



Integrated Cloud Applications & Platform Services

Java SE: Programming II

Student Guide - Volume I

D102474GC10

Edition 1.0 | May 2019 | D105722

Learn more from Oracle University at 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.

Introduction



ORACLE®



1

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

Adolfo De+la+Rosa (adolfo.delarosa.2012@gmail.com) has a
non-transferable license to use this Student Guide.

Course Objectives

After completing this course, you should be able to:

- Create Java technology applications that leverage the object-oriented features of the Java language, such as encapsulation, inheritance, and polymorphism
- Create applications that use the Collections framework
- Search and filter collections using lambda expressions
- Implement error-handling techniques using exception handling
- Describe Java's new module system
- Identify and address common requirements in migrating older applications to modularity



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

Course Objectives

- Use Lambda Expression concurrency features
- Implement input/output (I/O) functionality to read from and write to data and text files and understand advanced I/O streams
- Manipulate files, directories, and file systems using the NIO.2 specification
- Understand and apply Java secure guidelines
- Perform multiple operations on database tables, including creating, reading, updating, and deleting, using the JDBC API



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



Introductions

Meet your classmates and briefly introduce yourself:

- Name
- Title or position
- Company
- Experience with Java programming and Java applications
- Reasons for attending



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

Audience

The target audience includes those who have:

- Completed the *Java SE Programming I* course or have experience with the Java language and can create, compile, and execute programs
- Experience with at least one programming language
- An understanding of object-oriented principles
- Experience with basic database concepts and a basic knowledge of SQL



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

Prerequisites

To successfully complete this course, you must know how to:

- Develop applications using the Java programming language
- Use object-oriented programming techniques
- Use primitives and classes commonly found in Java programs, such as arrays, collections, and streams
- Perform basic operating system administration from the command line



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

Course Roadmap

Unit 1: Fast-Track to Object-oriented Programming

Unit 2: Functional Programming

Unit 3: Modular Programming

Unit 4: Streams and Parallel Streams

Unit 5: Java API Programming and Secure Coding Concepts

▶ Lesson 1: Course Introduction

▶ Lesson 2: Java OOP Review

▶ Lesson 3: Exception Handling and Assertions

▶ Lesson 4: Java Interfaces

▶ Lesson 5: Collections and Generics



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



Course Roadmap

Unit 1: Fast-Track to Object-oriented Programming

Unit 2:Functional Programming

Unit 3:Modular Programming

Unit 4:Streams and Parallel Streams

Unit 5: Java API Programming and Secure Coding Concepts

► Lesson 6: Functional Interface and Lambda Expressions

► Lesson 7: Collections, Streams, and Filters

► Lesson 8: Lambda Built-in Functional Interfaces

► Lesson 9: More Lambda Expressions



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



Course Roadmap

Unit 1: Fast-Track to Object-oriented Programming

Unit 2: Functional Programming

Unit 3: Modular Programming

Unit 4: Streams and Parallel Streams

Unit 5: Java API Programming and Secure Coding Concepts

▶ **Lesson 10: The Modules System**

▶ **Lesson 11: Migrating to a Modular Application**

▶ **Lesson 12: Services in a Modular Application**



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



Course Roadmap

Unit 1: Fast-Track to Object-oriented Programming

Unit 2: Functional Programming

Unit 3: Modular Programming

Unit 4: Streams and Parallel Streams

Unit 5: Java API Programming and Secure Coding Concepts

► Lesson I3: Concurrency

► Lesson 14: Parallel Streams

► Lesson 15: Terminal Operations: Collectors

► Lesson 16: Creating Custom Streams



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



Course Roadmap

Unit 1: Fast-Track to Object-oriented Programming

Unit 2: Functional Programming

Unit 3: Modular Programming

Unit 4: Streams and Parallel Streams

Unit 5: Java API Programming and Secure Coding Concepts

► Lesson 17: I/O Fundamentals and NIO2

► Lesson 18: Java Secure Coding Concepts

► Lesson 19: JDBC

► Lesson 20: Localization



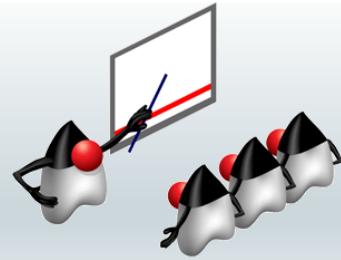
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Lesson Format

Lecture / Student Guide (50%)

- Traditional slides
- Sample Code
- Demos
- Short quizzes



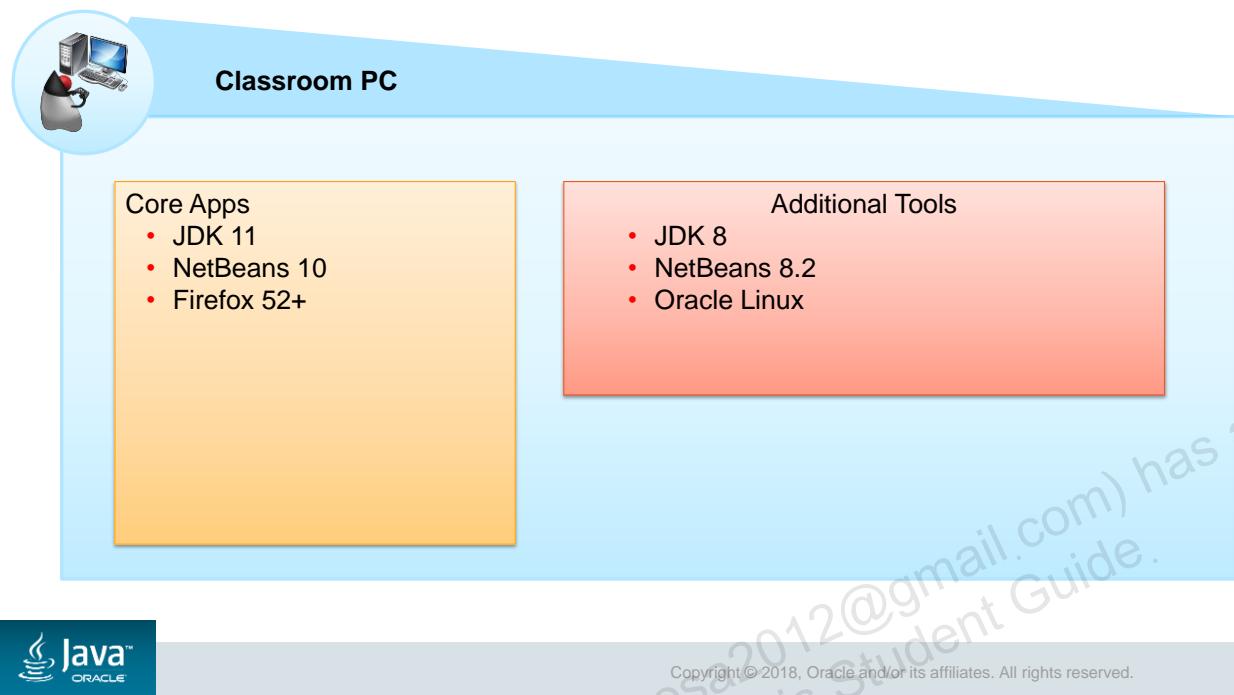
Practices / Activity Guide (50%)

- Hands-on learning
- Work with Java code
- Intended for the OU Practice Environment



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

Practice Environment



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

The following products are preinstalled for the lesson practices:

- **JDK 11**
 - **JLink**
 - **JShell**
 - **JDeprscan**
- **Firefox 52+**
- **NetBeans 10 rc3.**
- **Oracle Linux** is Oracle's enterprise implementation of Linux. It's compatible with RedHat Linux.

How Do You Learn More After the Course?

- In the Oracle Learning Library, there is a list of resources that you can use to learn more about Java programming. Look for the collection on the oracle.com/oll/java page.
- *Oracle Learning Library:*
 - <http://www.oracle.com/goto/oll>



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

Additional Resources

Resource	Website
Education and Training	http://education.oracle.com
Product Documentation	http://www.oracle.com/technology/documentation
Product Downloads	http://www.oracle.com/technology/software
Product Articles	http://www.oracle.com/technology/pub/articles
Product Support	http://www.oracle.com/support
Product Forums	http://forums.oracle.com
Product Tutorials	http://www.oracle.com/technology/obe
Sample Code	http://www.oracle.com/technology/sample_code



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

The table in the slide lists web resources where you can obtain additional information about Java.

Additional Resources

Resource	Website
Java Documentation	https://docs.oracle.com/javase
API Documentation	https://docs.oracle.com/javase/10/docs/api/index.html



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

The table in the slide lists web resources where you can obtain additional information about Java.

Summary

In this lesson, you reviewed the course objectives and the tentative class schedule. You met your fellow students, and you saw an overview of the computer environment that you will use during the course.

Enjoy the next five days of *Java SE Programming II*



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

Practice 1: Overview

This practice covers the following topics:

- 1-1: Logging in to Oracle Linux
- 1-2: Opening Terminal Windows in Oracle Linux
- 1-3: Verifying the Version of Java
- 1-4: Opening a Text File in Oracle Linux
- 1-5: Starting NetBeans and Opening a Project



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.

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.



ORACLE®

Java OOP Review

2



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

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 do the following:

- Create Java classes
- Use encapsulation in Java class design
- Construct abstract Java classes and subclasses
- Override methods
- Use virtual method invocation
- Use varargs to specify variable arguments
- Apply the final keyword in Java
- Distinguish between top-level and nested classes
- Use enumerations



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



Java Language Review

This lesson is a review of fundamental Java and programming concepts. It is assumed that students are familiar with the following concepts:

- The basic structure of a Java class
- Program block and comments
- Variables
- Branching constructs
- Iteration with loops
- Overloading of methods
- Encapsulation
- Inheritance
- Polymorphism
- Abstract Classes

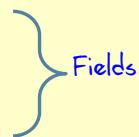


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A Simple Java Class: Employee

A Java class is often used to represent a concept.

```
1 package com.example.domain; Package declaration
2 public class Employee { Class declaration
3     public int empId;
4     public String name;
5     public String ssn;
6     public double salary;
7
8     public Employee () { Constructor
9     }
10
11    public int getEmpId () { Method
12        return empId;
13    }
14 }
```



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

A Java class is often used to store or represent data for the construct that the class represents. For example, you could create a model (a programmatic representation) of an employee. An `Employee` object defined by using this model contains values for `empId`, `name`, Social Security Number (`ssn`), and `salary`.

A constructor is used to create an instance of a class. Unlike methods, constructors do not declare a return type and are declared with the same name as their class. Constructors can take arguments, and you can declare more than one constructor.

Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all the fields `private`.

- The `Employee` class currently uses `public` access for all of its fields.
- To encapsulate the data, make the fields `private`.

```
public class Employee {  
  
    private int empId;  
    private String name;  
    private String ssn;  
    private double salary;  
  
    //... constructor and methods  
}
```

Declaring fields `private` prevents direct access to this data from a class instance.
// illegal!
emp.salary =
1_000_000_000.00;



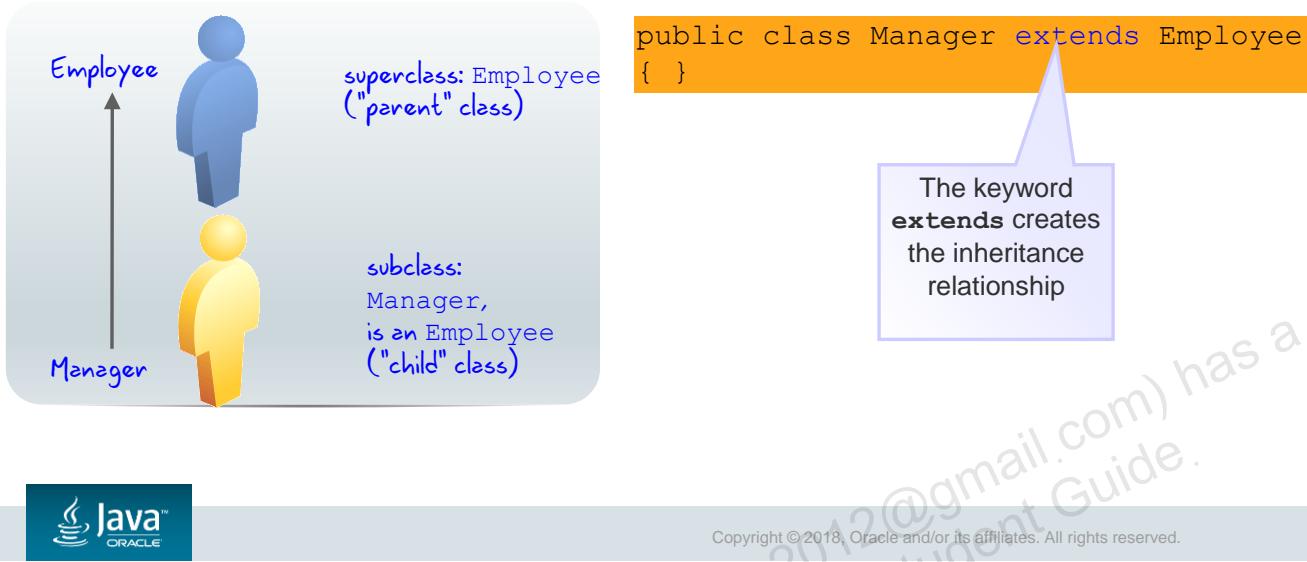
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In Java, we accomplish encapsulation through the use of visibility modifiers (and also through the module system introduced in the lesson “Modules Overview”). Declaring Java fields `private` makes it invisible outside of the methods in the class itself.

In this example, the fields `custID`, `name`, and `amount` are now marked `private`, making them invisible outside of the methods in the class itself.

Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.



In the case of a subclass access to its superclass, it has **access** to all superclass fields but only inherits the nonprivate attributes and methods.

The code snippet in the slide demonstrates the Java syntax for subclassing.

The diagram in the slide demonstrates an inheritance relationship between the `Manager` class and, its parent, the `Employee` class.

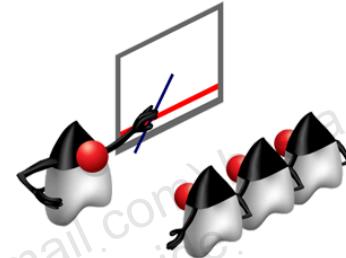
- The `Manager` class, by extending the `Employee` class, inherits all of the nonprivate data fields and methods from `Employee`.
- Since a manager is also an employee, then it follows that `Manager` has all of the same attributes and operations of `Employee`.

Note: The `Manager` class declares its own constructor. Constructors are *not* inherited from the parent class. There are additional details about this in the next slide.

Constructors in Subclasses

Although a subclass inherits all of the methods and fields from a parent class, it doesn't inherit constructors. There are two ways to gain a constructor:

- Write your own constructor.
- Use the default constructor.
 - If you do not declare a constructor, a default no-arg constructor is provided for you.
 - If you declare your own constructor, the default constructor is no longer provided.



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

Every subclass inherits the nonprivate fields and methods from its parent (superclass). However, the subclass does not inherit the constructor from its parent. It must provide a constructor.

The *Java Language Specification* includes the following description:

“Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.”

Using super

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- In its constructor, Manager calls the constructor of Employee.
- The `super` keyword is used to call a parent's constructor.
- It must be the first statement of the constructor.
- If it is not provided, a default call to `super()` is inserted for you.
- The `super` keyword may also be used to invoke a parent's method or to access a parent's (nonprivate) field.

```
super (empId, name, ssn, salary);
```



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

Using Access Control

- You have seen the keywords `public` and `private`.
- There are four access levels that can be applied to data fields and methods.
- Classes can be default (no modifier) or `public`.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes



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

Protected Access Control: Example

```
package demo;
public class Foo {
    protected int result = 20; ← subclass-friendly declaration
    int num= 25;
}
```

```
package test;
import demo.Foo;
public class Bar extends Foo {
    private int sum = 10;
    public void reportSum () {
        sum += result;
        sum +=num; ← compiler error
    }
}
```



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

Inheritance: Accessibility of Overriding Methods

The overriding method cannot be less accessible than the method in the parent class.

```
public class Employee {  
    //... other fields and methods  
    public String getDetails() { ...}  
}
```

```
public class BadManager extends Employee {  
    private String deptName;  
    // lines omitted  
    @Override  
    private String getDetails() { // Compile error  
        return super.getDetails () + 23 " Dept: " + deptName;  
    }  
}
```



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

Final Methods

A method can be declared `final`. Final methods may not be overridden.

```
public class MethodParentClass {  
    public final void printMessage() {  
        System.out.println("This is a final method");  
    }  
}
```

```
public class MethodChildClass extends MethodParentClass {  
    // compile-time error  
    public void printMessage() {  
        System.out.println("Cannot override method");  
    }  
}
```



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

Performance Myths

There is little to no performance benefit when you declare a method as `final`. Methods should be declared as `final` only to disable method overriding.

Final Classes

A class can be declared `final`. Final classes may not be extended.

```
public final class FinalParentClass { }
```

```
// compile-time error  
public class ChildClass extends FinalParentClass { }
```



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

Applying Polymorphism

Suppose that you are asked to create a new class that calculates a bonus for employees based on their salary and their role (employee, manager, or engineer):

```
public class BadBonus {  
    public double getBonusPercent(Employee e) {  
        return 0.01;  
    }  
  
    public double getBonusPercent(Manager m) {  
        return 0.03;  
    }  
    public double getBonusPercent(Engineer e) {  
        return 0.01;  
    }  
    // Lines omitted
```

not very
object-oriented!



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

Design Problem

What is the problem in the example in the slide? Each method performs the calculation based on the type of employee passed in and returns the bonus amount.

Consider what happens if you add two or three more employee types. You would need to add three additional methods and possibly replicate the code depending upon the business logic required to compute shares.

Clearly, this is not a good way to treat this problem. Although the code will work, this is not easy to read and is likely to create much duplicate code.

Applying Polymorphism

A good practice is to pass parameters and write methods that use the most generic possible form of your object.

```
public class GoodBonus {  
    public static double getBonusPercent(Employee e) {  
        // Code here  
    }  
  
// In the Employee class  
public double calcBonus() {  
    return this.getSalary() * GoodBonus.getBonusPercent(this);  
}
```

- One method will calculate the bonus for every type.



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

Use the Most Generic Form

A good practice is to design and write methods that take the most generic form of your object possible. In this case, `Employee` is a good base class to start from. But how do you know what object type is passed in? You learn the answer in the next slide.

Overriding methods of Object Class

The root class of every Java class is `java.lang.Object`.

- All classes subclass `Object` by default.
 - You don't have to declare that your class extends `Object`. The compiler does that for you.

```
public class Employee { //... }
```



```
public class Employee extends Object { //... }
```

- The `Object` class contains several methods, but there are three that are important to consider overriding:
 - `toString`, `equals`, and `hashCode`



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

Overriding methods of Object Class: `toString` Method

The `toString` method returns a String representation of the object.

```
Employee e = new Employee (101, "Jim Kern", ...)  
System.out.println(e);
```

- You can use `toString` to provide instance information:

```
public String toString () {  
    return "Employee id: " + empId + "\n"+  
           "Employee name:" + name;  
}
```

- This is a better approach to getting details about your class than creating your own `getDetails` method.



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

The `println` method is overloaded with a number of parameter types. When you invoke `System.out.println(e);` the method that takes an `Object` parameter is matched and invoked. This method in turn invokes the `toString()` method on the object instance.

Note: Sometimes you may want to be able to print out the name of the class that is executing a method. The `getClass()` method is an `Object` method used to return the `Class` object instance, and the `getName()` method provides the fully qualified name of the runtime class. `getClass().getName(); //` returns the name of this class instance. These methods are in the `Object` class.

Overriding methods of Object Class: equals Method

The `equals` method compares only object references.

- If there are two objects `x` and `y` in any class, `x` is equal to `y` if and only if `x` and `y` refer to the same object. For example:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

- In case you want to test the contents of the `Employee` object, you need to override the `equals` method:

```
public boolean equals (Object o) { ... }
```



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

The `equals` method of `Object` determines (by default) only if the values of two object references point to the same object. Basically, the test in the `Object` class is simply as follows:

If `x == y`, return true.

For an object (like the `Employee` object) that contains values, this comparison is not sufficient, particularly if we want to make sure there is one and only one employee with a particular ID.

Overriding methods of Object Class: equals Method

For example, overriding the `equals` method in the `Employee` class compares every field for equality:

```
@Override  
public boolean equals (Object o) {  
    boolean result = false;  
    if ((o != null) && (o instanceof Employee)) {  
        Employee e = (Employee)o;  
        if ((e.empId == this.empId) &&  
            (e.name.equals(this.name)) &&  
            (e.ssn.equals(this.ssn)) &&  
            (e.salary == this.salary)) {  
            result = true;  
        }  
    }  
    return result;  
}
```



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

This simple `equals` test first tests to make sure that the object passed in is not null and then tests to make sure that it is an instance of an `Employee` class (all subclasses are also employees, so this works). Then the `Object` is cast to `Employee`, and each field in `Employee` is checked for equality.

Note: For String types, you should use the `equals` method to test the strings character by character for equality.

@Override annotation

This annotation is used to instruct the compiler that the method annotated with `@Override` is an overridden method from super class or interface. When this annotation is used, the compiler check is to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that your method does not actually override as you think it does. Secondly, it makes your code easier to understand when you are overriding methods.

Overriding methods of Object Class: hashCode Method

The general contract for `Object` states that if two objects are considered equal (using the `equals` method), then integer `hashCode` returned for the two objects should also be equal.

```
@Override //generated by NetBeans
public int hashCode() {
    int hash = 7;
    hash = 83 * hash + this.empId;
    hash = 83 * hash + Objects.hashCode(this.name);
    hash = 83 * hash + Objects.hashCode(this.ssn);
    hash = 83 * hash + (int) (Double.doubleToLongBits(this.salary) ^
(Double.doubleToLongBits(this.salary) >>> 32));
    return hash;
}
```



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

Overriding hashCode

The Java documentation for the `Object` class states:

"... It is generally necessary to override the `hashCode` method whenever this method [`equals`] is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes."

The `hashCode` method is used in conjunction with the `equals` method in hash-based collections, such as `HashMap`, `HashSet`, and `Hashtable`.

This method is easy to get wrong, so you need to be careful. The good news is that IDEs such as NetBeans can generate `hashCode` for you.

To create your own hash function, the following will help approximate a reasonable hash value for equal and unequal instances:

- 1) Start with a nonzero integer constant. Prime numbers result in fewer hashcode collisions.
- 2) For each field used in the `equals` method, compute an `int` hash code for the field. Notice that for the `Strings`, you can use the `hashCode` of the `String`.
- 3) Add the computed hash codes together.
- 4) Return the result.

Casting Object References

After using the `instanceof` operator to verify that the object you received as an argument is a subclass, you can access the full functionality of the object by casting the reference:

```
public static void main(String[] args) {  
    Employee e = new Manager(102, "Joan Kern",  
    "012-23-4567", 110_450.54, "Marketing");  
    if (e instanceof Manager){  
        Manager m = (Manager) e;  
        m.setDeptName("HR");  
        System.out.println(m.getDetails());  
    }  
}
```

Without the cast to Manager, the `setDeptName` method would not compile.



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

Although a generic superclass reference is useful for passing objects around, you may need to use a method from the subclass.

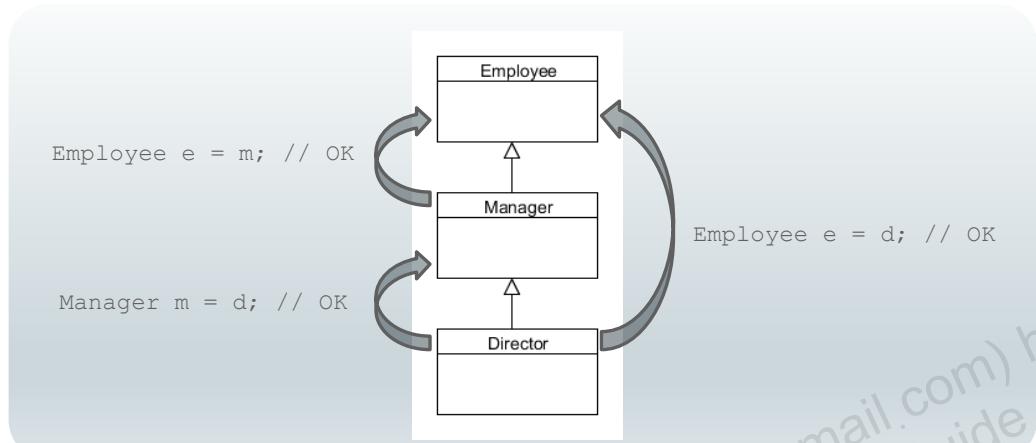
In the slide, for example, you need the `setDeptName` method of the `Manager` class. To satisfy the compiler, you can cast a reference from the generic superclass to the specific class.

However, there are rules for casting references. You see these in the next slide.

Upward Casting Rules

Upward casts are always permitted and do not require a cast operator.

```
Director d = new Director();
Manager m = new Manager();
```

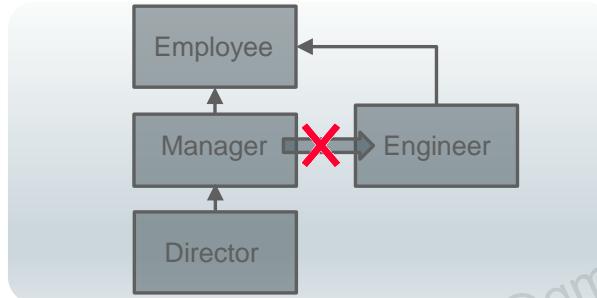


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Downward Casting Rules

For downward casts, the compiler must be satisfied that the cast is possible.

```
Employee e = new Manager(102, "Joan Kern",
    "012-23-4567", 110_450.54, "Marketing");
Manager m = (Manager)e; // ok
Engineer eng = (Manager)e; // Compile error
System.out.println(m.getDetails());
```



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

Downward Casts

With a downward cast, the compiler simply determines if the cast is possible; if the cast down is to a subclass, then it is possible that the cast will succeed.

Note that at run time, the cast results in a `java.lang.ClassCastException` if the object reference is of a superclass and not of the class type or a subclass.

Finally, any cast that is outside the class hierarchy will fail, such as the cast from a `Manager` instance to an `Engineer`. A `Manager` and an `Engineer` are both `employees`, but a `Manager` is not an `Engineer`.

Methods Using Variable Arguments

A variation of method overloading is when you need a method that takes any number of arguments of the same type:

```
public class Statistics {  
    public float average (int x1, int x2) {}  
    public float average (int x1, int x2, int x3) {}  
    public float average (int x1, int x2, int x3, int x4) {}  
}
```

- These three overloaded methods share the same functionality. It would be nice to collapse these methods into one method.

```
Statistics stats = new Statistics ();  
float avg1 = stats.average(100, 200);  
float avg2 = stats.average(100, 200, 300);  
float avg3 = stats.average(100, 200, 300, 400);
```



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

Methods with a Variable Number of the Same Type

One case of overloading is when you need to provide a set of overloaded methods that differ in the number of the same type of arguments. For example, suppose you want to have methods to calculate an average. You may want to calculate averages for 2, 3, or 4 (or more) integers.

Each of these methods performs a similar type of computation—the average of the arguments passed in, as in this example:

```
public class Statistics {  
    public float average(int x1, int x2) { return (x1 + x2) / 2; }  
    public float average(int x1, int x2, int x3) {  
        return (x1 + x2 + x3) / 3;  
    }  
    public float average(int x1, int x2, int x3, int x4) {  
        return (x1 + x2 + x3 + x4) / 4;  
    }  
}
```

Java provides a convenient syntax for collapsing these three methods into just one and providing for any number of arguments.

Methods Using Variable Arguments

- Java provides a feature called *varargs* or *variable arguments*.

```
public class Statistics {  
    public float average(int... nums) {  
        int sum = 0;  
        for (int x : nums) { // iterate int array nums  
            sum += x;  
        }  
        return ((float) sum / nums.length);  
    }  
}
```

The varargs notation treats the `nums` parameter as an array.

- Note that the `nums` argument is actually an array object of type `int[]`. This permits the method to iterate over and allow any number of elements.



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

Using Variable Arguments

The `average` method shown in the slide takes any number of integer arguments. The notation `(int... nums)` converts the list of arguments passed to the `average` method into an array object of type `int`.

Note: Methods that use varargs can also take no parameters—an invocation of `average()` is legal. You will see varargs as optional parameters in use in the NIO.2 API in the lesson titled “Java File I/O.” To account for this, you could rewrite the `average` method in the slide as follows:

```
public float average(int... nums) {  
    int sum = 0; float result = 0;  
    if (nums.length > 0) {  
        for (int x : nums) // iterate int array nums  
            sum += x;  
        result = (float) sum / nums.length;  
    }  
    return (result);  
}
```

Static Imports

A static import statement makes the static members of a class available under their simple name.

- Given either of the following lines:

```
import static java.lang.Math.random;  
import static java.lang.Math.*;
```

- Calling the `Math.random()` method can be written as:

```
public class StaticImport {  
    public static void main(String[] args) {  
        double d = random();  
    }  
}
```



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

Nested Classes

A nested class is a class declared within the body of another class:

- Have multiple categories
 - **Inner classes**
 - Member classes
 - Local classes
 - Anonymous classes
 - **Static nested classes**
- Are commonly used in applications with GUI elements
- Can limit utilization of a "helper class" to the enclosing top-level class



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

An **inner** nested class is considered part of the outer class and inherits access to all the private members of the outer class.

A **static** nested class is not an inner class, but its declaration appears similar to an additional `static` modifier on the nested class. Static nested classes can be instantiated before the enclosing outer class and, therefore, are denied access to all nonstatic members of the enclosing class.

Note: Anonymous classes are covered in detail in the lesson titled “Interfaces and Lambda Expressions.”

Reasons to Use Nested Classes

The following information is obtained from

<http://download.oracle.com/javase/tutorial/java/javaOO/nested.html>.

- **Logical Grouping of Classes**
 - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **Increased Encapsulation**
 - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **More Readable, Maintainable Code**
 - Nesting small classes within top-level classes places the code closer to where it is used.

Example: Member Class

```
public class BankEMICalculator {  
    private String CustomerName;  
    private String AccountNo;  
    private double loanAmount;  
    private double monthlypayment;  
    private EMICalculatorHelper helper = new EMICalculatorHelper();  
  
    /*Setters ad Getters*/  
  
    private class EMICalculatorHelper {  
        int loanTerm = 60;  
        double interestRate = 0.9;  
        double interestpermonth=interestRate/loanTerm;  
  
        protected double calcMonthlyPayment(double loanAmount)  
        {  
            double EMI= (loanAmount * interestpermonth) / ((1.0) - ((1.0) / Math.pow(1.0 +  
            interestpermonth, loanTerm)));  
            return(Math.round(EMI));  
        }  
    }  
}
```

Inner class,
EMICalculatorHelper



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

The example in the slide demonstrates an inner class, EMICalculatorHelper, which is defined in the BankEMICalculator class.

What Are Enums?

- An enum is a special data type that represents a fixed set of constants.
- For example:
 1. Directions
 2. Colors of the rainbow
 3. Planets in the solar system

- Defining an enum:

```
public enum Directions {  
    NORTH, SOUTH, EAST, WEST //; semi-colon is optional here  
}
```

enum values are declared in capitals as they are constants

- Accessing an enum values:

Enum constants can be accessed as `Directions.EAST` and `Directions.SOUTH`.



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

Complex Enums

Enums can have fields, methods and constructors.

```
public enum Department {  
  
    HR("DEPT-01"), OPERATIONS("DEPT-02"), LEGAL("DEPT-03"), MARKETING("DEPT-  
    04");//semi-colon is not optional here  
  
    Department(String deptCode) {  
        this.deptCode=deptCode;  
    }  
  
    private String deptCode;  
  
    public String getDeptCode() {  
        return deptCode;  
    }  
}
```



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

Enum Constructors

You may not instantiate an enum instance with `new`.

Methods in Enums

Enum has a few special methods and are very useful.

- `values()`: Returns an array of all enum constants of that enum type

```
public class EnumTest {  
    public static void main(String args[]) {  
  
        for(Department dept:Department.values()) {  
            System.out.println(dept+" Department Code:  
"+dept.getDeptCode());  
        }  
    }  
}
```

Output:

```
HR Department Code: DEPT-01  
OPERATIONS Department Code: DEPT-02  
LEGAL Department Code: DEPT-03  
MARKETING Department Code: DEPT-04
```



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

All enums implicitly extend `java.lang.Enum<E>`. Methods `name()`, `ordinal()` and `valueOf()`, inherited from `Enum<E>`, are available on all enums.

Methods in Enums

- `ordinal()`: Returns an `int` value equal to the enum constant's ordinal position in enum declaration, starting from the value 0

```
public class EnumTest {  
    public static void main(String args[]) {  
        for(Department dept:Department.values()) {  
            System.out.println("dept+"ordinal value-> "+dept.ordinal());  
        }  
    }  
}
```

Output:

```
HR ordinal value-> 0  
OPERATIONS ordinal value-> 1  
LEGAL ordinal value-> 2  
MARKETING ordinal value-> 3
```



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

Enum Constructors

You cannot instantiate an enum instance with `new` operator.

Summary

In this lesson, you should have learned how to:

- Create Java classes
- Use encapsulation in Java class design
- Construct abstract Java classes and subclasses
- Override methods
- Use virtual method invocation
- Use varargs to specify variable arguments
- Apply the final keyword in Java
- Distinguish between top-level and nested classes
- Use enumerations



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



Practice 2: Overview

This practice covers the following topics:

- Practice 2-1: Overriding and overloading methods
- Practice 2-2: Using Java enumerations



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

Quiz



Suppose that you have an `Account` class with a `withdraw()` method and a `Checking` class that extends `Account` that declares its own `withdraw()` method. What is the result of the following code fragment?

```
1 Account acct = new Checking();  
2 acct.withdraw(100);
```

- a. The compiler complains about line 1.
- b. The compiler complains about line 2.
- c. Runtime error: incompatible assignment (line 1)
- d. Executes `withdraw` method from the `Account` class
- e. Executes `withdraw` method from the `Checking` class



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

Quiz



Suppose that you have an Account class and a Checking class that extends Account. The body of the if statement in line 2 will execute.

```
1 Account acct = new Checking();  
2 if (acct instanceof Checking) { // will this block run? }
```

- a. True
- b. False



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

Quiz



Suppose that you have an `Account` class and a `Checking` class that extends `Account`. You also have a `Savings` class that extends `Account`. What is the result of the following code?

```
1 Account acct1 = new Checking();  
2 Account acct2 = new Savings();  
3 Savings acct3 = (Savings)acct1;
```

- a. `acct3` contains the reference to `acct1`.
- b. A runtime `ClassCastException` occurs.
- c. The compiler complains about line 2.
- d. The compiler complains about the cast in line 3.



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

Quiz



Which two of the following should an abstract method not have to compile successfully?

- a. A return value
- b. A method implementation
- c. Method parameters
- d. private access



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

Quiz



A final field (instance variable) can be assigned a value either when declared or in all constructors.

- 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.

3

Exception Handling and Assertions



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



ORACLE®

Objectives

After completing this lesson, you should be able to:

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



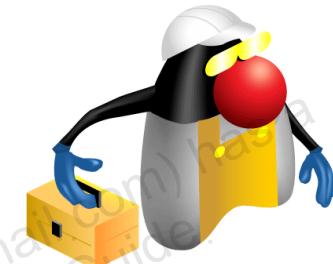
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Error Handling

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

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



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

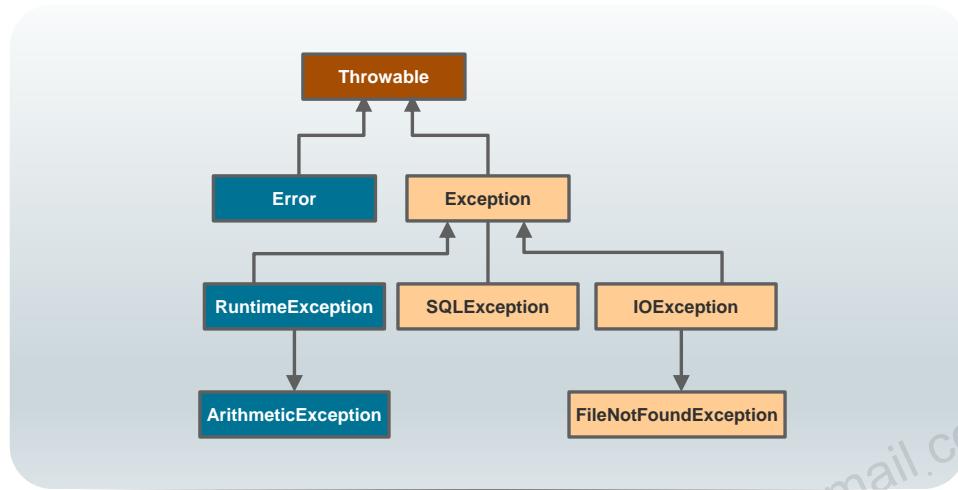
Returning a Failure Result

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

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

Exception Types

- An exception is an instance of a class derived directly or indirectly from the `java.lang.Throwable` class.
- Two predefined Java classes are derived from `Throwable`- `Error` and `Exception`.



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

Dealing with Exceptions

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

Unchecked Exceptions

`java.lang.RuntimeException` and `java.lang.Error` and their subclasses are categorized as unchecked exceptions. These types of exceptions should not normally occur during the execution of your application. You can use a `try-catch` statement to help discover the source of these exceptions.

However, when an application is ready for production use, there should be a little code remaining that deals with `RuntimeException` and its subclasses. The `Error` subclasses represent errors that are beyond your ability to correct, such as the JVM running out of memory. Common `RuntimeExceptions` that you may have to troubleshoot include:

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

Exception Types

- The exceptions derived from the `Error` class represent problems with the JVM and normally can't be recovered, and there is little that a programmer will do with these exceptions.
- The subclasses of `Exception` class support two types of exceptions:
 - Checked: These are exceptions that you need to handle within the code
 - Unchecked: These are exceptions that you don't need to handle within the code



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

Checked exceptions include all exceptions derived from the `Exception` class and are not derived from the `RuntimeException` class. These must be handled in code or the code will not compile cleanly, resulting in compile-time errors.

Unchecked exceptions are all other exceptions. They include exceptions, such as division by zero and array subscripting errors. These do not have to be caught, but like the `Error` exceptions, if they are not caught, the program will terminate.

Exception Handling Techniques in Java

There are two general techniques you can use in Java:

1. Handling an exception: You must add in a code block to handle the error
 - a. try block
 - b. try-with-resources block
2. Declaring an exception: You declare that a method may fail to execute successfully.



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

The Handle or Declare Rule

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

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

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

Exception Handling Techniques: try Block

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

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

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



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

Exception Handling Techniques: finally Clause

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



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

Closing Resources

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

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

Exception Handling Techniques: try-with-resources

- The `try-with-resources` block:
 - Declares one or more resources. All the resources opened within the block are automatically closed upon exit from the block.
 - No need to close the resources explicitly in the code.
 - Any class that implements `java.lang.AutoCloseable` can be used as a resource.

```
try (BufferedReader reader = Files.newBufferedReader((Path) new FileReader("src.txt"));
      BufferedWriter writer = Files.newBufferedWriter((Path) new FileWriter("dest.txt"));
{
    String input;
    while ((input = reader.readLine()) != null) {
        writer.write(input);
        writer.newLine();
    }
}
catch(URIException ex) {...}
catch(IOException ex){...}
```

Two resources declared within the try-with-resources block



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

Closeable Resources

The use of the previous technique can be cumbersome when multiple resources are opened and a failure occurs. It can result in multiple try-catch blocks that become hard to follow.

In Java 7, the `try-with-resources` block was introduced to address this situation. The `try-with-resources` statement can eliminate the need for a lengthy `finally` block. Resources opened by using the `try-with-resources` statement are always closed. If a resource should be autoclosed, its reference must be declared within the `try` statement's parentheses.

Resources declared with a `try-with-resources` block must be separated by semicolons, otherwise a compile-time error will be generated.

Exception Handling Techniques: try-with-resources Improvements

- Concise `try-with-resources` statements in JDK 9:

If you already have a resource as a final or effectively final variable, you can use that variable in the `try-with-resources` statement without declaring a new variable in the `try-with-resources` statement.

- For example, given resource declarations like:

```
// A final resource  
final Resource resource1 = new Resource("resource1");  
// An effectively final resource  
Resource resource2 = new Resource("resource2");
```

- The old way to write the code to manage these resources would be something like:

```
// Original try-with-resources statement from JDK 7 or 8  
try (Resource r1 = resource1;  
     Resource r2 = resource2) {  
    // Use of resource1 and resource 2 through r1 and r2.  
}
```

Resources are declared within the try-with-resources statement



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

Exception Handling Techniques: try-with-resources Improvements

- try-with-resources statement in JDK 9 and later versions.

```
BufferedReader reader = Files.newBufferedReader((Path) new FileReader("src.txt"));
BufferedWriter writer = Files.newBufferedWriter((Path) new FileWriter("dest.txt"));

try (reader ; writer) {
    String input;
    while ((input = reader.readLine()) != null) {
        writer.write(input);
        writer.newLine();
    }
}

catch (URISyntaxException ex) {
}
catch (IOException ex) {
}
```

Resources declared outside as
final or effectively final are used
within the try-with-resources
statement



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

Exception Handling Techniques: | operator in a catch block

- Consider the situation where two exceptions are potentially thrown and are handled in the same way, for example:

```
try {...}
catch (IOException e) {
    e.printStackTrace();
}
catch (NumberFormatException e){
    e.printStackTrace();
}
```

- Instead of duplicating the code in each catch block, you can use a vertical bar to permit one catch block to capture more than one exception.

```
try {...}
catch (IOException | NumberFormatException e) {
    e.printStackTrace();}
```

Multiple exception types
are separated with a
vertical bar.

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



Exception Handling Techniques: Declaring Exceptions

- You may declare that a method throws an exception instead of handling it:
 - Is used when the current method is not the appropriate place to handle the exception.
 - Allows the exception to be propagated higher into the sequence of method calls.
- For example:

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

`readByteFromFile()` may encounter some condition where it may throw `IOException`. Instead of dealing with the exception in `readByteFromFile()`, the `throws` keyword in the method definition results in the exception being passed to the code that called this method.



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

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

If an exception is generated while executing the method, the method stops executing and the exception is thrown to the caller. Uncaught exceptions are propagated to the next higher context until they are caught or they are thrown from main, where an error message and stack trace will be printed.

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

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

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

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

The method will skip all of the remaining lines of code in the method and immediately return to the caller.

Creating Custom Exceptions

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

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



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

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

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

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

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

Assertions

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



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

Why Use Assertions

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

Assertion Syntax

There are two forms of the `assert` statement:

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



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

The `assert` Statement

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

Internal Invariants

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

Internal Invariant



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

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

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

Control Flow Invariants

```
switch (suit) {  
    case Suit.CLUBS: // ...  
        break;  
    case Suit.DIAMONDS: // ...  
        break;  
    case Suit.HEARTS: // ...  
        break;  
    case Suit.SPADES: // ...  
        break;  
    default:  
        assert false : "Unknown playing card suit";  
        break;  
}
```

Control Flow Invariant



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

Class Invariants

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

Class Invariant



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

Controlling Runtime Evaluation of Assertions

- Assertion checks are disabled by default. You can enable assertions with either of the following commands:

```
java -ea MyProgram
```

```
java -enableassertions MyProgram
```



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

Enabling Assertions in Netbeans

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

Summary

In this lesson, you should have learned how to:

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

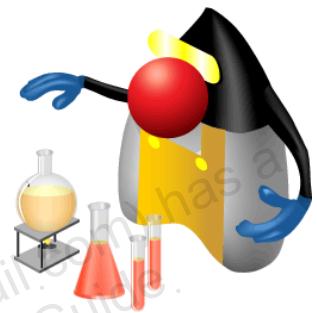


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice 3: Overview

This practice covers extending exception and using `throw` and `throws` clause.



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

Quiz



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

- a. True
- b. False



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

Quiz



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

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



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

Quiz



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

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



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

Quiz



Assertions should be used to perform user-input validation.

- a. True
- b. False



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

Java Interfaces



ORACLE



4

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

Adolfo De+la+Rosa (adolfo.delarosa.2012@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Use `default` methods in interfaces
- Identify when it's desirable to implement a `default` method in an interface
- Identify how inheritance rules apply to methods implemented in interfaces
- Use `private` methods in interfaces
- Identify when it's desirable to implement a `private` method in an interface
- Define anonymous classes



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

Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- Are similar to abstract classes containing only public abstract methods
- Outline methods that must be implemented by a class
 - Methods must not have an implementation {braces}.
- Can contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns



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

In Java, an interface outlines a contract for a class. The contract outlined by an interface mandates the methods that must be implemented in a class. Classes implementing the contract must fulfill the entire contract or be declared abstract.

Java SE 7 Interfaces

- Interfaces are Java's solution to safely facilitate multiple inheritance.
- Interfaces originally contained only:
 - `static` variables
 - `abstract` methods

An example is the `Accessible` interface. This interface is meant to be implemented in classes for financial products where people access money through deposits and withdrawals.

```
public interface Accessible{  
    public static final double OVERDRAFT_FEE = 25;  
  
    public abstract double verifyDeposit(double amount, int pin);  
    public abstract double verifyWithdraw(double amount, int pin);  
}
```



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

An example of a nonaccessible financial product would be a savings bond. You're not able to access the money after the bond is purchased.

Implementing Java SE 7 Interface Methods

`abstract` methods must be implemented later.

- If one class implements an interface, you'll write your implementation logic once.
- If many classes implement the same interface, you'll write your implementation logic many times.

What if most classes implement the exact same logic?

- Must you duplicate the same code in many places?
- Isn't code duplication bad?



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

Example: Implementing **abstract** Methods

Notice the logic found in these methods:

```
public class BasicChecking implements Accessible{  
    ...  
    public double verifyDeposit(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
    public double verifyWithdraw(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
        //Verify account balance won't go negative  
    }  
}
```



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

Example: Duplicating Logic

The same logic is largely duplicated by other classes that implement Accessible.

```
public class RestrictedChecking implements Accessible{  
    ...  
    public double verifyDeposit(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
    public double verifyWithdraw(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
        //Verify account balance won't go negative  
        //Verify the withdrawal is under the transaction limit.  
    }  
}
```



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

You'll have a chance to work with both the BasicChecking and RestrictedChecking classes in this lesson's practices. RestrictedChecking is a financial product that offers enhanced security by imposing a limit on the amount of money that can be withdrawn in a single transaction.

Implementing Methods in Interfaces

- Java SE 8 allows you to implement special types of methods within interfaces:
 - `static` methods
 - `default` methods
- `default` methods help minimize code duplication.
 - They provide a single location to write and edit code.
 - They can be overridden later if necessary.
 - They're overridden with per-class precision.



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

Example: Implementing `default` Methods

Previously duplicated logic can be written once in the Accessible interface.

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
        //Verify account balance won't go negative  
    }  
}
```



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

Adding the keyword "public" is optional for default methods. A default method is always public.

Example: Inheriting `default` Methods

Simply let the class implement the `Accessible` interface.

```
public class BasicChecking implements Accessible{  
    ...  
}
```



Example: Overriding a `default` Method

You can override a default method and call the interface's implementation.

- For example, you might override the default method to enhance the verification.

```
public class RestrictedChecking implements Accessible{  
    ...  
    public double verifyWithdraw(double amount, int pin){  
        //Call the interface's implementation  
        Accessible.super.verifyWithdraw(amount, pin);  
  
        //Verify the withdrawal is under the transaction limit.  
    }  
}
```



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

After calling the interface's version of the `vervityWithdraw` method, additional logic can be written into the method to support the features of the `RestrictedChecking` class.

What About the Problems of Multiple Inheritance?

Multiple inheritance of...	Is possible...	Using this mechanic...
Type	✓	A class implements multiple interfaces.
Behavior	✓	A class implements interfaces containing multiple default methods. Special rules exist to prevent complications.
State	✗	A call to a variable cannot have multiple potential values.*

*This is why multiple inheritance is problematic in C++.



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

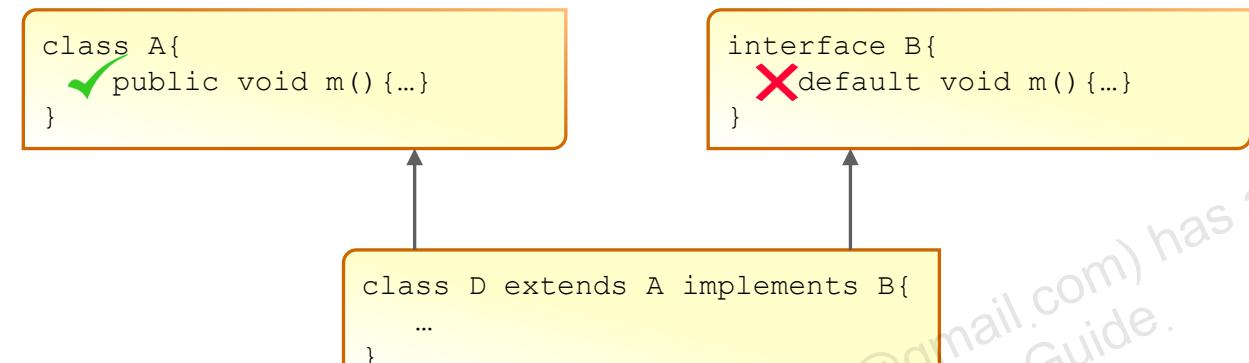
The idea of being able to implement methods within interfaces is still unfamiliar to many developers.

Inheritance Rules of `default` Methods

Rule 1:

A superclass method takes priority over an interface default method.

- The superclass method may be concrete or abstract.
- Only consider the interface default if no method exists from the superclass.



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

The code for this is found in the project Ex_04_01_Interfaces.

If you instantiate an instance of class D and call method m, the version of the method that is used comes from the superclass A.

Inheritance Rules of `default` Methods

Rule 2:

A subtype interface's default method takes priority over a super-type interface's default method.

```
interface B{  
    ✗default void m() {...}  
}
```

```
interface C extends B{  
    ✓default void m() {...}  
}
```

```
class D implements C{  
    ...  
}
```



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

The code for this is found in the project Ex_04_02_Interfaces.

If you instantiate an instance of class D and call method m, the version of the method that is used comes from interface C.

Inheritance Rules of `default` Methods

Rule 3:

If there is a conflict, treat the default method as abstract.

- The concrete class must provide its own implementation.
- This may include a call to a specific interface's implementation of the method.

```
interface B{  
    default void m() {...}  
}
```

```
interface C{  
    default void m() {...}  
}
```

```
class D implements B,C{  
    void m(){  
        B.super.m();  
    }  
}
```



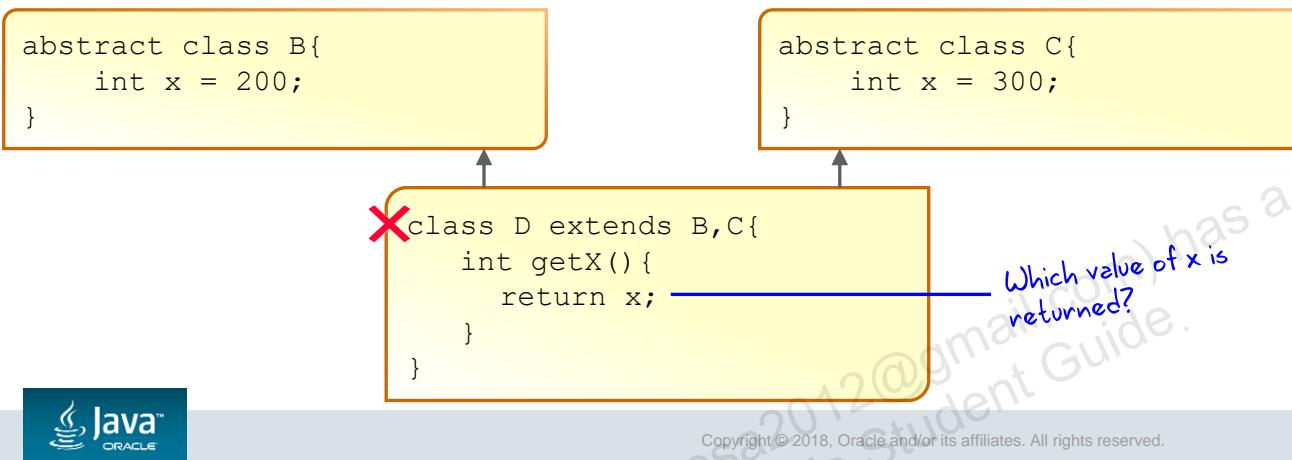
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The code for this is found in the project Ex_04_03_Interfaces.

If you instantiate an instance of class D and call method m, you'll need to specify within class D how the method should be implemented.

Interfaces Don't Replace Abstract Classes

- An interface doesn't let you store the state of an instance.
- An `abstract` class may contain instance fields.
- To avoid complications caused by multiple inheritance of state, a class cannot extend multiple `abstract` classes.



The code for this is found in the project Ex_04_04_Interfaces.

If the design of your program requires fields to be shared among different classes, you may want to write this field once in a common location. Abstract classes can accommodate this. Interfaces cannot. However, there is a trade-off. Multiple inheritance is not allowed with abstract classes.

What If `default` Methods Duplicate Logic?

Can anything be done to reduce this duplication?

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
        //Verify account balance won't go negative  
    }  
}
```

Duplicated

Duplicated



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

Returning to our scenario...

Duplication Between `default` Methods

One strategy is to put duplicated logic within its own `default` method.

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        verifyTransaction(amount, pin);  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        verifyTransaction(amount, pin);  
        //Verify account balance won't go negative  
    }  
    public default boolean verifyTransaction(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
}
```



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

You'll see in the practice solution why the `verifyTransaction` method returns a boolean.

The Problem with This Approach

- `default` methods must be public.
- They can be called from almost anywhere (including the main method).
 - Returned values may not mean anything outside the context of other methods.
 - It's dangerous if the method returns information you don't want exposed.
- They can be overridden at any time.
 - The result of calling the method may not be predictable.

```
public class TestClass{  
    public static void main(String[] args){  
  
        RestrictedChecking acct1 = new RestrictedChecking(1000, 1111);  
        acct1.verifyTransaction(200,1111);  
    }  
}
```



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

What if `verifyTransaction` was overridden somewhere in the program to always deposit one million dollars into an account? As long as the method remains public, this undesirable behavior (from the bank's perspective) remains a possibility.

Introducing `private` Methods in Interfaces

- A better strategy is to make the method `private`.
- `private` interface methods are more secure.
 - They can't be called from elsewhere.
 - They limit the risk of exposing sensitive information.
- `private` interface methods lead to more predictable programs.
 - They can't be overridden.
 - They can't be called from a class that implements the interface.
- `private` interface methods lead to more maintainable and readable code.
 - Common logic can be stored and edited in one location.
- `private` interface methods don't lead to inheritance complications.



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

Example: Using `private` Methods to Reduce Duplication Between `default` Methods

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        verifyTransaction(amount, pin);  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        verifyTransaction(amount, pin);  
        //Verify account balance won't go negative  
    }  
    private boolean verifyTransaction(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
}
```



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

You'll see in the practice solution why the `verifyTransaction` method returns a boolean.

Types of Methods in Java SE 9 Interfaces

Access Modifier and Method Type	Supported?
<code>public abstract</code>	Yes
<code>private abstract</code>	Compiler Error
<code>public default</code>	Yes
<code>private default</code>	Compiler Error
<code>public static</code>	Yes
<code>private static</code>	Yes
<code>private</code>	Yes



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

Adding the keyword "public" is optional for abstract methods. An abstract method is always public.
Adding the keyword "public" is optional for default methods. A default method is always public.

Anonymous Inner Classes

- Define a class in place instead of in a separate file.
- An Anonymous class doesn't have a name. Since it doesn't have a name, it cannot have a constructor.
 - You define an anonymous class and create its object at the same time.
 - It's always created using the new operator as part of an expression.
- Why would you do this?
 - Logically group code in one place
 - Increase encapsulation
- Syntax:

```
new <interface-name or class-name> (<argument-list>) {  
    // Anonymous class body goes here  
}
```



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

Creating an anonymous class:

The new operator is used to create an instance of the anonymous class. It's followed by either an existing interface name or an existing class name. Note that the interface name or class name is not the name for the newly created anonymous class. Rather, it is an existing interface/class name. If an interface name is used, the anonymous class implements the interface. If a class name is used, the anonymous class inherits from the class. The <argument-list> is used only if the new operator is followed by a class name.

Anonymous Inner Class: Example

- Example method call with concrete class

```
20     // Call concrete class that implements StringAnalyzer  
21     ContainsAnalyzer contains = new ContainsAnalyzer();  
22  
23     System.out.println("==Contains==");  
24     Analyzer.searchArr(strList, searchStr, contains);
```

- Anonymous inner class example

```
22     Analyzer.searchArr(strList, searchStr,  
23         new StringAnalyzer(){  
24             @Override  
25             public boolean analyze(String target, String searchStr){  
26                 return target.contains(searchStr);  
27             }  
28         });
```

- The anonymous inner class, `StringAnalyzer`, and its object is created in place.



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

The example shows how an anonymous inner class can be substituted for an object. Assume the class `Analyzer` has a static method `searchArr` that performs analysis based on a set of sample strings and a search string. The method takes the following arguments:

- `String[] strList` (the sample strings)
- `String searchStr` (the search String)
- `StringAnalyzer contains` (determines how the search functions)

By having the third argument, you can pass functionality to the `searchArr` method. For example, the search could be for anywhere in the string or just at the start. It may have to match case or not.

Here is the source code for `ContainsAnalyzer` that checks if a string starts with the search string:

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String searchStr) {  
        return target.startsWith(searchStr);  
    }  
}
```

The anonymous inner class specifies no name but implements almost exactly the same code (but has slightly different search functionality). The syntax is a little complicated as the class is defined where a parameter variable would normally be.

The advantages of passing functionality to a method are explained further in the lesson “Functional Interfaces and Lambda Expressions.”

Summary

In this lesson, you should have learned how to:

- Use `default` methods in interfaces
- Identify when it's desirable to implement a `default` method in an interface
- Identify how inheritance rules apply to methods implemented in interfaces
- Use `private` methods in interfaces
- Identify when it's desirable to implement a `private` method in an interface
- Define anonymous classes



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

Practice 4: Overview

This practice covers the following topics:

- Practice 4-1: Java SE 8 Default Methods
- Practice 4-2: Java SE 9 Private Methods



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

Quiz



What is true about code duplication?

- A. Duplication makes your code longer. This is good because it makes your colleagues believe you're really smart and capable of handling complex code.
- B. Duplication is good because it builds redundancy into the system.
- C. If you need to make an edit, you'll have to search for all cases where the code is duplicated. This is tedious and inefficient.



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

Quiz



The Accessible interface only verifies if a transaction is appropriate. Why can't this interface also store and modify an account's balance?

- A. `private` interface methods can't return `void`.
- B. Changing balance is a behavior. This could potentially lead to multiple inheritance of behavior.
- C. If the interface had a `balance` instance field, this could potentially lead to multiple inheritance of state.



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.

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.

Collections and Generics

5



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



ORACLE®

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 autoboxing and auto-unboxing feature
- Create a generic method and class
- Describe Java Collections framework
- Create different Collection implementations
- Order collections
- Explain wildcards in generics
- Describe Convenience methods for Collections



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

Type-Wrapper Classes

- Each primitive type has a corresponding type-wrapper class (in `java.lang` package).
 - Enables you to manipulate primitive-type values as objects
 - This is important, because the various implementations of the Collections manipulate and share objects—they cannot manipulate variables of primitive types.
- The following table lists wrapper classes:

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



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

Autoboxing and Auto-Unboxing

- Java provides boxing and unboxing conversions that automatically convert between primitive-type values and type-wrapper objects.
 - Boxing:** Converts a primitive value to an object of the corresponding type-wrapper class.
 - Unboxing:** Converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions are performed automatically and are called as autoboxing and auto-unboxing.

```
Integer[] integerArray = new Integer[5]; // create an array of Integer  
integerArray[0] = 10; // Autoboxing, assign primitive 10 to Integer array  
int value = integerArray[0]; // Auto-unboxing, get int value of Integer
```



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

Autoboxing and auto-unboxing are Java language features that enable you to make assignments without formal casting syntax. The casting is performed automatically by the compiler.

Note: Be careful when using autoboxing in a loop. There is a performance cost to using this feature.

Generic Methods

Consider a scenario where you have many overloaded methods, i.e., methods performing the same operation but differing in the argument type. For example:

```
public static void main(String args[]){
    Integer[] intArray = {50, 10, 20, 100, 50};
    Character[] charArray = {'J', 'A', 'V', 'A'};

    public static void displayArray(Integer[] inputArray) {
        for (Integer element : inputArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }

    public static void displayArray(Character[] inputArray) {
        for (Character element : inputArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }
}
```



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

A Generic Method

You can instead replace the overloaded methods by a generic method.

- For example, `displayArray` method in which actual type names are replaced with a generic type name (in this case `T`).

```
public static void displayArray(T [] inputArray) {  
    for (T element : inputArray) {  
        System.out.printf("%s ", element);  
    }  
    System.out.println();  
}
```

- The compiler infers the type argument based on the type of the actual arguments passed.



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

Generic Classes

- Generic classes provide a means to implement a class in a type-independent manner.
 - You can then instantiate type-specific objects of the generic class.
 - The compiler ensures the type safety of your code.

```
public class CacheString {  
    private String message;  
    public void add(String message){  
        this.message = message;  
    }  
    public String get(){  
        return this.message;  
    }  
}
```

```
public class CacheShirt {  
    private Shirt ;  
    public void add(Shirt shirt){  
        this.shirt = shirt;  
    }  
    public Shirt get(){  
        return this.shirt;  
    }  
}
```



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

Generic Cache Class

```
public class CacheAny <T>{  
    private T ;  
    public void add(T t) {  
        this.t = t;  
    }  
  
    public T get(){  
        return this.t;  
    }  
}
```



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

To create a generic version of the CacheAny class, a variable named `T` is added to the class definition surrounded by angle brackets. In this case, `T` stands for “type” and can represent any type. As the example in the slide shows, the code has changed to use `t` instead of a specific type of information. This change allows the CacheAny class to store any type of object.

`T` was chosen not by accident but by convention. Specific letters are commonly used with generics.

Note: You can use any identifier you want. The following values are merely strongly suggested.

Here are the conventions:

- `T`: Type
- `E`: Element
- `K`: Key
- `V`: Value
- `S, U`: Used if there are second types, third types, or more

Testing the Generic Cache Class

```
public static void main(String args[]){
    CacheString myMessage = new CacheString(); // Type
    CacheShirt myShirt = new CacheShirt(); // Type

    //Generics
    CacheAny<String> myGenericMessage = new CacheAny<String>();
    CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();

    myMessage.add("Save this for me"); // Type
    myGenericMessage.add("Save this for me"); // Generic

}
```



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

Note how the one generic version of the class can replace any number of type-specific caching classes. The `add()` and `get()` functions work exactly the same way. In fact, if the `myMessage` declaration is changed to generic, no changes need to be made to the remaining code.

The example code can be found in the Generics project in the `TestCacheAny.java` file.

Generics with Type Inference Diamond Notation

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>).
- The compiler determines the type arguments from the context.
- The pair of angle brackets <> is called the **diamond** notation.
- For example, you can create an instance of CacheAny<String> with the following statement:

```
CacheAny<String> myMessage = new CacheAny<>();
```



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

The type inference diamond is a new feature in JDK 7. In the generic code, notice how the right-side type definition is always equivalent to the left-side type definition. In JDK 7, you can use the diamond to indicate that the right type definition is equivalent to the left. This helps to avoid typing redundant information over and over again.

Example: TestCacheAnyDiamond.java

Note: In a way, it works in an opposite way from a “normal” Java type assignment. For example, Employee emp = new Manager(); makes emp object an instance of Manager.

But in the case of generics:

```
ArrayList<Manager> managementTeam = new ArrayList<>();
```

The left side of the expression (rather than the right side) determines the type.

Java SE 9: Diamond Notation with Anonymous Inner Classes

- Prior to Java SE 9, you need to specify class name in angular brackets for inner classes, but in Java SE 9 diamond operator is added for inner class as well.
- For example:

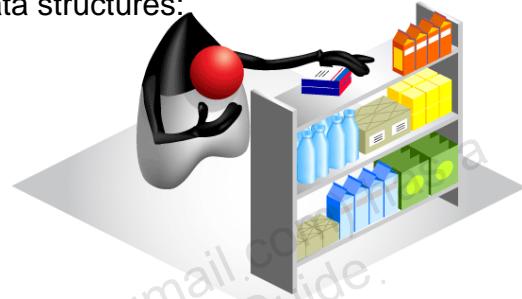
```
public class HelloDemo {  
  
    public static void main(String args[]) {  
  
        // Hello<String> hello = new Hello<String>("Prior to Java SE 9") {  
            Hello<String> hello = new Hello<>("Java SE 9") {  
                @Override  
                void hello() {  
                    System.out.println("Hello " + name);  
                }  
            };  
            hello.hello();  
        }  
    }  
}
```



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

Collections

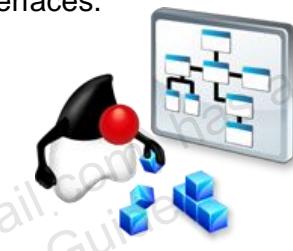
- A collection is a single object designed to manage a group of objects.
- Objects in a collection are called *elements*.
 - Used to store, retrieve, and manipulate aggregate data.
 - Represent data items that form a natural group, for example:
 - A mail folder (a collection of letters)
 - A telephone directory (a mapping of names to phone numbers).
- Various collection types implement many common data structures:
 - Stack, queue, tree, and linked list
 - The Collections API relies heavily on generics for its implementation.



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

Collections Framework in Java

- A collections framework is a unified architecture for representing and manipulating collections.
- All collections frameworks contain the following:
 - Interfaces: Allow collections to be manipulated independently of the details of their representation.
 - Implementations: These are the concrete implementations of the collection interfaces; that is, they are data structures.
 - Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.



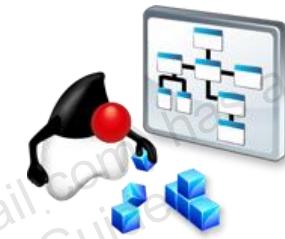
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Collections classes are all stored in the `java.util` package. The `import` statements are not shown in the following examples, but the `import` statements are required for each collection type:

- `import java.util.List;`
- `import java.util.ArrayList;`
- `import java.util.Map;`

Benefits of the Collections Framework

- The Java Collections Framework provides the following benefits:
 - Reduces programming effort
 - Increases program speed and quality
 - Reduces effort to design new APIs
 - Fosters software reuse

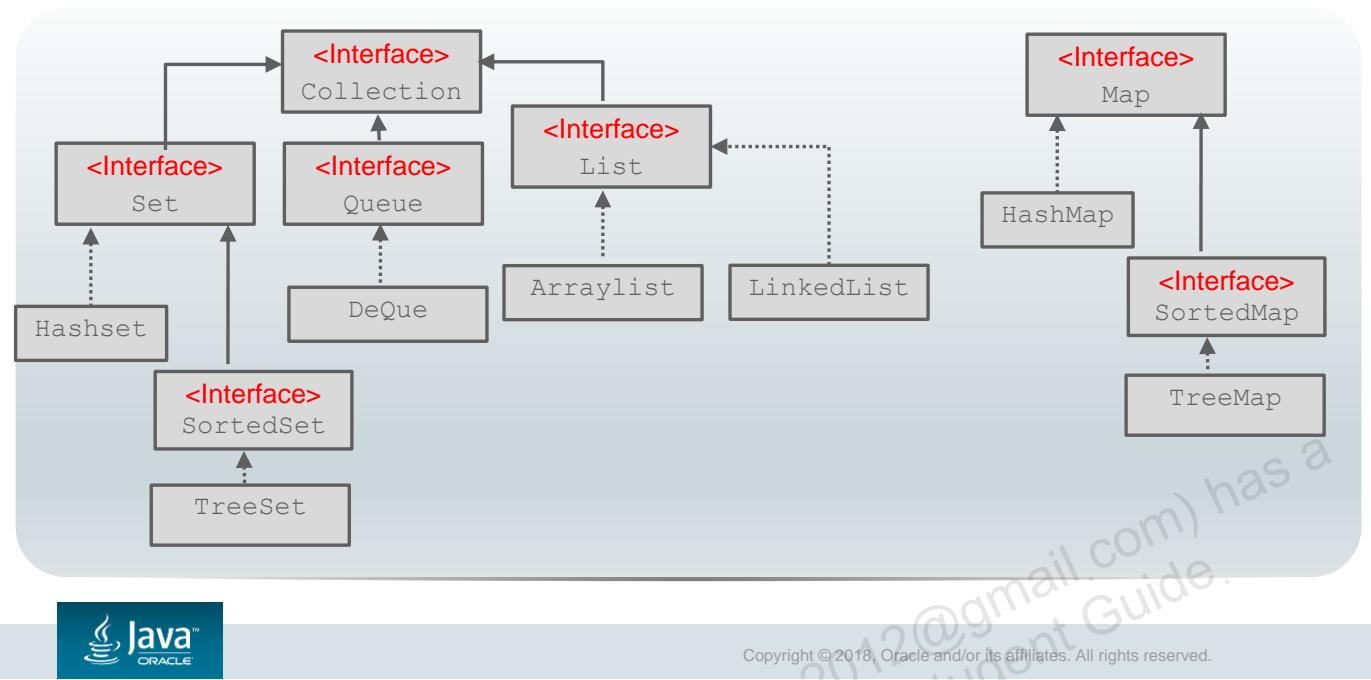


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Collection Types



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

The diagram in the slide shows the main interfaces of the Collection framework. The framework is made up of a set of interfaces and some of their implementations for working with a group (collection) of objects.

Characteristics of the Collection Framework

List, Set, Queue, and Map are interfaces in Java, and many concrete implementations of them are available in the Collections API.

The Map interface is a separate inheritance because it doesn't extend the Collection interface because it represents mappings and not a collection of objects.

Key Collections Interfaces

Interfaces	Implementations
<ul style="list-style-type: none">• List<ul style="list-style-type: none">– Elements are ordered according to how they're added.– Elements are searchable by index.– Duplicates are allowed.	<ul style="list-style-type: none">– ArrayList– LinkedList
<ul style="list-style-type: none">• Set<ul style="list-style-type: none">– Elements cannot be searched by index.– Duplicates are not allowed.	<ul style="list-style-type: none">– HashSet– LinkedHashSet– TreeSet
<ul style="list-style-type: none">• Map<ul style="list-style-type: none">– Elements are a key/value pair.– Duplicates are not allowed.	<ul style="list-style-type: none">– HashMap– LinkedHashMap– HashTable– TreeMap
<ul style="list-style-type: none">• Queue<ul style="list-style-type: none">– First-in, first-out or Last-in, first-out collection that models a waiting line.	<ul style="list-style-type: none">– Deque



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

The table in the slide shows the commonly used interfaces and their popular implementation.

ArrayList

- Is an implementation of the `List` interface
 - The list automatically grows if elements exceed initial size.
- Has a numeric index
 - Elements are accessed by index.
 - Elements can be inserted based on index.
 - Elements can be overwritten.
- Allows duplicate items

```
List<Integer> partList = new ArrayList<>(3);  
partList.add(new Integer(1111));  
partList.add(new Integer(2222));  
partList.add(new Integer(3333));  
partList.add(new Integer(4444)); // ArrayList auto grows  
System.out.println("First Part: " + partList.get(0)); // First item  
partList.add(0, new Integer(5555)); // Insert an item by index
```



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

ArrayList Without Generics

```
public class OldStyleArrayList {  
    public static void main(String args[]){  
        List partList = new ArrayList(3);  
  
        partList.add(new Integer(1111));  
        partList.add(new Integer(2222));  
        partList.add(new Integer(3333));  
        partList.add("Oops a string!");  
  
        Iterator elements = partList.iterator();  
        while (elements.hasNext()) {  
            Integer partNumberObject = (Integer)(elements.next()); //  
error?  
            int partNumber = partNumberObject.intValue();  
  
            System.out.println("Part number: " + partNumber);  
        }  
    }  
}
```

Runtime error:
ClassCastException



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

In the example in the slide, a part number list is created by using an `ArrayList`. There is no type definition when using syntax prior to Java version 1.5. So any type can be added to the list as shown on line 8. It is up to the programmer to know what objects are in the list and in what order. If the list was only for `Integer` objects, a runtime error would occur on line 12.

On lines 10–16, with a nongeneric collection, an `Iterator` is used to iterate through the list of items. Notice that a lot of casting is required to get the objects back out of the list so you can print the data.

In the end, there is a lot of needless “syntactic sugar” (extra code) working with collections in this way.

If the line that adds the `String` to the `ArrayList` is commented out, the program produces the following output:

```
Part number: 1111  
Part number: 2222  
Part number: 3333
```

Generic ArrayList

```
public class GenericArrayList {  
    public static void main(String args[]) {  
        List<Integer> partList = new ArrayList<>(3);  
  
        partList.add(new Integer(1111));  
        partList.add(new Integer(2222));  
        partList.add(new Integer(3333));  
        partList.add("Bad Data"); // compiler error now  
  
        Iterator<Integer> elements = partList.iterator();  
        while (elements.hasNext()) {  
            Integer partNumberObject = elements.next();  
            int partNumber = partNumberObject.intValue();  
  
            System.out.println("Part number: " + partNumber);  
        }  
    }  
}
```

No cast required.



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

With generics, things are much simpler. When the `ArrayList` is initialized on line 3, any attempt to add an invalid value (line 8) results in a compile-time error.

Note: On line 3, the `ArrayList` is assigned to a `List` type. Using this style enables you to swap out the `List` implementation without changing other code.

TreeSet: Implementation of Set

```
public class SetExample {  
    public static void main(String[] args) {  
        Set<String> set = new TreeSet<>();  
  
        set.add("one");  
        set.add("two");  
        set.add("three");  
        set.add("three"); // not added, only unique  
  
        for (String item:set){  
            System.out.println("Item: " + item);  
        }  
    }  
}
```



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

Set Interface :

A Set is an interface that contains only unique elements.

A Set has no index.

Duplicate elements are not allowed.

You can iterate through elements to access them.

TreeSet provides sorted implementation.

The example in the slide uses a TreeSet, which sorts the items in the set. If the program is run, the output is as follows:

Item: one

Item: three

Item: two

Map Interface

- A collection that stores multiple key-value pairs
 - Key: Unique identifier for each element in a collection
 - Value: A value stored in the element associated with the key
- Called “associative arrays” in other languages

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt



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

TreeMap: Implementation of Map

```
public class MapExample {  
  
    public static void main(String[] args){  
        Map <String, String> partList = new TreeMap<>();  
        partList.put("S001", "Blue Polo Shirt");  
        partList.put("S002", "Black Polo Shirt");  
        partList.put("H001", "Duke Hat");  
  
        partList.put("S002", "Black T-Shirt"); // Overwrite value  
        Set<String> keys = partList.keySet();  
  
        System.out.println("== Part List ==");  
        for (String key:keys){  
            System.out.println("Part#: " + key + " " +  
                partList.get(key));  
        }  
    }  
}
```



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

Some of the key implementation classes include:

- **TreeMap**: A map where the keys are automatically sorted
- **Hashtable**: A classic associative array implementation with keys and values. Hashtable is synchronized.
- **HashMap**: An implementation just like Hashtable except that it accepts null keys and values. Also, it is not synchronized.

The example in the slide shows how to create a Map and perform standard operations on it. The output from the program is:

```
== Part List ==  
Part#: H002 Duke Hat  
Part#: S001 Blue Polo Shirt  
Part#: S002 Black T-Shirt
```

Stack with Deque: Example

```
public class MapExample {  
  
    public static void main(String[] args){  
        Map <String, String> partList = new TreeMap<>();  
        partList.put("S001", "Blue Polo Shirt");  
        partList.put("S002", "Black Polo Shirt");  
        partList.put("H001", "Duke Hat");  
  
        partList.put("S002", "Black T-Shirt"); // Overwrite value  
        Set<String> keys = partList.keySet();  
  
        System.out.println("== Part List ==");  
        for (String key:keys){  
            System.out.println("Part#: " + key + " " +  
                partList.get(key));  
        }  
    }  
}
```



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

Ordering Collections

- The Comparable and Comparator interfaces are used to sort collections.
 - Both are implemented by using generics.
- Using the Comparable interface:
 - Overrides the compareTo method
 - Provides only one sort option
- The Comparator interface:
 - Is implemented by using the compare method
 - Enables you to create multiple Comparator classes
 - Enables you to create and use numerous sorting options



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

The Collections API provides two interfaces for ordering elements: Comparable and Comparator.

- **Comparable:** Is implemented in a class and provides a single sorting option for the class
- **Comparator:** Enables you to create multiple sorting options. You plug in the designed option whenever you want

Both interfaces can be used with sorted collections, such as TreeSet and TreeMap.

Comparable: Example

```
public class ComparableStudent implements Comparable<ComparableStudent>{
    private String name; private long id = 0; private double gpa = 0.0;

    public ComparableStudent(String name, long id, double gpa) {
        // Additional code here
    }
    public String getName() { return this.name; }
    // Additional code here

    public int compareTo(ComparableStudent s) {
        int result = this.name.compareTo(s.getName());
        if (result > 0) { return 1; }
        else if (result < 0){ return -1; }
        else { return 0; }
    }
}
```



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

The example in the slide implements the Comparable interface and its compareTo method. Notice that because the interface is designed by using generics, the angle brackets define the class type that is passed into the compareTo method. The if statements are included to demonstrate the comparisons that take place. You can also merely return a result.

The returned numbers have the following meaning.

- **Negative number:** s comes before the current element.
- **Positive number:** s comes after the current element.
- **Zero:** s is equal to the current element.

In cases where the collection contains equivalent values, replace the code that returns zero with an additional code that returns a negative or positive number.

Comparable Test: Example

```
public class TestComparable {  
    public static void main(String[] args) {  
        Set<ComparableStudent> studentList = new TreeSet<>();  
  
        studentList.add(new ComparableStudent("Thomas Jefferson",  
1111, 3.8));  
        studentList.add(new ComparableStudent("John Adams", 2222,  
3.9));  
        studentList.add(new ComparableStudent("George Washington",  
3333, 3.4));  
  
        for(ComparableStudent student:studentList){  
            System.out.println(student);  
        }  
    }  
}
```



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

In the example in the slide, an `ArrayList` of `ComparableStudent` elements is created. After the list is initialized, it is sorted by using the `Comparable` interface. The output of the program is as follows:

Name: George Washington ID: 3333 GPA:3.4

Name: John Adams ID: 2222 GPA:3.9

Name: Thomas Jefferson ID: 1111 GPA:3.8

Note: The `ComparableStudent` class has overridden the `toString()` method.

Comparator Interface

- Is implemented by using the `compare` method
- Enables you to create multiple Comparator classes
- Enables you to create and use numerous sorting options



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

The example in the next slide shows how to use `Comparator` with an unsorted interface such as `ArrayList` by using the `Collections` utility class.

Comparator: Example

```
public class StudentSortName implements Comparator<Student>{  
    public int compare(Student s1, Student s2){  
        int result = s1.getName().compareTo(s2.getName());  
        if (result != 0) { return result; }  
        else {  
            return 0; // Or do more comparing  
        }  
    }  
}
```

```
public class StudentSortGpa implements Comparator<Student>{  
    public int compare(Student s1, Student s2){  
        if (s1.getGpa() < s2.getGpa()) { return 1; }  
        else if (s1.getGpa() > s2.getGpa()) { return -1; }  
        else { return 0; }  
    }  
}
```

Here the compare logic is reversed and results in descending order.



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

The example in the slide shows the Comparator classes that are created to sort based on Name and GPA. For the name comparison, the if statements have been simplified.

Comparator Test: Example

```
public class TestComparator {  
    public static void main(String[] args){  
        List<Student> studentList = new ArrayList<>(3);  
        Comparator<Student> sortName = new StudentSortName();  
        Comparator<Student> sortGpa = new StudentSortGpa();  
  
        // Initialize list here  
  
        Collections.sort(studentList, sortName);  
        for(Student student:studentList){  
            System.out.println(student);  
        }  
  
        Collections.sort(studentList, sortGpa);  
        for(Student student:studentList){  
            System.out.println(student);  
        }  
    }  
}
```



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

The example in the slide shows how the two Comparator objects are used with a collection.

Note: Some code has been commented out to save space.

Notice how the Comparator objects are initialized on lines 4 and 5. After the sortName and sortGpa variables are created, they can be passed to the sort() method by name. Running the program produces the following output.

```
Name: George Washington  ID: 3333  GPA:3.4  
Name: John Adams      ID: 2222  GPA:3.9  
Name: Thomas Jefferson  ID: 1111  GPA:3.8  
Name: John Adams      ID: 2222  GPA:3.9  
Name: Thomas Jefferson  ID: 1111  GPA:3.8  
Name: George Washington  ID: 3333  GPA:3.4
```

Note

- The Collections utility class provides a number of useful methods for various collections. Methods include min(), max(), copy(), and sort().
- The Student class has overridden the toString() method.

Wildcards

- In generics code, the question mark (?) is called the wildcard and represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable.
- For example, the `sum` method totals the elements in a `List` using a wildcard in the `List` parameter.

```
public static double sum(List<? extends Number> list) {  
    double total = 0; //  
    for (Number element : list) {  
        total += element.doubleValue();  
    }  
    return total;  
}
```



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

Wildcards: Upper Bound

- In the `sum` method, `List<? extends Number>` indicates the wildcard extends class `Number`, which means that the wildcard has an **upper bound** of `Number`.
- Thus, the unknown-type argument, `?` must be either `Number` or a subclass of `Number`.
- With the wildcard type argument, method `sum` can receive an argument a `List` containing any type of `Number`, such as a `List<Integer>`, `List<Double>`, or `List<Number>`.

```
public static void main(String[] args) {  
    Integer[] integers = {1, 2, 3, 4, 5};  
    List<Integer> integerList = new ArrayList<>();  
    //Insert elements into the integerList and then invoke sum method  
    sum(integerList);  
  
    Double[] doubles = {1.1, 3.3, 5.5};  
    List<Double> doubleList = new ArrayList<>();  
    //Insert elements into the doubleList and then invoke sum method  
    sum(doubleList);  
}
```



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

Why Use Generics?

In a nutshell, code that uses generics has many benefits over nongeneric code:

- Stronger type checks at compile time.
 - A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.
 - Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
- Elimination of casts.
 - The following code snippet with generics doesn't require casting:

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0); // no cast
```

- Enabling programmers to implement generic algorithms.

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.



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

Java SE 9: Convenience Methods for Collections

Java SE 9 adds new static convenience factory methods to interfaces `List`, `Set`, and `Map` that enables you to create small immutable collections; that is, they cannot be modified once they are created.

- Their goals are to:
 - Reduce boilerplate code
 - Improve readability
 - Improve performance
 - Get the same amount of work done by typing less code



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

Please refer to JEP 269 on convenience methods for collections.

The word *factory* indicates that these methods create objects. These methods are **convenient** because you simply pass the elements as arguments to a convenience factory method, which creates the collection and adds the elements to the collection for you.

In lessons on Advanced and Parallel streams, you'll see how using lambdas and streams with immutable entities can help you create “parallelizable” code that will run more efficiently on today's multi-core architectures.

of Convenience Method

- Java SE 8: Collections require one line of code to add each element. For example:

```
List<String> testList = new ArrayList<>();  
testList.add("A");  
testList.add("B");  
testList.add("C");  
testList.add("D");  
testList.add("E");
```

- Java SE 9: The same work is done in one line with the of method.

```
List<String> testList = List.of("A", "B", "C", "D", "E"); // List  
Set<String> testSet = Set.of("A", "B", "C", "D", "E"); // Set  
Map<String, Integer> testMap = Map.of("A", 1, "B", 2, "C", 3, "D", 4, "E", 5); // Map  
Key Value
```

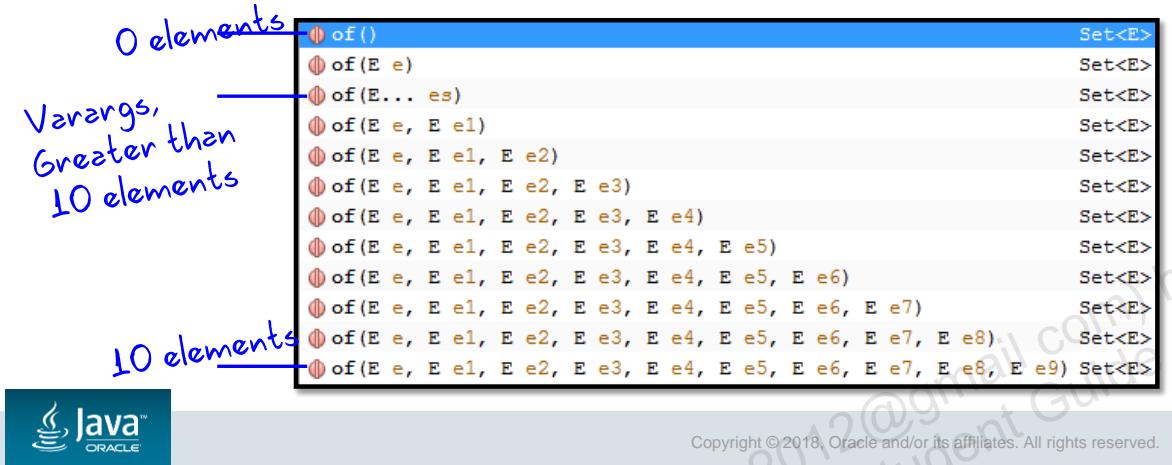


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

This slide demonstrates the convenience factory method, `of`, for a List, a Set, and a Map.

Overloading of Method

- Most variants return a collection of a specific size.
- The smallest returns a collection of 0 elements.
- The largest returns a collection of 10 elements.
- The varargs variant is used for all other sizes or greater than 10 elements.



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

Why Overload the `of` Method for zero to 10 elements?

The `of` method is overloaded for zero to 10 elements because research showed that these handle the vast majority of cases in which immutable collections are needed. *They eliminate the extra overhead of processing variable-length argument lists. This improves the performance of applications that create small immutable collections.*

For performance and space efficiency.

Varargs cause performance overhead relating to:

- Temporary array allocation
- Initialization
- Garbage collection

Specifying variants with 10 parameters or less covers the majority of use cases. This avoids the need to use the vararg variant and the associated performances overhead. This is the case for `Lists` and `Sets`. `Maps` are slightly different.

ofEntries Method for Maps

```
Map<String, Integer> testMap = Map.ofEntries(  
    entry("A", 1),  
    entry("B", 2),  
    entry("C", 3),  
    entry("D", 4),  
    entry("E", 5),  
    entry("F", 6),  
    entry("G", 7),  
    entry("H", 8),  
    entry("I", 9),  
    entry("J", 10),  
    entry("K", 11));
```



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

This slide demonstrates creating a Map using method `ofEntries` for more than 10 key and value pairs.

If a Map needs more than 10 elements, key and value pairs must be boxed as an entry.

There is no varargs variant for the `of` method in the Map interface. Instead, `ofEntries` accommodates returning a Map of indeterminate size. `ofEntries` is a static factory method of the Map interface.

Features of Convenience Methods

- **Immutability:**
 - `of` and `ofEntries` return immutable collections.
 - Methods like `add`, `set`, and `remove` throw `UnsupportedOperationException`.
 - It's thread-safe.
- **No null values:**
 - Null values are disallowed as `List` or `Set` elements, `Map` keys or values. This avoids later `NullPointerException`.
- **No duplicates:**
 - Duplicates in Sets and Maps cause an `IllegalArgumentException`.

```
Set<String> testSet = Set.of("A", "B", null, "D", "E");
```



```
Set<String> testSet = Set.of("A", "B", "A", "D", "E");
```



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

Summary

In this lesson, you should have learned how to:

- Describe autoboxing and auto-unboxing feature
- Create a generic method and class
- Describe Java Collections framework
- Create different Collection implementations
- Order collections
- Explain wildcards in generics
- Describe Convenience methods for Collections



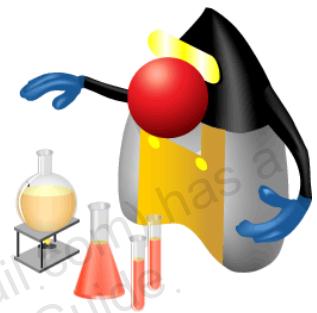
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice 5: Overview

This practice covers the following topics:

- 5-1: Creating a map to store a part number and count
- 5-2: Using convenience method `of`
- 5-3: Using convenience method `ofEntries`



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

Quiz



Which of the following is *not* a conventional abbreviation for use with generics?

- A. T: Table
- B. E: Element
- C. K: Key
- D. V: Value



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

Quiz



Which interface would you use to create multiple sort options for a collection?

- A. Comparable
- B. Comparison
- C. Comparator
- D. Comparinator



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

Quiz



The `of` convenience method is overloaded. How many variants of the `of` method exist in the `Set` interface?

- A. 2
- B. 3
- C. 4
- D. 8
- E. 12



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

Quiz



Which two are important consideration when deciding to use an `of` factory method to create a `List`?

- A. Whether elements need to be added to the `List` later
- B. Whether you will have more than 10 elements
- C. Whether the `List` may contain a null value
- D. Whether the `List` should contain duplicates



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.

Functional Interfaces and Lambda Expressions

6



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 do the following:

- Explain functional programming concepts
- Define a functional interface
- Define a predicate
- Describe how to pass a function as a parameter
- Define a lambda expression
- Define a statement lambda
- Describe lambda parameters
- Describe local-variable syntax for lambda parameters



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



Problem Statement

- Given a list of Person objects, perform operations like making an automated call to selected Persons by filtering the List based on selection criteria like age or gender.
- The Person class has the following properties:

```
public class Person {  
    private String givenName;  
    private String surName;  
    private int age;  
    private Gender ;  
    private String eMail;  
    private String phone;  
    private String address;  
    private String city;  
    private String state;  
    private String code;
```



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

The Person class also includes getters and setters for each field and code that implements the Builder pattern. The code in the slide only shows the fields.

RoboCall Class

- The RoboCall class has the following properties:

```
public class RoboCall {  
  
    public static robocall(String number) {  
  
        /*Code to place an automated call.  
         This code will connect to the phone system  
         using the supplied number and place the call.  
        */  
    }  
}
```



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

RoboCall is a class that has a static method that can place an automated phone call to a supplied phone number. The program that uses this class needs to filter the list of Person objects based on the selection criteria and then pass the filtered Persons' phone number to the RoboCall.robocall() method which will place the automated call.

RoboCall Every Person

```
void robocallEveryPerson() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        String num=p.getPhoneNumber();  
        RoboCall.robocall(num);  } }
```



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

RoboCall Use Case: Eligible Drivers

```
void robocallEligibleDrivers() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        if(p.getAge()>= 16){  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    }  } }
```



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

RoboCall Use Case: Eligible Voters

```
void robocallEligibleVoters() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        if(p.getAge()>= 18){  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    }  } }
```



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

RoboCall Use Case: Legal Drinking Age

```
void robocallLegalDrinkers() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        if(p.getAge()>= 21){  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    }  } }
```

- Problem: Notice the code in these three use cases; there is duplicated code, i.e. same boiler plate code with some tweaks inside!



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

Call only persons over 21.

Solution: Parameterization of Values

Add a parameter, age.

```
public static void robocallPersonOlderThan(int age) {  
    for(Person p: pl){  
        if(p.getAge() >= age){  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    }  }}
```

Age is
parameterized
and the
constant value
for age is
replaced.



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

age is passed as a parameter instead of a constant; you are passing a value.

Solution: Parameterized Methods

Add a parameter, age.

```
void robocallEligibleDrivers () {  
    robocallPersonOlderThan (16); }  
  
void robocallEligibleVoters () {  
    robocallPersonOlderThan (18); }  
  
void robocallLegalDrinkers () {  
    robocallPersonOlderThan (21); }
```

Removes the
duplication seen
earlier.



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

Parameters for Age Range

Additional Use Case: To enroll into the National Service Program, the age range is 18 to 25 (inclusive).

```
public static void robocallPersonsInAgeRange(int low, int high) {  
    for (Person p : pl) {  
        if (low <= p.getAge() && high < p.getAge()) {  
            String num = p.getPhoneNumber();  
            RoboCall.robocall(num);    } } }
```



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

The code has been made more flexible by now looking at an age range.

Using Parameters for Age Range

```
void robocallEligibleDrivers () {  
    robocallPersonsInAgeRange (16,MAX); }  
  
void robocallEligibleVoters () {  
    robocallPersonsInAgeRange (18,MAX); }  
  
void robocallLegalDrinkers () {  
    robocallPersonsInAgeRange (21,MAX); }  
  
void robocallSelectiveService () {  
    robocallPersonsInAgeRange (18,26); }
```

→ INTEGER.MAX_VALUE



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

This is a good example of *delegation*. A method is called and that method determines what other method to call and with what parameters. The caller is only aware of the methods on this page.

Corrected Use Case

To enroll into the National Service Program, age range is 18 to 25 (inclusive) and only **men** can enroll!

```
enum GENDER{MALE, FEMALE}

void robocallSelectiveService () {
    robocallPersonInRange(18,25,MALE);
}
```

This works, but what about queries
that don't care about gender?



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

Parameterized Computation

- Value parameterization is good, but only up to a point.
- Problems:
 - Values must be in the given range
 - Sometimes some special values work: 0, -1, null, INTEGER.MAX_VALUE
 - Add more boolean or enum parameters
- Paradigm Shift: Parameterize a method's **behavior** instead of its values:
 - Make a method's parameter a **function** instead of a value



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

How To Pass a Function in Java?

Ideally pass the function - the actual behavior itself - `p.getAge ()>= someValue`

Instead of parametrizing values, you parametrize the behavior.

```
/* For each Person in the list, perform a test: are they within an age  
range? Are they Male or Female? Are they in a certain city. And there could  
be many, currently unidentified, tests needed in the future as well.  
If they pass the test, get their phone number and Robocall their number. */  
  
// How would it look to pass the function as an argument?  
void robocallEligibleDrivers(){  
    robocallPersonOlderThan(<a function to test if a person is 16 or over>);}  
  
void robocallEligibleVoters(){  
    robocallPersonOlderThan(<a function to test if a person is 18 or over>);}  
  
// How can this be done in Java?
```



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

A better solution is to use the Functional Programming paradigm to handle situations where you find yourself adding more methods to cover more and more cases.

Prior to Java SE 8: Pass a Function Wrapped in an Object

```
public class DriverEligibilityTester {  
    public boolean isEligible(Person p) {  
        return p.getAge() >= 16;    }}  
  
// Pass isEligible method wrapped within an object  
DriverEligibilityTester eTester = new DriverEligibilityTester();  
robocallEligible(eTester);  
  
public static void robocallEligible(DriverEligibilityTester tester) {  
    for (Person p : pl) {  
        if (tester.isEligible(p)) { // call the wrapped method  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);        } } }  
// Notice you can use a more generic method to make the calls. By passing  
// the test condition as an argument instead of passing literal values,  
// the same method can be used for many different eligibility selections.
```



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

Prior to SE 8: Abstract Behavior With an Interface

While using a concrete class will work, it is more flexible and a better design to use an interface:

```
public interface EligiblePerson {  
    boolean isEligible(Person p);  
  
}  
  
public class DriverEligibilityTester implements EligiblePerson{  
    @Override  
    public boolean isEligible(Person p) {  
        return p.getAge() >= 16;  
    }  
  
}  
  
// main()...  
EligiblePerson eTester = new DriverEligibilityTester();  
robocallEligible(eTester);  
  
public static void robocallEligible(EligiblePerson tester) {  
    for (Person p : pl) {  
        if (tester.isEligible(p)) {  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);  
        }  
    }  
}
```

roboCallEligible() can accept any class that implements the EligiblePerson interface



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

Prior to SE 8: Replace Implementation Class with Anonymous Inner Class

```
//old way  
//EligiblePerson eTester = new EligibilityTester();  
//robocallEligible(eTester);  
  
// New way:  
robocallEligible(new EligiblePerson() {  
    public boolean isEligible(Person p) {  
        return p.getAge() >= 16;    }  });  
  
public static void robocallEligible(EligiblePerson tester) {  
    for (Person p : pl) {  
        if (tester.isEligible(p)) {  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    }  } }
```

Note what's different. The object being passed is now instantiated within the method call. It is an implementation of EligiblePerson but has no class name. It is an anonymous inner class.

There is a lot of "boilerplate" code – is it all needed? In SE 8 and later you can replace anonymous inner classes with a Lambda expression.



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

Lambda Solution: Replace Anonymous Inner Class with Lambda Expression

Java SE 8 introduced the Lambda syntax. A functional interface can now be expressed more simply and clearly.

This:

```
robocallEligibile( new EligiblePerson () {  
    public boolean isEligible(Person p){  
        return p.getAge() >= 16; }});
```

Can be replaced with this syntax:

```
robocallEligibile( Person p ) -> p.getAge() >= 16;  
  
public static void robocallEligibile(EligiblePerson tester) {  
    for (Person p : pl) {  
        if (tester.isEligible(p)) { ...
```



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

If the intent is to pass a method as a parameter, before SE 8 you would have needed to create an interface and the use an anonymous inner class to create the object to be passed at runtime. This anonymous class can be replaced with a lambda expression in SE 8.

The Java SE 8 JDK (compiler and runtime) understands and accepts a lambda expression for an anonymous inner class whose interface has one abstract method.

It's now easy to use a lambda expression to create much more complex filters for the person to be robocalled. What if you needed to call everyone who is 18-25 and lives in Denver?

```
(Person p) p-> getAge() >=18 && p.getAge() <=25 && p.getCity.equals("Denver");
```

As long as the return type is Boolean and takes a Person as an argument, any expression will work:

```
(Person p) p-> p.getSurName().length() > 10 characters // works
```

Rewriting the Use Cases Using Lambda

```
void robocallEligibleDrivers() {  
    robocallEligible(p -> p.getAge() >= 16); }  
  
void robocallEligibleVoters() {  
    robocallEligible(p -> p.getAge() >= 18); }  
  
void robocallLegalDrinkers () {  
    robocallEligible(p -> p.getAge() >= 21); }  
  
void robocallEligibleForService() {  
    robocallEligible(p -> p.getGender() = MALE &&  
                     p.getAge() >= 18 &&  
                     p.getAge() <= 25); }
```



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

What Is a Lambda?

- Essentially an anonymous function:
 - Allows one to treat data as a function
 - Provides parameterization of **behavior** as opposed to **values** or **type**
- Combined with JVM and class library changes, it supports:
 - Explicit but unobtrusive parallelism
 - High productivity
- Suitable for simple everyday programming and also heavy-lifting (parallel)



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

What is a Functional Interface

- A functional interface in Java is one that has only **one** method.
- For example:
 - Some widely used functional interfaces
 - `java.util.Comparator` has only `compare()` method
 - `java.awt.event.ActionListener` has only `actionPerformed(ActionEvent)` method
- `Arrays.sort()` will take an array of objects and sort them, providing the object implements the `Comparable` interface – has a method that compares elements to determine order.
- But, what if the object array **does not** implement `Comparable`?
 - Another `Arrays.sort()` method takes the object array and the sorting behavior as a parameter (a `Comparator`):

```
Arrays.sort((p1,p2)-> p1.getSurName().compareTo(p2.getSurName()))
```



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

Sorting collections is interesting to examine.

The default approach is that the object being sorted must implement `Comparable`. `Comparable` tells the sort method what the sorting behavior should be. This is quite an elegant approach, but doesn't handle objects that haven't implemented `Comparable`, and perhaps cannot be refactored to do so.

But there's another `sort()` method that can sort objects that *don't* implement `Comparable`. If the object doesn't implement `Comparable`, how can the sorting behavior be established? The answer is by passing in this *behavior* as a parameter.

This is done with the `Comparator` function. It's interesting that this bit of functional style programming has been part of Java since 1.2! It's just so useful that even though you have to use an anonymous class or concrete class to express the functionality in the `Comparator`, it's been the way to do it.

Then lambda really just makes this passing of functionality easy enough that it becomes useful, readable, and elegant for all kinds of designs. Streams would probably be pretty much unusable without lambda - certainly they'd lose their "fluent" description.

`Arrays.sort()` and `Collections.sort()` are part of the API, but any design situation where you find yourself adding more methods to cover more and more cases is a good example of where passing functionality is useful. Adding new methods is ugly and error-prone; at the very least adding a new method means changing more than one class. Using the functional approach instead is much better, the method doesn't need to change, only the caller of the method.

Which of These Interfaces Are Functional Interfaces?

```
public interface DoAddition{
    int add(int a, int b);
}

public interface SmartSubtraction{
int subtract(int a, int b);
int subtract(double a, double b);

}

public interface SmartAddition extends DoAddition{
    int add(double a, double b);
}

public interface NothingToSeeHere{
}
```

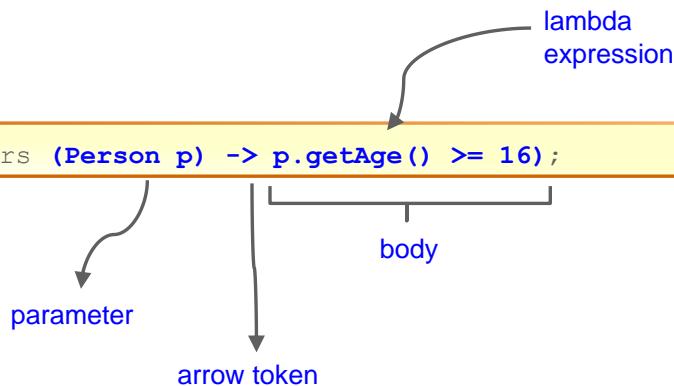


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The second is not a functional interface, as there are two abstract methods. The third is not as it adds a second method. The fourth is not because it has no methods.

Lambda Expression

```
robocallEligibleDrivers (Person p) -> p.getAge() >= 16;
```



- A lambda expression is like a method; it provides a list of formal parameters and a body.
- A lambda body is either a single expression or a block, expressed in terms of those parameters.
 - Like a method body, a lambda body describes code that will be executed whenever an invocation occurs.

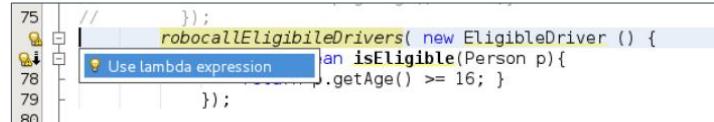


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The syntax of the lambda is based on the functional notation discussed earlier. When you write a lambda expression in Java SE 8 or later, it's converted into an instance of a functional interface.

Using Lambdas

- Lambdas work just like an interface or class and can be used anywhere you would use a functional interface.
- This syntax change can be detected by IDEs like NetBeans, and they can suggest it or an anonymous class if you prefer.



- Using Lambda expressions you can simplify your code and improve readability.
- They can even make frameworks and advanced and sophisticated functionality like streams feasible.



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

Methods (“functions”) can now be passed as parameters and called, referenced and passed by object references and “stored” within an object reference. How? The method (lambda expression) is referenced like an object.

Which of The Following Are Valid Lambda Expressions?

1. `() -> {}`
2. `() -> "Duke"`
3. `() -> {return "Kenny";}`
4. `(Integer i) -> return "Larry" + i;`
5. `(String s) -> {"IronMan";}`

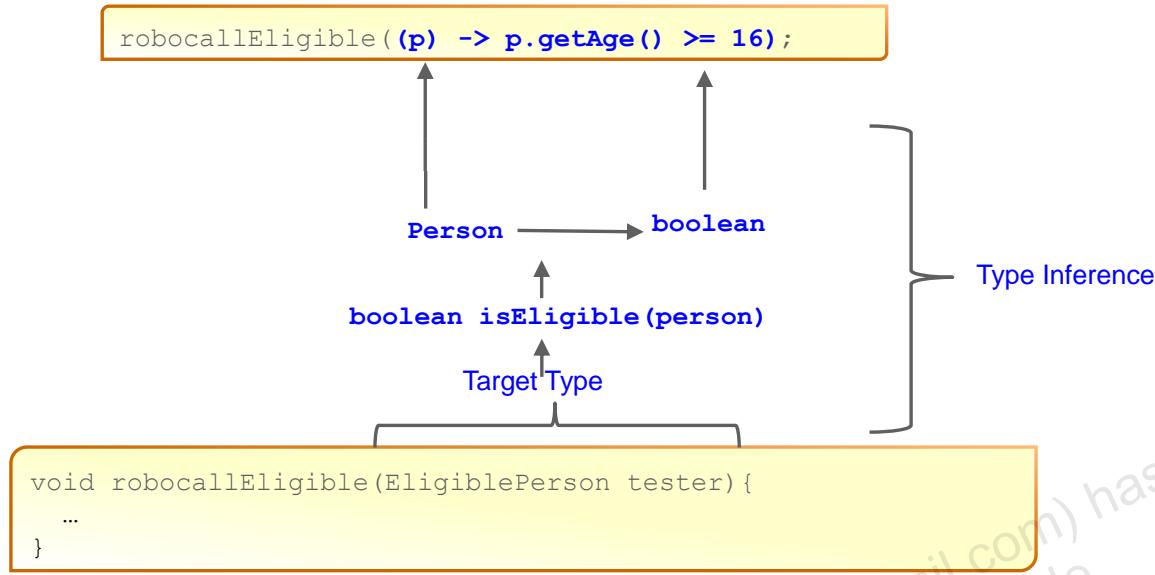


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Answer: Only 4 and 5 are invalid lambdas.

1. This lambda has no parameters and returns void. It's similar to a method with an empty body. For example:
`public void run() { }.`
2. This lambda has no parameters and returns a `String` as an expression.
3. This lambda has no parameters and returns a `String` (using an explicit return statement).
4. `return` is a control-flow statement. To make this lambda valid, curly braces are required as follows:
`(Integer i) -> {return "Larry" + i; }.`
5. `"IronMan"` is an expression, not a statement. To make this lambda valid, you can remove the curly braces and semicolon as follows:
`(String s) -> "IronMan".`
Or if you prefer, you can use an explicit return statement as follows:
`(String s) -> {return "IronMan"; }.`

Lambda Expression: Type Inference



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

What is the type of `p`?

The compiler performs the type inference as illustrated in the slide to determine the type of `p`. The compiler verifies that in the lambda expression the parameter to the left of the arrow: `p` is of the type and the expression to the right of the arrow returns a boolean value. You don't have to explicitly specify the type.

To Create a Lambda Expression

It's essentially an anonymous inner class without the boilerplate code.

Include the essentials: parameter list, block of code to execute and match the return type of the method.

```
// pass as a parameter to a method:  
robocallEligible((p, age) -> p.getAge() >= 16);  
  
// store in variable and pass as a variable to the method as a parameter.  
EligiblePerson eTest = (p) -> p.getAge() >= 16;  
robocallEligible(eTest);  
  
// store in an array  
EligiblePerson[] eTests = {(Person p) -> p.getAge() >= 16,  
                           (Person p) -> p.getAge() >= 18};  
robocallEligible(eTests[0]);
```



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

Getting started creating lambdas can be tricky. One technique that can help is to get your code working with an anonymous inner class first, then replace the boilerplate code with the lambda.

Examples of Lambdas

A boolean expression	(List<String> list) -> list.isEmpty() // Predicate
Creating objects	() -> new Person(10) // Supplier
Consuming from an object	(Person p) -> {System.out.println(p.getSurName());} // Consumer
Select/extract from an object	(String s) -> s.length()
Combine two values	(int a, int b) -> a * b //Function
Compare two objects	(Person p1, Person p2) -> p1.getAge().compareTo(p2.getAge())



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

Lambdas really just make this passing of functionality easy enough that it becomes useful, readable, and elegant for all kinds of designs.

Statement Lambdas: Lambda with Body

```
robocallMatchingPersons (p -> p.getAge ()>=16,  
                         num -> {System.out.println("Calling");  
                           robocall (num); });  
  
robocallMatchingPersons (p -> p.getAge ()>=18,  
                         num -> {System.out.println("Calling");  
                           txtMsg (num); });
```

These are called **statement** lambdas instead of **expression** lambdas as they contain a block; it's similar to executing an ordinary block of code in Java within braces.



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

Examples: Lambda Expression

```
(() -> {}           // No parameters; result is void  
() -> 42           // No parameters, expression body  
(int x) -> x+1     // Single declared-type parameter  
(x) -> x+1         // Single inferred-type parameter  
(int x, int y) -> x+y // Multiple declared-type parameters  
(x, y) -> x+y      // Multiple inferred-type parameters
```



Lambda Parameters

- The formal parameters of a lambda expression may have either declared types or inferred types.
 - It's not possible to declare the types of some of its parameters but leave others to be inferred.
 - Only parameters with declared types can have modifiers.

```
(x, int y) -> x+y      // Illegal: can't mix inferred and declared types  
(x, final y) -> x+y   // Illegal: no modifiers with inferred types
```



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

Local-Variable Syntax for Lambda Parameters

- This is a new feature introduced in Java SE 11.
- Allows `var` keyword to be used when declaring the formal parameters of implicitly typed lambda expressions.
- A lambda expression may be implicitly typed, where the types of all its formal parameters are inferred:
`(x, y) -> x.process(y) // implicitly typed lambda expression`
- Java SE 10 makes implicit typing available for local variables:

```
var x = new Foo();
for (var x : xs) { ... }
try (var x = ...) { ... } catch ...
```

- For uniformity with local variables, in Java SE 11 you can add `var` for the formal parameters of an implicitly typed lambda expression:

```
(var x, var y) -> x.process(y) // implicit typed lambda expression
```



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

Local-Variable Syntax for Lambda Parameters

- An implicitly typed lambda expression must use `var` for all its formal parameters or for none of them.
 - In addition, `var` is permitted only for the formal parameters of implicitly typed lambda expressions.
 - Explicitly typed lambda expressions continue to specify manifest types for all their formal parameters.
 - The following examples are illegal:

```
(var x, y) -> x.process(y) // Cannot mix 'var' and 'no var' in implicitly typed lambda expression
```

```
(var x, int y) -> x.process(y) // Cannot mix 'var' and manifest types in explicitly typed lambda expression
```



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

JEP 323: Local-Variable Syntax for Lambda Parameters:

<http://openjdk.java.net/jeps/323>

JEP 286: Local-Variable Type Inference

<http://openjdk.java.net/jeps/286>

Functional Interfaces: Predicate

You don't have to create a new interface each time you wish to use a lambda expression.

```
public interface Predicate<T> {  
    public boolean test(T t);  
  
    robocallEligible((Person p) -> p.getAge() >= 16);  
  
    public static void robocallEligible(Predicate pred) {  
        for (Person p : pl) {  
            if (pred.test(p))  
                RoboCall.robocall(p.getPhoneNumber()); } } }
```

The method signature of the lambda expression matches the interface, so it compiles.

- **Lambda method signature:** boolean methodName (Person p)
- **Required method signature:** boolean methodName (Type t)



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

Functional interfaces are covered in detail in the lesson “Built-in Functional Interfaces”.

Using Functional Interfaces

```
// DriverEligible interface is not needed!

robocallEligibile((Person p) -> p.getAge() >= 16); // Drivers
robocallEligibile((Person p) -> p.getAge() >= 18); //Voters
robocallEligibile((Person p) -> p.getCity() >= "Denver"); // Residents
robocallEligibile((Person p) -> p.getAge()>=18 && p.getAge()<=25); //age range

public static void robocallEligibile(Predicate pred) {
    for (Person p : pl) {
        if (pred.test(p)) {
            RoboCall.roboCall(p.getPhoneNumber()); } } }
// compiler needs a method signature that matches our lambda (and vice versa)
```



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

Using existing functional interfaces makes your code more robust and predictable. The method calls will always be passing an existing type. Also as you'll see in the lesson "Built-in Functional Interfaces", functional interfaces often have useful functionality expressed in their default methods.

Quiz



When a developer creates an anonymous inner class, the new class is typically based on which one of the following?

- A. enums
- B. Executors
- C. Functional interfaces
- D. Static variables



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

Quiz



Which is true about the parameters passed into this lambda expression:

`(t, s) -> t.contains(s)`

- A. Their type is inferred from the context.
- B. Their type is executed.
- C. Their type must be explicitly defined.
- D. Their type is undetermined.



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

Summary

In this lesson, you should have learned how to:

- Explain functional programming concepts
- Define a functional interface
- Define a predicate
- Describe how to pass a function as a parameter
- Define a lambda expression
- Define a statement lambda
- Describe lambda parameters
- Describe local-variable syntax for lambda parameters



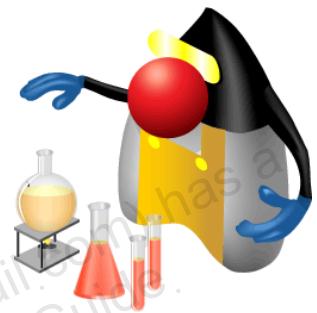
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice 6: Overview

This practice covers creating lambda expressions.

- Practice 6-1: Refactor Code to Use Lambda Expressions
- Practice 6-2: Refactor Code to Reuse Lambda Expressions



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.

Collections, Streams, and Filters

7



ORACLE®



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

Adolfo De+la+Rosa (adolfo.delarosa2022@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods
- Define pipelines in terms of lambdas and collections



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

Collections, Streams, and Filters

- Iterate through collections using forEach
- Streams and Filters



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

The RoboCall App

RoboCall app is used for automating the communication with groups of people.

- It can contact individuals by phone, email, or regular mail.
- In the lesson examples, the app will be used to contact three groups of people.
 - Drivers: Persons over the age of 16
 - Draftees: Male persons between the ages of 18 and 25
 - Pilots (specifically commercial pilots): Persons between the ages of 23 and 65
- It uses the `Person` class and creates the master list of persons you want to contact.
 - An `ArrayList` of `Person` objects is used for the examples that follow.



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

Collection Iteration and Lambdas

Iterating with `forEach` method.

```
public class RoboCallTest06 {  
  
    public static void main(String[] args) {  
  
        List<Person> pl = Person.createShortList();  
  
        System.out.println("\n==== Print List ====");  
        pl.forEach(p -> System.out.println(p));  
  
    }  
}
```



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

RoboCallTest07: Stream and Filter

```
public class RoboCallTest07 {  
  
    public static void main(String[] args){  
  
        List<Person> pl = Person.createShortList();  
        RoboCall105 robo = new RoboCall105();  
  
        System.out.println("\n==== Calling all Drivers Lambda ===");  
        pl.stream()  
            .filter(p -> p.getAge() >= 23 && p.getAge() <= 65)  
            .forEach(p -> robo.roboCall(p));  
    }  
}
```

Fluent
programming
style.

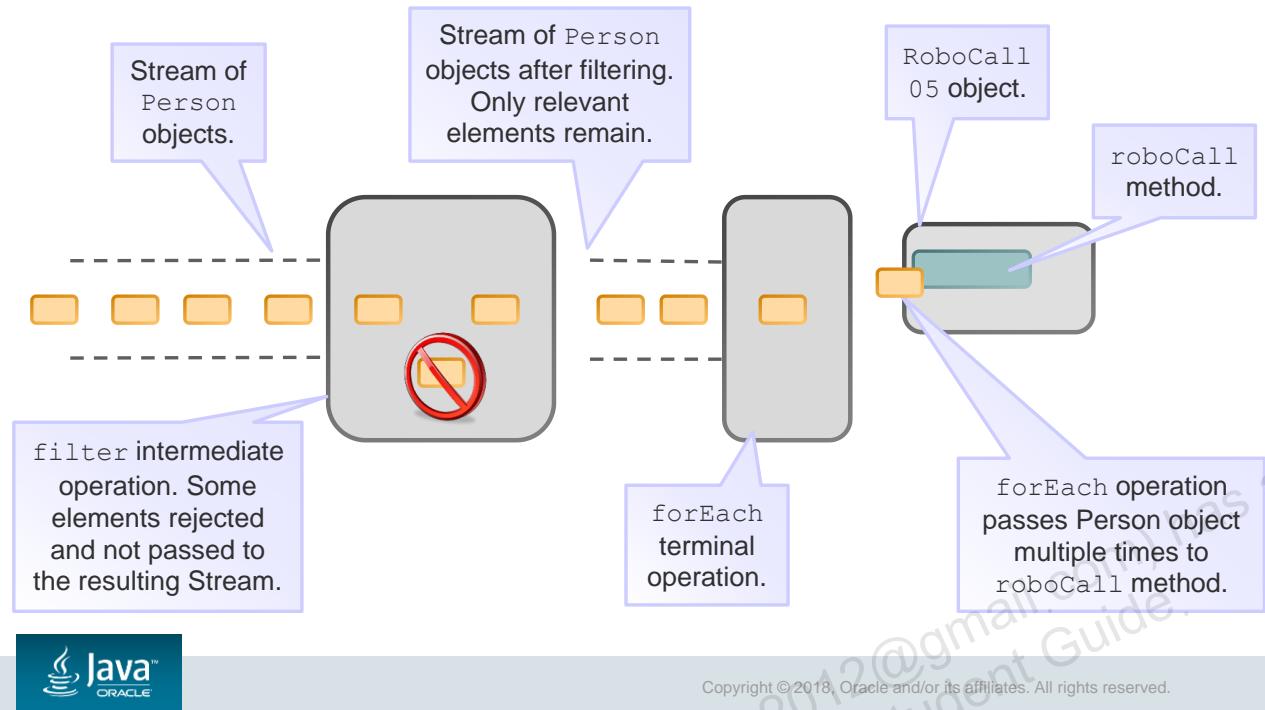


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adding the `stream()` method to the statement opens up a whole host of new operations on collections. The `filter()` method, shown in the slide, is an example. It takes a `Predicate` as a parameter and filters the result so that only collection elements that match the `Predicate` criteria are returned to `forEach`. Also, and this is very important, it returns a `Stream`, ensuring that further method calls can be made on the `Stream` by simply chaining them. This is fluent programming.

This is a big improvement on the looping shown in the previous code. First, the collection statement with a `stream` really describes what is happening (take this collection, filter out these elements, and return the results). Second, the looping methods in `RoboCall105` are no longer needed. The selection of elements and their output are handled in one statement.

Stream and Filter



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

RobocallTest08: Stream and Filter Again

```
public class RoboCallTest08 {  
    public static void main(String[] args){  
        List<Person> pl = Person.createShortList();  
        RoboCall05 robo = new RoboCall05();  
  
        // Predicates  
        Predicate<Person> allPilots =  
            p -> p.getAge() >= 23 && p.getAge() <= 65;  
  
        System.out.println("\n==== Calling all Drivers Variable ===");  
        pl.stream()  
            .filter(allPilots)  
            .forEach(p -> robo.roboCall(p));  
    }  
}
```



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

SalesTxn Class

- Class used in examples and practices to follow
- Stores information about sales transactions
 - Seller and buyer
 - Product quantity and price
- Implemented with a Builder class
- Buyer class
 - Simple class to represent buyers and their volume discount level
- Helper enumerations
 - BuyerClass: Defines volume discount levels
 - State: Lists the states where transactions take place
 - TaxRate: Lists the sales tax rates for different states



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

Java Streams

- Streams
 - `java.util.stream`
 - A sequence of elements on which various methods can be chained
- Method chaining
 - Multiple methods can be called in one statement
- Stream characteristics
 - They are immutable.
 - After the elements are consumed, they are no longer available from the stream.
 - A chain of operations can occur only once on a particular stream (a pipeline).
 - They can be sequential (default) or parallel.



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

The Filter Method

- The `Stream` class converts a collection to a pipeline:
 - Immutable data
 - Can only be used once and discarded
- The `filter` method uses a `Predicate` object (usually written as a lambda expression) to select items.
- Syntax:

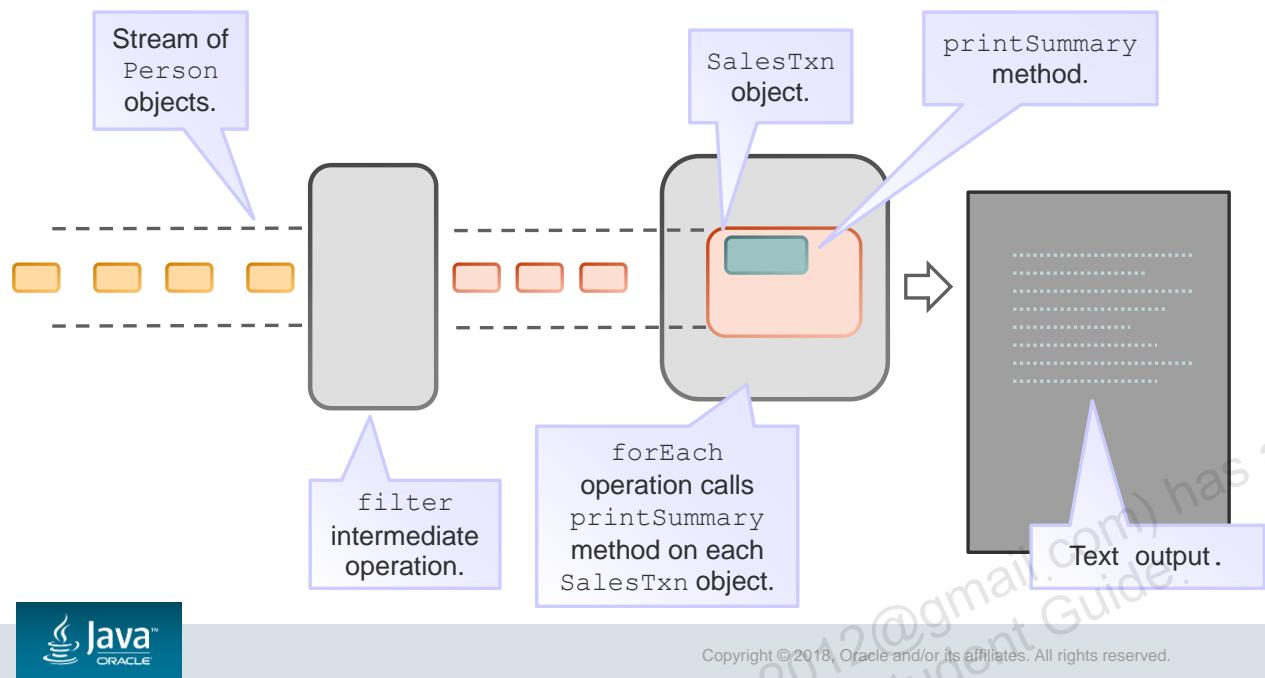
```
System.out.println("\n== CA Transactions Lambda ==");  
tList.stream()  
    .filter(t -> t.getState().equals("CA"))  
    .forEach(SalesTxn::printSummary);
```



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

The `filter` method takes a lambda expression as a parameter (actually a `Predicate` object) and filters the data based on the logical expression provided. The `Predicate` object is the target type of the filter. The `filter` method returns a new `Stream`, and the `forEach` method is then called on this `Stream`. The `forEach` method calls the `printSummary` method of `SalesTxn` on each element of the `Stream`.

Filter and SalesTxn Method Call



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

Method References

- In some cases, the lambda expression merely calls a class method.
 - `.forEach(t -> t.printSummary())`
- In these cases, you can use a method reference.
 - `.forEach(SalesTxn::printSummary);`
- You can use a method reference in the following situations:
 - Reference to a static method
 - `ContainingClass::staticMethodName`
 - Reference to an instance method
 - Reference to an instance method of an arbitrary object of a particular type (for example, `String::compareToIgnoreCase`)
 - Reference to a constructor
 - `ClassName::new`



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

In many situations, a method reference can be substituted for a lambda expression. If a lambda expression merely calls a method on that object, a Method Reference can be substituted.

Method Chaining

- Pipelines allow method chaining (like a builder).
- Methods include filter and many others.
- For example:

```
tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .filter(t -> t.getBuyer().getName().equals("Acme Electronics"))
    .forEach(SalesTxn::printSummary);
```



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

Developers are not limited to only one method call. Multiple methods can be chained together in a single statement. This is called "method chaining." The statement structure looks similar to that of the builder pattern, and it shares the approach that each method call returns an object on which further method calls can be made.

Analogy: If you think about it, these statements are very similar to SQL statements with `where` clauses. The syntax is different, but the idea is very similar; elements are not addressed individually; instead, the processing is aggregate.

Method Chaining

- You can use compound logical statements.
- You select what is best for the situation.

```
System.out.println("\n== CA Transactions for ACME ==");
tList.stream()
    .filter(t -> t.getState().equals("CA") &&
           t.getBuyer().getName().equals("Acme Electronics"))
    .forEach(SalesTxn::printSummary);

tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .filter(t -> t.getBuyer().getName().equals("Acme Electronics"))
    .forEach(SalesTxn::printSummary);
```

Compound
logical
expression.

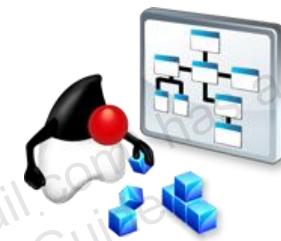
Method
chaining.



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

Pipeline Defined

- A stream pipeline consists of:
 - A source
 - Zero or more intermediate operations
 - One terminal operation
- Examples
 - Source: A Collection (could be a file, a stream, and so on)
 - Intermediate: filter, map
 - Terminal: forEach



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

In this lesson, streams have only been filtered and the results printed out. However, many more Stream methods can be chained, each one returning a Stream to be used by the next method call. Connecting these Stream operations together in a single statement is called a stream pipeline.

Note that while many Stream methods return a Stream of the same type, a method like map can return a Stream of a different type, and some methods return non-Stream types. For example, some methods return an Optional (though this can then be used to generate a Stream if that is required).

A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc.), zero or more intermediate operations (which transform a stream into another stream, such as filter(Predicate)), and a terminal operation (which produces a result or side-effect, such as count or forEach method). Streams are lazy; computation on the source data is performed only when the terminal operation is initiated, and source elements are consumed only as needed.

Summary

In this lesson, you should have learned how to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods together
- Define pipelines in terms of lambdas and collections



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

Practice 7: Overview

This practice covers the following topics:

- Practice 7-1: Update RoboCall to use Streams
- Practice 7-2: Mail Sales Executives using Method Chaining
- Practice 7-3: Mail Sales Employees over 50 using Method Chaining
- Practice 7-4: Mail Male Engineering Employees under 65 using Method Chaining



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.

Lambda Built-in Functional Interfaces



ORACLE®



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

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:

- List the main built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



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

Built-in Functional Interfaces

- Lambda expressions rely on functional interfaces
 - Important to understand what an interface does
 - Concepts make using lambdas easier
- Focus on the purpose of main functional interfaces
- Become aware of many primitive variations
- Lambda expressions have properties like those of a variable
 - Use when needed
 - Can be stored and reused



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

The `java.util.function` Package

- **Predicate:** An expression that returns a boolean
- **Consumer:** An expression that performs operations on an object passed as argument and has a void return type
- **Function:** Transforms a T to a U
- **Supplier:** Provides an instance of a T (such as a factory)
- **Primitive variations**
- **Binary variations**



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

Predicate is not the only functional interface provided with Java. A number of standard interfaces are designed as a starter set for developers.

Example Assumptions

- The following two declarations are assumed for the examples that follow:

```
List<SalesTxn> tList = SalesTxn.createTxnList();  
SalesTxn first = tList.get(0);
```



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

One or both of the declarations pictured are assumed in the examples that follow.

Predicate

```
package java.util.function;

public interface Predicate<T> {
    public boolean test(T t);
}
```



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

A **Predicate** takes a generic class and returns a boolean. It has a single method, namely `test`.

Predicate: Example

```
Predicate<SalesTxn> massSales =  
    t -> t.getState().equals(State.MA);  
  
System.out.println("\n== Sales - Stream");  
tList.stream()  
    .filter(massSales)  
    .forEach(t -> t.printSummary());  
  
System.out.println("\n== Sales - Method Call");  
for(SalesTxn t:tList){  
    if (massSales.test(t)){  
        t.printSummary();  
    }  
}
```

The test method of the predicate is being called from within the stream.

You can call the test method of the predicate directly.



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

In this example, a `SalesTxn` is tested to see if it was executed in the state of MA. The `filter` method takes a predicate as a parameter. In the second example, notice that the predicate can call its `test` method with a `SalesTxn` object as a parameter. This is what the stream does internally.

Consumer

```
1 package java.util.function;  
2  
3 public interface Consumer<T> {  
4  
5     public void accept(T t);  
6  
7 }
```



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

A Consumer takes a generic and returns nothing. It has a single method accept.

Consumer: Example

```
Consumer<SalesTxn> buyerConsumer = t ->
    System.out.println("Id: " + t.getTxnId()
        + " Buyer: " + t.getBuyer().getName());
System.out.println("== Buyers - Lambda");
tList.stream().forEach(buyerConsumer);
System.out.println("== First Buyer - Method");
buyerConsumer.accept(first);
```



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

Function

```
package java.util.function;

public interface Function<T, R> {

    public R apply(T t);
}
```



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

A Function takes one generic type and returns another. Notice that the input type comes first in the list and then the return type. So the `apply` method takes a T and returns an R.

Function: Example

```
Function<SalesTxn, String> buyerFunction =  
    t -> t.getBuyer().getName();  
  
System.out.println("\n== First Buyer");  
System.out.println(buyerFunction.apply(first));
```



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

The example takes a SalesTxn and returns a String. The Function interface is used frequently in the update Collection APIs.

Supplier

```
package java.util.function;

public interface Supplier<T> {

    public T get();
}
```



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

The `Supplier` returns a generic type and takes no parameters.

Supplier: Example

```
List<SalesTxn> tList = SalesTxn.createTxnList();  
Supplier<SalesTxn> txnSupplier =  
    () -> new SalesTxn.Builder()  
        .txnid(101)  
        .salesPerson("John Adams")  
        .buyer(Buyer.getBuyerMap().get("PriceCo"))  
        .product("Widget")  
        .paymentType("Cash")  
        .unitPrice(20)  
//... Lines omitted  
    .build();  
  
tList.add(txnSupplier.get());  
System.out.println("\n== TList");  
tList.stream().forEach(SalesTxn::printSummary);
```

calling `get` generates
a `SalesTxn` from the
lambda that was
defined earlier.



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

In the example, the Supplier creates a new SalesTxn.

Primitive Interface

- Primitive versions of all main interfaces
 - Will see these a lot in method calls
- Return a primitive
 - Example: `ToDoubleFunction`
- Consume a primitive
 - Example: `DoubleFunction`
- Why have these?
 - Avoids auto-boxing and unboxing



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

If you look at the API docs, there are a number of primitive interfaces that mirror the main types: `Predicate`, `Consumer`, `Function`, `Supplier`. These are provided to avoid the negative performance consequences of auto-boxing and unboxing.

Return a Primitive Type

```
package java.util.function;

public interface ToDoubleFunction<T> {

    public double applyAsDouble(T t);
}
```



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

The `ToDoubleFunction` interface takes a generic type and returns a `double`.

Return a Primitive Type: Example

```
ToDoubleFunction<SalesTxn> discountFunction =  
    t -> t.getTransactionTotal()  
        * t.getDiscountRate();  
  
System.out.println("\n== Discount");  
System.out.println(  
    discountFunction.applyAsDouble(first));
```



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

This example calculates a value from a transaction and returns a double. Notice that the method name changes a little, but this is still a Function. Pass in one type and return something else, in this case a double.

Process a Primitive Type

```
package java.util.function;

public interface DoubleFunction<R> {

    public R apply(double value);
}
```



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

Process Primitive Type: Example

```
9     A06DoubleFunction test = new A06DoubleFunction();  
10    DoubleFunction<String> calc =  
11        t -> String.valueOf(t * 3);  
12  
13    String result = calc.apply(20);  
14    System.out.println("New value is: " + result);  
15
```

The value $3 * 20$ will be generated, then converted to a String by the lambda expression, and this String result will be returned.



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

The example computes a value and then returns the result as a String.

Binary Types

```
package java.util.function;

public interface BiPredicate<T, U> {

    public boolean test(T , U u);
}
```



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

The binary version of the standard interfaces allows two generic types as input. In this example, the BiPredicate takes two parameters and returns a boolean.

Binary Type: Example

```
List<SalesTxn> tList = SalesTxn.createTxnList();
SalesTxn first = tList.get(0);
String testState = "CA";

BiPredicate<SalesTxn, String> stateBiPred =
    (t, s) -> t.getState().getStr().equals(s);

System.out.println("\n== First is CA?");
System.out.println(
    stateBiPred.test(first, testState));
```



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

This example takes a SalesTxn and a String to do a comparison and return a result. The test method merely takes two parameters instead of one.

Unary Operator

```
package java.util.function;

public interface UnaryOperator<T> extends Function<T, T> {
    @Override
    public T apply(T t);
}
```



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

The UnaryOperator takes a class as input and returns an object of the same class.

UnaryOperator: Example

- If you need to pass in something and return the same type, use the UnaryOperator interface.

```
UnaryOperator<String> unaryStr =  
    s -> s.toUpperCase();  
  
System.out.println("== Upper Buyer");  
System.out.println(  
    unaryStr.apply(first.getBuyer().getName()));
```



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

The UnaryOperator interface takes a generic type and returns that same type. This example takes a String and returns the String in uppercase.

Wildcard Generics Review

- Wildcards for generics are used extensively.
- ? super T
 - This class and any of its super types
- ? extends T
 - This class and any of its subtypes



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

A Closer Look at Consumer

```
public interface Consumer<T>
    void accept(T t)
```

- Its type, expressed as a generic, is T, and that is the type that will be passed into its accept method.
- Note in particular the type of its accept method.
- What is T and where does it come from?
 - Typically from the Collection or Stream type that uses a Consumer as a parameter to one of its methods, e.g. forEach.



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

Interface List<E> use of forEach method

Interface List<E>

Has super interface of:

Collection<E>, Iterable<E>

- In the API docs:
- Iterable<E> has method signature:
- default void forEach(Consumer<? super T> action)
- So <T> is type of the Iterable that defines forEach method



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

Example of Range of Valid Parameters

```
public class Fruit extends Plant{  
    String fruitType = ""; int amount = 0;  
  
    public Fruit(String type, int amount) {  
        this.amount = amount;  
        this.fruitType = type;  
    }  
    public String describe() {  
        return "Fruit description: "  
            + this.fruitType + ":" + this.amount;  
    }  
}  
  
public class Plant { }
```



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

Plant Class Example

```
List<Fruit> fruits = List.of(new Fruit("apple", 1), new
Fruit("orange", 2), new Fruit("pear", 4));
// SE 9 new convenience method
fruits.forEach(a -> System.out.println(a.describe()));

Fruit description: apple:1
Fruit description: orange:2
Fruit description: pear:4
```

- As the type is inferred by the compiler, it's passing in a `Consumer<Fruit>`.



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

Plant Class Example

- If you explicitly declare the type in the lambda expression, the code won't compile because `describe()` doesn't exist on the superclass.

```
fruits.forEach((Plant a) ->  
    System.out.println(a.describe()));
```

Will cause code to fail to compile.

- The following compiles and runs (it's calling the default `toString()` on `Object`).

```
fruits.forEach((Plant a) ->  
    System.out.println(a));  
  
sample.Fruit@67b64c45  
sample.Fruit@4411d970  
sample.Fruit@6442b0a6
```



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

TropicalFruit Class Example

```
class TropicalFruit extends Fruit {  
    public TropicalFruit(String type, int amount) {  
        super(type, amount);  
    }  
}
```

- The following does not work.

```
fruits.forEach((TropicalFruit a) ->  
    System.out.println(a));
```

method forEach in interface Iterable<T> cannot be applied to given types;

```
fruits.forEach((TropicalFruit a) -> System.out.println(a));
```



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

method forEach in interface Iterable<T> cannot be applied to given types;

```
fruits.forEach((TropicalFruit a) -> System.out.println(a));
```

^

required: Consumer<? super Fruit>

found: (TropicalFruit...)ln(a)

reason: argument mismatch; Consumer<TropicalFruit> cannot be converted to
Consumer<? super Fruit>

where T is a type-variable:

```
T extends Object declared in interface Iterable
```

Use of Generic Expressions and Wildcards

- Generic expressions can make methods look very complex.
- In many cases if you are writing a lambda expression that infers the type, the docs may, say:

```
map(Function<? super T, ? extends R> mapper)
```

But you can read it as simply:

```
map(Function<T, R> mapper)
```



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

Consumer andThen method

```
default Consumer<T> andThen(Consumer<? super T> after)
```

- Common in the `java.util.function` Interface types.
- Allows for function composition, where (in this case) `Consumer` objects can be chained together.
 - This extra functionality is a good reason that it's often better to use the standard functional types rather than creating your own.
 - The `andThen` method must be called on a `Consumer`



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

Using Consumer.andThen method

- In the code, the lambda expression is creating an object of type `Consumer<Fruit>` and passing the reference into the `forEach` method that then calls `accept()`.

```
fruits.forEach(a ->
    System.out.println(a.describe()));
```

- So you might expect this to work, but it doesn't.

```
fruits.forEach((a -> System.out.println(a)).andThen(..more
lambda code...));
```



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

The second example doesn't work because while `a -> System.out.println(a)` is the lambda expression that represents a `Consumer` when its method is called it returns `void` and can't therefore have the `andThen` method tacked on to the end. It won't compile, and the compiler will report a `void` referencing problem.

Using Consumer.andThen method

- The following, however, does work.

```
Consumer<Fruit> bag = b ->
    System.out.print(b.amount + " " + b.fruitType);
Consumer<Fruit> bagOutput = bag.andThen(a ->
    System.out.println(a.amount > 1?"s":""));

fruits.forEach(bagOutput);

1 apple
2 oranges
4 pears
```

Adds “s” where appropriate.



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

Summary

In this lesson, you should have learned how to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces

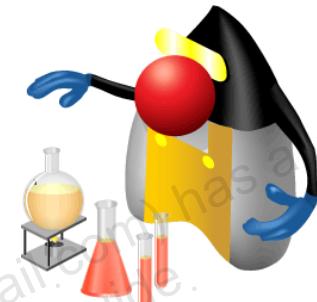


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Practice 8: Overview

This practice covers the following topics:

- Practice 8-1: Creating Consumer lambda expression
- Practice 8-2: Creating a Function lambda expression
- Practice 8-3: Creating a Supplier lambda expression
- Practice 8-4: Creating a BiPredicate lambda expression



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.

Lambda Operations



ORACLE®

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



Objectives

After completing this lesson, you should be able to:

- Extract data from an object by using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class



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

Streams API

- Streams
 - `java.util.stream`
 - A sequence of elements on which various methods can be chained:
- The Stream class has these properties:
 - Immutable data
 - Can only be used once
 - Encourages fluent programming through method chaining
- The Java API doc gives details of all `Stream` methods.
- Classes
 - `Stream<T>` handles non-numerical objects.
 - `DoubleStream`, `IntStream`, `LongStream` handle primitive `int`, `long`, and `double` types.



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

A stream pipeline consists of a source, zero, or more intermediate operations (which transform a stream into another stream) and a terminal operation that ends the use of a stream.

To perform a computation, stream operations are composed into a stream pipeline.

A stream pipeline consists of:

- **A source:** An array, a collection, a generator function, an I/O channel
- Zero or more intermediate operations, which transform a stream into another stream, e.g. `filter`
- A terminal operation, which produces a result or side effect, e.g. `count` or `forEach`

Streams may be lazy. Computation on the source data is performed only when the terminal operation is initiated, and source elements are consumed only if needed.

Types of Operations

- **Intermediate**
 - filter() map() peek() dropWhile()
- **Intermediate short-circuit**
 - limit() takeWhile()
- **Terminal**
 - forEach() count() sum() average() min() max() collect()
- **Terminal short-circuit**
 - findFirst() findAny() anyMatch() allMatch() noneMatch()
 - takeWhile()



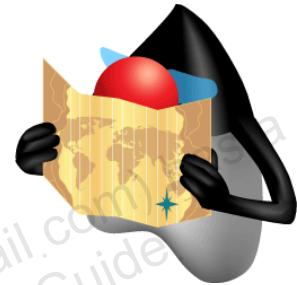
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The above is a list of stream methods by their operation type.

Extracting Data with Map

```
map(Function<? super T,? extends R> mapper)
```

- A map takes one Function as an argument.
 - A Function takes one generic type and returns the same type or something else.
- Primitive versions of map method
 - mapToInt mapToLong mapToDouble



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

Taking a Peek

`peek(Consumer<? super T> action)`

- The peek method performs the operation specified by the lambda expression and returns the elements to the stream.
- Useful for printing intermediate results



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

Search Methods: Overview

- `findFirst()`
 - Returns the first element that meets the specified criteria
- `allMatch()`
 - Returns `true` if all the elements meet the criteria
- `noneMatch()`
 - Returns `true` if none of the elements meet the criteria
- All of the above are short-circuit terminal operations.



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

The `findFirst` method of the `Stream` class finds the first element in the stream specified by the filters in the pipeline. The `findFirst` method is a terminal short-circuit operation. A terminal operation ends the processing of a pipeline.

The `allMatch` method returns whether all elements of this stream match the provided `predicate`. The method may not evaluate the `predicate` on all elements if not necessary for determining the result. If the stream is empty, `true` is returned and the `predicate` is not evaluated.

The `noneMatch` method returns whether no elements of this stream match the provided `predicate`. It will not evaluate the `predicate` on all elements if this is not necessary for determining the result. If the stream is empty, `true` is returned and the `predicate` is not evaluated.

Search Methods

- Nondeterministic search methods
 - Used for nondeterministic cases, in effect, situations where parallel is more effective
 - Results may vary between invocations.
- `findAny()`
 - Returns the first element found that meets the specified criteria
 - Results may vary when performed in parallel.
- `anyMatch()`
 - Returns true if any elements meet the criteria
 - Results may vary when performed in parallel.



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

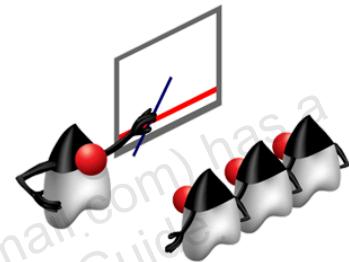
Nondeterministic means that the search may return a different result on each invocation, but any of these are correct and usable.

The `findAny` method returns an `Optional<T>` describing some element of the stream or an empty `Optional<T>` if the stream is empty. The behavior of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (If a stable result is desired, use `findFirst()` instead.) This is a short-circuiting terminal operation.

The `anyMatch` method returns whether any elements of this stream match the provided predicate. The method may not evaluate the predicate on all elements if it is not necessary for determining the result. If the stream is empty, `false` is returned and the predicate is not evaluated. This is a short-circuiting terminal operation.

Optional Class

- `Optional<T>`
 - A container object that may or may not contain a non-null value
 - If a value is present, `isPresent()` returns true.
 - `get()` returns the value.
 - Many other methods available including `stream` to return a new Stream object if necessary
 - In `java.util` package
- **Optional primitives**
 - `OptionalDouble` `OptionalInt` `OptionalLong`



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

Short-Circuiting Example

Performs only required operations

Using
findFirst
short-
circuiting
terminal
operation.

```
== First CO Bonus ==  
Stream start  
Stream start  
Stream start  
Stream start  
Stream start  
Stream start  
Executives  
CO Executives  
Bonus paid: $7,200.00
```

```
== CO Bonuses ==  
Stream start  
Stream start  
Stream start  
Stream start  
Stream start  
Stream start  
Executives  
CO Executives  
Bonus paid: $6,600.00  
Stream start  
Executives  
CO Executives  
Bonus paid: $8,400.00
```

Using
forEach
terminal
operation.



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

The slide shows two lists of operations on a list of Employees. The list on the right must go through all the employee elements as it uses the `forEach` terminal operation. The list on the left uses the `findFirst` method and, thus, when the first element is found, stream processing terminates.

Stream Data Methods

`count()`

- Returns the count of elements in this stream

`max(Comparator<? super T> comparator)`

- Returns the maximum element of this stream according to the provided Comparator

`min(Comparator<? super T> comparator)`

- Returns the minimum element of this stream according to the provided Comparator



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

Performing Calculations

Primitive streams have `average` and `sum` methods:

- `DoubleStream`, `IntStream`, `LongStream`

`average()`

- Returns an `OptionalDouble` describing the arithmetic mean of elements of this stream
- Returns an empty `Optional` if this stream is empty

`sum()`

- Returns the sum of elements in this stream



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

Sorting

`sorted()`

- Returns a stream consisting of the elements sorted according to natural order

`sorted(Comparator<? super T> comparator)`

- Returns a stream consisting of the elements sorted according to the Comparator



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

The `sorted` method can be used to sort stream elements based on their natural order. This is an intermediate operation.

Comparator Updates

```
comparing(Function<? super T,? extends U> keyExtractor)
```

- Allows you to specify any field to sort on based on a method reference or lambda
- Primitive versions of the Function also supported

```
thenComparing(Comparator<? super T> other)
```

- Specify additional fields for sorting.

```
reversed()
```

- Reverse the sort order by appending to the method chain.



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

Saving Data from a Stream

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

- Allows you to save the result of a stream to a new data structure
- A number of useful collectors are available from the `Collectors` class
 - Examples
 - `stream().collect(Collectors.toList());`
 - `stream().collect(Collectors.toMap());`
- If a static import of the `Collectors` class is used in the source file, the code can be simplified for readability to just the method call:
 - `stream().collect(toList());`

toList and toMap are just two static methods of the Collectors class that return a Collector.



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

The `collect` method allows you to save the results of all the filtering, mapping, and sorting that takes place in a pipeline. Notice how the `collect` method is called. It takes a `Collector` as a parameter. The `Collectors` class provides a number of collectors that can be combined in many ways to return the elements remaining in a pipeline after intermediate operations.

The `Collectors` class and the many collectors that it provides is covered in much more detail in the lesson titled “Terminal Operations: Collectors.”

Collectors Class

- **averagingDouble(ToDoubleFunction<? super T> mapper)**
 - Produces the arithmetic mean of a double-valued function applied to the input elements
- **groupingBy(Function<? super T, ? extends K> classifier)**
 - A "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a map
- **joining()**
 - Concatenates the input elements into a String, in encounter order
- **partitioningBy(Predicate<? super T> predicate)**
 - Partitions the input elements according to a Predicate



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

The `groupingBy` method of the `Collectors` class allows you to generate a `Map` based on the elements contained in a stream. The keys are based off a selected field in a class. Matching objects are placed into an `ArrayList` that becomes the value for the key.

The `joining` method of the `Collectors` class allows you to join together elements returned from a stream.

The `partitioningBy` method offers an interesting way to create a `Map`. The method takes a `Predicate` as an argument and creates a `Map` with two `boolean` keys. One key is `true` and includes all the elements that meet the true criteria of the `Predicate`. The other key, `false`, contains all the elements that resulted in `false` values as determined by the `Predicate`.

Quick Streams with Stream.of

- The Stream.of method allows you to easily create a stream.

```
public static void main(String[] args) {  
  
    Stream.of("Monday", "Tuesday", "Wednesday", "Thursday")  
        .filter(s -> s.startsWith("T"))  
        .forEach(s -> System.out.println("Matching Days: " + s));  
}
```



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

Flatten Data with flatMap

- Use the flatMap method to flatten data in a stream.

```
Path file = new File("tempest.txt").toPath();  
  
try{  
  
    long matches = Files.lines(file)  
        .flatMap(line -> Stream.of(line.split(" ")))  
        .filter(word -> word.contains("my"))  
        .peek(s -> System.out.println("Match: " + s))  
        .count();  
  
    System.out.println("# of Matches: " + matches);  
}
```

Because flatMap returns a stream, the lambda function must produce a stream.



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

The flatMap method can be used to convert data into a stream. When called on a stream, it has the effect of turning each element of the stream into a new stream. It therefore is used to “flatten” the data structure.

In order to search for the occurrence of a particular word, you need a stream of type String, where each word is a String element. Then filter to only include the word of your choice in the stream, peek, to show a match being made, and finally count, to count the matches.

Output for peek is:

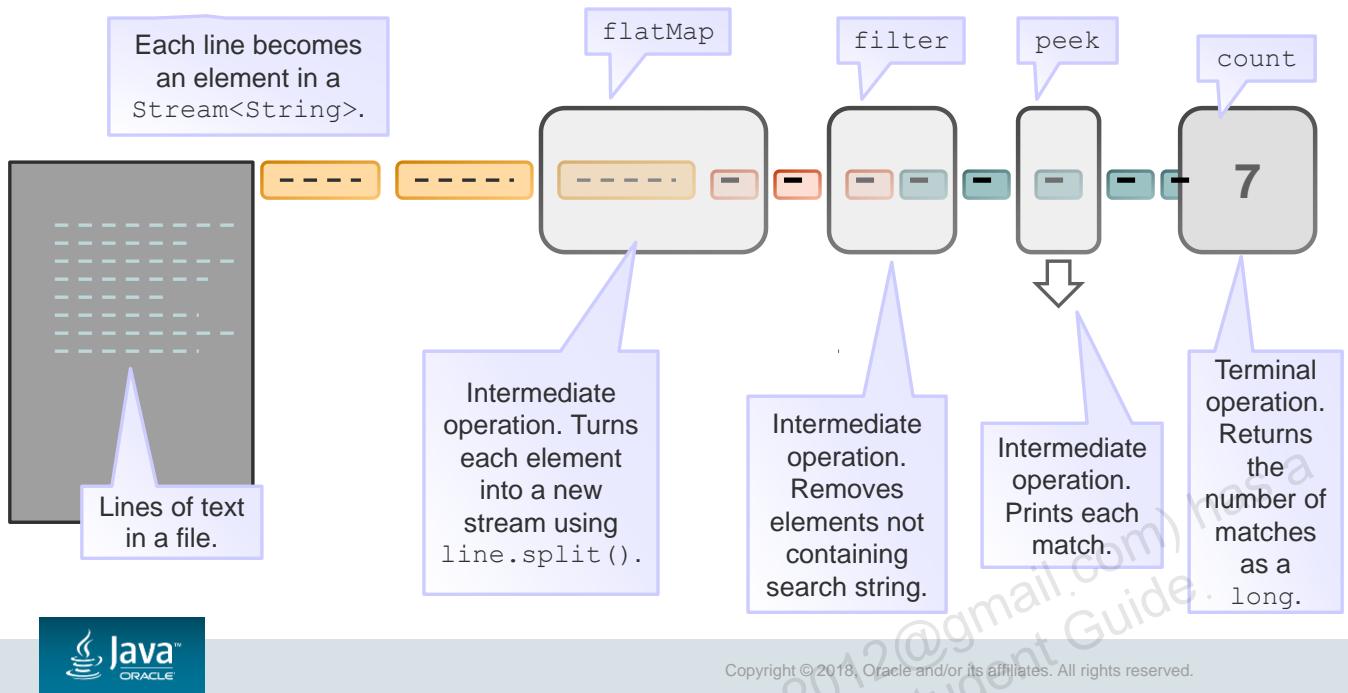
- **Match:** your
- **Match:** yourself
- **Match:** your

Output for matches (i.e. count): 3

But why is flatMap needed?

Because `File.lines(file)` returns a stream of type `String` of entire lines. To instead have a stream of every word, each line is used to generate a new stream of single words.

flatMap in Action



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

In order to search for the occurrence of a particular word, you need a stream of type `String`, where each word is a `String` element. Then filter to only include the word of your choice in the stream, peek, to show a match being made, and finally count, to count the matches.

Output for peek is:

- **Match:** my
- **Match:** rummy
- **Match:** myself
- **Match:** ...

Output for matches (i.e. count): 7

But why is flatMap needed?

Because `File.lines(file)` returns a stream of type `String` of entire lines. To instead have a stream of individual words, each line is used to generate a new stream of single words.

Summary

In this lesson, you should have learned how to:

- Extract data from an object using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class

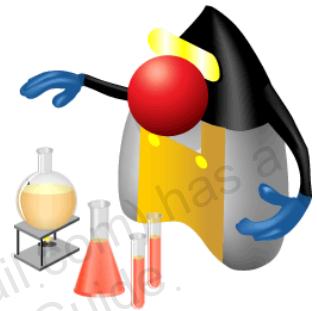


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Practice 9: Overview

This practice covers the following topics:

- Practice 9-1: Using Map and Peek
- Practice 9-2: FindFirst and Lazy Operations
- Practice 9-3: Analyzing Transactions with Stream Methods
- Practice 9-4: Performing Calculations with Primitive Streams
- Practice 9-5: Sorting Transactions with Comparator
- Practice 9-6: Collecting Results with Streams
- Practice 9-7: Joining Data with Streams
- Practice 9-8: Grouping Data with Streams



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.

The Module System

10



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



ORACLE®

Adolfo De+la+Rosa (adolfo.delarosa.2012@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Describe the purpose (in general terms) of the `module-info` class
- Create modules with defined module dependencies and module encapsulation
- Compile modules and create modular JAR files on the command line
- Describe how NetBeans IDE organizes its folders for source, compiled modules, and modular JARs



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

Module System

- The module system:
 - Supports programming in the “large”
 - Is built into the Java language
 - Is usable at all levels:
 - Applications
 - Libraries
 - The JDK itself
 - Addresses reliability, maintainability, and security
 - Supports creation of applications that can be scaled for small computing devices



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

Programming in the “large” means programming techniques and component organization at the level of the unit of distribution, rather than at class level. For example, lambda can be considered programming in the “small,” whereas designing modules is programming in the “large.”

Because the module system is concerned with programming in the “large,” designing modules is not likely to be done by all developers; it will be done by architects, and some developers will program only in the “small.”

Module System: Advantages

- Addresses the following issues at the unit of distribution/reuse level:
 - Dependencies
 - Encapsulation
 - Interfaces
- The unit of reuse is the module.
 - It is a full-fledged Java component.
 - It explicitly declares:
 - Dependencies on other modules
 - Which packages it makes available to other modules
 - Only the public interfaces in those available packages are visible outside the module.



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

Java Modular Applications



- No missing dependencies
- No cyclic dependencies
- No split packages



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

Dependencies are fully checked during compilation and run time. The program will not attempt to run if all dependencies are not found.

Cyclic dependencies—module A requires module B and vice versa—are not permitted. This ensures more robust applications.

Split packages—module A and module B contain packages with the same names—are not permitted.

All these are explored further later in this lesson.

What Is a Module?

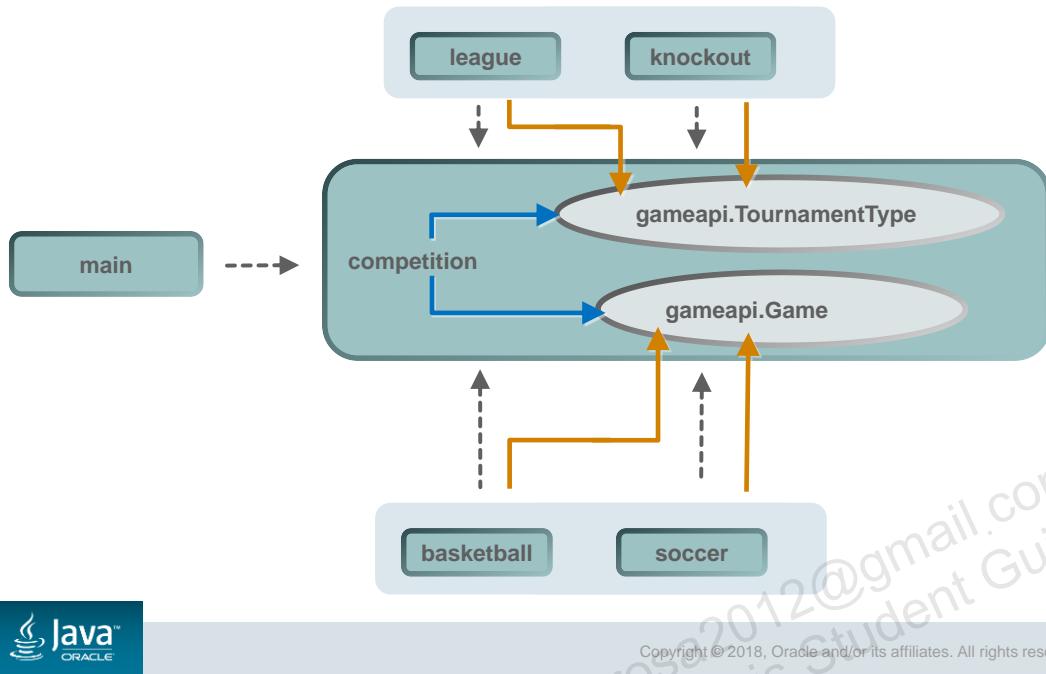
A module is a set of packages that make sense being grouped together.

- Modularity was introduced in JDK SE 9.
- Modules add a higher level of aggregation above packages.
 - They are the basic unit of distribution and reuse.
- A module is a reusable group of related packages, as well as resources (such as images and XML files) and a module descriptor; that is, **programs are modules**.
- In a module, some of the packages are:
 - Exported packages: Intended for use by code outside the module
 - Concealed packages: Internal to the module; they can be used by code inside the module but not by code outside the module.



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

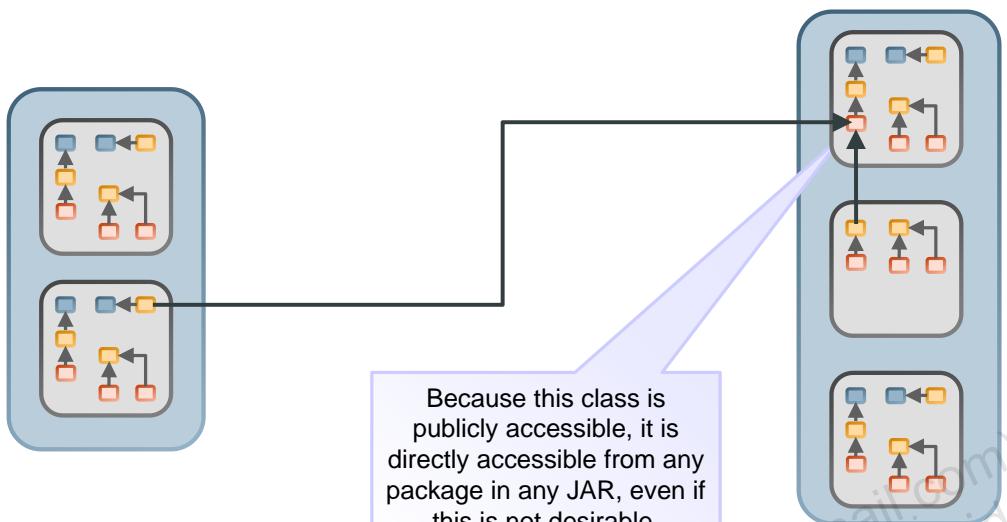
A Modular Java Application



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

The slide shows a dependency diagram for the TeamGameManager application. The types of games supported can easily be added to it, as can the types of competition.

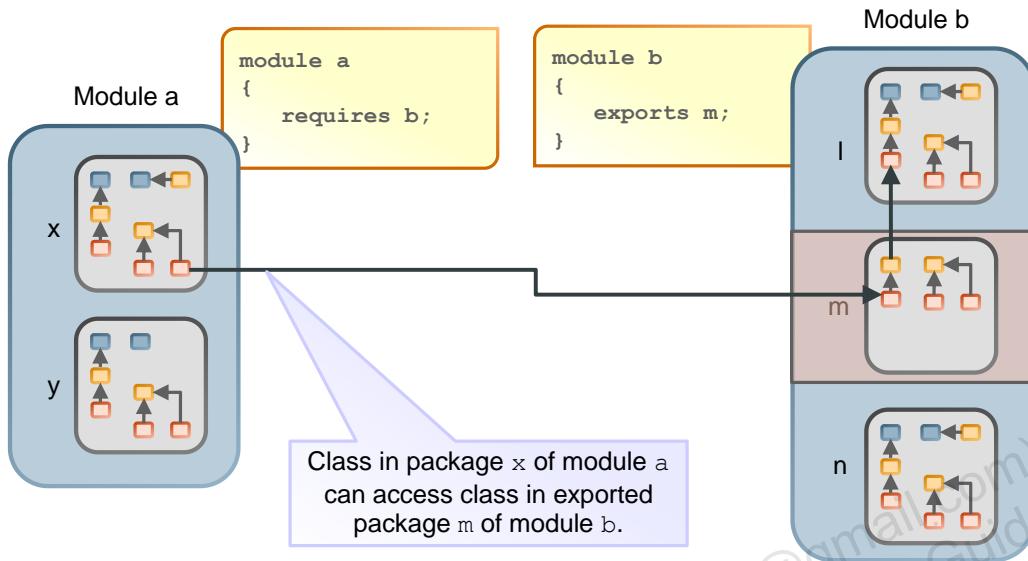
Issues with Access Across Nonmodular JARs



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

If a class needs to be made accessible to a different package in the same JAR, it must be made public. This makes it also accessible to any class in any package.

Dependencies Across Modules

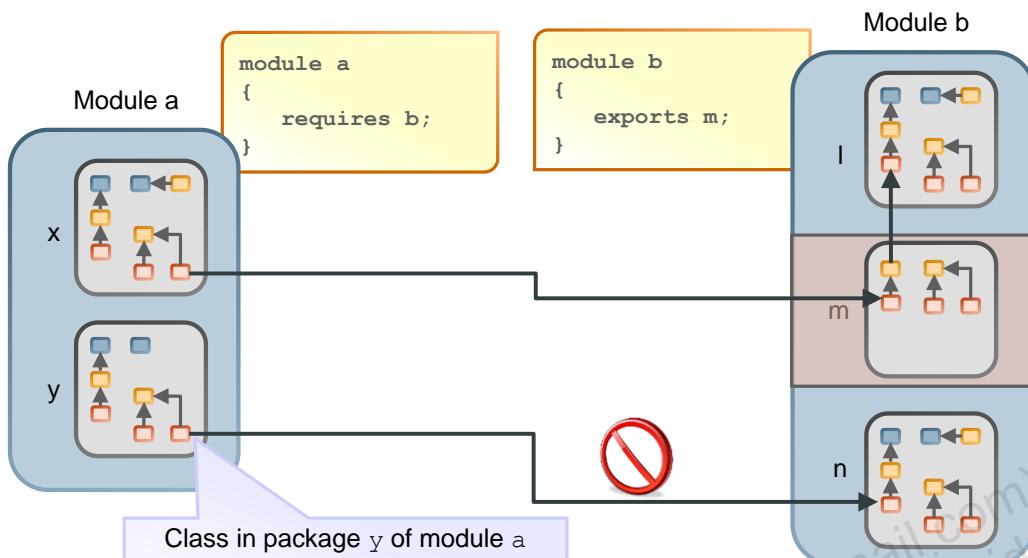


In the example shown, module a has a dependency on module b; it “requires” module b. But module b only makes its m package available (it “exports” package m).

The class in package x can access the class in package m, because m is exported, but it cannot access any class in packages l or n, even if these are public classes. Classes in modules l or n are concealed packages only accessible (if public) within module b.

Note that the requires and exports keywords are introduced here just to illustrate that modules offer explicit dependencies and encapsulation at the module level. They, and other module-related directives, will be covered in detail later in this lesson.

Dependencies Across Modules



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

Note that the class in package **y** cannot access the class in package **n** (assuming the class in **l** is public), even though module **a** requires module **b**. Access across modules is based on the following:

- Modules must explicitly require other modules. This gives the module system reliable dependencies. These are checked during compilation and before running the code.
- Modules must explicitly export the packages they want to make visible. This delivers encapsulation at the module level.
- The class being accessed must be public.

What Is a Module?

A module:

- Contains one or more packages and other resources such as images or xml files
- Is defined in its module descriptor (`module-info.class`), which is stored in the module's root folder
 - The module descriptor must contain the module name.
 - Additionally the module descriptor can contain details of:
 - Required module dependencies (other modules that this module depends on)
 - Packages that this module exports, making them available to other modules
 - Otherwise all packages in the module are implicitly unavailable to other modules.
 - Permissions to open content of this module to other modules via the use of reflection
 - Services this module offers to other modules
 - Services this module consumes



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

```
module <this module name> {  
    requires <another module name>  
    exports <packages of this module to other modules that require any  
           public types they contain>  
    opens <packages, including non-public types, to other modules>  
    uses <services provided via a an implementation>  
    provides <services to any module> with <a service implementation>  
    version <value>  
}
```

Module Dependencies with `requires`

A module defines that it needs another module using the `requires` directive.

- `requires` specifies a normal module dependency (this module needs access to some content provided by another module).
- `requires transitive` specifies a module dependency and makes the module dependent upon available to other modules.
- `requires static` indicates module dependency at compile time, but not at the run time.



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



`requires`—module a is dependent on module b; it needs at least one class or interface in b.

`requires transitive`—module a is dependent on module b; it needs at least one class or interface in b, AND it will make that class or interface in b available to other modules.

This is a very brief summary—it's covered in much more details later.

Module Package Availability with exports

A module defines what content it makes available for other modules using the `exports` directive.

- Exporting a package makes all of its public types available to other modules.
- There are two directives to specify packages to export:
 - The `exports` directive specifies a package whose public types are accessible to all other modules.
 - The `exports ... to` directive restricts the availability of an exported package to a list of specific modules.
 - Accepts a comma-separated list of module names after the `to` keyword.



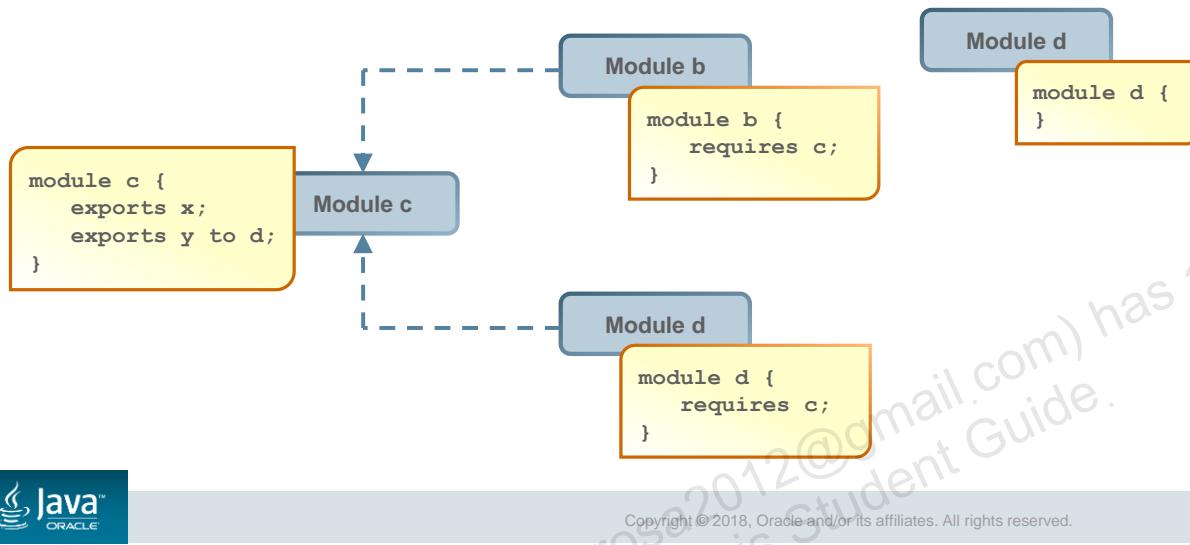
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



- Only classes (and their variables and methods) that have public access can be accessed from another module.
- To access types in an exported package, the module requiring the use of those types must use the `requires` directive.

Module Graph 1

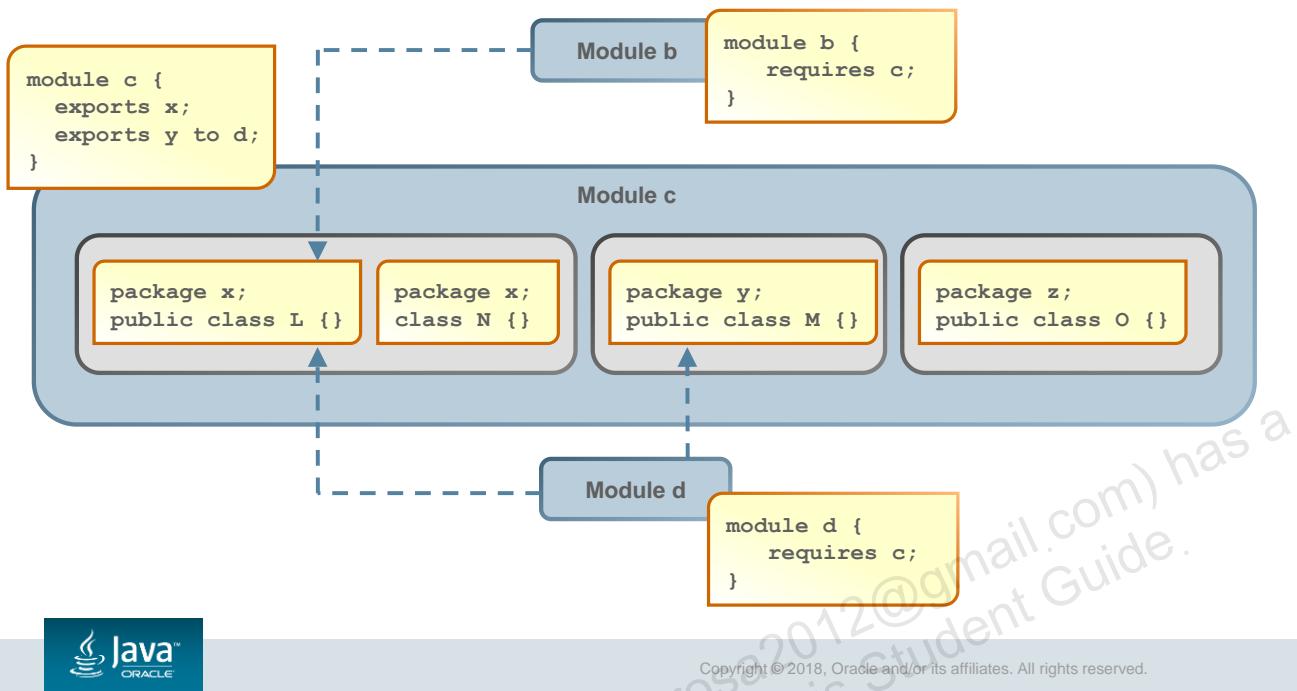
- The module system resolves all dependencies expressed in the `requires` directives.
 - This can be illustrated as a module graph.



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

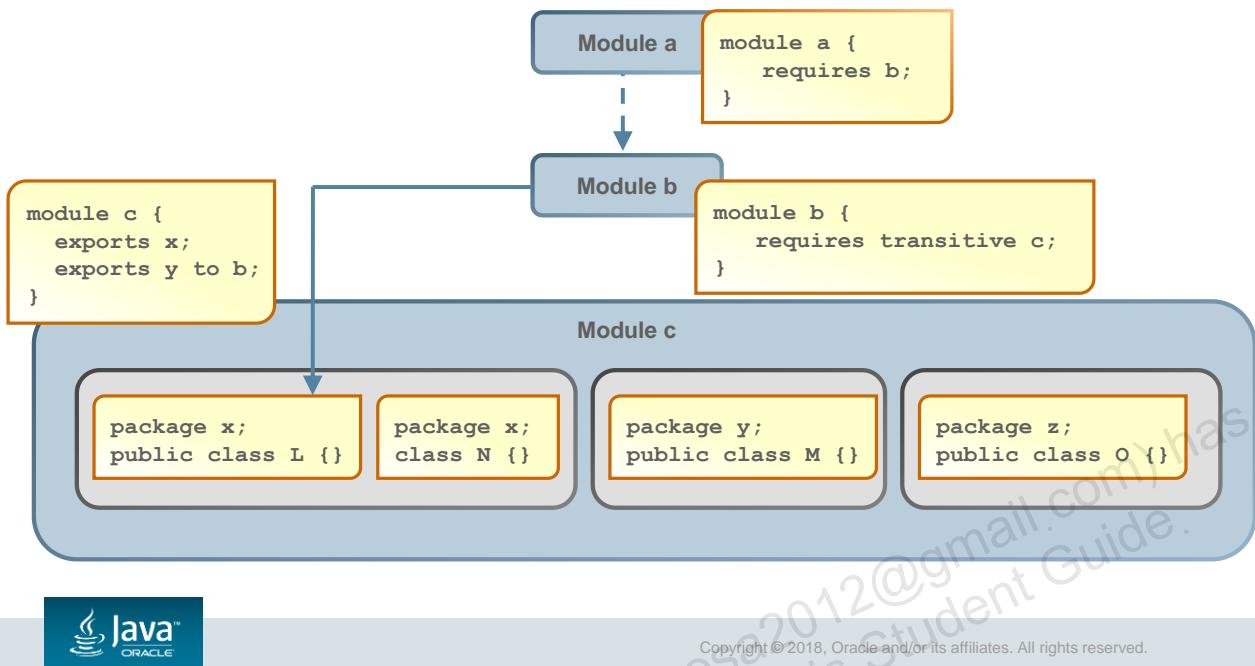
The module system *resolves* the dependencies expressed in its **requires** clauses by locating additional observable modules to fulfill those dependencies and then resolves the dependencies of those modules, and so forth, until every dependency of every module is fulfilled. The result of this transitive-closure computation is a *module graph* that, for each module with a dependency that is fulfilled by some other module, contains a directed edge from the first module to the second.

Module Graph 2



- In the example above, module c exports package x to all other interested modules and package y just to module c.
- Thus:
 - Class L is visible to modules b and d.
 - Class N is not visible to modules b and d because it is not public.
 - Class M is not visible to module b because its package has only been exposed to module d.
 - Class O is not visible to anyone outside of the module c because package z is not exported.

Transitive Dependencies



Imagine that module `b` provides an API and for this reason is required by module `a`. In addition, imagine that the API returns a type `L` that exists in package `x` in module `c`. In order for module `a` to have access to this type, `L`, either:

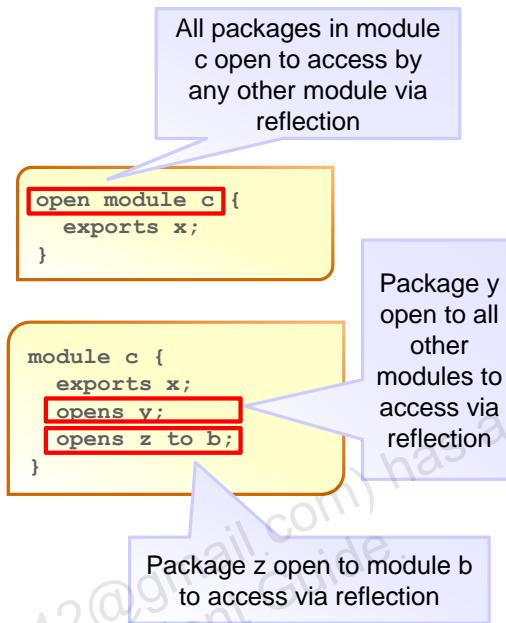
- Module `b` must transitively require `c` (as shown here) OR
- Module `a` must state its own requirement for module `c`

In the example shown, module `b` has a transitive dependency on module `c`. Note that dependencies between modules are shown with a dotted line and transitive dependencies with a solid line.

Access to Types via Reflection

A module may be set up to allow runtime-only access to a package by using the `opens` directive.

- The `opens` directive makes a package available to all other modules at run-time but not at compile time.
- The `opens ... to` directive makes a package available to a list of specific modules at run-time but not compile time.
- Using `opens` for a package is similar to using `exports`, but it also makes all of its nonpublic types available via reflection.
 - Modules that contain injectable code should use the `opens` directive, because injections work via reflection.
- All packages in a module can be made available to access via reflection by using the `open` directive before the `module` directive.



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

Before Java 9, reflection could be used to learn about all types in a package and all members of a type—even its private members—whether you wanted to allow this capability or not. Thus, nothing was truly encapsulated.

A key motivation of the module system is strong encapsulation. By default, a type in a module is not accessible to other modules unless it's a public type and you export its package. You expose only the packages you want to expose. With Java 9 and later, this also applies to reflection.

The `opens` directive allows you to specify a package so that all its types (and all of its types' members) are accessible via reflection (this includes types with nonpublic access).

The `opens... to` directive is the same as the `opens` directive except it allows you to limit the access by reflection to a set of specified modules.

Note that if the types are public, `exports <package name>` makes all the types in a package available via reflection, but `opens` is required to make nonpublic types available.

Opening packages to access via reflection is covered in more details in the lesson titled “Migration.”.

Example Hello World Modular Application Code

Here is a simple Hello World application with two modules. It will be used for examples later in this lesson.

Module greeting

```
package greeting;
import java.util.logging.Logger;
import world.World;
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello " +
            World.say());
    }
}
```

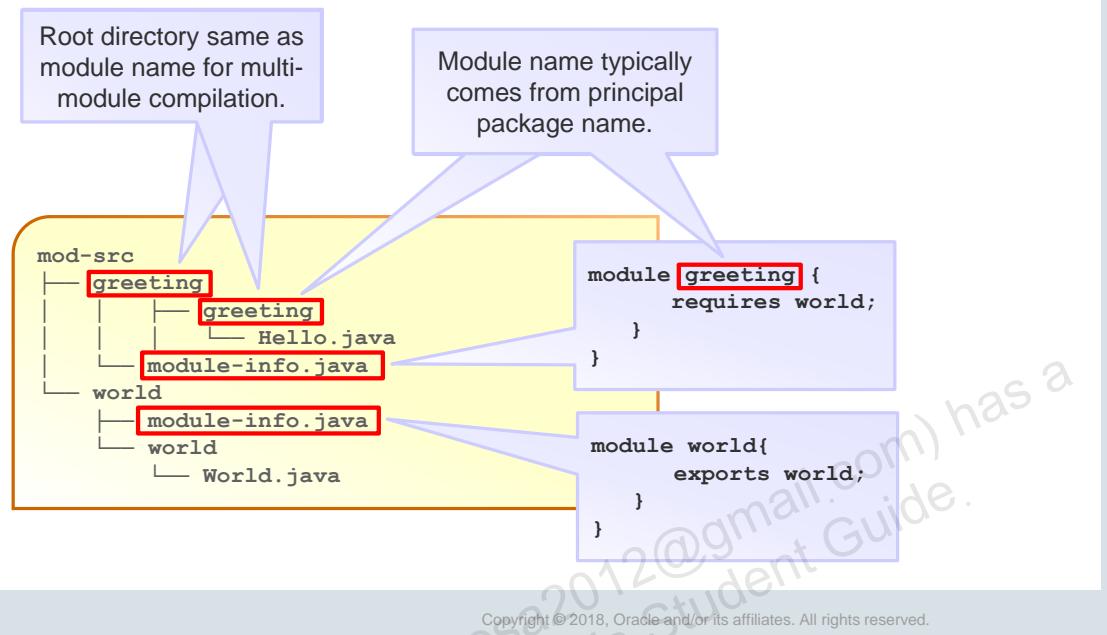
Module world

```
package world;
public class World {
    public static String say() {
        return "World!";
    }
}
```



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

Example Hello World Modular File Structure



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

Compiling a Modular Application

Single module compilation:

```
javac -d <output folder> <list of source code file paths including module-info>
```

Multi-module compilation:

```
javac -d <output folder>
--module-source-path <root directory of the module source> \
<list of source code file paths>
```

Get description of the compiled module:

```
java --module-path <path to compiled module>
--describe-module <module name>
```



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

When compiling a single module, all source files must be listed, including the module-info.java file. The output folder needs to be set to the directory you wish the modules to be placed in. A multi-module application can be compiled by using a sequence of single module compilations, but this is not ideal as it requires a number of commands, and it cannot guarantee the dependencies.

Multi-module compilation is achieved by using the `--module-source-path` option to point to the root directory of the source file structure.

You can get a description of a compiled module by using the `java` command with `--module-path` and `--describe-module` options, with `--module-path` pointing to either the module folder or to the containing folder for the module folder. If the module being described has not explicitly defined that it requires `java.base` and relied upon implicit inclusion, then `--describe-module` displays:

`java.base` mandated for this module.

Single Module Compilation Example

You can compile the application module by module.

Output folder points to module folder.

```
javac -d mods/world src/world/module-info.java \
src/world/world/World.java
javac -d mods/greeting --module-path mods src/greeting/module-info.java \
src/greeting/greeting>Hello.java
```

```
mod-src
└── greeting
    └── greeting
        └── Hello.java
    └── module-info.java
└── world
    └── module-info.java
    └── world
        └── World.java
```



```
mods
└── greeting
    └── greeting
        └── Hello.class
    └── module-info.class
└── world
    └── module-info.class
    └── world
        └── World.class
```



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

In the example above:

- For each module to be compiled, the `module-info.java` file must be included in the list of source files.
- `world` is compiled first as it has no dependencies on any other module.
- The second module to be compiled must use `--module-path` to point to the compiled `world` module.
- `-d` (the output folder specification) has the name of the module to be compiled as the name of the final directory in its path. This is not mandatory for single module compilation, but it is recommended.
- The module name is based on the package name. It is recommended as good practice to name the module the same as the principal package in the module, but this is not mandatory.

Multi Module Compilation Example

- Passing just the filename for the source of the main class.

```
javac -d mods --module-source-path src src/greeting/greeting>Hello.java
```

Output folder points to
containing folder for modules.

- Passing all source filenames.

```
javac -d mods --module-source-path src $(find src -name "*.java")
```

- Checking module contents.

```
java --module-path mods --describe-module greeting
greeting file:///home/<username>/demo/mods/greeting
exports greeting
requires java.base mandated
requires world
```



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

The javac examples above use `--module-source-path` so javac can find the source code.

In the first example, even though `javac` is only passed `Hello.java` as a file to compile, it can determine that modular compilation is required. This means that:

- The greeting module will be compiled (`Hello.java` and the `greeting` module's `module-info.java` file).
- The world module will be compiled (`World.java` and the `world` module's `module-info.java` file).
- For the first compilation (assume `mods` directory does not exist), all necessary files will be compiled.
- For compilation where the compiled files already exist in the destination directory, only `Hello.java` will be compiled.

In the second example, the unix `find` command is used to ensure that all source files are compiled.

Creating a Modular JAR

- Use the `jar` command to create a modular JAR:

```
jar --create -f <path and name of JAR file>
  --main-class <package name>.<main class name>
  -C <path to compiled module code> .
```

- Hello World application example:

```
jar --create -f jars/world.jar -C mods/world .
jar --create -f jars/hello.jar --main-class greeting.Hello -C mods/greeting/ .
```



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

The `jar` command options shown are:

- `--create` instructs the `jar` utility to create new jar file.
- `-f` sets path and name of the JAR file.
- `-C` sets path to compiled code of the module.
- `--main-class` sets the main class of the JAR, so it doesn't need to be passed to the `java` command on the command line.

Running a Modular Application

- Running an unpackaged module application:

```
java --module-path <path to compiled module or modules> \
--module <module name>/<package name>.<main class name>
```

- Running an application packaged into modular JARs (assuming main class specified when creating JARs):

```
java --module-path <path to JARs> --module <module name>
```

- Running the Hello World application example:

```
java -p jars -m greeting
```



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

Some options have short versions:

- `--module-path` has a short version, `-p`.
- `--module` has a short version, `-m`.

The Modular JDK



- In JDK 9, the monolithic JDK is broken into several modules.
It now consists of about 90 modules.
- Every module is a well-defined piece of functionality of the JDK:
 - All the various frameworks that were part of the prior releases of JDK are now broken down into their modules.
 - For example: Logging, Swing, and Instrumentation
- The modular JDK:
 - Makes it more scalable to small devices
 - Improves security and maintainability
 - Improves application performance



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

Java SE Modules

java.se:

- This module doesn't contain any code but has only dependencies declared in the module descriptor:

```
module java.se {  
    requires transitive java.desktop;  
    requires transitive java.sql;  
    requires transitive java.xml;  
    requires transitive java.prefs;  
    // ... many more  
}
```

- In the module descriptor, a `requires transitive` clause is listed for each module that is part of the Java SE specification.
- When you say `requires java.se` in a module, all these modules will be available.



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

Java SE Modules

These modules are classified into two categories:

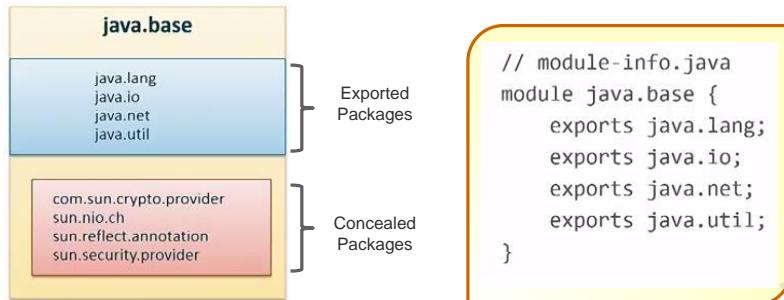
1. Standard modules (`java.*` prefix for module names):
 - Part of the Java SE specification.
 - For example: `java.sql` for database connectivity, `java.xml` for XML processing, and `java.logging` for logging
2. Modules not defined in the Java SE 9 platform (`jdk.*` prefix for module names):
 - Are specific to the JDK.
 - For example: `jdk.jshell`, `jdk.policytool`, `jdk.httpserver`



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

The Base Module

- The base module is `java.base`.
 - Every module depends on `java.base`, but this module doesn't depend on any other modules.
 - The base module exports all of the platform's core packages.



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

Finding the Right Platform Module

You can get a list of the packages a platform module contains with the `--describe-module` switch:

```
/home/oracle$ java --describe-module java.base
```

Partial Output is shown:

```
exports java.io  
exports java.lang  
exports java.lang.annotation  
exports java.lang.invoke  
exports java.lang.module  
exports java.lang.ref  
exports java.lang.reflect  
exports java.math  
exports java.net  
.....
```

1

The `java.base` module exports the `java.math` package.



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

Illegal Access to JDK Internals in JDK 9

- You can disable the warning message on a library-by-library basis by using the `--add-opens` command-line flag.
- For example, you can start Jython in the following way:

```
$java --add-opens java.base/sun.nio.ch=ALL-UNNAMED --add-opens  
java.base/java.io=ALL-UNNAMED -jar jython-standalone-2.7.0.jar  
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
```

- This time the warning is not issued because the Java invocation explicitly acknowledges the reflective access.
- As you can see, you may need to specify multiple `--add-opens` flags to cover all the reflective access operations that are attempted by libraries on the class path.



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

To better understand the behavior of tools and libraries, you can use the `--illegal-access=warn` command line flag. This flag causes a warning message to be issued for every illegal reflective-access operation. In addition, you can obtain detailed information about illegal reflective-access operations, including stack traces.

What Is a Custom Runtime Image?



- You can create a special distribution of the Java run time containing only the necessary modules.
 - Application modules and only those platform modules used by your application
- You can do this in Java SE 9 with *custom runtime image*.
- A custom runtime image is self-contained:
 - It bundles the application modules and platform modules with the JVM and everything else it needs to execute your application.



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

Link Time



- In Java SE 9, an optional **link time** is introduced between the compilation and runtime phase.
- Link time:
 - Requires a linking tool that will assemble and optimize a set of modules and their transitive dependencies to create a runtime image.



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

Using jlink to Create a Runtime Image

- A basic invocation of jlink:

```
jlink [options] --module-path modulepath --add-modules mods  
--output path
```

- You will have to specify the following three parameters:
 - modulepath: The module path where the platform and application modules to be added to the image are located. Modules can be modular JAR files, JMOD files, or exploded directories.
 - mods: The list of the modules to be added to the runtime image. The jlink tool adds these modules and their *transitive dependencies*.
 - path: The output directory where the generated runtime image will be stored.



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

JMOD format:

JDK 9 introduced a new format, called JMOD, to package modules. JMOD files are designed to handle more content types than JAR files can. The JDK 9 modules are packaged in JMOD format for you to use at compile time and link time. JMOD format is not supported at run time. You can package your own modules in JMOD format. Files in the JMOD format have a .jmod extension. JDK 9 ships with a new tool called jmod. It is located in the `JDK_HOME\bin` directory. It can be used to create a JMOD file, list the contents of a JMOD file, and print the description of a module.

Example: Using jlink to Create a Runtime Image

The following command creates a new runtime image:

```
/Hello$ jlink  
--module-path dist/Hello.jar:/usr/java/jdk-9/jmods  
--add-modules com.greeting  
--output myimage
```

- **--module-path:** This constructs a module path where HelloWorldApp is present, and the \$JAVA_HOME/jmods directory contains the platform modules.
- **--add-modules:** This indicates that com.greeting is the module that needs to be added in the runtime image.
- **--output:** This directory is where the runtime image generated, myimage, is stored.

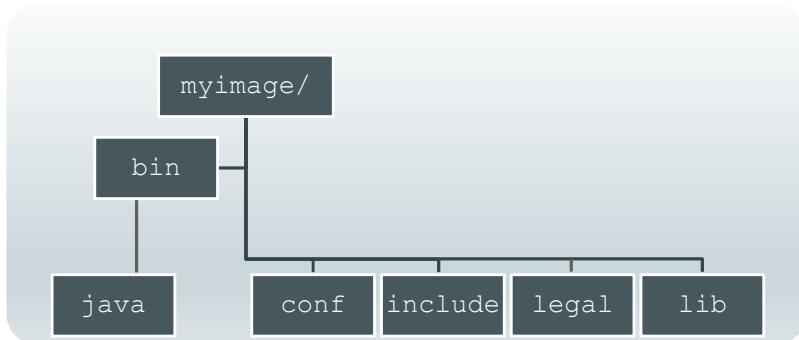
Note: In Windows, the path separator is ; instead of :



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

Examining the Generated Image

The generated image, `myimage`, has the following directory layout:



The custom run time generated is:

- Fully self-contained. It bundles the application modules with the JVM and everything else it needs to execute your application.
- Platform-specific and is not portable to other platforms.



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

bin: Contains executable files. On Windows, it also contains dynamically linked native libraries (.dll files).

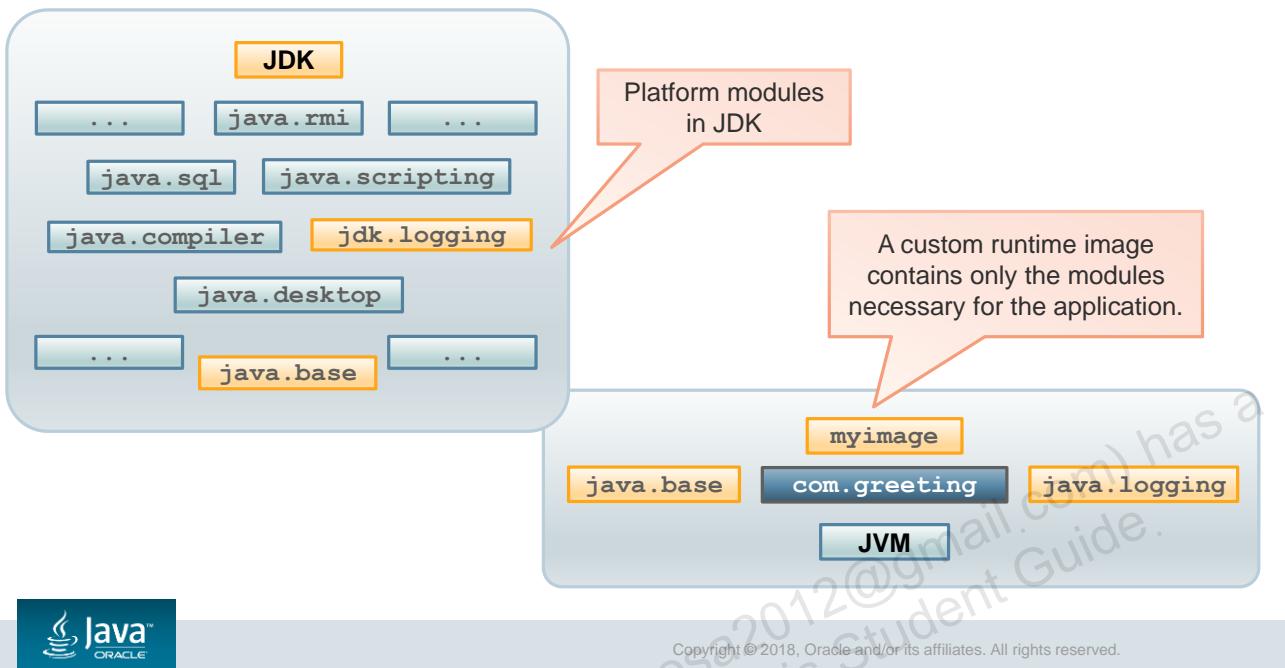
conf: Contains the editable configuration files such as .properties and .policy

include: Contains C/C++ header files

legal: Contains legal notices

lib: Contains, among other files, the modules added to the runtime image

Modules Resolved in a Custom Runtime Image



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

Advantages of a Custom Runtime Image

Creating a custom runtime image is beneficial for several reasons:

- Ease of use:
 - Can be shipped to your application users who don't have to download and install JRE separately to run the application.
- Reduced footprint:
 - Consists of only those modules that your application uses and therefore is much smaller than a full JDK
 - Can be used on resource-constrained devices or to run an application in the cloud
- Performance:
 - Runs faster because of link time optimizations that are otherwise too costly.



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

JIMAGE Format



- The runtime image is stored in a special format called JIMAGE, which is:
 - Optimized for space and speed
 - A much faster way to search and load classes than from JAR and JMOD file
- JDK 9 ships with the `jimage` tool to let you explore the contents of a JIMAGE file.



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

Running the Application

- You can use the `java` command, which is in `myimage`, to launch your application.

```
$ myimage/bin/ java --module com.greeting
```

```
$ myimage/bin/ java -m com.greeting
```

Name of the
module

- You don't have to set the module path. The custom runtime image is in its own module path.



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

Running the Application

The `jlink` command has a `--launcher` option that creates a platform-specific executable in the `bin` directory.

```
/Hello$ jlink  
--module-path dist/Hello.jar:/usr/java/jdk-9/jmods  
--add-modules com.greeting  
--launcher Hello=com.greeting  
--output myimage
```

Name of the module

You can use this executable to run your application:

```
$ myimage/bin Hello
```



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

The `--launcher` option makes `jlink` create a platform-specific executable such as a `Hello.bat` file on Windows in the `bin` directory. You can use this executable to run your application. The file contents are simply a wrapper for running the main class in this module. You can use this file to run the application.

Summary

In this lesson, you should have learned how to:

- Describe the purpose (in general terms) of the module-info class
- Create modules with defined module dependencies and module encapsulation
- Compile modules and create modular JAR files on the command line
- Describe how NetBeans IDE organizes its folders for source, compiled modules, and modular JARs



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

Practice 10: Overview

This practice covers the following topics:

- 10-1: Creating a modular application from the Command Line
- 10-2: Compiling modules from the Command Line
- 10-3: Creating a modular application from NetBeans
- 10-4: Requiring a module transitively
- 10-5: Beginning to modularize an older Java application
- 10-6: Creating and Optimizing a Custom Runtime Image by Using `jlink`
- 10-7: Using NetBeans to Create and Optimize a Runtime Image



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



Adolfo De+la+Rosa (adolfoleralrosa2012@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.

Migrating to a Modular Application



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



ORACLE®

Adolfo De+la+Rosa (adolfo.delarosa.2012@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Use `jdeps` to check the dependencies of individual JARs in a Java SE 8 application
- Describe the difference between top-down and bottom-up migration
- Use the class path and the module path to run a Java SE 9 application
- Describe split packages and how they can occur
- Describe cyclic dependencies and a way to address them



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

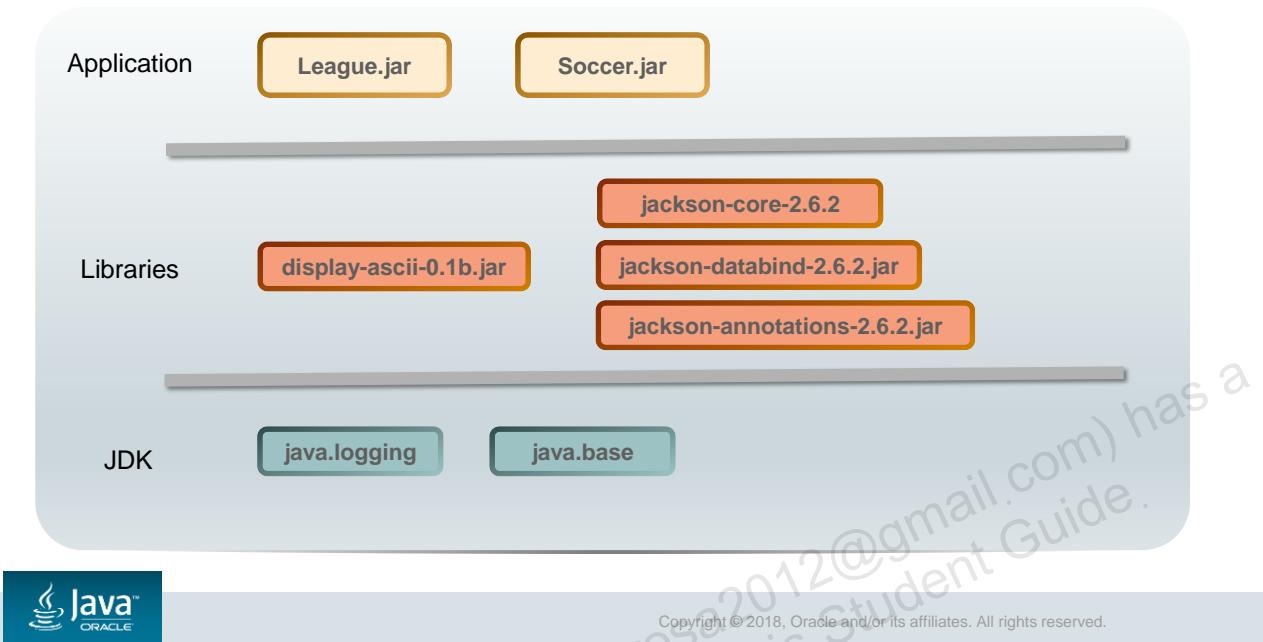
Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on libraries
- Split packages
- Cyclic dependencies



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

The League Application



The graphic in the slide shows a typical application. It has three layers: the application JAR files at the top, the JDK at the bottom, and a number of library JAR files in the middle. In an application written prior to Java SE 9, all of these three layers comprise JAR files. To run them, they are simply added to the class path and loaded as needed by the JVM.

Assume that even though the application and library JARs were written using a Java version prior to SE 9, you now want to run them on as a modularized application.

Given this three-layer application, it's important to understand that while the JDK is modularized, JARs can be run on the module path or on the class path as desired. This flexibility allows you to start by migrating the lowest layer, the libraries, first, while still running the application JARs on the class path without modularizing them. You can also migrate the application JARs first and run these modularized JARs with library JARs that have not been modularized. This is covered next in this lesson.

In this and subsequent slides, modules will be shown without the `.jar` suffix. (Also, note that modules do not need to be placed in a JAR file to function as modules.)

Run the Application

Using the class path:

```
java -cp \
    dist/League.jar: \
    lib/Soccer.jar: \
    lib/Basketball.jar: \
    lib/display-ascii-0.1b.jar: \
    lib/jackson-core-2.6.2: \
    lib/jackson-databind-2.6.2.jar: \
    lib/jackson-annotations-2.6.2.jar \
    main.Main
```



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

Here's the entire application being run on the class path, exactly as you would do for an application running on Java SE 8. But assuming this is running on a modular JDK, something slightly different is going on here as the modular JDK works only with modules. What happens is that if a JAR file is placed on the class path, as shown in this example, the module system will essentially treat all the packages as if they're in a special module called the unnamed module.

The Unnamed Module

- All types must be associated with a module in Java SE 9.
- A type is considered a member of the unnamed module if it is:
 - In a package not associated with any module
 - Loaded by the application
- Unnamed modules:
 - Read all other modules
 - Export all their packages
 - Cannot have any dependencies declared on them
 - Cannot be accessed by a named module
 - A named module is one with a `module-info.java` file.



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

Topics

- Application migration overview
- **Top-down migration**
- Bottom-up migration
- More on libraries
- Split packages
- Cyclic dependencies



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

Top-down Migration and Automatic Modules

Top-down migration refers to the order in which two JARs may be migrated to corresponding modules (where the migration is one to one).

- Given JARs `a.jar` and `b.jar` where some classes in `a.jar` depend on some classes in `b.jar`, top-down migration involves migrating `a.jar` first.
 - As a consequence, `b.jar` must be run on the module path as otherwise it cannot be accessed by what is now module `a`.
 - If run on the class path, it would become an unnamed module, and named modules cannot access unnamed modules.
 - By running `b.jar` on the module path, it becomes an automatic module with a name and can be accessed by module `a`.
 - Module `a` can now, in its `module-info.java` file, declare its dependency on the automatic module created from `b.jar`.
 - For example, if “`b`” is the name given to this automatically generated module based on `b.jar`, module `a` can refer to this automatic module with the directive:
 - `requires b;`

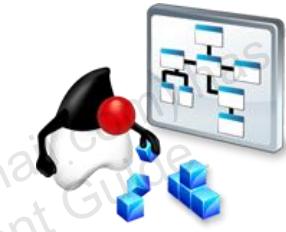


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Note that the distinction here is that in the case of two JARs, `a.jar` and `b.jar`, top-down migration means first migrating the JAR that depends on the other JAR. Classes in `a.jar` depend on classes in `b.jar`, so top-down migration means migrating `a.jar` first.

Automatic Module

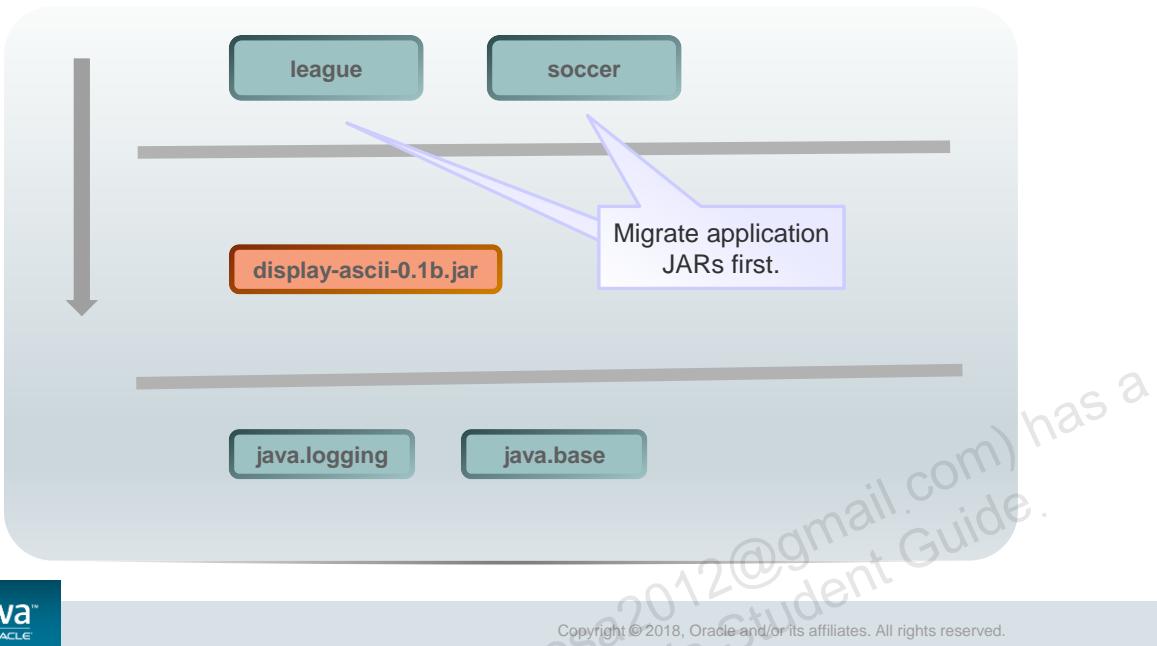
- Is a JAR file that does not have a module declaration and is placed on the module path
- Is a “real” module
- Requires no changes to someone else's JAR file
- Is given a name derived from the JAR file (either from its name or from metadata)
- Requires all other modules
- Can be required by other modules
- Exports all of its packages



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

Two unique things about automatic modules, that they require all other modules and that they export all their packages, are necessitated by the fact that they don't have their own `module-info.java` file.

Top-Down Migration



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

The graphic in the slide shows an example in which the migration has been performed top down. Here, the JAR files, League.jar and Soccer.jar, have been converted into two modules, league and soccer. This is done by adding a module-info.java file to the source code for each of the JAR files. This is shown in more detail in the following slides.

In this example, the JAR files, League.jar and Soccer.jar, were modularized at the same time, but this is still top-down migration because classes in League.jar depend on classes in display-ascii-0.1b.jar. Therefore, display-ascii-0.1b.jar will need to be become an automatic module.

Creating module-info.java—Determining Dependencies

Consider each JAR file that will become an application module—what does it require and what does it export?

- Module league requires?
- Module league exports?
- Module soccer requires?
- Module soccer exports?



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

Check Dependencies

Run **jdeps** to check dependencies:

```
jdeps -s lib/display-ascii-0.1b.jar lib/Soccer.jar dist/League.jar
League.jar -> lib/Soccer.jar
League.jar -> lib/display-ascii-0.1b.jar
League.jar -> java.base
League.jar -> java.logging
Soccer.jar -> java.base
Soccer.jar -> java.logging
display-ascii-0.1b.jar -> java.base
```



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

The **jdeps** command enables you to see what the dependencies are by looking at the JAR files. If **League.jar** becomes the league module, it will require the soccer module (created from **Soccer.jar**). But what about the requirement for **display-asci-0.1b.jar**? If you are performing top-down migration, the whole point is that you can migrate the application first, but use the libraries as they are. The next slides show how this is possible.

Library JAR to Automatic Module



Put the library JAR file on
the module path and ...

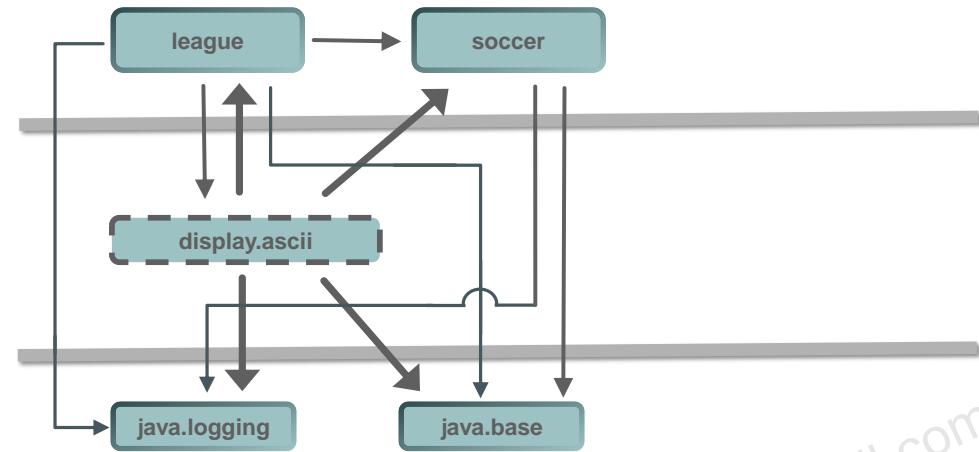
... it becomes an automatic
module.

display-ascii-0.1b.jar



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

Typical Application Modularized



This is the result. The `league` module can now require the `display.ascii` module, and the `display.ascii` module behaves like any other module, except that it requires every other module.

Note that in the graphics in this lesson, automatic modules are denoted with a dotted line rather than a solid one.



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

Topics

- Application migration overview
- Top-down migration
- **Bottom-up migration**
- More on Libraries
- Split packages
- Cyclic dependencies



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

Bottom-up Migration

Bottom-up migration refers to the order in which two JARs may be migrated to corresponding modules (where the migration is JAR to module one to one).

Given two JARs `a.jar` and `b.jar`, where some classes in `a.jar` depend on some classes in `b.jar`:

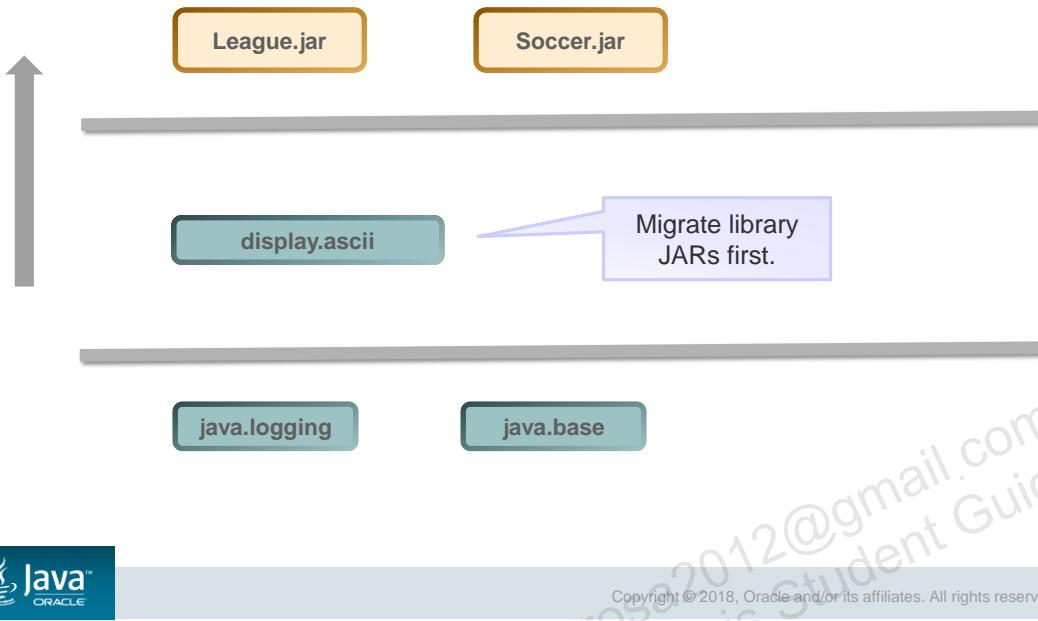
- Bottom-up migration involves migrating `b.jar` first.
 - This migrated module, `b`, can now be run on the module path.
 - JAR `a.jar` can run on the class path as an unnamed module or on the module path as an automatic module.



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

Note that the distinction here is that in the case of two JARs, `a.jar` and `b.jar`, bottom-up migration means first migrating the JAR that has the dependency on it. Classes in `a.jar` depend on classes in `b.jar`, so bottom-up migration means migrating `b.jar` first.

Bottom-Up Migration

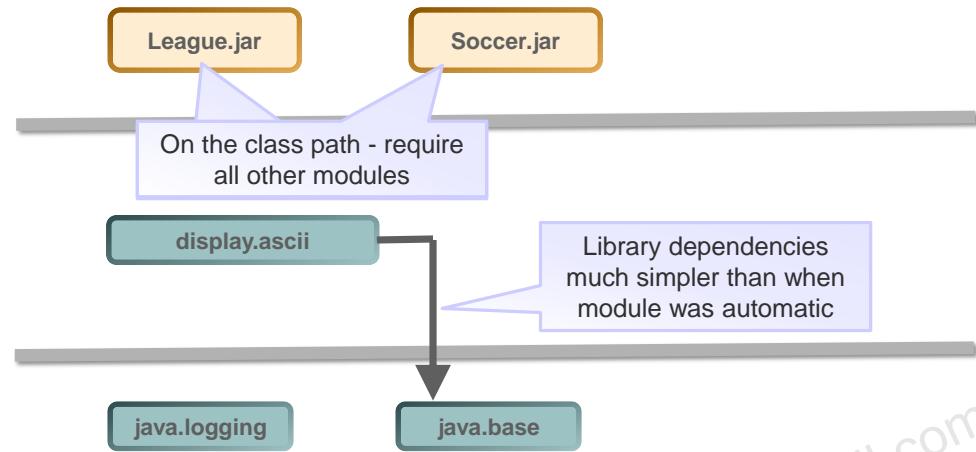


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The graphic in the slide shows bottom-up migration, where the library JAR, which was previously named `display-ascii-0.1b.jar`, has been converted to a module named `display.ascii`. The slide shows `League.jar` and `Soccer.jar` as not having been converted to modules.

Because the class path and the module path can be combined, the original application JARs can be run on the class path, while the `display.ascii` library module is run on the module path. It would also be possible to run `League.jar` and `Soccer.jar` on the module path—they would then run as automatic modules.

Modularized Library



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

Run Bottom-Up Migrated Application

Note use of `-add-modules`

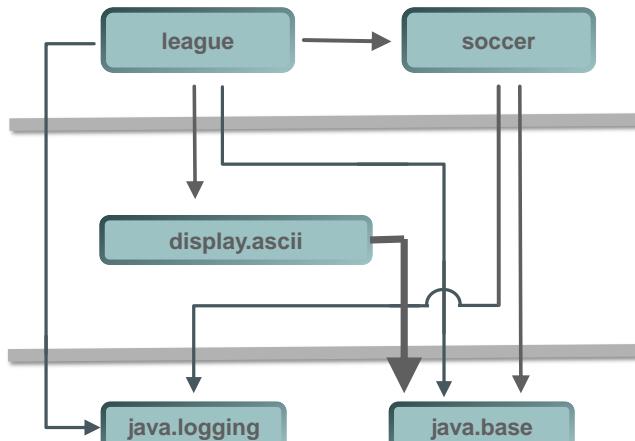
```
java -cp dist/League.jar:lib/Soccer.jar -p lib/display.ascii \
main.Main
java -cp dist/League.jar:lib/Soccer.jar -p mods/display.ascii \
--add-modules display.ascii main.Main
```



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

You need to use the `-add-modules` option in the `java` command because `League.jar` is running as an unnamed module. The Java run time cannot determine what modules to resolve, so `display.ascii` must be added manually as shown in the slide.

Fully Modularized Application



```
java -p dist/League.jar: \
      lib/Soccer.jar: \
      lib/display.ascii.jar \
      -m League.jar/main.Main
```

Robins win the league!

	Robins	Sparrows	Magpies	Crows	Points	Goals
Robins	X	0 - 1	0 - 4	1 - 1	2	3
Sparrows	2 - 1	X	1 - 1	1 - 0	8	7
Magpies	1 - 0	2 - 2	X	0 - 1	7	9
Crows	1 - 1	2 - 0	1 - 1	X	7	6



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

Whether starting from the bottom-up or the top-down, the migration process is completed when all JARs have been converted, that is, both the application JARs and the library JARs. In the example in the slide, the library JAR (after being modularized) is `display.ascii.jar`. This assumes that this library JAR is under the control of the organization doing the migration. But in many cases, this may not be true, and you may have to either continue to use the library as an automatic module or wait until the library maintainer creates a modularized version of the library.

Module Resolution

Use `--show-module-resolution` with `--limit-modules` to limit output.

```
java --limit-modules java.base,display.ascii,Soccer \
--show-module-resolution -p mods:lib -m league/main.Main
root league file:///home/oracle/examples/mods/league/
    league requires display.ascii
    file:///home/oracle/examples/league/mods/display.ascii/
    league requires Soccer
    file:///home/oracle/examples/league/mods/soccer/
    league requires java.logging jrt:/java.logging
    soccer requires java.logging jrt:/java.logging
    java.base binds java.logging jrt:/java.logging
```



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

The `--show-module-resolution` option will list all the modules that are being resolved as the application starts up. If you know exactly what modules are required, you can limit the modules that are resolved and thus the output with the `--limit-module` option.

Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on Libraries
- Split packages
- Cyclic dependencies

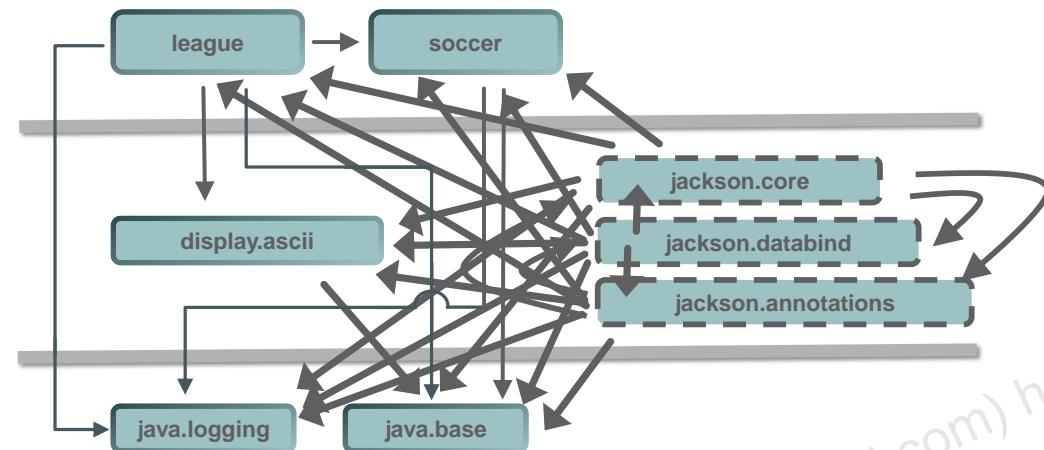


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

More About Libraries



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



The slide shows an example where there are some additional libraries that remain as automatic modules. The libraries in question are the Jackson JSON libraries. Because there are three JARs and because automatic modules require all other modules transitively, the resultant module graph is a little messy. But it still has much more dependency information in the overall graph than the corresponding class path.

Run Application with Jackson Libraries

- Compile:

```
javac -p lib -d mods --module-source-path src-9 \
      $(find src-9 -name "*.java")
```

- Run:

```
java -p mods:lib -m league/main.Main
Unable to make field private soccer.SoccerTeam soccer.Soccer.homeTeam
accessible: module Soccer does not "opens soccer" to module
jackson.databind (through reference chain: soccer.Soccer[0])
Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
...
```



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

You will find that many libraries use reflection. This will cause a problem because they won't be able to reflect on types within your application unless their packages have been exported. For types that are marked as private, exporting is not enough, and another directive `opens` is available.

`opens` is covered in more detail in the lesson titled “Working with the Module System.”

Open Soccer to Reflection from Jackson Libraries

- Open the entire module.

```
open module soccer {  
    requires java.logging;  
    exports soccer;  
    exports util;  
}
```

- Open just the package needed to all modules or to a specific module (as shown).

```
module soccer {  
    requires java.logging;  
    exports soccer;  
    exports util;  
    opens soccer to jackson.databind;  
}
```



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

Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on Libraries
- **Split packages**
- Cyclic dependencies



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

Split Packages

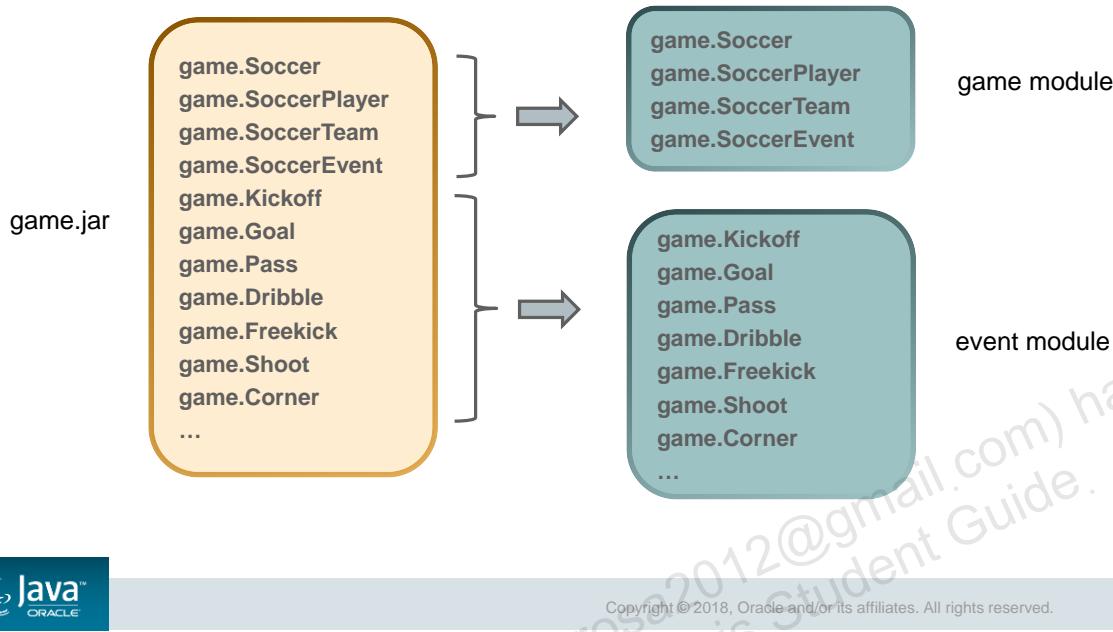
How they can occur during migration

- Splitting Java 8 application into modules
- Converting a Java 8 application that was organized for access control



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

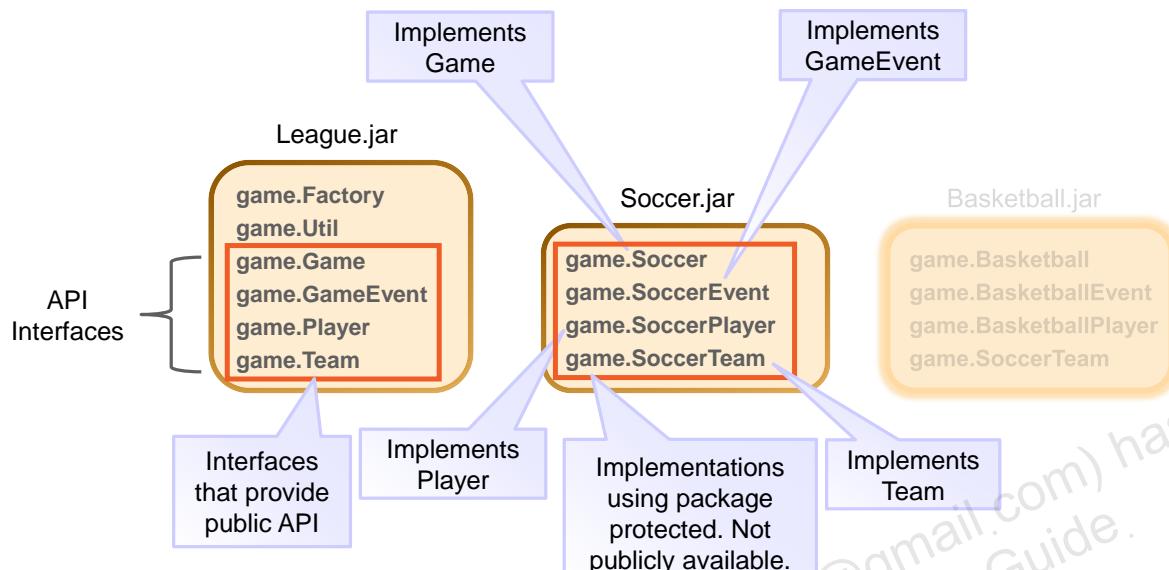
Splitting a Java 8 Application into Modules



In the example in the slide, the problem is caused by modifying a simply designed original Java SE 8 application so that it now uses modules in a logical way.

The solution is straightforward: refactor the classes to use a number of packages in a logical way, similar to the logical split shown by module in the diagram.

Java SE 8 Application Poorly Designed with Split Packages

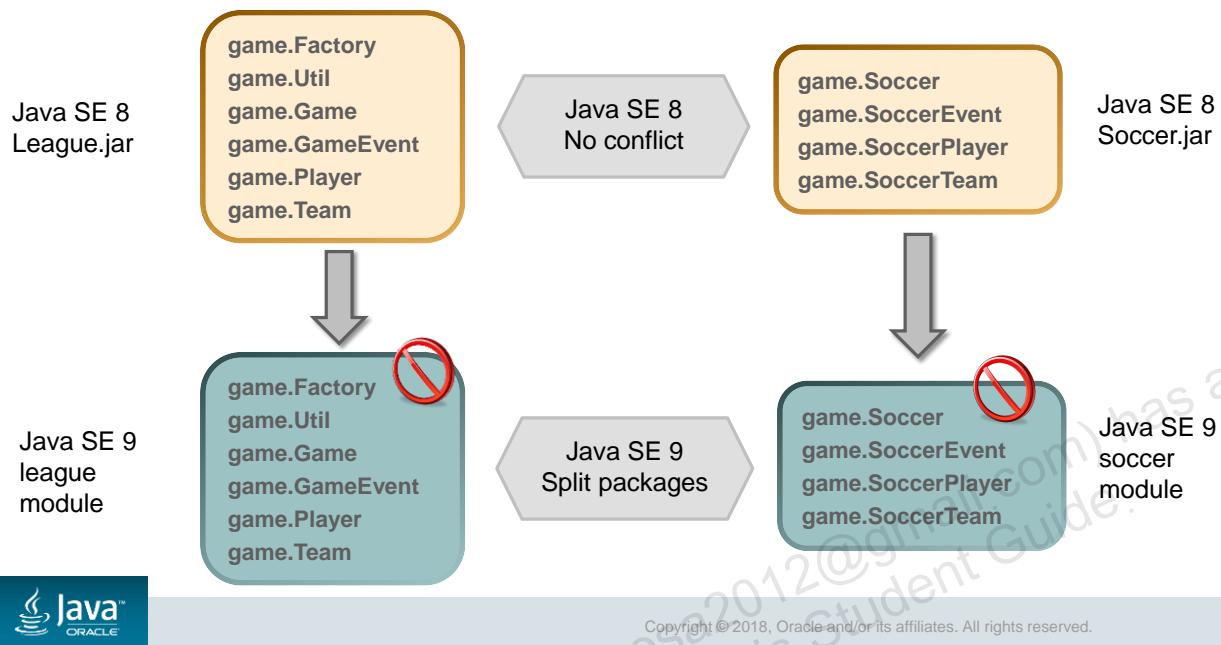


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

This example illustrates a split package problem, where split packages are used intentionally to allow the classes in a JAR to implement interfaces in another JAR but not have the implementations public. The idea is that if a package name is reused in another JAR, the classes in that package could be given package access and, therefore, could be accessible to classes in the same package that are stored in another JAR, but be inaccessible to other classes (that are not in that package).

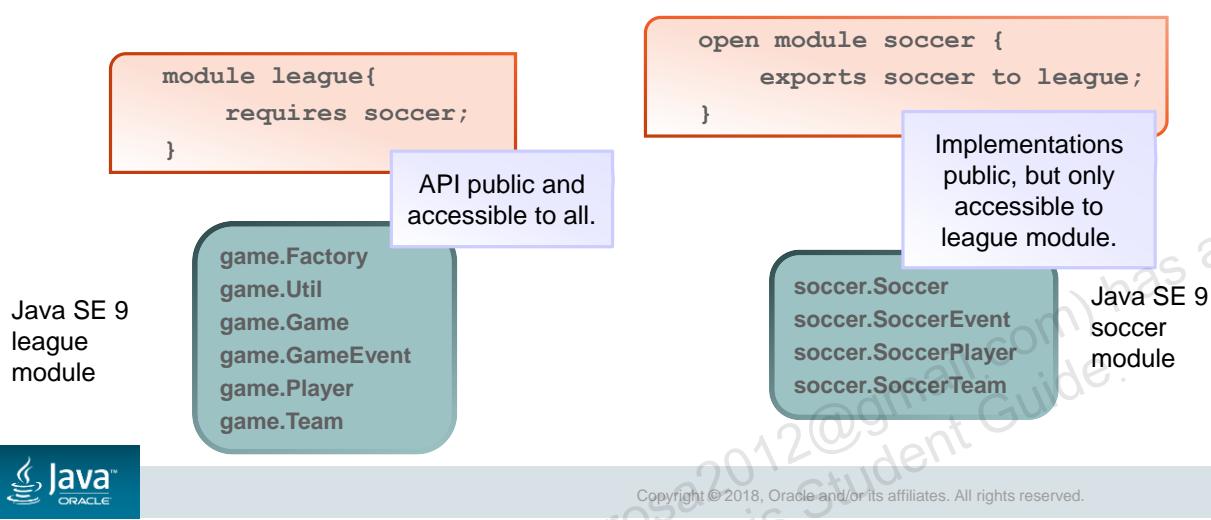
Note that this is not considered a good approach; it is a nonstandard and problematic attempt to achieve what is very easily achievable with modular Java.

Migration of Split Package JARs to Java SE 9



Addressing Split Packages

Encapsulation is achieved at the modular JAR level in Java SE 9 through the use of qualified export.



Topics

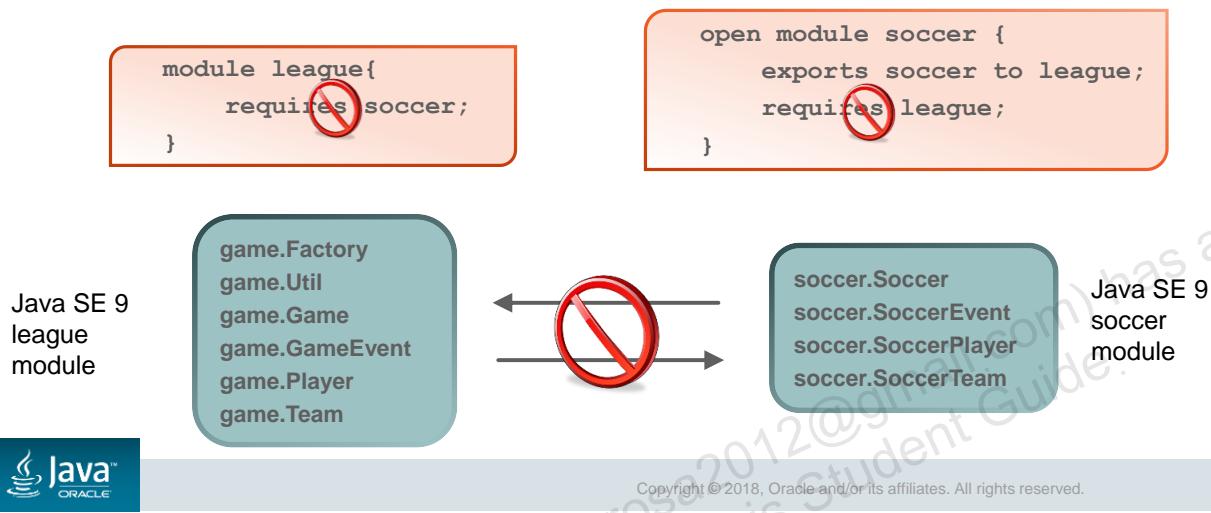
- Application migration overview
- Top-down migration
- Bottom-up migration
- More on Libraries
- Split packages
- Cyclic dependencies



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

Cyclic Dependencies

Cyclic module dependencies are not permitted in Java SE 9.

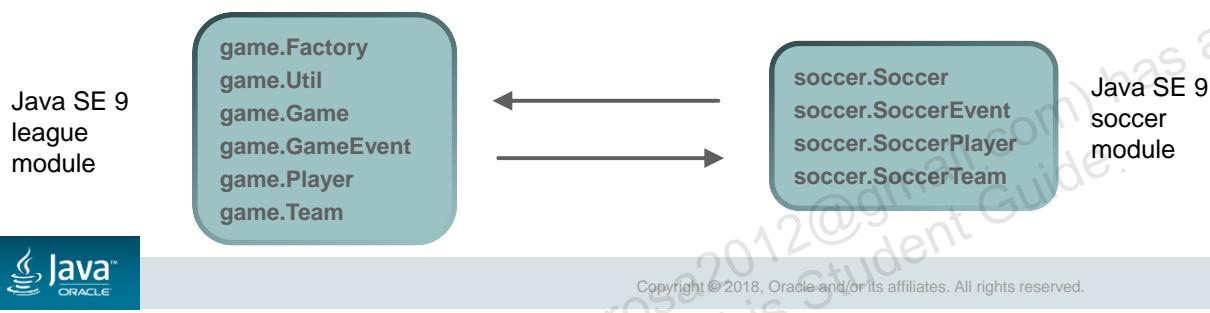


Migrating the previous split packages example will raise another issue. Implementation classes in the `soccer` module require access to the interfaces in the `league` module.

The `Factory` class in the `league` module needs access to implementation classes in the `soccer` module. This is a cyclic dependency and is not permitted in the module system.

Addressing Cyclic Dependency 1

One possible approach is to remove the dependency soccer has on league...



Addressing Cyclic Dependency 2

...and instead create a new module that both league and soccer are dependent on.



The graphic in the slide shows one way of addressing this cyclic dependency. You can create a new module to hold the various interfaces of the API so that the cycle no longer exists. This is not a solution for every case, but often the consideration of where to put the interface types may help suggest an answer.

There is another answer though, and that is the use of services. This is covered in the lesson titled “Services.”

Top-down or Bottom-up Migration Summary

Top-down or Bottom-up migration refers to the order in which JARs may be migrated to corresponding modules (where the migration is one to one JAR to module or modular JAR). Given two JARs `a.jar` and `b.jar`, where some classes in `a.jar` depend on some classes in `b.jar`:

- Top-down migration involves migrating `a.jar` first.
 - As a consequence, `b.jar` must become an automatic module as otherwise it cannot be accessed by what is now module `a`.
 - Both modules must be run on the module path.
- Bottom-up migration involves migrating `b.jar` first.
 - This migrated module, `b`, can now be run on the module path.
 - JAR `a.jar` can run on the class path as an unnamed module or on the module path as an automatic module.



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

Summary

In this lesson, you should have learned how to:

- Use `jdeps` to check the dependencies of individual JARs in a Java SE 8 application
- Describe the difference between top-down and bottom-up migrations
- Use the class path and the module path to run a Java SE 9 application
- Describe split packages and how they can occur
- Describe cyclic dependencies and a way to address them



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

Practice 11: Overview

This practice covers the following topics:

- Practice 11-1: Examining the League Application
- Practice 11-2: Using `jdeps` to Determine Dependencies
- Practice 11-3: Migrating the Application
- Practice 11-4: Adding a main Module
- Practice 11-5: Migrating a Library
- Practice 11-6: Bottom-Up Migration
- Practice 11-7: Adding the Jackson Library



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



Adolfo De+la+Rosa (adolfoleralrosa2012@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.