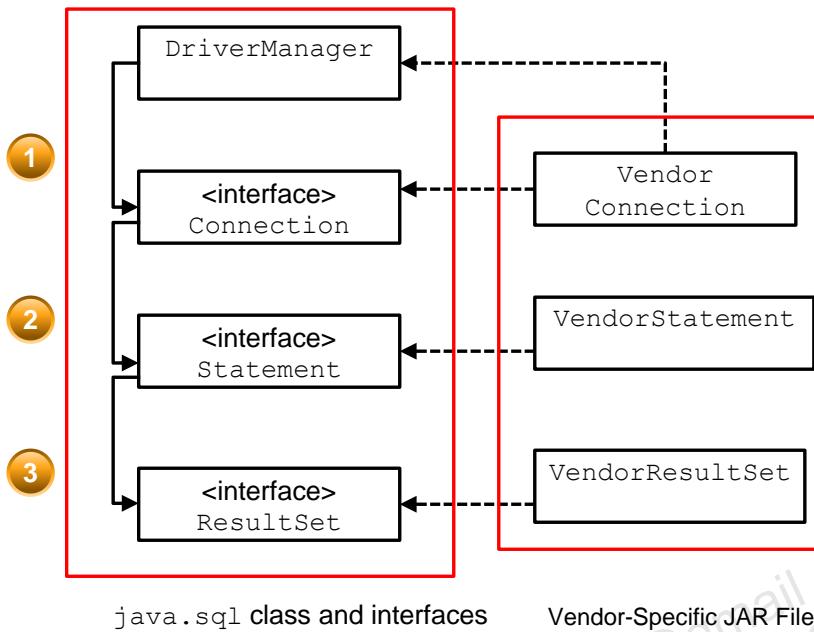
The `DriverManager` class is used to get an instance of a `Connection` object by using the JDBC driver named in the connection URL:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

# Using the JDBC API



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Package `java.sql`

Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language. It consists of 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.

# 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 © 2018, 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).

## 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
executeQuery(sqlString)	ResultSet	SELECT statement
executeUpdate(sqlString)	int (rows affected)	INSERT, UPDATE, DELETE, or a DDL
execute(sqlString)	boolean (true if there was a ResultSet)	Any SQL command or commands



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

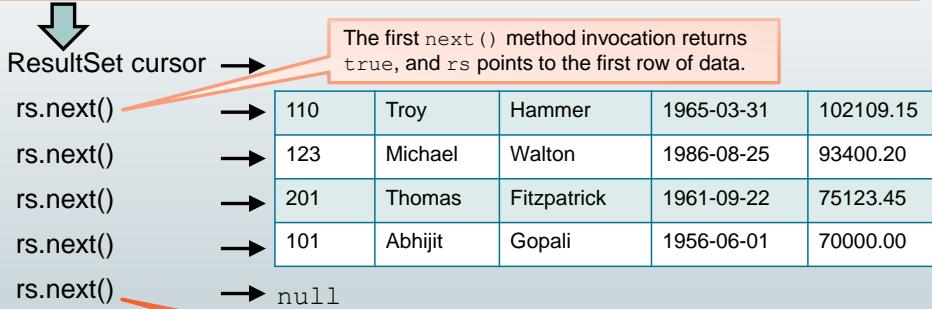
A SQL statement is executed against a database using an instance of a Statement object. The Statement object is a wrapper object for a query. A Statement object is obtained through a Connection object—the database connection. So it makes sense that from a Connection, you get an object that you can use 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 © 2018, 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 © 2018, 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 © 2018, 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 SimpleJDBCExample 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,  
9.                  password)) {  
10.             Statement stmt = con.createStatement();  
11.             String query = "INSERT INTO Employee VALUES (500, 'Jill',  
12.                     'Murray', '1950-09-21', 150000)";  
13.             if (stmt.executeUpdate(query) > 0) {  
14.                 System.out.println("A new Employee record is added");  
15.             }  
16.             String query1 = "select * from Employee";  
17.             ResultSet rs = stmt.executeQuery(query1);  
18.             //code to display the rows  
19.         }  
20.     }
```

Query to insert a row  
in the Employee.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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 © 2018, 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

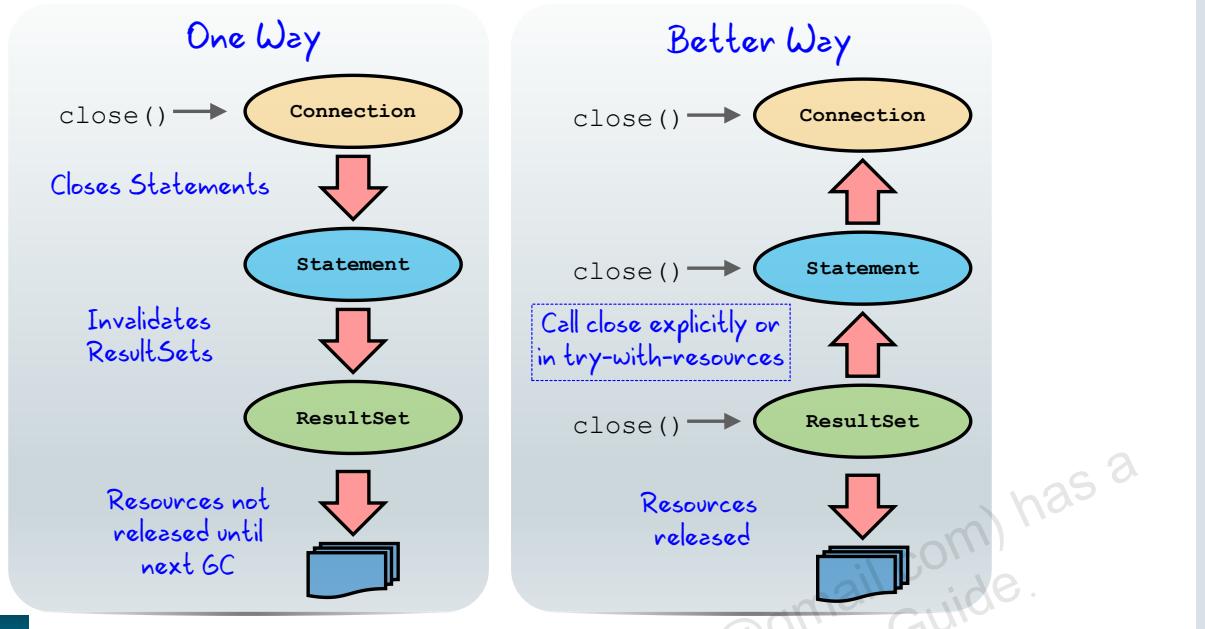


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- 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



Copyright © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## 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;           The IN parameter is passed in  
cStmt.setInt (1, age); to the stored procedure.  
  
ResultSet rs = cStmt.executeQuery();  
cStmt.registerOutParameter(2, Types.INTEGER);  
boolean result = cStmt.execute(); The OUT parameter is returned  
int count = cStmt.getInt(2); from the stored procedure.  
  
System.out.println("There are " + count +  
    " Employees over the age of " + age);
```

- Stored procedures are executed on the database.



Copyright © 2018, 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:

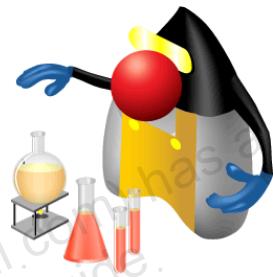
- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## Practice 19: Overview

- This practice covers working with the Derby Database and JDBC.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Localization



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

20



Adolfo De+la+Rosa (adolfodelarosa2020@gmail.com) has a  
non-transferable license to use this Student Guide.

## 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



Copyright © 2018, 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



Copyright © 2018, 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:



Copyright © 2018, 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.
- See Java 9 supported locales [here](#).



Copyright © 2018, 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.



Copyright © 2018, 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 © 2018, 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 -Dpropertynname=value -Dpropertynname=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 © 2018, 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.



Copyright © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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 © 2018, 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. Régler à l'anglais
2. Régler 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

# Displaying Currency

- Format currency:
  - Get a currency instance from `NumberFormat`.
  - Pass the `Double` to the `format` method.
- Sample currency output:

```
1 000 000 ₽ ru_RU  
1 000 000,00 € fr_FR  
¥1,000,000.00 zh_CN  
£1,000,000.00 en_GB
```



Copyright © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

# 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.

## 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 © 2018, 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

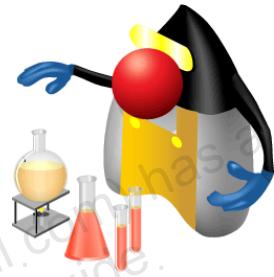


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



## Practice 20: Overview

This practice covers creating a localized date application.



Copyright © 2018, 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 © 2018, 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 © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Annotations



ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Objectives

After completing this lesson, you should be able to:

- Describe the purpose of annotations and typical usage patterns
- Apply annotations to classes and methods
- Describe commonly used annotations in the JDK
- Declare custom annotations



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Topics

- Common Annotations in the JDK
- Meta Annotations
- Custom Annotations
- Annotations from Frameworks



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Scenario

- You write a functional interface to enable a lambda expression:

```
public interface TestInterface {  
    int abstractMethod(int x);  
  
}
```

```
public class TestClass {  
    TestInterface lambda = (x) -> (2*x);  
  
}
```

- Will your colleagues understand this intention?
- What's to stop them from adding another method to the interface?

```
public interface TestInterface {  
    int abstractMethod(int x);  
    int abstractMethod2(int y);  
}
```

```
public class TestClass {  
    TestInterface lambda = (x) -> (2*x);  
}  
//This won't compile
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

A functional interface contains exactly one abstract method. The lambda expressions you write elsewhere are derived from a functional interface's one abstract method. Although an interface is still valid if it contains a second abstract method, it no longer works as a functional interface. A second abstract method ruins the functional interface and risks breaking code elsewhere. Your colleague may not realize this issue because the code breaks where they're not looking.

## @FunctionalInterface Annotation

Solution: Insert this annotation above the interface declaration.

- This enforces the definition of a functional interface.
- A colleague's deviation is immediately flagged.

```
@FunctionalInterface  
public interface TestInterface {  
    int abstractMethod(int x);  
}
```

```
@FunctionalInterface  
public interface TestInterface {  
    int abstractMethod(int x);  
    int abstractMethod2(int y);  
}  
//This won't compile
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Insert the `@FunctionalInterface` annotation before the interface's definition. This forces the interface to abide by the rules of being a functional interface, which has exactly one abstract method. If a colleague unknowingly deviates by adding a second abstract method, the interface won't compile. The issue is immediately flagged in the very file your colleague is editing. Your colleague should immediately notice and fix the issue.

This code is available in the `functionalInterface` package of the `AnnotationTesting` project.

## Annotation Characteristics

- Annotations start with the @ symbol.
- Annotations are a form of metadata.
  - They provide data about a program that's not part of the program itself.
  - They have no direct effect on the operation of the code they annotate.
- The metadata is useful at the source-code level, compile time, or run time.
- Annotations are applicable to declarations of a class, method, field, the use of types, and to other annotations.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## @Override Annotation

Scenario: A clumsy colleague forgets parameters when overriding methods.

- The method becomes overloaded, not overridden.
- Your program risks generating incorrect results.

Solution: Insert this annotation before the method declaration.

- The method signature must match an inherited method's.
- Otherwise, the code won't compile.

```
public class A {  
    void method(int x) {...}  
}
```

```
public class B extends A {  
    void method() {...}  
}
```

```
public class B extends A {  
    @Override  
    void method() {...}  
}  
//This won't compile
```

```
public class B extends A {  
    @Override  
    void method(int x) {...}  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

As method signatures grow complex, it's increasingly likely you may forget a parameter when attempting to override a method. The slide shows a simplified version of this scenario. Nevertheless, the effects of this mistake cause the method to be overloaded, not overridden. The program may generate incorrect results, as a method call may run the superclass's implementation rather than the subclass implementation you anticipated. You'll encounter similar issues if you misspell the method name.

The `@Override` annotation guards against this scenario. The compiler generates an error if the annotated method fails to correctly override an inherited method.

This code is available in the `override` package of the `AnnotationTesting` project.

## @Deprecated Annotation

Scenario: You realize that an old method in the library you maintain is unsafe. Its use must be discouraged.

- The old method is dangerous and inefficient.
- A better alternative exists.

Solution: Insert this annotation before the method declaration.

- Deprecated code is crossed out wherever it's mentioned in the IDE.
- The compiler generates a warning when deprecated code is used.

```
public class TestAPI {  
    public static void oldMethod() {...}  
}
```

```
public static void main(String[] args){  
    TestAPI.oldMethod();  
}
```

```
public class TestAPI {  
    @Deprecated  
    public static void oldMethod() {...}  
}
```

```
public static void main(String[] args){  
    TestAPI.oldMethod();  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## @Deprecated Annotation Recommendations and Options

Document the deprecation.

- Use Javadoc tag `@deprecated`.
- Note the reason for deprecation.
- Recommend an alternative.

Provide additional information.

- Note the version number when depreciation occurred.
- Note if the deprecated code will be entirely removed in future.

```
public class TestAPI {  
    /**  
     * @deprecated  
     * This method is deprecated because it's  
     * unsafe. Please use newMethod() instead.  
     */  
    @Deprecated(since="11", forRemoval=true)  
    public static void oldMethod() {...}  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

It's strongly recommended that you document the reason for deprecating a program element using the `@deprecated` Javadoc tag. Documentation should also suggest and link to any recommended replacement API. A replacement API often has subtly different semantics, which should also be noted. The use of the at sign (@) in both Javadoc comments and annotations is not coincidental: they are related conceptually. Also, note the Javadoc tag starts with a lowercase d and the annotation starts with an uppercase D.

`@Deprecated` has a String element `since`. Its value indicates the version when deprecation first occurred. In this case, `oldMethod` was deprecated in version 11. It's recommended you specify a `since` value for any newly deprecated program elements.

`@Deprecated` has a boolean element `forRemoval`. A `true` value indicates intent to remove the deprecated element in a future version. A `false` value indicates that although use of the deprecated element is discouraged, there's no intent to remove it yet.

## @SuppressWarnings Annotation

Scenario: You really want to use a deprecated API and suppress those pesky warnings.

- Warnings clutter and slow development.
- There's no better option.
- You can live with the consequences.

Solution: Insert this annotation before the method call.

- Apply it to the entire class or the specific calling method. More specificity is better.
- Warnings belong to categories.
- You may target multiple categories.

```
public static void main(String[] args){  
    TestAPI.oldMethod();  
}
```

```
@SuppressWarnings("deprecation")  
public static void main(String[] args){  
    TestAPI.oldMethod();  
}
```

```
@SuppressWarnings(  
    {"unchecked", "deprecation"})  
public class MainClass(){  
    public static void main(String[] args){  
        TestAPI.oldMethod();  
    }  
}
```

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## @SafeVarargs Annotation

Scenario: The pesky warning you want to suppress relates to varargs.

- You're positive nothing will go awry from heap pollution.
- You can live with the consequences.

Solution: Insert this annotation before the method declaration.

```
@SafeVarargs  
static void m(List<Integer> ... args) {  
    System.out.println(args.length);  
}
```



```
@SafeVarargs  
static void m2(List<Integer> ... args) {  
    Object[] objectArray = args;  
    objectArray[0] = List.of("Bug");  
    int broken = args[0].get(0);  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

The `@SafeVarargs` annotation is applicable to methods and constructors. It asserts your code doesn't perform potentially unsafe operations on its varargs parameter. These unsafe operations may result in heap pollution. Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. Because this is difficult for the compiler to verify, a warning is issued instead. Your code may be perfectly safe and free of heap pollution issues. If this is true, the compiler warnings may feel like annoying clutter. The `@SafeVarargs` annotation lets you suppress these unchecked warnings.

Be very sure of your assertion before using this annotation. The example on the right misuses `@SafeVarargs`. This code results in a `ClassCastException`. The reason has to do with how parameterized varargs are handled. When the compiler encounters a varargs method, it translates the varargs parameter into an array. In this example, the compiler translates the varargs parameter `List<Integer>... args` to an array parameter `List<Integer>[] args`. However, Java does not allow arrays with parameterized types. Parametrized information may be stripped away at compile time thanks to type erasure, resulting in an array of `Lists` with `Objects`. The example creates an `Object` array that points to the same location in memory as the `List<Integer>[] args`. This provides a means of assigning a `String` value to the array where an `Integer` should be.

This code is available in the `safeVarargs` package of the `AnnotationTesting` project.

# Topics

- Common Annotations in the JDK
- Meta Annotations
- Custom Annotations
- Annotations from Frameworks



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Examples from the Deprecated Annotation

- Meta annotations are applied to other annotations.
- `@Documented` specifies the annotation information appear in Javadoc-generated documentation.
- `@Retention` specifies where the annotation and its affects are retained.
- `@Target` specifies where the annotation is applicable.
- To supply more than one value, surround the list of values with curly braces.

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({CONSTRUCTOR, FIELD, LOCAL_VARIABLE,
         METHOD, PACKAGE, MODULE,
         PARAMETER, TYPE})
public @interface Deprecated {
    String since() default "";
    boolean forRemoval() default false;
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

`@Retention` accepts `RetentionPolicy` enum values.

- `RetentionPolicy.SOURCE` is retained in the source code, but discarded by the compiler.
- `RetentionPolicy.CLASS` is retained by the compiler, but ignored by the JVM.
- `RetentionPolicy.RUNTIME` is retained by the JVM and readable at run time.

In this example, `@Target` specifies where the `@Deprecated` annotation can be applied. `@Target` accepts Element Type enum values. There is an example in the `targetType` package of the AnnotationTesting project.

- `ElementType.ANNOTATION_TYPE` can be applied to an annotation type.
- `ElementType.CONSTRUCTOR` can be applied to a constructor.
- `ElementType.FIELD` can be applied to a field including enum constants.
- `ElementType.LOCAL_VARIABLE` can be applied to a local variable.
- `ElementType.METHOD` can be applied to a method-level annotation.
- `ElementType.MODULE` can be applied to a module.
- `ElementType.PACKAGE` can be applied to a package declaration.
- `ElementType.PARAMETER` can be applied to the parameters of a method.
- `ElementType.TYPE` can be applied to a class, interface, annotation, or enum declaration.
- `ElementType.TYPE_PARAMETER` can be applied to a type parameter declaration.
- `ElementType.TYPE_USE` can be applied to the use of a type.

## @Documented Meta Annotation Effects

The screenshot shows the JavaDoc interface for the NashornException class. At the top, there are tabs for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted), USE, TREE, DEPRECATED, INDEX, and HELP. To the right of the tabs, it says "Java SE 11 & JDK 11". Below the tabs, there's a search bar with a magnifying glass icon and a close button. Underneath the search bar, there are links for ALL CLASSES, SUMMARY: NESTED | FIELD | CONSTR | METHOD, and DETAIL: FIELD | CONSTR | METHOD. The main content area starts with "Module jdk.scripting.nashorn" and "Package jdk.nashorn.api.scripting". Below that, the "Class NashornException" is listed. A small box labeled "Class NashornException" is positioned to the right of the class name. Under the class name, there's a list of inheritance: java.lang.Object, java.lang.Throwable, java.lang.Exception, java.lang.RuntimeException, and jdk.nashorn.api.scripting.NashornException. Then, there's a section titled "All Implemented Interfaces:" which lists Serializable. Below this, there's a code snippet with a red box around the @Deprecated annotation:

```
@Deprecated(since="11",
           forRemoval=true)
public abstract class NashornException
extends RuntimeException
```

A blue arrow points from the text "Without @Documented applied to @Deprecated, this wouldn't appear automatically in the documentation." to the red box around the @Deprecated annotation.

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Topics

- Common Annotations in the JDK
- Meta Annotations
- **Custom Annotations**
- Annotations from Frameworks



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Scenario

- Company guidelines require you to cite authorship information at the beginning of each class.
- This has been traditionally done through comments.
- Colleagues make mistakes:
  - Forgotten categories
  - Misspelt categories
  - Commenting in different locations
  - Different data formats

```
// Author: Maureen DeLawn  
// Revision: 6  
// Revision Date: 4/15/2019  
// Reviewers: Ben Ng, Yu Wong
```

```
public class Gen1List{  
}
```

```
public class Gen2List extends Gen1List{  
    // Author: Yu Wong  
    // Version: Alpha  
    // Reverion Date: 4/15/2019
```

```
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Solution: Write a Custom Annotation

- Annotations enforce structure:
  - What data to include
  - Where it can be written
- Create a file for your annotation.
- Declare its type as `@interface`. Annotations are an interface variant.
- Declare any annotation elements.
  - Elements store values.
  - An element declaration looks similar to a method declaration.
  - You may set default values.
  - You may declare an element as an array to contain many values.

`ClassPreamble.java`

```
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface ClassPreamble {
    String author();
    int revision() default 0;
    String revisionDate() default "N/A";
    String[] reviewers();
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

You can have your IDE create a Java annotations file like you would stub any class or interface. The type is declared as `@interface`. In fact, annotations are an interface form. The symbol `@` is a play on words. Pronounced "AT", it's an acronym for "Annotation Type".

The `@Retention` and `@Target` meta annotations are optional. `@Target` ensures the `ClassPreamble` annotation is only applicable to classes, interfaces, and enums.

There are four annotation elements declared in this example: `author`, `revision`, `revisionDate`, and `reviewers`. Element declaration looks similar to method declaration. You can declare elements to contain different types of data, including `Strings`, `ints`, and arrays. You may also set default values for elements.

This code is available in the `customAnnotation` package of the `AnnotationTesting` project.

## Applying a Custom Annotation

- Apply your annotation using the @ symbol.
- List any element-value pairs between parentheses, separated by commas.
- If not specified, elements take on their default value.
- You must set a value for all elements that lack a default value.
- Values must match their declared type:
  - Good: revision = 6
  - Bad: revision = "Alpha"
- Array values require curly braces.

```
@ClassPreamble(  
    author = "Maureen DeLawn",  
    revision = 6,  
    revisionDate = "4/15/2019",  
    reviewers = {"Ben Ng", "Yu Wong"}  
)  
public class Gen1List {}
```

```
@ClassPreamble(  
    author = "Yu Wong",  
    reviewers = {"Ben Ng", "Robin Peck"}  
)  
public class Gen2List extends Gen1List{}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

@ClassPreamble can now be written at the top of classes. This is followed by a set of parentheses where you specify the values of each annotation element. List any element-value pairs, separated by commas. Make sure your values match the expected type. An array annotation element may have more than one value. Use curly braces to supply multiple values.

The second example shows how you do not need to specify an element's value if the annotation provides a default value. You must specify element values that have no default value. The program won't compile otherwise.

## Reading Annotation Elements Through Reflection

```
public static void main(String[] args) {  
    System.out.println(  
        Gen1List.class.getAnnotation(ClassPreamble.class) +"\n\n"  
        +Gen2List.class.getAnnotation(ClassPreamble.class) +"\n\n"  
        +Gen2List.class.getAnnotation(ClassPreamble.class).author());  
}
```

```
@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,  
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})  
  
@customAnnotation.ClassPreamble(revisionDate="N/A", revision=0, author="Yu Wong",  
reviewers={"Ben Ng", "Robin Peck"})  
  
Yu Wong
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Other Details

- You may omit an element's name when assigning a value if it's named `value`.

```
public @interface ClassPreamble {  
    String value();  
}
```

```
@ClassPreamble(value = "Ben Ng") ✓
```

```
@ClassPreamble("Ben Ng") ✓
```

- You may omit curly braces if an array contains only one value.

```
public @interface ClassPreamble {  
    String[] value();  
}
```

```
@ClassPreamble({"Ben Ng"}) ✓
```

```
@ClassPreamble("Ben Ng") ✓
```

- Beware of null values.

```
public @interface ClassPreamble {  
    String value() default null;  
}
```

```
@ClassPreamble(null) ✗
```

- Although annotations are an interface form, they cannot extend other interfaces.

```
public @interface ClassPreamble extends List {} ✗
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

An element named `value` gets special treatment. If you name an element `value`, you can omit its name later when you apply the annotation and assign its value.

When assigning a value to an array element, you may omit the curly braces if the array contains only one value.

Beware of null values. These examples won't compile.

Although annotations are an interface form, they cannot extend other interfaces.

## Inheriting an Annotation

**Scenario:** Management believes that because superclass edits impact subclasses, authorship data from superclasses should also be reflected on subclasses.

**Solution:** Add the `@Inherited` meta annotation.

- A subclass without an annotation gains the inherited annotations from a superclass.
- A subclass cannot inherit an annotation from an interface.

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface ClassPreamble {
    String author();
    int revision() default 0;
    String revisionDate() default "N/A";
    String[] reviewers();
}
```

```
@ClassPreamble(
    author = "Maureen DeLawn",
    revision = 6,
    revisionDate = "4/15/2019",
    reviewers = {"Ben Ng", "Yu Wong"})
public class Gen1List {}
```

```
public class Gen2List extends Gen1List{}
```

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



The meta annotation `@Inherited` is added to the annotation type `ClassPreamble`. The `Gen1List` class is annotated with `@ClassPreamble`. `Gen2List` extends `Gen1List`, but contains no annotation. Because `@ClassPreamble` is now an inherited annotation, `Gen2List` behaves as if it were annotated with the same `@ClassPreamble` found in the superclass. We'll examine the impacts of this on the next slide.

## Reading an Inheriting an Annotation Through Reflection

```
public static void main(String[] args) {  
    System.out.println(  
        Gen1List.class.getAnnotation(ClassPreamble.class) +"\n\n"  
        +Gen2List.class.getAnnotation(ClassPreamble.class) +"\n\n"  
        +Gen2List.class.getAnnotation(ClassPreamble.class).author());  
}
```

```
@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,  
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})  
  
@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,  
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})  
  
Maureen DeLawn
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

When you query the annotation type for a class that lacks the annotation, the class' superclass is then queried for the annotation type. The `Gen2List` class lacks `@ClassPreamble`. When that annotation type is queried through reflection, the information is gathered by pulling from the superclass' `@ClassPreamble`.

## Repeating an Annotation

Scenario: Management wants authorship data available for each edit.

Solution: Store repeated annotations in a single compiler-generated container.

- Add the `@Repeatable` meta annotation.
  - In parentheses, specify a container.
- Declare the container annotation type.
  - Add an element that's an array of the repeated annotation type.
  - Name this element `value`.

```
@Repeatable(PreamblesContainer.class)
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface ClassPreamble {
    String author();
    int revision() default 0;
    String revisionDate() default "N/A";
    String[] reviewers();
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface PreamblesContainer {
    ClassPreamble[] value();
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Repeating annotations are stored in a container annotation that's automatically generated by the Java compiler. For the compiler to do this, two declarations are required in your code.

First, mark your annotation with the meta annotation `@Repeatable`. In parentheses is the container annotation type that the Java compiler generates to store repeating annotations. In this example, the container is `PreamblesContainer`.

Second, declare the container annotation type. It must have an element that's an array type of your repeatable annotation. In this example, `ClassPreamble[]`. Name this element `value`.

## Repeating Annotation Example

```
@ClassPreamble(  
    author = "Robin Peck",  
    revision = 7,  
    revisionDate = "5/15/2019",  
    reviewers = {"Ben Ng", "Yu Wong"}  
)  
@ClassPreamble(  
    author = "Maureen DeLawn",  
    revision = 6,  
    revisionDate = "4/15/2019",  
    reviewers = {"Ben Ng", "Yu Wong"}  
)  
public class Gen1List {}
```

```
@Repeatable(PreamblesContainer.class)  
@Retention(RetentionPolicy.RUNTIME)  
@Target(TYPE)  
public @interface ClassPreamble {  
    String author();  
    int revision() default 0;  
    String revisionDate() default "N/A";  
    String[] reviewers();  
}
```

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(TYPE)  
public @interface PreamblesContainer {  
    ClassPreamble[] value();  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

`@ClassPreamble` can be repeated as many times as you need on the same class.

# Reading a Repeatable Annotation Through Reflection

```
public static void main(String[] args) {
    System.out.println(
        Gen1List.class.getAnnotation(PreamblesContainer.class).value()[0]      +"\n\n"
        +Gen1List.class.getAnnotation(PreamblesContainer.class).value()[1]      +"\n\n"
        +Gen1List.class.getAnnotationsByType(ClassPreamble.class)[1]           +"\n\n"
        +Gen1List.class.getAnnotationsByType(ClassPreamble.class)[1].author());
}

@customAnnotation.ClassPreamble(revisionDate="5/15/2019", revision=7, author="Robin
Peck", reviewers={"Ben Ng", "Yu Wong"})

@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})

@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})

Maureen DeLawn
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Topics

- Common Annotations in the JDK
- Meta Annotations
- Custom Annotations
- Annotations from Frameworks



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Frameworks

- There are many Java frameworks.
- Annotations may act as the interface between Java code and Java frameworks.
- You'll often use annotations to leverage framework features.

Popular Java frameworks include:

- Spring Boot
- Spring MVC
- JUnit
- Struts
- JavaServer Faces (JSF)
- Dropwizard



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## @NonNull Type Annotation and Pluggable Type Systems

- It's a non-standardized annotation.
- It's found through various frameworks and tools.
- It's a type annotation, which are applicable anywhere you'd use or declare a type.
- Use it in conjunction with defect detection tools to warn or protect against possible null values.

```
public class TestClass {  
    @NonNull String str;  
    BiConsumer lambda = (@NonNull var x, final var y) -> x.process(y);  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Summary

In this lesson, you should have learned how to:

- Describe the purpose of annotations and typical usage patterns
- Apply annotations to classes and methods
- Describe commonly used annotations in the JDK
- Declare custom annotations



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfoldelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Security Survey



ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



B

Adolfo De+la+Rosa (adolfo.delarosa.2012@gmail.com) has a  
non-transferable license to use this content.

## Objectives

After completing this lesson, you should be able to:

- Identify potential Denial of Service vulnerabilities
- Secure confidential information
- Support data integrity
- Explain reasons for input validation
- Limit accessibility and extensibility of sensitive objects
- Consider security measures for serialization



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## About This Lesson

- This lesson offers a sampling to get you thinking about security.
- There's much more to learn before becoming a security expert.
- In general, it's good to be paranoid and think critically of your code.
- Consider all the angles.
- Paranoia keeps you on guard against even the slightest vulnerabilities.
  - Assume bad actors are always out to get you.
  - Assume developers and users will make mistakes.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



# Denial of Service (DoS) Attack

## Symptoms

- Legitimate users are unable to access resources and services.
- There is excessive resource consumption. No resources remain to do legitimate work.

## Causes

- A file or code construct grows too large.
- A service or connection is overwhelmed with bogus requests.

## Prevention

- Use permissions to restrict access to code that consumes vulnerable resources.
- Validate all inputs into a system.
- Release resources in all cases.
- Monitor excessive resource consumption disproportionate to that used to request a service.

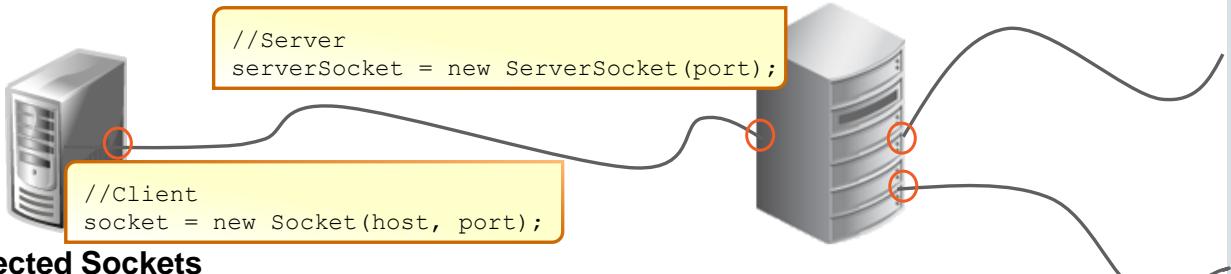


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

# Sockets

## About Sockets

- Sockets are the endpoints in a connected network.
- Ensure these endpoints aren't vulnerable to malicious connections.



## Unprotected Sockets

- Unprotected sockets may be overwhelmed with bogus or malicious connection requests.
- Restrict connections and privileged access.
- Validate server addresses and check permissions before creating new sockets.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Assume variables `socket` and `serverSocket` are declared elsewhere.

## Other DoS Examples

### Zip Bomb

- An innocuously sized file requires an outrageous amount of space after decompression.
- Limit the decompressed data size. Don't rely on the file's compressed size or metadata.

### Billion Laughs

- XML entity expansion causes an XML document to grow dramatically during parsing.
- Set `XMLConstants.FEATURE_SECURE_PROCESSING` to enforce reasonable limits.

### Unsustainable Growth

- Ensure that constructs like collections and buffers can't grow to unmanageable sizes.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Zip bombs are perpetrated by short file that are very highly compressed. For instance, ZIPs, GIFs and gzip encoded http contents. When decompressing files, it's better to limit the decompressed data size rather than relying on the compressed size or metadata.

Billion laughs attacks occur when XML entity expansion causes an XML document to grow dramatically during parsing. This is explained more later in this lesson. To prevent this type of attack, set the `XMLConstants.FEATURE_SECURE_PROCESSING` feature to enforce reasonable limits.

Collections are sized dynamically. Ensure that a collection cannot grow so big that it unreasonably hogs all available memory.

## Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



# Enemies Target Confidential Information

## About Confidential Information

- This includes users' personal data, security credentials, or information revealing how your system functions.
- Enemies exploit this information for gain or chaos, harming your users and your system.
- If information is exposed or tampered with, your system is no longer trustable.

## Prevention

- Clearly identify and encapsulate confidential information.
- Be as restrictive as possible with permissions.
- Limit scope and access to objects holding confidential information.
- Store trusted information in unmodifiable or immutable objects.
- Do not reveal confidential information to untrusted code, libraries, exceptions, or logs.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Purge Sensitive Information from Exceptions

- Sanitize exceptions so confidential information is not reported.
- This example output dangerously clues enemies to the location of your application's configuration file.

```
Cannot find main configuration file:
```

```
C:\config\badfile.properties (The system cannot find the path specified)
java.io.FileNotFoundException: C:\config\badfile.properties
(The system cannot find the path specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java:138)
at java.io.FileInputStream.<init>(FileInputStream.java:97)
at com.example.sec.FileException.readProps(FileException.java:20)
at com.example.sec.FileException.main(FileException.java:12)
```

```
Java Result: -1
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Internal exceptions should be caught and sanitized before propagating them to upstream callers. The type of an exception may reveal sensitive information, even if the message has been removed. For instance, exceptions related to file access like `FileNotFoundException` may reveal whether a file exists. An attacker can learn about your directory system by providing various file names as input and analyzing the resulting exceptions.

## Do Not Log Highly Sensitive Information

- This example log exposes a user's ID, address, and credit card number.
- Your enemies may use this information to commit fraud and identity theft.

```
Feb 08, 2020 10:55:14 AM com.example.sec.DetailedLogger logMessages
SEVERE: ID 123-45-6778 for User John Adams does not match.
Feb 08, 2020 10:55:14 AM com.example.sec.DetailedLogger logMessages
SEVERE: User John Adams located at 123 Drury Lane, Quincy, MA
Feb 08, 2020 10:55:14 AM com.example.sec.DetailedLogger logMessages
SEVERE: Cannot find exp date for credit card 1111-1111-1111-1111
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

This example log reveals a user's ID, address, and credit card number, which are highly sensitive. This type of information should not be kept for longer than necessary nor where it may be seen, even by administrators. It should not be sent to log files, and its presence should not be detectable through searches.

## Topics

- Denial of Service
- Confidential Information
- **Integrity of Inputs**
- Developing Secure Objects
- Secure Serialization and Deserialization



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Validate Inputs

- Expect all kinds of crazy inputs:
  - Casual typos
  - Misunderstandings in formatting
  - Special characters, which could be disguised commands
  - Devious inputs that reveal information about the system
  - Malicious inputs to hack and exploit security holes
- Programs must anticipate this unpredictability and elegantly recover from any issues.
  - Never trust inputs from untrusted sources.
  - Use well-tested, specialized libraries and Java APIs.
  - Avoid ad hoc code for validation.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Numbers That Make Programs Go Awry

- Be careful checking resource limits and dealing with numeric inputs.
- There are APIs to guard against overflowed number space:

```
final int MAX = Integer.MAX_VALUE;
System.out.println(MAX);                                // 2147483647
System.out.println(MAX +1);                            // -2147483648
System.out.println(Math.addExact(MAX, 1));           // java.lang.ArithmaticException
                                                       // Also try multiplyExact
                                                       // and decrementExact
```

- There are APIs to guard against bad floating point values:

```
if (Double.isInfinite(untrusted_double)) {
    // guards against 1/Double.MIN_VALUE
    // and 1/-Double.MIN_VALUE
}
```

```
if (Double.isNaN(untrusted_double)){
    // guards against division by 0.0
    // and infinities minus infinities
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Directory Traversal Attacks with . . /

- . . / requests the parent directory
- Your enemies may include . . / in paths.
  - They'll gradually learn your directory structure.
  - They want a path to sensitive information.
- Absolute paths may contain . . /.
- Canonicalize your path names before validation.
- Canonical paths are absolute, unique, and stripped of . . /
- Consider these methods:
  - String getCanonicalPath() //java.io.File
  - File getCanonicalFile() //java.io.File
  - Path toRealPath(LinkOption... options) //java.nio.file
  - Path normalize() //java.nio.file

```
File f = new File("../project");
String s = f.getCanonicalPath();
// "C:\\Windows\\project"
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Both `getCanonicalPath` and `getCanonicalFile` are part of the original Java io API. `getCanonicalPath` returns a canonical path as a `String`. `getCanonicalFile` is similar but returns a `File` object instead of a `String`. These methods are called on `File` objects. The example calling `getCanonicalPath` shows how a file path containing . . / can be saved as a `String`. The resulting canonical path is absolute with . . / stripped out.

Both `toRealPath` and `normalize` are part of Java's NIO2 file API. `toRealPath` returns a `Path` for a file that must exist. An array of `LinkOptions` may be supplied that indicates how to handle symbolic links like " . . ". `normalize` also returns a `Path`, but the file doesn't necessarily have to exist. These methods are called on instances that implement the `Path` interface. The actual behaviors of these methods may vary depending on how the interface is implemented.

## SQL Injection Through Dynamic SQL

- Like Java Strings, SQL statements can be assembled dynamically by concatenating a mix of hard-coded text, variables, and user input values.
- Matching single quotes ('') mark the end points of text.

```
String query = "SELECT * FROM Employees WHERE LastName ='" +userInput +"';
```

- Using another '' , an input field is exploited to inject malicious code into the statement.
- Assume the user enters this in an input dialog for the userInput String:

```
'; DROP DATABASE criticalDataBase; //
```

- The resulting String query is set to run two queries.
  - A meaningless query to find an employee with a blank last name.
  - A malicious query that deletes a critical database!



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Safer SQL

- PreparedStatements and substituted parameters shown in the JDBC lesson are a better approach.
- They ensure that parameter values are never interpreted as SQL.

```
String query = "SELECT * FROM Employee WHERE LastName = ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setString(1, userInput);
```

- In cases where PreparedStatements do not work, use validation methods provided by JDBC. For example, Statement.enquoteLiteral.

```
StubStatement stmt = new StubStatement();
String lastName = stmt.enquoteLiteral("O'Kelly");
// lastName is: 'O''Kelly'
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

In the first example, the query is still stored as a String. Each ? in query indicates a parameter to substitute. A PreparedStatement pStmt is created based on the query. The Connection con is defined elsewhere. PreparedStatement setXXX methods set values for each ? and help validate input. The first argument indicates which ? to set. The second argument indicates the replacement value.

The second example creates an instance of StubStatement, which is an example class that implements the Statement interface. The default method enquoteLiteral is part of the Statement interface. It returns a String enclosed in single quotes. Any occurrence of a single quote within the String is replaced by two single quotes. In this case, calling the method on the String argument "O'Kelly" returns 'O''Kelly'.

## XML Inclusion

- Many types of attacks exploit XML Document Type Definitions (DTDs), resulting in various degrees of harm:
  - Denial of Service
  - Exposed private files and data
- What's the purpose of a DTD?
  - Outlining how XML elements should be structure
  - Defining substitutable values known as entities

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE employee[
    <!ELEMENT employee (name, company)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT company (#PCDATA)>
    <!ENTITY sun "Sun Microsystems">
]>

<employee>
    <name>Kenny O'Kelly</name>
    <company>&sun;</company>
</employee>
```

Name: Kenny O'Kelly  
Company: Sun Microsystems

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



At the top of this XML example file, DOCTYPE is the DTD that outlines the XML for representing an employee. Elements are used to outline an employee's name and company. An entity is used to define a substitutable value of "Sun Microsystems". When the actual employee is created later in the file, &sun; is used to substitute in the value of the sun entity. When this is read into Java code, the resulting name is Kenny O'Kelly and company is Sun Microsystems.

This code is available in the `xml` package of the `SecurityTesting` project.

## The Problem with XML Entities

- Entities may also be defined externally in other files and URLs.
- This leaves XML susceptible to XML External Entity (XXE) attacks.

```
... <!ENTITY sun SYSTEM "http://[REDACTED].com/malicious.dtd">  
...
```

- A Billion Laughs DoS occurs when `malicious.dtd` recursively require more and more entities.
  - Private data may be exposed through any element referencing `&sun;` if `malicious.dtd` points to a sensitive file.
- If you must use XML, guard against attacks by:
    - Setting resource limits
    - Validating inputs
    - Reducing privileges by using the most restrictive XML parser configurations
    - Disabling external entities and DTDs all together

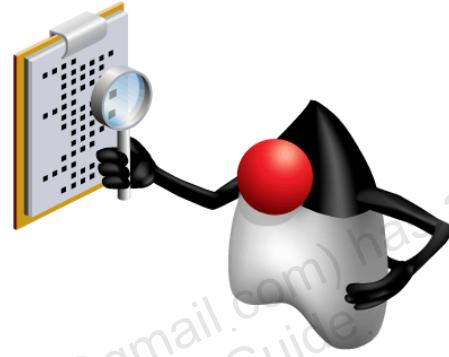


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

XXE attacks are similar to SQL injection in the sense that malicious code from the outside world is introduced into your application. The previous example has been changed. The value for `sun` is no longer defined internally. Instead, it's defined externally in a dtd file. This could potentially be from an untrusted or malicious source.

## Failure to Verify Bytecode

- Verify bytecode against tampering and dangerous behavior.
- Many sources may generate or modify bytecode you need. Don't trust these sources.
- Beware of the command line arguments `-Xverify:none` or `-noverify`, which disable bytecode verification.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- **Developing Secure Objects**
- Secure Serialization and Deserialization

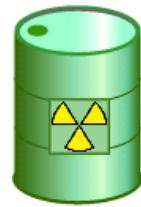


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Isolate Unrelated Code

- Isolate and contain less-trustworthy code.
- Don't allow interference between unrelated code.
- Encapsulate with the most restrictive permissions possible.
- `public static` fields should be `final`.
  - Mutable statics are a vulnerability for code that directly or indirectly depends on their values.



```
public class ClassA {  
    public static int year = 1974;  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Encapsulate with the most restrictive permissions possible. Traditionally, classes and interfaces may be `public` if they're part of a published API. Otherwise, declare them package-private. Class members of an API may similarly be `public` or `protected` as appropriate. Otherwise, declare them `private` or package-private.

## Stronger Encapsulation with Modules

- Restrict access further through modules and packages.
- In addition to private, package, and protected, there are 3 types of public.
- The `modules-info` files determine who may instantiate this example public class:

```
package pkgA;  
  
public class ClassA {...}
```

1. public to everyone.

```
module A{  
    exports pkgA;  
}
```

```
module B{  
    requires A  
}
```

```
module C{  
    requires A  
}
```

2. public to specific modules.

```
module A{  
    exports pkgA to B;  
}
```

```
module B{  
    requires A  
}
```

```
module C{  
    requires A  
}
```

3. public only in its own module.

```
module A{  
    //exports nothing  
}
```

```
module B{  
    requires A  
}
```

```
module C{  
    requires A  
}
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Use modules and packages as another means of restricting access as much as possible.

In this example, `ClassA` exists in `pkgA` package within module `A`. The class is public, but it's not necessarily accessible to other classes and modules. The same thing is true about the class's public fields and methods. It all depends on declarations made in `module-info` files. In scenario 1, the class is public to everyone because its package is exported from module `A`. Modules `B` and `C` require module `A`, meaning classes inside these modules may instantiate `ClassA`. In scenario 2, the class is public only to specific modules. In this case module `B`. Classes within Module `B` may instantiate `ClassA`, but classes within Module `C` may not. Scenario 3 has the most restrictive form of public access. `ClassA` is public only within Module `A`. Classes within Modules `B` and `C` may not instantiate `ClassA`.

This code is available in the `ModulesTest` project.

## Stronger Encapsulation Against Reflection

With modules, private data is no longer accessible via reflection.

```
module A{  
    exports pkgA;  
}  
  
module B{  
    requires A  
}  
  
package pkgB; //in Module B  
public class NewMain {  
  
    public static void main(String[] args) {  
        ClassA test = new ClassA();  
        try {  
            Field secret = Class.forName("pkgA.ClassA").getDeclaredField("secretCrush");  
            secret.setAccessible(true);  
            System.out.println(secret.get(test));  
        }...  
    }  
}  
  
package pkgA; //in Module A  
public class ClassA {  
    public static final int YEAR = 1974;  
    private String secretCrush = "Kenny O'Kelly";  
}
```

java.lang.reflect.InaccessibleObjectException: Unable to make field private java.lang.String pkgA.ClassA.secretCrush accessible: module A does not "opens pkgA" to module B



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Limit Extensibility

Non-final and non-private classes and methods can be maliciously overridden by an attacker.

- Hide methods as necessary.
- Design classes and methods for inheritance, or declare them final or private.
- Instantiate with a constructor after the potential object is verified safe.
- Don't call overridable methods from constructors.

```
// Unsubclassable class
public final class SensitiveClass {
    private final Behavior behavior;

    // Hidden constructor
    private SensitiveClass(Behavior behavior) {
        this.behavior = behavior;
    }

    // Guarded construction
    public static SensitiveClass newSensitiveClass
        (Behavior behavior) {
        // ... validate any arguments ...
        // ... perform security checks ...
        return new SensitiveClass(behavior);
    }
}
```



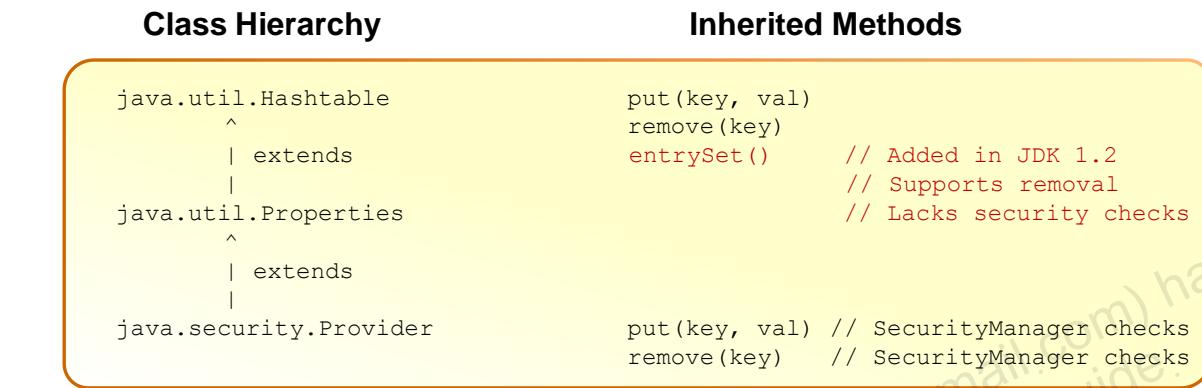
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

This code shows an example of a secure, un subclassable class. The class is final, and therefore un subclassable. Its fields and constructor are private. The only way to create an instance of this object type is to first call the static factory method newSensitiveClass. This method first validates any arguments and performs security checks before calling the constructor to actually instantiate the object. A safe and secure instance of a SensitiveClass is returned. Overridable methods shouldn't appear within this method or the constructor. Such methods may be maliciously overridden by an attacker to circumvent security checks, expose sensitive data, or perform other compromising actions.

This approach also guards against partially initialized objects, which are vulnerable to attack. When a constructor in a non-final class throws an exception, attackers can attempt to gain access to partially initialized instances. You should ensure a non-final class remains totally unusable until its constructor completes successfully. By guarding the constructor like in this example, we ensure only safe and secure instances are sent for construction.

# Beware of Superclass Changes

- Superclass changes affect subclass behavior by:
  - Changing the implementation of an inherited method that's not overridden
  - Introducing new methods
- Changes may break subclass assumptions and lead to security vulnerabilities.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

In this hierarchy, the `Provider` subclass inherits certain methods from `Hashtable`, including `put` and `remove`. `Provider.put` maps a cryptographic algorithm name, like RSA, to a class that implements that algorithm. To prevent malicious code from affecting its internal mappings, `Provider` overrides `put` and `remove` to enforce `SecurityManager` checks.

The `Hashtable` class was enhanced in JDK 1.2 to include a new method, `entrySet`, which supports the removal of entries from the `Hashtable`. The `Provider` class was not updated to override this new method. This oversight allowed an attacker to bypass the `SecurityManager` check enforced in `Provider.remove`, and to delete `Provider` mappings by simply invoking the `Hashtable.entrySet` method.

## Problem: Vulnerable Object Fields

- Encapsulation may not be enough to protect fields from malicious manipulation.
- Keys to exploitation are:
  - Objects provided to a constructor
  - Objects returned from a getter method

```
public class Example {  
    private final Date date;  
    public Example(Date date) {  
        this.date = date;  
    }  
    public Date getDate() {  
        return date;  
    }  
}
```

```
...  
Date badDate = new Date();  
Example obj = new Example(badDate);  
  
badDate.setYear(666);  
obj.getDate().setMonth(6);  
System.out.println(obj.getDate());
```

```
Tue Jul 01 18:48:08 EDT 2566
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Date is an object field in the Example class. The field is set with a constructor. Although the field is private and has a traditional getter method, this isn't enough to prevent unwanted manipulation, which effectively negates the security benefits of encapsulation.

The code on the right creates a Date object badDate and supplies it to the constructor. Both this reference, and the Example class's date field point to the same location in memory. This means the badDate reference has access to freely manipulate a private field in another class! The getter method provides the same dangerous level of access because it returns the date object. You can see from the output how both means are effective.

This code is available in the date package of the SecurityTesting project.

## Solution: Create Copies of Mutable or Subclassable Input Values

- Most methods like constructors and setters should first copy any objects they receive.
- Getters should return copies. There are a few ways to do this, including:
  - Calling a constructor
  - Calling the `clone` method, if the object type is trustable or `final`

```
public class Example {  
    private final Date date;  
    public Example(Date date) {  
        this.date = new Date(date.getTime());  
    }  
    public Date getDate() {  
        return (Date)date.clone();  
    }  
}
```

```
...  
Date badDate = new Date();  
Example obj = new Example(badDate);  
  
badDate.setYear(666);  
obj.getDate().setMonth(6);  
System.out.println(obj.getDate());
```

Mon Apr 01 18:51:08 EDT 2019



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## File Security Bug Reports

- Reports let others know what to beware of and plan accordingly until a fix is ready.
- Customers may not appreciate you keeping secrets when they're at risk.
- It's dangerous to mess with code you don't understand.
- Reports help the right people fix issues and anticipate/prevent similar issues in the future.



## Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Serialization and Deserialization

- Serialization converts an object's state into a byte stream.
  - Instance fields
  - Not static fields
  - Not methods
- Deserialization converts the byte stream into a copy of the object.
- Serialization is useful for storing state in a database or transferring state over a network.
- The class or its superclass must implement the `Serializable` interface.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Problem: Fields Are Accessible After Serialization

**Solution 1:** Hide sensitive fields from serialization by declaring them `transient`.

```
public class Employee implements Serializable {  
    private int salary;  
    private transient String fear;  
    ...  
}
```

salary: 80000  
fear: Spiders

10010101

salary: 80000  
fear: null

**Solution 2:** Declare a special array field `serialPersistentFields`, which specifies each field to serialize.

```
public class Employee implements Serializable {  
    private int salary;  
    private String fear;  
    private static final ObjectStreamField[]  
        serialPersistentFields = {  
            new ObjectStreamField("salary", int.class);  
    };  
    ...  
}
```

salary: 80000  
fear: Spiders

10010101

salary: 80000  
fear: null



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

You should avoid serializing security-sensitive classes, and carefully guard sensitive information within classes you do serialize. Java's access controls cannot be enforced after serialization. In effect, serialization provides a public interface to fields. An attacker may analyze the byte stream to learn sensitive data.

Guard against this by declaring sensitive fields `transient`. A `transient` field is not serialized. This is illustrated by the first example. An employee's deepest fear is considered sensitive information, and is marked `transient` to prevent serialization. When the example object is serialized and then deserialized, the resulting copy knows nothing of the original object's fears. The `fear` field is `null`.

Alternatively, you may declare a special array field `serialPersistentFields`, which specifies each field you want to serialize. Solution 2 produces the same results. This solution overrides any `transient` declarations. Even if the second example were edited to declare the `salary` field `transient`, the resulting object after deserialization still has the same `salary` as the original object.

These solutions are available in the `serialize` package of the `SecurityTesting` project.

## Solution 3a: Implement writeObject and readObject Methods

- `writeObject` specifies data serialized to the byte stream.
- `readObject` deserializes from the byte stream and specifies how copies are created.
- The order of statements within these methods must align.

```
public class Employee implements Serializable {  
    private int salary;  
    private String fear;  
  
    ...  
    private void writeObject(ObjectOutputStream out) throws IOException {  
        out.writeObject(this.salary);  
        out.writeObject("Fearless Copy");  
    }  
  
    private void readObject(ObjectInputStream in) throws  
        IOException, ClassNotFoundException {  
        this.salary = (int)in.readObject();  
        this.fear = (String)in.readObject()  
    }  
}
```

salary: 80000  
fear: Spiders

10010101

salary: 80000  
fear: Fearless Copy



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

The `writeObject` method specifies what data to serialize to the byte stream. You're not limited to serializing just class fields. You can serialize whatever data you want from this method. In this example, the Employee's fear still must be kept secret. Instead of adding the actual vulnerable value to the byte stream, a replacement String is serialized.

The `readObject` method deserializes the byte stream and specifies how copies are made. The state of the copies are derived from the byte stream in this example. But this isn't necessarily required. For example, instead of serializing "Fearless Copy" to the byte stream, we could achieve the same effect by letting the `readObject` method set `this.fear` to "Fearless Copy".

The order that data is serialized and deserialized matters greatly. If the order of statements in `readObject` were reversed in this example, a `ClassCastException` is thrown due to the mismatch in data types between `int` and `String`.

This solution also circumvents any transient declarations. Even if the `salary` field were declared transient, the resulting object after deserialization still has the same salary as the original object.

## Solution 3b: Implement writeObject with PutField, and readObject with GetField

- Use ObjectOutputStream.PutField to selectively serialize data.
- The order of statements is less important.
- This solution is brittle. Refactored fields may mismatch Strings in these methods.

```
public class Employee implements Serializable {
    private int salary;
    private String fear;
    ...
    private void writeObject(ObjectOutputStream out) throws IOException {
        ObjectOutputStream.PutField fields = out.putFields();
        fields.put("salary", this.salary);
        fields.put("fear", "Fearless Copy");
        out.writeFields();
    }
    private void readObject(ObjectInputStream in) throws
        IOException, ClassNotFoundException {
        ObjectInputStream.GetField fields = in.readFields();
        this.fear = (String) fields.get("fear", null);
        this.salary = fields.get("salary", 0);
    }
}
```

10010101

salary: 80000  
fear: Fearless Copy



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

The `writeObject` method still specifies what data to serialize to the byte stream. Now individual fields are specified by name, along with their intended values. An instance of `ObjectOutputStream.PutField` is created. Data is serialized by specifying key-value pairs with the instance's `put` method. Finally, `writeFields` is called on the output stream.

The `readObject` deserializes the byte stream to set the value of a copy's fields. An instance of `ObjectInputStream.GetField` is first created. The order of statements afterwards is less important, because data is retrieved based on key-value pairs from the `get` method. The `get` method also specifies a default value. This method returns primitives and `Objects`, which means an `Object` may need to be cast before it's set as a copy's field value.

Although the order that data is serialized and deserialized is less important when fields are handled by name, the solution is more-brittle. String names used in the `put` and `get` methods must match the class's field names. If a field is refactored, the corresponding String name might not be updated. This results in an `IllegalArgumentException`.

This solution does not circumvent transient declarations. Fields aren't recognized if they're declared `transient`. For example, if the `fear` field were declared `transient`, the code fails to compile.

## Solution 4: Implement a Serialization Proxy Pattern

- Don't expose your object to serialization.
- Instead, serialize a proxy object type with writeReplace.
- The proxy is written as an inner class.
- Deserialize the proxy back to a copy of the original object type with readResolve.
- Conclude with another readResolve for safety.

```
public class Employee implements Serializable {  
    private int salary;  
    private String fear;  
...  
    private static class ProxyEmployee implements Serializable{  
        private final int rank;  
        public ProxyEmployee(Employee emp){  
            rank = (int)(emp.salary/10000 - 5);  
        }  
        private Object readResolve() throws ObjectStreamException {  
            return new Employee((rank+5)*10000, "Fearless by Proxy");  
        }  
        private Object writeReplace() throws ObjectStreamException {  
            return new ProxyEmployee(this);  
        }  
        private void readObject(ObjectInputStream ois) throws  
            InvalidObjectException {  
            throw new InvalidObjectException("A proxy was not used!");  
        }  
    }  
}
```

salary: 80000  
fear: Spiders

rank: 3

salary: 80000  
fear: Fearless by Proxy

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



## Deserialize Cautiously

- Deserialization is a form of object construction.
- It effectively creates a public constructor that may sidestep security checks.
- Deserialization of untrusted data is dangerous.
- Like other means of setting state, you should first validate values.



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

## Summary

In this lesson, you should have learned how to:

- Identify potential Denial of Service vulnerabilities
- Secure confidential information
- Support data integrity
- Explain reasons for input validation
- Limit accessibility and extensibility of sensitive objects
- Consider security measures for serialization



Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



For more information on the Java secure coding guideline, see:

<https://www.oracle.com/technetwork/java/seccodeguide-139067.html>



Adolfo De+la+Rosa (adolfoldelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.