



Java SE 11: Programming Complete

Student Guide

D107120GC10 | 107910

Learn more from Oracle University at education.oracle.com



© 2018 International & Oracle Academy / Use Only

Author

Vasily Strelnikov

**Technical Contributors
and Reviewers**

Joe Greenwald
Svetlana Savvina
Jacobo Marcos

Editors

Moushmi Mukherjee
Aju Kumar

Graphic Editor

Kavya Bellur

Publishers

Sujatha Nagendra
Veena Narasimhan
Pavithran Adka
Srividya Rameshkumar

Copyright © 2020, 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.

1003242020

Contents

1 Introduction to Java

- Course Goals 1-2
- Audience 1-3
- Course Schedule 1-4
- Course Practices 1-5
- Lesson Objectives 1-6
- What Is Java? 1-7
- How Java Works? 1-8
- Classes 1-9
- Objects 1-10
- Inheritance 1-11
- Java APIs 1-12
- Java Keywords, Reserved Words, and a Special Identifier 1-13
- Java Naming Conventions 1-14
- Java Basic Syntax Rules 1-16
- Define Java Class 1-17
- Access Classes Across Packages 1-18
- Use Access Modifiers 1-19
- Create Main Application Class 1-20
- Compile Java Program 1-21
- Execute Java Program 1-22
- Comments and Documentation 1-23
- Summary 1-25
- Practices 1-26

2 Primitive Types, Operators, and Flow Control Statements

- Objectives 2-2
- Declare and Initialize Primitive Variables 2-4
- Restrictions on Primitive Declarations and Initializations 2-5
- Java Operators 2-6
- Assignment and Arithmetic Operators 2-7
- Arithmetic Operations and Type Casting 2-8
- More Mathematical Operations 2-9
- Binary Number Representation 2-10
- Bitwise Operators 2-11

Equality, Relational, and Conditional Operators	2-12
Short-Circuit Evaluation	2-13
Flow Control Using if/else Construct	2-14
Ternary Operator	2-15
Flow Control Using switch Construct	2-16
JShell	2-17
Summary	2-18
Practices	2-19

3 Text, Date, Time, and Numeric Objects

Objectives	3-2
String Initialization	3-3
String Operations	3-4
String Indexing	3-5
StringBuilder: Introduction	3-6
Wrapper Classes for Primitives	3-7
Representing Numbers Using BigDecimal Class	3-8
Method Chaining	3-9
Local Date and Time	3-10
More Local Date and Time Operations	3-11
Instants, Durations, and Periods	3-13
Zoned Date and Time	3-14
Represent Languages and Countries	3-15
Format and Parse Numeric Values	3-17
Format and Parse Date and Time Values	3-18
Localizable Resources	3-21
Format Message Patterns	3-22
Formatting and Localization: Example	3-23
Summary	3-24
Practices	3-25

4 Classes and Objects

Objectives	4-2
UML: Introduction	4-3
Modeling Classes	4-4
Modeling Interactions and Activities	4-6
Designing Classes	4-7
Creating Objects	4-8
Define Instance Variables	4-9
Define Instance Methods	4-10
Object Creation and Access: Example	4-11

Local Variables and Recursive Object Reference	4-12
Local Variable Type Inference	4-14
Define Constants	4-15
Static Context	4-16
Accessing Static Context	4-17
Combining Static and Final	4-18
Other Static Context Use Cases	4-19
NetBeans IDE: Introduction	4-21
Summary	4-22
Practices	4-23

5 Improved Class Design

Objectives	5-2
Overload Methods	5-3
Variable Number of Arguments	5-4
Define Constructors	5-5
Reuse Constructors	5-6
Access Modifiers Summary	5-7
Define Encapsulation	5-8
Define Immutability	5-9
Constants and Immutability	5-10
Enumerations	5-11
Complex Enumerations	5-12
Java Memory Allocation	5-13
Parameter Passing	5-14
Java Memory Cleanup	5-15
Summary	5-16
Practices	5-17

6 Inheritance

Objectives	6-2
Extend Classes	6-3
Object Class	6-4
Reuse Parent Class Code Through Inheritance	6-6
Instantiating Classes and Accessing Objects	6-7
Rules of Reference Type Casting	6-8
Verify Object Type Before Casting the Reference	6-9
Reference Code Within the Current or Parent Object	6-10
Define Subclass Constructors	6-11
Class and Object Initialization Summary	6-12
Override Methods and Use Polymorphism	6-14

Reuse Parent Class Logic in Overwritten Method	6-16
Define Abstract Classes and Methods	6-17
Define Final Classes and Methods	6-19
Override Object Class Operations: <code>toString</code>	6-20
Override Object Class Operations: <code>equals</code>	6-21
Override Object Class Operations: <code>hashCode</code>	6-22
Compare String Objects	6-23
Factory Method Pattern	6-24
Summary	6-25
Practices	6-26

7 Interfaces

Objectives	7-2
Java Interfaces	7-3
Multiple Inheritance Problem	7-4
Implement Interfaces	7-5
Default, Private, and Static Methods in Interfaces	7-6
Interface Hierarchy	7-7
Interface Is a Type	7-8
Functional Interfaces	7-9
Generics	7-10
Use Generics	7-11
Examples of Java Interfaces: <code>java.lang.Comparable</code>	7-13
Examples of Java Interfaces: <code>java.util.Comparator</code>	7-14
Examples of Java Interfaces: <code>java.lang.Cloneable</code>	7-15
Composition Pattern	7-16
Summary	7-17
Practices	7-18

8 Arrays and Loops

Objectives	8-2
Arrays	8-3
Combined Declaration, Creation, and Initialization of Arrays	8-4
Multidimensional Arrays	8-5
Copying Array Content	8-6
Arrays Class	8-7
Loops	8-8
Processing Arrays by Using Loops	8-9
Complex for Loops	8-10
Embedded Loops	8-11
Break and Continue	8-12

Summary 8-13

Practices 8-14

9 Collections

Objectives 9-2

Introduction to Java Collection API 9-3

Java Collection API Interfaces and Implementation Classes 9-4

Create List Object 9-5

Manage List Contents 9-6

Create Set Object 9-7

Manage Set Contents 9-8

Create Deque Object 9-9

Manage Deque Contents 9-10

Create HashMap Object 9-11

Manage HashMap Contents 9-12

Iterate through Collections 9-13

Other Collection Behaviors 9-14

Use java.util.Collections Class 9-15

Access Collections Concurrently 9-16

Prevent Collections Corruption 9-17

Legacy Collection Classes 9-19

Summary 9-20

Practices 9-21

10 Nested Classes and Lambda Expressions

Objectives 10-2

Types of Nested Classes 10-3

Static Nested Classes 10-6

Member Inner Classes 10-7

Local Inner Classes 10-9

Anonymous Inner Classes 10-10

Anonymous Inner Classes and Functional Interfaces 10-12

Understand Lambda Expressions 10-13

Define Lambda Expression Parameters and Body 10-14

Use Method References 10-15

Default and Static Methods in Functional Interfaces 10-16

Use Default and Static Methods of the Comparator Interface 10-17

Use Default and Static Methods of the Predicate Interface 10-18

Summary 10-19

Practices 10-20

11 Java Streams API

- Objectives 11-2
- Characteristics of Streams 11-3
- Create Streams Using Stream API 11-4
- Stream Pipeline Processing Operations 11-5
- Using Functional Interfaces 11-6
- Primitive Variants of Functional Interfaces 11-7
- Bi-argument Variants of Functional Interfaces 11-9
- Perform Actions with Stream Pipeline Elements 11-10
- Perform Filtering of Stream Pipeline Elements 11-11
- Perform Mapping of Stream Pipeline Elements 11-12
- Join Streams using flatMap Operation 11-13
- Other Intermediate Stream Operations 11-14
- Short-Circuit Terminal Operations 11-15
- Process Stream Using count, min, max, sum, average Operations 11-16
- Aggregate Stream Data using reduce Operation 11-17
- General Logic of the collect Operation 11-19
- Using Basic Collectors 11-20
- Perform a Conversion of a Collector Result 11-21
- Perform Grouping or Partitioning of the Stream Content 11-22
- Mapping and Filtering with Respect to Groups or Partitions 11-23
- Parallel Stream Processing 11-25
- Parallel Stream Processing Guidelines 11-26
- Restrictions on Parallel Stream Processing 11-27
- Summary 11-29
- Practices 11-30

12 Handle Exceptions and Fix Bugs

- Objectives 12-2
- Using Java Logging API 12-3
- Logging Method Categories 12-4
- Guarded Logging 12-6
- Log Writing Handling 12-7
- Logging Configuration 12-9
- Describe Java Exceptions 12-10
- Create Custom Exceptions 12-11
- Throwing Exceptions 12-12
- Catching Exceptions 12-13
- Exceptions and the Execution Flow 12-14
- Example Throwing an Unchecked Exception 12-15

Example Throwing a Checked Exception	12-16
Handling Exceptions	12-17
Resource Auto-Closure	12-18
Suppressed Exceptions	12-19
Handle Exception Cause	12-20
Java Debugger	12-21
Debugger Actions	12-22
Manipulate Program Data in Debug Mode	12-23
Validate Program Logic Using Assertions	12-24
Normal Program Flow with No Exceptions	12-25
Program Flow Producing a Runtime Exception	12-27
Program Flow Catching Specific Checked Exception	12-29
Program Flow Catching Any Exceptions	12-31
Summary	12-33
Practices	12-34

13 Java IO API

Objectives	13-2
Java Input-Output Principals	13-3
Java Input-Output API	13-4
Reading and Writing Binary Data	13-5
Basic Binary Data Reading and Writing	13-6
Reading and Writing Character Data	13-8
Basic Character Data Reading and Writing	13-9
Connecting Streams	13-10
Standard Input and Output	13-11
Using Console	13-12
Understand Serialization	13-14
Serializable Object Graph	13-15
Object Serialization	13-16
Serialization of Sensitive Information	13-17
Customize Serialization Process	13-18
Serialization and Versioning	13-19
Working with Filesystems	13-20
Constructing Filesystem Paths	13-22
Navigating the Filesystem	13-24
Analyse Path Properties	13-25
Set Path Properties	13-26
Create Paths	13-28
Create Temporary Files and Folders	13-29
Copy and Move Paths	13-30

Delete Paths	13-31
Handle Zip Archives	13-32
Represent Zip Archive as a FileSystem	13-33
Access HTTP Resources	13-35
Summary	13-36
Practices	13-37

14 Java Concurrency and Multithreading

Objectives	14-2
Java Concurrency Concepts	14-3
Implement Threads	14-5
Thread Life Cycle	14-6
Interrupt Thread	14-7
Block Thread	14-8
Make Thread Wait Until Notified	14-9
Common Thread Properties	14-10
Create Executor Service Objects	14-11
Manage Executor Service Life Cycle	14-15
Implementing Executor Service Tasks	14-17
Locking Problems	14-19
Writing Thread-Safe Code	14-20
Ensure Consistent Access to Shared Data	14-22
Non-Blocking Atomic Actions	14-23
Ensure Exclusive Object Access Using Intrinsic Locks	14-24
Intrinsic Lock Automation	14-25
Non-Blocking Concurrency Automation	14-27
Alternative Locking Mechanisms	14-28
Summary	14-29
Practices	14-30

15 Java Modules

Objectives	15-2
Compile, Package, and Execute Non-Modular Java Applications	15-3
What Is a Module?	15-5
Java Platform Module System (JPMS)	15-7
JPMS Module Categories	15-8
Define Module Dependencies	15-9
Export Module Content	15-10
Modules Example	15-11
Open Module Content	15-12
Open an Entire Module	15-13

Produce and Consume Services	15-14
Services Example	15-15
Multi-Release Module Archives	15-16
Compile and Package a Module	15-17
Execute a Modularized Application	15-18
Migrating Legacy Java Applications Using Automatic Modules	15-19
Create Custom Runtime Image	15-20
Execute Runtime Image	15-22
Optimize a Custom Runtime Image	15-23
Summary	15-25
Practices	15-26

A Annotations

Objectives	A-2
Introduction to Annotations	A-3
Design Annotations	A-4
Apply Annotations	A-5
Dynamically Discover Annotations	A-7
Document the Use of Annotations	A-9
Annotations that Validate Design	A-10
Deprecated Annotation	A-11
Suppress Compiler Warnings	A-12
Var-args and Heap Pollution	A-13
Summary	A-14

B Java Database Connectivity

Objectives	B-2
Java Database Connectivity (JDBC)	B-3
JDBC API Structure	B-4
Manage Database Connections	B-5
Create and Execute Basic SQL Statements	B-6
Create and Execute Prepared SQL Statements	B-7
Create and Execute Callable SQL Statements	B-8
Process Query Results	B-9
Control Transactions	B-11
Discover Metadata	B-12
Customize ResultSet	B-13
Set Up ResultSet Type	B-14
Set Up ResultSet Concurrency and Holdability	B-16
Summary	B-17

C Java Security

- Objectives C-2
- Security Threats C-3
- Denial of Service (DoS) Attack C-4
- Define Security Policies C-5
- Control Access Using Permissions C-6
- Execute Privileged Code C-7
- Secure Filesystem and IO Operations C-8
- Best Practices for Protecting your Code C-9
- Erroneous Value Guards C-10
- Protect Sensitive Data (Part 1) C-11
- Protect Sensitive Data (Part 2) C-12
- Prevent JavaScript Injections C-14
- Prevent XML Injections C-15
- Discover and Document Security Issues C-16
- Summary C-17

D Advanced Generics

- Objectives D-2
- Compiler Erases Information About Generics D-3
- Generic and Raw Type Compatibility D-4
- Generics and Type Hierarchy D-5
- Wildcard Generics D-6
- Upper Bound Wildcard D-7
- Lower Bound Wildcard D-8
- Collections and Generics Best Practices D-9
- Summary D-10

Introduction to Java



ORACLE®

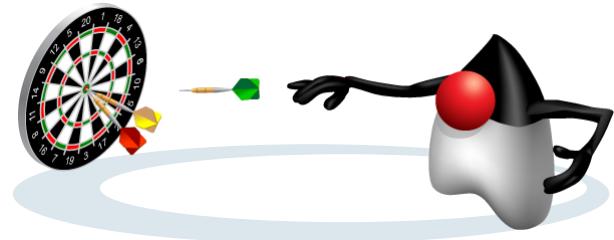


Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Course Goals

In this course, you learn how to implement application logic using Java SE:

- Describe the object-oriented programming approach
- Explain Java syntax and coding conventions
- Use Java constructs and operators
- Use core Java APIs, such as Collections, Streams, IO, and Concurrency
- Deploy Java SE applications



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Audience

The target audience includes those who:

- Have some non-Java programming experience and want to learn Java
- Have basic knowledge of Java and want to improve it
- Prepare for the Java SE 11 Certification exams

Briefly introduce yourself:

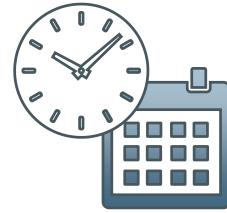
- Name
- Title or position
- Company
- Experience with programming
- Reasons for attending



Course Schedule

Day One

- Lesson 1: Introduction to Java
- Lesson 2: Primitive Types, Operators, and Flow Control Statements
- Lesson 3: Text, Date, Time, and Numeric Objects

**Day Two**

- Lesson 4: Classes and Objects
- Lesson 5: Improved Class Design
- Lesson 6: Inheritance

Day Three

- Lesson 7: Interfaces
- Lesson 8: Arrays and Loops
- Lesson 9: Collections

Day Four

- Lesson 10: Nested Classes and Lambda Expressions
- Lesson 11: Java Streams API
- Lesson 12: Handle Exceptions and Fix Bugs

Extras:

- Appendix A: Annotations
- Appendix B: JDBC API
- Appendix C: Security
- Appendix D: Generics

Day Five

- Lesson 13: Java IO API
- Lesson 14: Java Concurrency and Multithreading
- Lesson 15: Java Modules



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

This schedule is for guidance purposes only. Depending on the pace of practices, exact timings may vary.

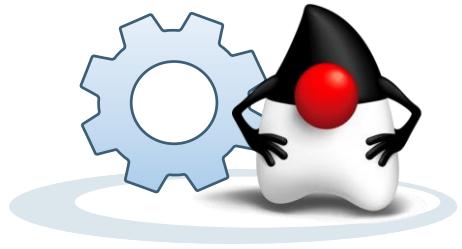
Course Practices

During the course practice sessions, you:

- Explore the features of Java language
- Apply the knowledge gained throughout the course to develop a product management application

The practice environment uses:

- JDK 11
- JShell
- NetBeans 11



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Lesson Objectives



After completing this lesson, you should be able to:

- Discover Java language origins and use-cases
- Explain Java portability and provider neutrality
- Explain object-oriented concepts
- Describe Java syntax and coding conventions
- Create Java class with main method
- Compile and execute a Java application

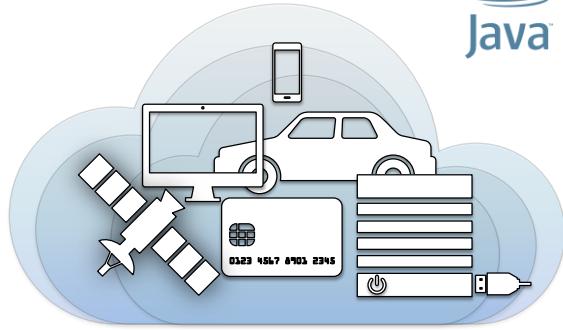


Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

What Is Java?

- General purpose programming language similar to C and C++
- Object-Oriented and Platform-Independent
- Originally designed in 1995 for use in consumer electronics
- Modern uses include writing applications for Internet of things, Cloud computing, and so on.
- This course covers Java SE (Standard Edition) version 11.



❖ **Java Editions:**

*Java Card - Smart card Edition
Java ME - Micro Edition
Java SE - Standard Edition
Java EE - Enterprise Edition*

❖ **Java SE is the base edition on which other editions are based.**



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

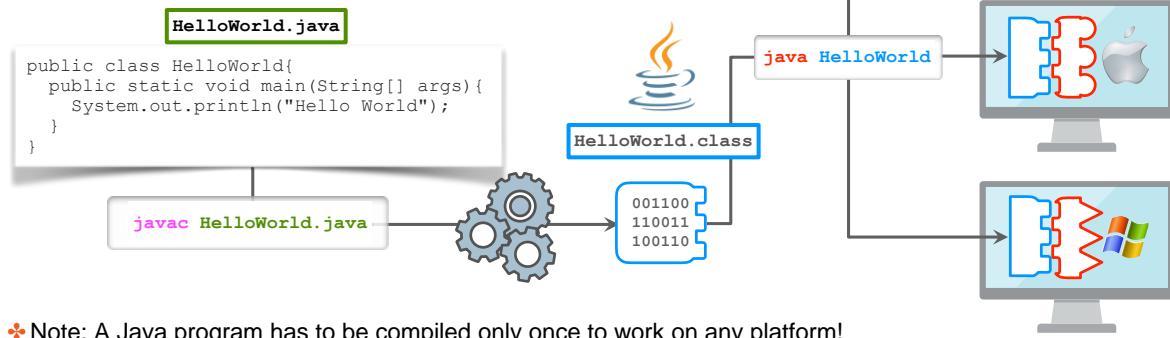
You start learning Java with Java SE because this is the base edition, representing the core of the Java language. All other Java editions represent more specialized use-cases of Java that are targeting particular environments, such as SIM cards (Java card), smart TVs (Java ME), or Application Servers (Java EE).

Starting from version 9, new versions of Java are released every 6 months. However, only some of these new versions are considered to be LTS (Long Term Support) versions. This course is based on Java SE version 11, which is the LTS version, previous LTS version is 8.

How Java Works?

Java is a platform-independent programming language.

- Java **source code** is written as plain text .java files.
- Source code is **compiled** into **byte-code** .class files for JVM.
- **Java Virtual Machine** must be installed on a target computer.
- JVM executes your application by translating Java byte-code instructions to platform-specific code.



❖ Note: A Java program has to be compiled only once to work on any platform!



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

You need to have Java Virtual Machine (JVM) installed on a target computer where you want your Java program to be executed. JVM is itself platform specific and its purpose is to translate platform-independent byte-code instructions contained in .class files into platform-specific instructions that the target computer would be able to execute.

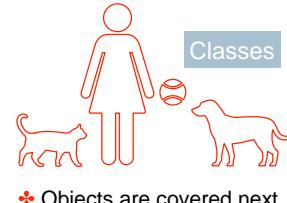
Optionally, since Java 9, it is possible to package your Java program. Its deployment would contain a JVM and can be installed to a target computer as a platform-specific executable; therefore, you do not require a separate JVM installation.

Classes

Class and Object are two key object-oriented concepts.

Java code is structured with **classes**.

- Class represents a type of thing or a concept, such as Dog, Cat, Ball, Person.
- Each class defines what kind of information (**attributes**) it can store:
 - A Dog could have a name, color, size.
 - A Ball would have type, material, and so on.
- Each class defines what kind of behaviors (**operations**) containing program logic (**algorithms**) it is capable of:
 - A Dog could bark and fetch a Ball.
 - A Cat could meow but is not likely to play fetch.



✖ Objects are covered next

```
class Person {
    void play() {
        Dog dog = new Dog();
        dog.name = "Rex";
        Ball ball = new Ball();
        dog.fetch(ball);
    }
}

class Dog {
    String name;
    fetch(Ball ball) {
        ball.find();
        ball.chew();
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

You may consider the words "function," "procedure," "operation," "method," and "behavior" to be synonymous. They all represent the same concept, a way of defining logic and algorithms within the class.

You may also consider the words "variable," "attribute," and "field" to be synonymous. They all represent the same concept, a way of defining information and data storage within the class.

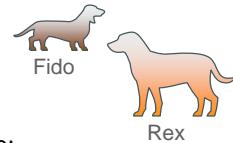
Java has procedural language capabilities, that is, you write algorithms contained within functions defined by your classes.

Objects

An Object is a specific **instance** (example of) a Class.

- Each object would be capable of having **specific values** for each attribute defined by a class that represents its type. For example:
 - A dog could be called Fido, and be brown and small.
 - Another could be called Rex, and be orange and big
- To operate on an object, you can **reference** it using a variable of a relevant type.
- Each object would be capable of behaviors defined by a class that represents its type:
 - At run time, objects **invoke operations** upon each other to execute program logic.

Objects



```
class Person {
    void play() {
        Dog dog = new Dog();
        dog.name = "Rex";
        Ball ball = new Ball();
        dog.fetch(ball);
    }
}

class Dog {
    String name;
    fetch(Ball ball) {
        ball.find();
        ball.chew();
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

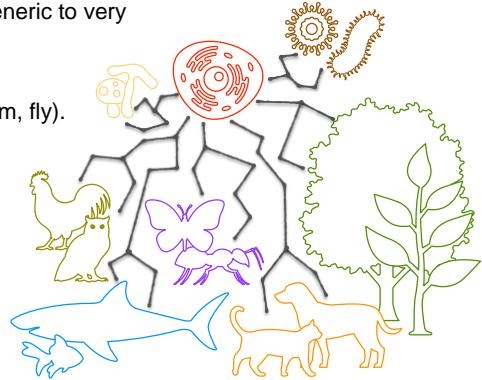
Note: The term Class means type and Object means instance of that type. However, sometimes people use these terms interchangeably, which can be quite confusing.

Inheritance

You can reuse (inherit) attributes and behaviors across class hierarchy.

- Classes can form hierarchical relationships.
- Superclass represents a more generic, parent type (living organism).
- Superclasses define common attributes and behaviors (eat, propagate).
- A subclass represents a more specific, child type (animal, plant, and so on).
- There could be any number of levels in the hierarchy, from very generic to very specific child types (dog, cat, and so on).
- Subclasses inherit all attributes and behaviors from their parents.
- Subclasses can define more specific attributes and behaviors (swim, fly).

```
class Animal extends LivingOrganism {  
    // generic attributes and behaviours  
}  
  
class Dog extends Animal {  
    // specific attributes and behaviours  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Java APIs

Java Development Kit (JDK) provides hundreds of classes for various programming purposes:

- To represent basic data types, for example, `String`, `LocalDateTime`, `BigDecimal`, and so on
- To manipulate collections, for example, `Enumeration`, `ArrayList`, `HashMap`, and so on
- To handle generic behaviors and perform system actions, for example, `System`, `Object`, `Class`, and so on
- To perform input/output (I/O) operations, for example, `FileInputStream`, `FileOutputStream`, and so on
- Many other API classes are used to access databases, manage concurrency, enable network communications, execute scripts, manage transactions, security, logging, build graphical user interfaces, and so on

- ❖ Application Programming Interface (API) is a term that describes a collection of classes that are designed to serve a common purpose.
- ❖ All Java APIs are thoroughly documented for each version of the language. For example, Java 11 documentation can be found at:
<https://docs.oracle.com/en/java/javase/11/docs/api/>



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Java Keywords, Reserved Words, and a Special Identifier

Keywords available since 1.0

Keywords no longer in use

Keywords added in 1.2

Keywords added in 1.4

Keywords added in 5.0

Keywords added in 9.0

Reserved words for literals values

Special identifier added in 10

if	implements	boolean	assert
else	extends	try	enum
continue	interface	catch	module
break	class	finally	requires
for	static	throw	transitive
do	final	throws	exports to
while	return	new	uses
switch	transient	this	provides
case	void	super	with
default	byte	instanceof	opens to
private	short	native	
protected	int	synchronized	true
public	long	volatile	false
import	char	goto	null
package	float	const	
abstract	double	strictfp	var

Notes

- ❖ Keywords and Literals cannot be used as identifiers (names of classes, variables, methods, and so on).
- ❖ Actual use and meaning of these keywords and literals are covered later in this course.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

All Java keywords and literals are lowercase. You cannot use a keyword or a literal as an identifier, that is, the name of a package, variable, class, or method. You can use a special identifier as a variable name (bad idea, code looks very confusing), but not as a class name.

Java Naming Conventions

- Java is case-sensitive; Dog is not the same as dog.
- Package name is a reverse of your company domain name, plus the naming system adopted within your company.
- Class name should be a noun, in mixed case with the first letter of each word capitalized.
- Variable name should be in mixed case starting with a lowercase letter; further words start with capital letters.
- Names should not start with numeric characters (0-9), underscore _ or dollar \$ symbols.
- Constant name is typically written in uppercase with underscore symbols between words.
- Method name should be a verb, in mixed case starting with a lowercase letter; further words start with capital letters.

```
package: com.oracle.demos.animals
class: ShepherdDog
variable: shepherdDog
constant: MIN_SIZE
method: giveMePaw
```

```
package: animals
class: Shepherd Dog
variable: _price
constant: minSize
method: xyz
```



✖ Note: The use of the _ symbol as a first or only character in a variable name produces a compiler warning in Java 8 and an error in Java 9 onward.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

The prefix of a unique package name is always written in lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.

Class names should be nouns, in mixed case, with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words; avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

Variable names are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign \$ characters, even though both are allowed.

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic, that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers, c, d, and e for characters.

A constant name is typically written in uppercase with the underscore symbol between words.

The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

Java Basic Syntax Rules

- All Java statements must be terminated with the " ; " symbol.
- Code blocks must be enclosed with " { " and " } " symbols.
- Indentations and spaces help readability, but are syntactically irrelevant.

```
package com.oracle.demos.animals;
class Dog {
    void fetch() {
        while (ball == null) {
            keepLooking();
        }
    }
    void makeNoise() {
        if (ball != null) {
            dropBall();
        } else {
            bark();
        }
    }
}
```

✿ Note: Example shows some constructs such as `if/else` and `while` that are covered later in the course.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

Define Java Class

- **Class name** is typically represented by one or more nouns, for example, Dog, SabreToothedCat, Person.
- Class must be saved into a file with the same name as the class and extension **.java**.
- Classes are grouped into packages.
- Packages are represented as folders where class files are saved.
- **Package name** is typically a reverse of your company domain name, plus a naming system adopted within your company.
Example: com.oracle.demos, org.acme.something
- Package and class name must form a unique combination.

/somepath/com/oracle/demos/animals/Dog.java

```
package <package name>;
class <ClassName> {

}

package com.oracle.demos.animals;
class Dog {
    // the rest of this class code
}
```

- ❖ Note: If package definition is missing, class would belong to a "default" package and would not be placed into any package folder. However, this is not a recommended practice.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

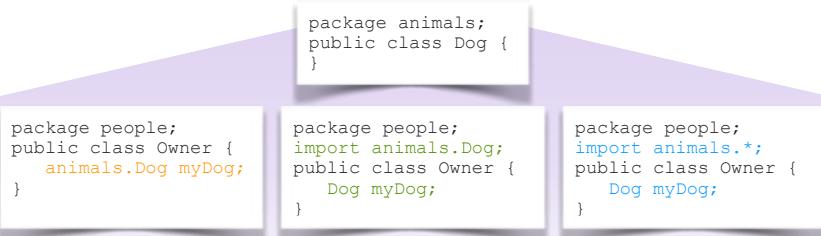
Typically each .java contains only one class definition. However, in some cases, for example with Nested and Inner classes, it is possible to define more than one class in a single .java file. These cases are covered in later lessons.

Access Classes Across Packages

To access a class in another package:

- Prefix the class name with the package name
- Use the import statement to import specific classes or the entire package content
 - The import of all classes from the `java.lang.*` package is implicitly assumed.

The example shows three alternative ways of referencing the class `Dog` in the package `animals` from the class `Owner` in the package `people`:



Notes

- ❖ Imports are not present in a compiled code. An `import` statement has no effect at runtime efficiency of the class. It is a simple convenience to avoid prefixing class name with package name throughout your source code.
- ❖ Access modifiers (such as `public`) are explained in the following slide.



Use Access Modifiers

Access modifiers describe the visibility of classes, variables, and methods.

- `public` - Visible to any other class
- `protected` - Visible to classes that are in the same package or to subclasses
- `<default>` - Visible only to classes in the same package
- `private` - Visible only within the same class

```
package <package name>;
import <package name>.<class name>;
import <package name>.*;
<access modifier> class <ClassName> {
    <access modifier> <variable definition>
    <access modifier> <method definition>
}
```

```
package b;
import a.*;
public class Y extends X {
    public void doThings() {
        X x = new X();
        x.y1; ✓✓
        x.y2; ✓✓
        x.y3; ✗
        x.y4; ✗
    }
}
```

```
package a;
public class X {
    public Y y1;
    protected Y y2;
    Y y3;
    private Y y4;
}
```

Notes

- ❖ `<default>` means that no access modifier is explicitly set.
- ❖ Subclass-Superclass relationship (use of the `extends` keyword) is covered later in the course.
- ❖ Any nonprivate parts of your class should be kept as stable as possible, because changes of such code may adversely affect any number of other classes that may be using your code.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

In this example, class Y is a subclass of class X, as defined by the `extends` clause. Therefore, class Y can access protected members of class X, although it is in a different package. However, because class Y is in a different package from class X, class Y is unable to access the default members of class X.

Create Main Application Class

The `main` method is the entry point into your application.

- It is the starting point of program execution.
- The method name must be called `main`.
- It must be `public`. You intend to invoke this method from outside of this class.
- It must be `static`. Such methods can be invoked without creating an instance of this class.
- It must be `void`. It does not return a value.
- It must accept `array of String objects` as the only parameter.
(The name of this parameter "args" is irrelevant.)

```
package demos;
public class Whatever {
    public static void main(String[] args) {
        // program execution starts here
    }
}
```

✿ Note: Use of `static` and `void` keywords and handling of arrays are covered later in the course.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

You can define the `main` method in different ways:

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

There is no practical difference between these approaches; however, the first one is probably the most common.

Compile Java Program

Compile classes with the **javac** Java compiler.

- The **-classpath** or **-cp** parameter points to **locations of other classes** that may be required to compile your code.
- The **-d** parameter points to a **path to store compilation result**.
(The compiler creates **package subfolders** with **compiled class files** in this path.)
- Provide **path to source code**.

```
javac -cp /project/classes -d /project/classes /project/sources/demos/Whatever.java
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

21

You do not have to use the **-cp** parameter to compile or execute a class if this class is not referencing any other classes of yours. You do not have to use the **-d** parameter if your class is in a default package; however, this is not a recommended practice.

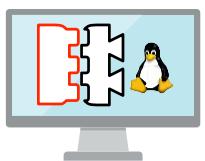
Execute Java Program

Execute program using `java` executable Java Virtual Machine (JVM).

- Specify `-classpath` or `-cp` to point to **folders where your classes are located**.
- Specify **fully qualified class name**. Use package prefix; do not use the `.class` extension.
- Provide a **space separated list of parameters** after the class name.

Access command-line parameters:

- Use `array object` to access parameters.
- Array `index` starts at **0** (first parameter).



```
java -cp /project/classes demos.Whatever Jo John "A Name" Jane
Hello John
```

```
package demos;
public class Whatever {
    public static void main(String[] args) {
        String param1 = args[1];
        System.out.println("Hello "+param1);
    }
}
```

- Since Java 11, it is also possible to run **single-file source code** as if it is a compiled class. JVM will interpret your code, but no compiled class file would be created:

```
java /project/sources/demos/Whatever.java
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

Also, consider this example:

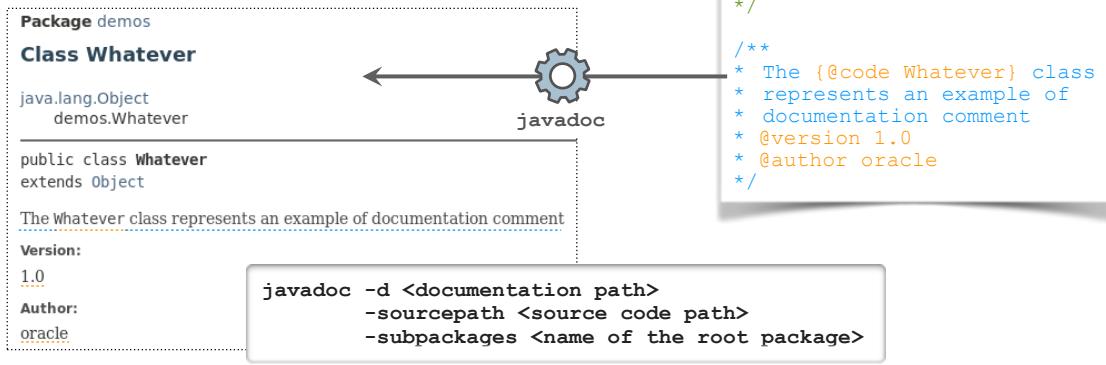
```
cd /projects/classes
javac -d /project/classes
/project/sources/demos/Whatever.java
java demos.Whatever
```

This example works without setting the class path because both Java compiler and virtual machine consider the current folder as the default class path. It is also possible to set class path as an environment variable.

Launching single-file source code as if it is a compiled program ability is a new feature in Java 11. However, it is not very practical; the alternative is to use the JShell utility, which is covered later in the course.

Comments and Documentation

- **Code Comments** can be placed anywhere in your source code.
- **Documentation Comments:**
 - May contain HTML markups
 - May contain **descriptive tags** prefixed with @ sing
 - Are used by the `javadoc` tool to generate documentation



✖ Note: All APIs in the Java development kit are documented using the `javadoc` utility.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

For more information about how to write doc comments for the Javadoc Tool, see:

- <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html>
- <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

The Javadoc tool parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing (by default) the public and protected classes, nested classes (but not anonymous inner classes), interfaces, constructors, methods, and fields. You can use it to generate the API (Application Programming Interface) documentation or the implementation documentation for a set of source files.

You can run the Javadoc tool on entire packages, individual source files, or both. When documenting entire packages, you can either use subpackages for traversing recursively down from a top-level directory or pass in an explicit list of package names. When documenting individual source files, you pass in a list of source (.java) filenames.

Javadoc tags:

Add author names: @author <list of names>

Display text in code font: {@code <text>}

Place relative path to the root directory from any documentation page:
{@docRoot}

Describe reasons why an API should no longer be used: @deprecated
<description text>

Describe an exception that a method throws: @exception <ClassName>
<description text>

The @throws and @exception tags are synonyms: @throws
<ClassName> <description text>

Inherit documentation comments from a superclass class or an interface that
class implements: {@inheritDoc}

Insert a link pointing to another Java documentation article: {@link
<package.Class#member> <text to display on the link>}

Describe method parameter: @param <ParameterName> <description
text>

Describe method return value: @return <description text>

Add "See Also" link to another Java documentation article: @see
<reference>

Add description of a serializable field: @serial <Field> | include |
exclude

Document the data written by the writeObject() or writeExternal()
methods: @serialData <description text>

Document an ObjectOutputStream component: @serialField
<FieldName> <FieldType> <description text>

Display the value of a constant: {@value package.class#field}

Describe in which Java release the API was introduced: @since <release>

Add version attribute: @version <version text>

Summary

In this lesson, you should have learned how to:

- Discover Java language origins and use-cases
- Explain Java portability and provider neutrality
- Explain object-oriented concepts
- Describe Java syntax and coding conventions
- Create a Java class by using the `main` method
- Compile and execute a Java application



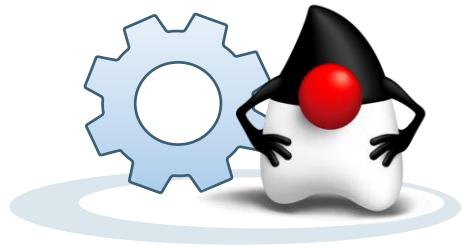
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

Practices

In this practice, you will:

- Verify the JDK Installation
- Create HelloWorld java application class with main method
- Compile and execute this application



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

26

Primitive Types, Operators, and Flow Control Statements



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Describe primitive types
- Describe operators
- Explain primitives type casting
- Use Math class
- Implement flow control with `if/else` and `switch` statements
- Describe JShell



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Java Primitives

Java language provides eight primitive types to represent simple numeric, character, and boolean values.

Whole numbers				Floating point numbers			
byte	short	int	long	float	double		
8 bits	16 bits	32 bits	64 bits	32 bits	64 bits		
-128	-32,768	-2,147,483,648	-9,223,372,036,854,780,000	1.4E-45	4.9E-324		
127	32,767	2,147,483,647	9,223,372,036,854,780,000	3.4028235E+38	1.7976931348623157E+308		
default value 0 or 0L				default value 0.0F or 0.0			
Binary 0b1001 Octal 072 Decimal 1234 Hex 0x4F Upper or lowercase L at the end indicates long value				Normal 123.4 or exponential notations 1.234E2; Upper or lowercase F at the end indicates float value			
Boolean		Character (represents single character value)					
boolean		char					
default value false		16 bits					
true or false		0					
		65,535					
		default value '\u0000'					
Character 'A' ASCII code '\101' Unicode '\u0041' Escape Sequences: tab '\t' backspace '\b' new line '\n' carriage return '\r' form feed '\f' single quote '\'' double quote '\"' backslash '\\'							



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

The slide shows the capacity of each type, minimum and maximum values for each type (except boolean), default value, and ways of expressing values.

You can print integral values, formatted as binary, octal, or hex, using Integer or Long class methods `toBinaryString` `toOctalString` `toHexString`.

Use uppercase L and F to indicate float and long values to make your code more readable.

The default value for the char type is '\u0000'. It does not correspond to any character on a keyboard and is not the same as space that has a code of '\u0020'.

Declare and Initialize Primitive Variables

Primitive declaration and initialization rules:

- Variable declaration and initialization syntax:
`<type> <variable name> = <value>;`
- A variable can be declared with no immediate initialization, so long as it is initialized before use.
- Numeric values can be expressed as binary, octal, decimal, and hex.
- Float and double values can be expressed in normal or exponential notations.
- Multiple variables of the same type can be declared and initialized simultaneously.
- Assignment of one variable to another creates a copy of a value.
- Smaller types are automatically promoted to bigger types.
- Character values must be enclosed in single quotation marks.

```
int a = 0b101010; // binary
short b = 052;    // octal
byte c = 42;      // decimal
long d = 0x2A;   // hex
float e = 1.99E2F;
double f = 1.99;
long g = 5, h = c;
float i = g;
char j = 'A';
char k = '\u0041', l = '\101';
int s;
s = 77;
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

In the example in the slide, variables a, b, c, d are all set to 42 using binary, octal, decimal, and hex value expressions.

Char values are stored as character codes. For example, character 'A' has a character code of 65 expressed as a decimal number. However, when expressing an ASCII code for this character, you should use its octal representation, which is '101', or for Unicode, use hex representation, which is '\u0041'.

Automatic promotion means that a value of a smaller type (for example, float) can be directly assigned to a variable of a bigger type (for example, double).

Restrictions on Primitive Declarations and Initializations

Primitive declaration and initialization restrictions:

- Variables must be initialized before use.
- A bigger type value cannot be assigned to a smaller type variable.
- Character values must not be enclosed in double quotation marks.
- A character value cannot contain more than one character.
- Boolean values can be expressed only as `true` or `false`.

```
byte a;
byte b = a;
byte c = 128;
int d = 42L;
float e = 1.2; 
char f = "a";
char g = 'AB';
boolean h = "true";
boolean i = 'false';
boolean j = 0;
boolean k = False;
```

✿ Note: Each incorrect example given here would cause Java code not to compile.



Java Operators

List of Java operators in the order of precedence:

Operators	Precedence
postfix increment and decrement	<code>++ --</code>
prefix increment and decrement, and unary	<code>++ -- + - ~ !</code>
multiplicative	<code>* / %</code>
additive	<code>+ -</code>
bit shift	<code><< >> >>></code>
relational	<code>< > <= >= instanceof</code>
equality	<code>== !=</code>
bitwise AND	<code>&</code>
bitwise exclusive OR	<code>^</code>
bitwise inclusive OR	<code> </code>
logical AND	<code>&&</code>
logical OR	<code> </code>
ternary	<code>? :</code>
assignment	<code>= += -= *= /= %= &= ^= = <<= >>= >>>=</code>

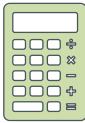


Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

Details of these operators are covered later in this lesson with the exception of the instanced operator, which is covered later in the course.

Assignment and Arithmetic Operators



Assignments and arithmetics

= + - * / %

Compound assignments are combinations of an operation and an assignment that is acting on the same variable.

+ = - = * = / = % =

Operator evaluation order can be changed using round brackets.

()

Increment and decrement operators has prefix and postfix positions:

y=++x x is incremented first and then the result is assigned to y.

y=--x x is decremented first and then the result is assigned to y.

y=x++ y is assigned the value of x first and then x is incremented.

y=x-- y is assigned the value of x first and then x is decremented.

++ --

```
int a = 1; // assignment (a is 1)
int b = a+4; // addition (b is 5)
int c = b-2; // subtraction (c is 3)
int d = c*b; // multiplication (d is 15)
int e = d/c; // division (e is 5)
int f = d%6; // modulus (f is 3)
```

```
int a = 1, b = 3;
a += b; // equivalent of a=a+b (a is 4)
a -= 2; // equivalent of a=a-2 (a is 2)
a *= b; // equivalent of a=a*b (a is 6)
a /= 2; // equivalent of a=a/2 (a is 3)
a %= a; // equivalent of a=a%a (a is 0)
```

```
int a = 2, b = 3;
int c = b-a*b; // (c is -3)
int d = (b-a)*b; // (c is 3)
```

```
int a = 1, b = 0;
a++; // increment (a is 2)
++a; // increment (a is 3)
a--;
a--; // decrement (a is 2)
--a; // decrement (a is 1)
b = a++; // increment postfix (b is 1, a is 2)
b = ++a; // increment prefix (b is 3, a is 3)
b = a--;
b = --a; // increment postfix (b is 3, a is 2)
b = a--; // increment prefix (b is 1, a is 1)
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Modulus finds a remainder of division of two numbers. The slide shows an example of 15%6, and it works like this: divide 15/6 and round the result down to get 2, then multiply 2*6, which is 12 and finally find the modulus as 15-12 which will be 3.

Compound assignments covered in this slide used arithmetic operators (+= - = * = / = % =). However, the same technique can also be used with Bitwise and Bit Shift operators (&= ^= |= <<= >>= >>>=). These operators are covered later.

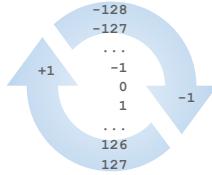
Operator – can be used to invert the sign of the expression.

Operator + can be used to indicate a positive number; however, it is considered excessive, since numbers are assumed to be positive anyway.

Arithmetic Operations and Type Casting

Rules of Java arithmetic operations and type casting:

- Smaller types are automatically casted (promoted) to bigger types.
byte->short->char->int->long->float->double
- A bigger type value cannot be assigned to a smaller type variable without explicit type casting.
- Type can be explicitly casted using the following syntax: (<new type>) <variable or expression>
- When casting a bigger value to a smaller type, beware of a possible overflow.
- Resulting type of arithmetic operations on types smaller than int is an int; otherwise, the result is of a type of a largest participant.



```

byte a = 127, b = 5;
✗ byte c = a+b;           // compilation fails
✓ int d = a + b;          // d is 132
✗ byte e = (byte)(a+b);    // e is -124 (type overflow, because 127 is the max byte value)
✗ int f = a/b;            // f is 25 (a/b is 25 because it is an int)
✗ float g = a/b;          // g is 25.0F (result of the a/b can be implicitly or
✗ float h = (float)(a/b); // h is 25.0F explicitly casted to float, but a/b is still 25)
✓ float i = (float)a/b;   // i is 25.4F (when either a or b
✓ float j = a/(float)b;  // j is 25.4F is float the a/b becomes float)
✗ b = (byte)(b+1);        // explicit casting is required, because b+1 is an int
✓ b++;
✓ char x = 'x';
✓ char y = ++x;          // arithmetic operations work with character codes
  
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

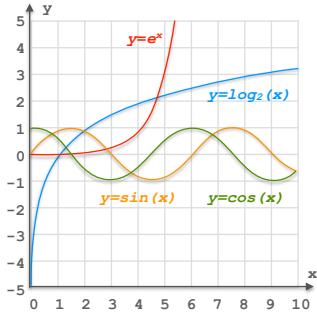
Type casting makes sense if you think about similarities between different types. For example, you could say that a whole number is a variant of a floating point number. For example, an int can be imagined as a double, which has no actual floating point part. Therefore, if you assign an int value to a double variable, it works perfectly fine and explicit type casting is not required. However, the other way around, assigning double to an int is possible only if you choose to ignore the floating point part of the double value. This could cause a truncation of the double value and would require explicit casting.

Note that char is treated as character code for the purposes of arithmetic operations.

More Mathematical Operations

Class `java.lang.Math` provides various mathematical operations:

- Exponential such as `ex`
- Logarithmic such as `log2(x)`
- Trigonometric such as `sin(x)` `cos(x)`
- and many more...



Examples of Math functions:

```
double a = 1.99, b = 2.99, c = 0;
c = Math.cos(a); // cosine
c = Math.acos(a); // arc cosine
c = Math.sin(a); // sine
c = Math.asin(a); // arc sine
c = Math.tan(a); // tangent
c = Math.atan(a); // arc tangent
c = Math.exp(a); // ea
c = Math.max(a,b); // greater of two values
c = Math.min(a,b); // smaller of two values
c = Math.pow(a,b); // ab
c = Math.sqrt(a); // square root
c = Math.random(); // random number 0.0>=c<1.0
```

Numeric rounding example :

```
int a = 11, b = 3;
long c = Math.round(a/b); // c is 3
double d = Math.round(a/b); // d is 3.0
double e = Math.round((double)a/b*100)/100.0; // e is 3.67
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

For more examples of Math class functions, see the Java documentation at:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

The number rounding example demonstrates that Java allows you to perform mathematical operations on numbers that are not necessarily decimal. To round a number to a set number of digits after the decimal point, you need to consider if this is a binary, octal, decimal, or hex number that you want to handle.

General rounding formula looks like this: `Math.round(a*bc) / bc` where `a` is a double or float number you want to round, `b` is the number base (2,8,10,16), and `c` is the number of fraction digits you need to get as a result of rounding. Also, because the result of any arithmetic operation on types smaller than int is an int, make sure that at least one participant of the expression is a float or a double number, in order for the result to be a floating point number.

This slide discusses mathematical rounding. Do not confuse with numeric text formatting, which is not operating on a number, but on text instead. This would be explained later in the course.

Binary Number Representation

All Java numeric primitives are signed (*that is, could represent positive and negative values*).

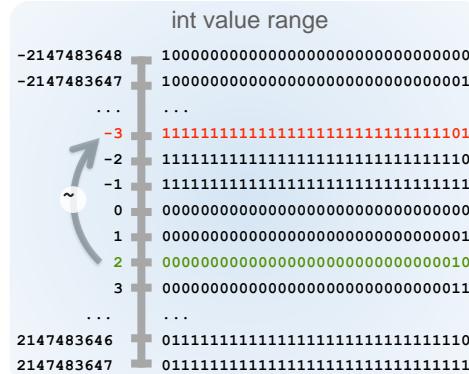
- Java uses a two's complement implementation of a signed magnitude representation of an integer.
- For example, a byte zero value is represented as the 00000000 sequence of bits.
- Changing a sign (negative or positive) is done by inverting all the bits and then adding one to the result.
- For example, a byte value of one is represented as 00000001 and minus one is 11111111.

Bitwise Complement operator inverts all bits of a number:

The result of the bitwise compliment operator `~a` would be its "mirrored" binary value: `-(a+1)`.

```
int a = 2; //  
int b = ~a; // b is -3
```

❖ The next slide shows the rest of the bitwise operators.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Byte zero value is stored as 00000000 (8 zero bits), short as 16 zero bits, int as 32, and so on.

Bitwise operators enable direct bit manipulations of numeric values. Their uses are mostly in low-level programming cases, such as communications over network sockets, encryption, compression, graphics, and so on. Most of the time Java application programmers are not directly exposed to such low-level operations and are using high-level APIs to program. However, under the hood, these high-level APIs would use these kind of operations anyway.

Bitwise Operators

Compare corresponding bits of two operands with bitwise operators:

- **Bitwise AND** & when both bits are 1, the result is 1, or either of the bits is not 1, the result is 0.
- **Bitwise OR** | when either of the bits is 1, the result is 1; otherwise, the result is 0.
- **Bitwise Exclusive OR** ^ when corresponding bits are different, the result is 1; otherwise, the result is 0.

```
byte a = 5;           // 000000101
byte b = 3;           // 000000011
byte c = (byte) (a & b); // 00000001 (c is 1)
byte d = (byte) (a | b); // 00000111 (d is 7)
byte e = (byte) (a ^ b); // 00000110 (e is 6)
```

Shift bits to the left or right with bitwise operators:

- **Signed Left Shift** << shifts each bit to the left by specified number of positions, fills low-order positions with 0 bit values
- **Signed Right Shift** >> shifts each bit to the right by specified number of positions
- **Unsigned Right Shift** >>> same as above, but fills high-order positions with 0 bit values

```
int a = 5;           // 00000000000000000000000000000000101
int b = -5;          // 11111111111111111111111111111111011
int c = a << 2;    // 0000000000000000000000000000000010100 (c is 20)
int d = b << 2;    // 111111111111111111111111111111110100 (d is -20)
int e = a >> 2;    // 000000000000000000000000000000000001 (e is 1)
int f = b >> 2;    // 111111111111111111111111111111110110 (f is -2)
int g = a >>> 2;  // 000000000000000000000000000000000001 (e is 1)
int h = b >>> 2;  // 001111111111111111111111111111110110 (h is 1073741822)
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Bitwise operators result in int type, just like other arithmetic operations.
Therefore, explicit casting is required if you want results of these operations to be assigned to types smaller than int.

Equality, Relational, and Conditional Operators

Compare values to determine the boolean result:

- **Equal to** ==
- **Not equal to** !=
- **Greater than** >
- **Greater than or equal to** >=
- **Less than** <
- **Less than or equal to** <=
- **NOT** ! (boolean inversion)
- **AND** && &
- **OR** || &
- **Exclusive OR** ^

== != > >= < <= !
&& || (short-circuit evaluation)
& | ^ (full evaluation)

```

int a = 3, b = 2;
boolean c = false;
c = (a == b);           // c is false
c = !(a == b);          // c is true
c = (a != b);          // c is true
c = (a > b);           // c is true
c = (a < b);           // c is false
c = (a <= b);          // c is false
c = (a > b && b == 2); // c is true
c = (a < b && b == 2); // c is false
c = (a < b || b == 2); // c is true
c = (a < b || b == 3); // c is false
c = (a > b ^ b == 2); // c is false
    
```

Notes

- ❖ Round brackets () are not required but could improve code readability.
- ❖ The difference between full and short-circuit evaluation is explained in the next slide.



Short-Circuit Evaluation

Short-circuit evaluation enables you to not evaluate the right-hand side of the AND and OR expressions, when the overall result can be predicted from the left-hand side value.

`&& ||` (*short-circuit evaluation*)

`& | ^` (*full evaluation*)

```
true && evaluated
false && not evaluated
false & evaluated
false || evaluated
true || not evaluated
true | evaluated
true ^ evaluated
false ^ evaluated
```

```
int a = 3, b = 2;
boolean c = false;
c = (a > b && ++b == 3); // c is true, b is 3
c = (a > b && ++b == 3); // c is false, b is 3
c = (a > b || ++b == 3); // c is false, b is 4
c = (a < b || ++b == 3); // c is true, b is 4
c = (a < b | ++b == 3); // c is true, b is 5
```

❖ Note: It is not advisable to mix boolean logic and actions in the same expression.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

In a short-circuit `&&` evaluation, when the left side of expression is true, the right side still has to be evaluated to determine the result. However when the left side of the expression is false, the overall result can be determined without evaluating the right-hand side of the expression. This is precisely what is going to happen—Java would not attempt to evaluate the right-hand side of the expression when the overall result can be predicted by looking at the left side. Likewise, the short-circuit `||` evaluation would not evaluate the right side of the expression, when the left side yields true and both sides will be evaluated when the left side yields false.

Do not confuse assignment `=` with equality `==` operator.

The following code example will not compile:

```
int x = 1, y = 2;
boolean z = (x=y);
```

The following code example will compile:

```
int x = 1, y = 2;
boolean z = ((x=y) == 2); // z is true
```

Flow Control Using if/else Construct

Conditional execution of the algorithm using the if/else construct:

- The if code block is executed when the boolean expression yields true; otherwise else block is executed.
- The else clause is optional.
- There is no else/if operator in Java, but you can embed an if/else inside another if/else construct.

```
if (<boolean expression>) {
    /* logic to be executed when
       if expression yields true */
} else {
    /* logic to be executed when
       if expression yields false */
}
```

❖ Note: Compilation fails because a block containing more than one statement must be enclosed with {}.

```
int a = 2, b = 3;
if (a > b)
    a--;
    b++;
else
    a++;
```



```
int a = 2, b = 3;
if (a > b) {           // is false
    a--;               // not executed
} else {                // algorithm enters else block
    if (a < b) {        // is true
        a++;             // a is 3
    } else {              // this else block is not executed
        b++;               // not executed
    }
}
```



❖ Note: It is optional to put curly brackets {} around if or else blocks of code, when they contain only a single statement. This code fragment is identical to the example above, but {} omissions could make it harder to read.

```
int a = 2, b = 3;
if (a > b)
    a--;
else if (a < b)
    a++;
else
    b++;
```



❖ Note: Carriage returns and indentations in Java improve readability, but are irrelevant from the compiler perspective.



Ternary Operator

The ternary operator is used to perform conditional assignment.

- You can use the ternary operator ?: instead of writing an if/else construct, if you only need to assign a value based on a condition.
- When the boolean expression yields true, value after the ? is assigned.
When the boolean expression yields false, value after the : is assigned.

```
<variable> = (<boolean expression>) ? <value one> : <value two>;
```

```
int a = 2, b = 3;
int c = (a >= b) ? a : b; // c is 3
```



These constructs produce identical results.

```
int a = 2, b = 3;
int c = 0;
if (a >= b) {
    c = a;
} else {
    c = b;
}
```



✿ Note: The ternary operator should be used to simplify conditional assignment logic. Do not use it instead of if/else statements to perform other actions, as it can make your code less readable.

```
int a = 2, b = 3;
int c = (a >= b) ? a : (--b == a) ? a : b; // c is 2
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

The if/else example works fine, but may be harder to understand that all it actually does is perform an assignment. Use of the ternary operator can disambiguate this.

Flow Control Using switch Construct

Control program flow using the `switch` construct.

- Switch expression must be of one of the following types:
`byte, short, int, char, String, enum`
- Case **labels** must match the expression type.
- Execution flow proceeds to the case in which the label matches the expression value.
- Execution flow continues until it reaches the end of switch or encounters an optional `break` statement.
- If the switch expression did not match any of the cases, then the **default case** is executed. It does not have to be the last case in the sequence and it is optional.

```
switch (<expression>) {
    case <label>:
        <case logic>
    case <label>:
        <case logic>
        break;
    default:
        <case logic>
}
```

Example cases:

- Special, increase price by 1
- New, increase price by 2
- Discounted, decrease price by 4
- Expired, set price to 0
- Any other, set price to 3

Execution path within the switch when status is 'N' (New)

```
char status = 'N';
double price = 10;
switch (status) {
    case 'S':
        price += 1; // not executed
    case 'N':
        price += 2; // price is 12
    case 'D':
        price -= 4; // price is 8
        break;
    case 'E':
        price = 0; // not executed
        break;
    default:
        price = 3; // not executed
}
//the rest of the program logic
```

❖ Note: Strings and enums are covered later in the course.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

JShell

- JShell is an interactive Read-Evaluate-Print Loop (REPL) command-line tool.
- Its purpose is to help to learn Java programming language and prototype Java code.
- It evaluates declarations, statements, and expressions as they are entered.
- It shows the results immediately.

JShell command reference:

JShell use example:

```
$ jshell
jshell> int x = 1
x ==> 1
jshell> int y = 1
y ==> 1
jshell> x+y
$3 ==> 2
jshell>/exit
$
```

/list	list the source you have typed
/edit	edit a source entry
/drop	delete a source entry
/save	save snippet source to a file
/open	open a file as source input
/vars	list the declared variables and their values
/methods	list the declared methods and their signatures
/types	list the type declarations
/imports	list the imported items
/exit	exit the jshell tool
/env	view or change the classpath or modules context
/reset	reset the jshell tool
/reload	reset and replay relevant history
/history	history of what you have typed
/help	or /? get information about using the jshell tool
/set	set configuration information
!*	rerun last snippet
/<id>	rerun snippets by ID or ID range
/-<n>	rerun n-th previous snippet



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Summary

In this lesson, you should have learned how to:

- Describe primitive types
- Describe operators
- Explain primitives type casting
- Use Math class
- Implement flow control with `if/else` and `switch` statements
- Describe JShell



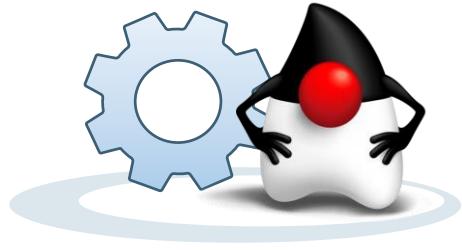
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

Practices

In this practice, you will:

- Use JShell tool
- Declare, Initialise and Perform Operations on Primitives
- Use the `if/else` and `switch` Constructs and use a Ternary Operator



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Text, Date, Time, and Numeric Objects



ORACLE®

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

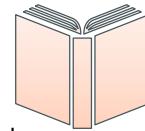


After completing this lesson, you should be able to:

- Manipulate text values using String and StringBuilder classes
- Describe primitive wrapper classes
- Perform string and primitive conversions
- Handle decimal numbers using BigDecimal class
- Handle date and time values
- Describe Localization and Formatting classes

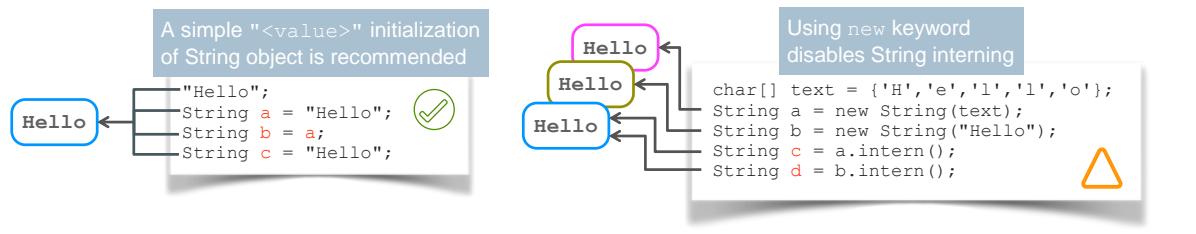


String Initialization



The `java.lang.String` class represents a sequence of characters.

- String is a class (not a primitive). Its instances represent sequences of characters.
- Just like any other Java object, the String object can be instantiated by using the `new` keyword.
- However, String is the only Java object that allows simplified instantiation as a text value enclosed with double quotes: "`some text`" and that is a recommended approach.
- JVM can optimize memory allocated to store String objects by maintaining a single copy of each **String literal** in the String Pool memory area, **regardless of how many variables reference this copy**. (This process is called interning.)
- The `intern()` method returns a reference to an interned (single) copy of a String literal.



- ❖ Reminder: A primitive `char` represents a single character. Its values are enclosed in single quotes: '`a`'.
- ❖ Note: Example uses a char array `char[]`; arrays are covered later in the course.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

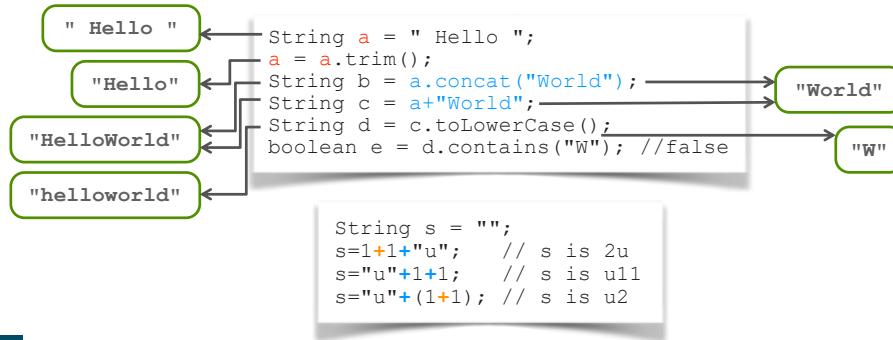
Interning String objects means that in the example on the right, only one copy of the String literal "Hello" will actually be placed in memory. All variables a, b, and c will be referring to the same copy of this String literal. This approach is guaranteed to be safe, because String objects are immutable. In the other example, String objects a and b are not interned, but objects c and d are.

From the perspective of the application developer, String objects appear to represent its content as `char[]`. However, because Java 9 onward String objects use the "Compact String" internal storage mechanism, JVM can store values not just as `char[]` but also as `byte[]`. It can also use UTF-16 encoding only as necessary, reducing the program's memory utilization and improving garbage collector performance.

String Operations

String objects are immutable.

- Once a string object is initialized, it cannot be changed.
- String operations such as `trim()`, `concat()`, `toLowerCase()`, `toUpperCase()`, and so on would always return a new String, but would not modify the original String.
- It is possible to **reassign the string reference** to point to another string object.
- For convenience reasons, String allows the use of the `+` operator instead of the `concat()` method. However, remember that `+` is also an arithmetic operator.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

There are two more methods that allow removing leading and trailing spaces from the String, `stripLeading()` and `stripTrailing()` (available since version 11), in addition to the `trim` method.

For more information about String class and its methods, see Java documentation:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html>

String Indexing

String contains a sequence character indexed by integer.

- String index starts from 0.
- When getting a substring of a string, the **begin index** is inclusive of the result, but **end index** is not.
- If a substring is not found, the `indexOf` method returns -1.
- Both `indexOf` and `lastIndexOf` operations are **overloaded** (have more than one version) and accept a char or a String parameter and may also accept a search starting from the index position.
- An attempt to access text beyond the last valid index position (length-1) will produce an exception.



```
String a = "HelloWorld";
String b = a.substring(0,5); // b is "Hello"
int c = a.indexOf('o'); // c is 4
int d = a.indexOf('o',5); // d is 6
int e = a.lastIndexOf('l'); // e is 8
int f = a.indexOf('a'); // f is -1
char g = a.charAt(0); // g is H
int h = a.length(); // h is 10
char i = a.charAt(10);
throw StringIndexOutOfBoundsException
```

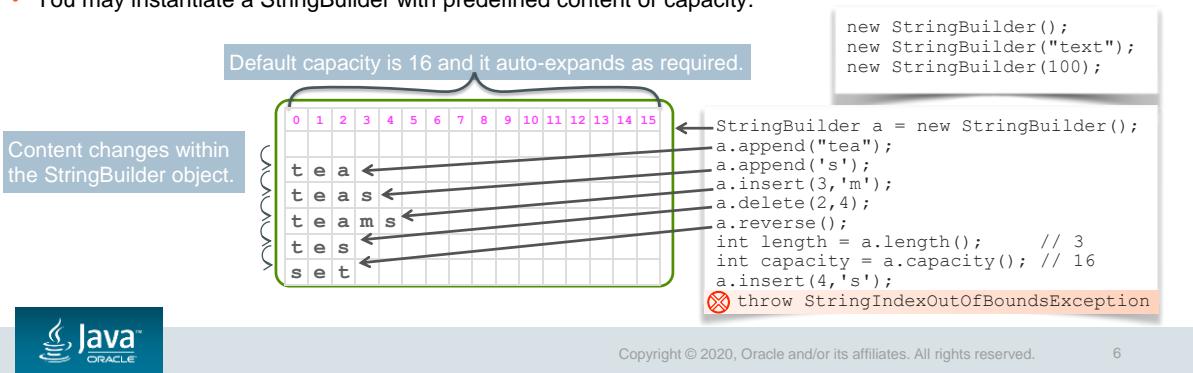
✖ Note: Exception handling is covered later in the course.



StringBuilder: Introduction

Another way of handling text in Java is with the `java.lang.StringBuilder` class.

- `StringBuilder` objects are mutable, allowing modifications of the character sequences they store.
- Handling text modifications with `StringBuilder` reduces the number of string objects you need to create.
- Some methods such as `substring`, `indexOf`, `charAt` are identical to that of a `String` class.
- Extra methods are available: `append`, `insert`, `delete`, `reverse` accepting `String` or `char` parameters.
- Sequence of characters must be continuous. It may contain spaces, but you may not leave gaps of no characters at all.
- Like other classes, `StringBuilder` objects are instantiated using the `new` keyword.
- You may instantiate a `StringBuilder` with predefined content or capacity.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

Strings represent an easy and convenient way to produce output or capture user input, but they could prove to be not very efficient when you need to perform more complex text handling. That is why Java provides an alternative class `StringBuilder` designed to manipulate with text in a more efficient way.

`StringBuilder` will automatically expand its capacity as needed (if you add more text), but performance-wise it is best to set the expected capacity immediately when you create a `StringBuilder` object.

For more information about `StringBuilder`, see Java documentation:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuilder.html>

There is an alternative to the `StringBuilder` class called `StringBuffer`. They generally work in the same way and provide equivalent operations; however, `StringBuffer` is designed as thread-safe and `StringBuilder` is not. Thread safety means that object would not allow multiple threads to modify its content concurrently. Unfortunately, thread-safety costs resources and has a decremental effect on program's performance. Many Java APIs provide thread-safe and unsafe versions of classes. It is generally recommended to use thread-unsafe versions, for performance reasons unless thread-safety is really required.

Wrapper Classes for Primitives



Wrapper classes apply object-oriented capabilities to primitives.

- A wrapper class is capable of holding a primitive value provided for every Java primitive.
- Construct wrapper object out of primitive or string using the `valueOf()` methods.
- Extract primitive values out of the wrapper using the `xxxValue()` methods.
- Instead of formal conversion of wrapper to primitive and back, you can use direct assignment known as **auto-boxing** and **auto-unboxing**.
- Create wrapper or primitive out of the string using the `parseXXX()` methods.
- You may convert a primitive to a string using the `String.valueOf()` method.
- Wrapper classes provide constants, such as `min` and `max values` for every type.

```
int a = 42;
Integer b = Integer.valueOf(a);
int c = b.intValue();
b = a;
c = b;
String d = "12.25";
Float e = Float.valueOf(d);
float f = d.parseFloat(d);
String g = String.valueOf(a);
Short.MIN_VALUE;
Short.MAX_VALUE;
```

Primitive Wrapper

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

Notes

- ❖ Advanced text formatting and parsing is covered later.
- ❖ Avoid too many auto-boxing and auto-unboxing operations for performance reasons.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Since Java 9, it is recommended not to use wrapper class constructors. `new Integer(<int>)` and `new Integer(<String>)` constructors are deprecated.

You may construct a wrapper out of text using customer radix value: `Integer.valueOf("B2B", 16)`; will construct a decimal value of 2859 out of the text representation of a 16-base number.

Auto boxing/unboxing process copies values between two memory areas known as stack and heap, which is an expensive operation. Using boxing/unboxing extensively would consume excessive amount of CPU resources.

Stack and heap memory areas are covered in the lesson titled "Improve Class Design."

Representing Numbers Using BigDecimal Class

The `java.math.BigDecimal` class is useful in handling decimal numbers that require exact precision.

- All primitive wrappers and `BigDecimal` are immutable and signed (cannot be changed and may represent positive or negative numbers).
- However, unlike other numeric wrapper classes, `BigDecimal` has arbitrary precision (for example, `Double` has a limited precision as a binary 64 bit number).
- It is designed to work specifically with decimal numbers and provide convenient `scale` and `round` operations.
- It provides arithmetic operations as methods such as `add`, `subtract`, `divide`, `multiply`, `remainder`.
- It is typically used to represent decimal number that require exact precision, such as fiscal values.

```
BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate); // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Classes that you have used so far in the course were located in the package `java.lang`. Content of the `java.lang` package is available (implicitly imported) to all other Java classes. However, you may wish to add an import statement for the `BigDecimal` class, since it is located in another package `java.math`.

`RoundingMode` is an enum from the `java.math` package that defines different rounding algorithms and is used to control exact rounding for the `BigDecimal` operations.

When working with decimal whole numbers, consider using the `java.math.BigInteger` class instead of simple primitives or wrapper classes. `BigInteger` provides similar decimal number handling capabilities as `BigDecimal`.

Method Chaining

When an operation returns an object you may invoke next operation upon this object immediately

- It is possible to **chain method invocations** technique with any operation that returns an object.
- Examples:
 - Arithmetic operations of `BigDecimal` return `BigDecimal` objects.
 - Text manipulating operations of `String` return `String` objects.

```
String s1 = "Hello";
String s2 = s1.concat("World").substring(3,6); // s2 is "loW"

BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate);           // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```

❖ Note: Method chaining is a common coding technique used when a method returns an object, followed by an invocation of another method upon that object and so on...

❖ Without method chaining, code appears to be cluttered with **unnecessary intermediate variables**:

```
BigDecimal taxedPrice = price.add(tax);
price = taxedPrice.setScale(2,RoundingMode.HALF_UP);
```



Local Date and Time

Local date and time API is used to handle date and time values.

- Classes `LocalDate`, `LocalTime`, and `LocalDateTime` from `java.time` package

- Date and time objects can be created using methods:

`now()` to get current date and time, or

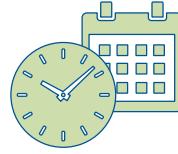
```
LocalDateTime.of(year, month, day, hours, minutes, seconds, nanoseconds)
```

```
LocalTime.of(hours, minutes, seconds, nanoseconds)
```

```
LocalDate.of(year, month, day) for specific date and time or
```

combining other date and time objects using `atTime()` and `of(localDate, localTime)` or

extracting date and time portions from `LocalDateTime` using `toLocalDate()` and `toLocalTime()`



```
LocalDate today = LocalDate.now();
LocalTime thisTime = LocalTime.now();
LocalDateTime currentDateTime = LocalDateTime.now();
LocalDate someDay = LocalDate.of(2019, Month.APRIL, 1);
LocalTime someTime = LocalTime.of(10, 6);
LocalDateTime otherDateTime = LocalDateTime.of(2019, Month.MARCH, 31, 23, 59);
LocalDateTime someDateTime = someDay.atTime(someTime);
LocalDate whatWasTheDate = someDateTime.toLocalDate();
```

✿ Note: Date and Time API (`java.time` package) was introduced in Java SE 8.
Earlier Java versions used `java.util.Date` class to represent date and time values.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

The `of()` operation is overloaded (has several versions) to enable constructing partial date or time values. For example, not setting nanoseconds or seconds would still work, but the time will set seconds and nanoseconds to zero.

Example on this page uses an enum `java.time.Month`, which represents 12 months. Another useful enum is `java.time.DayOfWeek`, which represents seven days of the week. Enums are covered later in the course.

The `java.util.Date` class is significantly less sophisticated than the new Java data and time API. It represents an offset (specified number of milliseconds) since the standard base time known as "the epoch," namely January 1, 1970, 00:00:00 GMT.

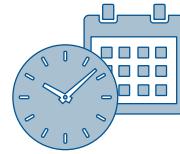
```
Date someTime = new Date(); // create current date and
time object
long milliseconds = someTime.getTime(); // get amount of
milliseconds elapsed from "the epoch"
someTime.setTime(milliseconds+(1000*60)); // modify date
object by adding one minute to it
```

Note that Date class is mutable; that is, you can change date and time value for the same instance of Date object.

More Local Date and Time Operations

Characteristics of local date and time operations

- Local Date and Time objects are immutable.
- All date and time manipulation methods will produce new date and time objects.
- New date and time objects can be produced out of existing objects using `plusXXX()` or `minusXXX()` or `withXXX()` methods.
- It is possible to chain method invocations together, because all date and time manipulation operations return date and time objects.
- Individual portions of date and time objects can be retrieved with `getXXX()` methods.
- Operations `isBefore()` and `isAfter()` check if a date or time is before or after another.



```
LocalDateTime current = LocalDateTime.now();
LocalDateTime different = current.withMinute(14).withDayOfMonth(3).plusHours(12);
int year = current.getYear();
boolean before = current.isBefore(different);
```

❖ Reminder: Method chaining helps to avoid cluttering code with unnecessary intermediate variables.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Classes that you have used so far in the course were located in the package `java.lang`. Content of the `java.lang` package is available (implicitly imported) to all other Java classes. However, this is not the case with other packages, so you would need to add import statements to access `java.time.LocalDateTime`, `java.time.LocalTime` and `java.time.LocalDate` classes.

Methods to create new date and time by changing specific portion of existing date or time:

```
withYear(int year)
withDayOfYear(int dayOfYear)
withMonth(int month)
withDayOfMonth(int dayOfMonth)
withHour(int hour)
withMinute(int minute)
withSecond(int second)
withNano(int nanoOfSecond)
```

Methods to create new date and time by adjusting existing date or time forward or backward:

```
plusYears(long years) minusYears(long years)
plusMonths(long months) minusMonths(long months)
plusWeeks(long weeks) minusWeeks(long weeks)
plusDays(long days) minusDays(long days)
plusHours(long hours) minusHours(long hours)
plusMinutes(long minutes) minusMinutes(long minutes)
plusSeconds(long seconds) minusSeconds(long seconds)

plusNanos(long nanos) minusNanos(long nanos)
```

Methods to get specific portions of date and time:

```
getYear()
getDayOfYear()
getMonth()
getDayOfMonth()
getDayOfWeek()
getHour()
getMinute()
getSecond()
getNano()
```

Instants, Durations, and Periods

In addition to dates and times, the API can also represent periods and durations of time.

- The `java.time.Duration` class can represent an amount of time in nanoseconds.
- The `java.time.Period` class can represent an amount of time in units such as years or days.
- The `java.time.Instant` class can represent an instantaneous point on the time-line (time-stamp).
- Just like the rest of the Local Date and Time API, Duration, Period, and Instant objects are immutable.
- Provide methods such as `now()`, `ofXXX()`, `plusXXX()`, `minusXXX()`, `withXXX()`, and `getXXX()`.
- Provide methods such as `between()`, `isNegative()` to handle distances between points in time.
- Use identical coding techniques such as method chaining.

```
LocalDate today = LocalDate.now();
LocalDate foolsDay = LocalDate.of(2019, Month, APRIL, 1);
Instant timeStamp = Instant.now();
int nanoSecondsFromLastSecond = timeStamp.getNano();
Period howLong = Period.between(foolsDay, today);
Duration twoHours = Duration.ofHours(2);
long seconds = twoHours.minusMinutes(15).getSeconds();
int days = howLong.getDays();
```



- ✿ Note: Instant and Duration are more suitable for implementing system tasks, such as using a timestamp when writing logs. Period is more suitable for implementing business logic.



Zoned Date and Time

Time zones can be applied to local date and time values.

The `java.time.ZonedDateTime` class:

- Represents date and time values according to time zone rules
- Has the same time management capabilities as `LocalDateTime`
- Provides time zone specific operations such as `withZoneSameInstant`
- Accounts for daylight saving time and time zone differences

```
ZoneId london = ZoneId.of("Europe/London");
ZoneId la = ZoneId.of("America/Los_Angeles");
LocalDateTime someTime = LocalDateTime.of(2019, Month.APRIL, 1, 07, 14);
ZonedDateTime londonTime = ZonedDateTime.of(someTime, london);
ZonedDateTime laTime = londonTime.withZoneSameInstant(la);
```

- The `java.time.ZoneId` class defines time zones.
- Timezone can be set as:

```
ZoneId.of("America/Los_Angeles");
ZoneId.of("GMT+2");
ZoneId.of("UTC-05:00");
ZoneId.systemDefault();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Represent Languages and Countries

Java provides APIs to make your application adjustable to different languages and locations around the world.

- The `java.util.Locale` class represents languages and countries.
- The ISO 639 language and ISO 3166 or country codes or UN M.49 area codes are used to set up locale objects.
- Locale can represent just language or a combination of language plus country or area.
- Variant is an optional parameter, designed to produce custom locale variations.
- Language tag string allows constructing locales for various calendars, numbering systems, currencies, and so on.

```

Locale uk = new Locale("en", "GB");           // English Britain
Locale uk = new Locale("en", "GB", "EURO");    // English Britain Euro (custom variant)
Locale us = new Locale("en", "US");           // English America
Locale fr = new Locale("fr", "FR");           // French France
Locale cf = new Locale("fr", "CA");           // French Canada
Locale fr = new Locale("fr", "029");          // French Caribbean
Locale es = new Locale("fr");                 // French
Locale current = Locale.getDefault();         // current default locale
// Example constructing locale that uses Thai numbers and Buddhist calendar:
Locale th = Locale.forLanguageTag("th-TH-u-ca-buddhist-nu-thai");

```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Commonly used locales are available as constants in the `Locale` class. For example:

`Locale us = Locale.US;`

is the same as

`Locale us = new Locale("en", "US");`

Instances of the `Locale` can also be constructed with a variant argument (custom variant), for example:

`new Locale("en", "GB", "EURO");`

Or with the use of the `Locale.Builder` class:

```

Locale uk = new
Locale.Builder().setLanguage("en").setRegion("GB").build();

```

Or with the use of the Language tag:

`Locale uk = Locale.forLanguageTag("en-GB");`

Examples of Language tag extension are:

cu (currency type)
fw (first day of the week)
rg (region override)
tz (time zone)
u-ca (calendar)
u-nu (numbering system)

For more information, see Java Locale documentation:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Locale.html>

Also, see the list of supported Locales, examples of calendars, and numbering systems:

<https://www.oracle.com/technetwork/java/javase/documentation/java11locales-5069639.html>

Format and Parse Numeric Values

The `java.text.NumberFormat` class is used to parse and format numeric values.

- Works with any Java Number, including primitives, wrapper classes and `BigDecimal` objects

```
BigDecimal price = BigDecimal.valueOf(2.99);
Double tax = 0.2;
int quantity = 12345;
Locale locale = new Locale("en", "GB");
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
String formattedPrice = currencyFormat.format(price);
String formattedTax = percentageFormat.format(tax);
String formattedQuantity = numberFormat.format(quantity);
```

value initializations

locale initialization

formatter initializations

formatting values

- Method `parse` return type is `Number` and it can be casted to numeric primitive wrappers or `BigDecimal` types

£2.99 20% 12,345

formatted results

```
BigDecimal newPrice = (BigDecimal)currencyFormat.parse("£1.75");
Double newTax = (Double)percentageFormat.parse("12%");
int newQuantity = numberFormat.parse("54,321").intValue();
```

parsing values

1.75 0.12 54321

parsed values



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

The `NumberFormat.parse()` method return type is `java.lang.Number`. Class `Number` is extended by all primitive wrapper classes as well as the `BigDecimal` class. This means that once the value is parsed, you can convert it to any type that you want to use in your program. In this example, values are converted to `BigDecimal`, `Double`, and `int`.

Number format parse method will throw a `ParseException` if parsed String does not conform to the expected format, such as wrong or missing currency symbols, wrong thousand or decimal separators, and so on.

Use the `java.text.DecimalFormat` class to set up custom format for decimal numeric values.

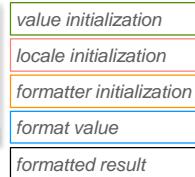
Format and Parse Date and Time Values

The `java.time.format.DateTimeFormatter` class is used to parse and format date and time values.

- You can set up **custom format pattern** or use **standard format patterns** defined by `java.time.format.FormatStyle enum`.

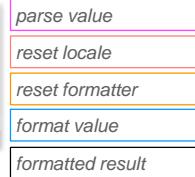
```
LocalDate date = LocalDate.of(2019, Month.APRIL, 1);
Locale locale = new Locale("en", "GB");
DateTimeFormatter format = DateTimeFormatter.ofPattern("EEEE dd MMM yyyy", locale);
String result = date.format(formatter);
```

Monday 01 Apr 2019



```
date = LocalDate.parse("Tuesday 31 Mar 2020", dateFormatter);
locale = new Locale("ru");
format = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM).localizedBy(locale);
result = date.format(formatter);
```

31 Mar. 2020 r.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

By default, Date and Time values are formatted using ISO format, in the order of years months days hours minutes seconds. For example: 2011-12-03T10:15:30

Alternatively, you can set specific format using patterns predefined by the `FormatStyle enum` (`FULL`, `LONG`, `MEDIUM`, `SHORT`) or define custom pattern `DateTimeFormatter.ofPattern(String pattern, Locale locale)`

Date and time pattern symbols:

Symbol	Meaning	Examples
G	era	AD; Anno Domini; A
u	year	2004; 04
y	year-of-era	2004; 04
D	day-of-year	189
M/L	month-of-year	7; 07; Jul; July; J
d	day-of-month	10
g	modified-julian-day	2451334
Q/q	quarter-of-year quarter	3; 03; Q3; 3rd

Y	week-based-year	1996; 96
w	week-of-week-based-year	27
W	week-of-month	4
E	day-of-week	Tue; Tuesday; T
e/c	localized day-of-week 2; 02;	Tue; Tuesday; T
F	day-of-week-in-month	3
a	am-pm-of-day	PM
h	clock-hour-of-am-pm (1-12)	12
K	hour-of-am-pm (0-11)	0
k	clock-hour-of-day (1-24)	24
H	hour-of-day (0-23)	0
m	minute-of-hour	30
s	second-of-minute	55
S	fraction-of-second	978
A	milli-of-day	1234
n	nano-of-second	987654321
N	nano-of-day	1234000000
V	time-zone ID	America/Los_Angeles;
Z; -08:30		
v	generic time-zone name	Pacific Time; PT
z	time-zone name	Pacific Standard Time; PST
O	localized zone-offset	GMT+8; GMT+08:00; UTC-08:00
X	zone-offset 'Z' for zero	Z; -08; -0830; -08:30; -083015; -08:30:15
x	zone-offset	+0000; -08; -0830; -08:30; -083015; -08:30:15
Z	zone-offset	+0000; -0800; -08:00
p	pad next	1
	escape for text	
'	single quote	

Example of custom pattern:

```
LocalDateTime someTime =  
    LocalDateTime.of(2019, Month.APRIL, 1, 17, 42);  
  
DateTimeFormatter dateFormatter =  
    DateTimeFormatter.ofPattern("EEEE dd MMMM YYYY, hh:mm a",  
        new Locale("en", "GB"));  
  
String result = dateFormatter.format(someTime);
```

Produces the following result:

Monday 01 April 2019, 05:42 pm

Note: LocalDate, LocalTime, LocalDateTime, ZonedDateTime and
DateTimeFormatter, all have format and parse methods; use whichever
is convenient.

Localizable Resources

Resource bundles contain localizable resources.

- Resource bundle can be represented as plain text file with the extension `.properties`.
- Resources, for example, user-visible messages, are placed into resource bundles as `<key>=<value>`.
- Bundles may contain messages or message patterns with **substitution parameters**.
(value substitutions are explained later)
- The `java.util.ResourceBundle` class loads bundles and retrieves resources.
- **Default bundle** can be used if no locale is specified `ResourceBundle.getBundle(<bundle name>)` or if the resource you're trying to get is not present in another (language and country specific) bundle.

```
Locale locale = new Locale("en", "GB");
ResourceBundle bundle = ResourceBundle.getBundle("resources.messages", locale);
String helloPattern = bundle.getString("hello");
String otherMessage = bundle.getString("other");
```

resources (package folder)
 messages.properties
 messages_en_GB.properties
 messages_ru.properties

hello=もしもし {0}
 product={0}, 價格 {1}, 分量 {2}, 賞味期限は {3}
 other=他に何か

default bundle, can
be in any language

hello=Hello {0}
 product={0}, price {1}, quantity {2}, best before {3}

hello=Привет {0}
 product={0}, цена {1}, количество {2}, годен до {3}



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

21

Resource bundles are treated as classes (loaded from classpath). This means that a bundle can be placed into a package folder, in which case, you need to specify package prefix to load the bundle file.

For example, assume that property file is placed into the "demos.resources" package folder: `demos\properties\messages_en_GB.properties`

In which case, use package prefix to fully qualify bundle name:

```
Locale locale = new Locale("en", "GB");
ResourceBundle bundle =
ResourceBundle.getBundle("demos.resources.messages", locale);
```

Resource bundle can be defined not only as a property file, but also as a Java class. However, this is not very commonly used because it could be harder to maintain than a plain text file.

Format Message Patterns

Formatter classes parse and format messages, numbers, date and time values.

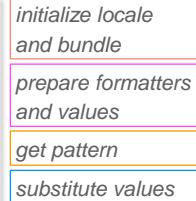
- The `java.text.MessageFormat` class substitutes values into message patterns.
- Message patterns can be stored in resource bundles. For example, `resources/messages_en_GB.properties` containing product message pattern:

```
product={0}, price {1}, quantity {2}, best before {3}
```

- After all required values are formatted, they can be substituted into the message:

```
Locale locale = new Locale("en", "GB");
ResourceBundle bundle = ResourceBundle.getBundle("resource.messages", locale);
// assume following values are already formatted:
String name = "Cookie",
String price = currency.format(price);
String quantity = number.format(quantity);
String bestBefore = date.format(dateFormatter);

String pattern = bundle.getString("product");
String message = MessageFormat.format(pattern, name, price, quantity, bestBefore);
```



Cookie, price £2.99, quantity 4, best before 1 Apr 2019



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

MessageFormat provides both format and parse capabilities, in the same way `DateTimeFormatter` or `NumberFormat` do:

```
MessageFormat formatter = new MessageFormat("{0}, price {1}, quantity {2}, best before {3}");
Object[] values = formatter.parse("Cookie, price £2.99, quantity 4, best before 1 Apr 2019");
```

This method returns an object array of values extracted from the string, according to the pattern set for this message format.

Formatting and Localization: Example

```
resources/messages_en_GB.properties  
  
product={0}, price {1}, quantity {2}, best before {3}  
  
String name = "Cookie";  
BigDecimal price = BigDecimal.valueOf(2.99);  
LocalDate bestBefore = LocalDate.of(2019, Month.APRIL, 1);  
int quantity = 4;  
  
Locale locale = new Locale("en", "GB");  
ResourceBundle bundle = ResourceBundle.getBundle("resource.messages", locale);  
  
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);  
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);  
DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("dd MMM yyyy", locale);  
  
String fPrice = currencyFormat.format(price);  
String fQuantity = numberFormat.format(quantity);  
String fBestBefore = dateFormat.format(bestBefore); // or bestBefore.format(dateFormat);  
  
String pattern = bundle.getString("product");  
String message = MessageFormat.format(pattern, name, fPrice, fQuantity, fBestBefore);
```

```
Cookie, price £2.99, quantity 4, best before 1 Apr 2019
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

Summary

In this lesson, you should have learned how to:

- Manipulate text values using String and StringBuilder classes
- Describe primitive wrapper classes
- Perform string and primitive conversions
- Handle decimal numbers using BigDecimal class
- Handle date and time values
- Describe Localization and Formatting classes



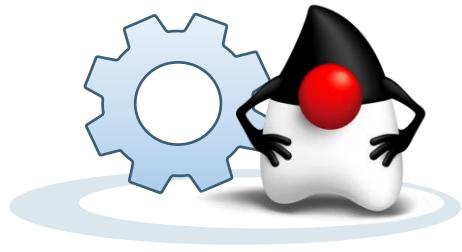
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

24

Practices

In this practice, you will:

- Manipulate text values using String and StringBuilder classes
- Manipulate and format Numeric, Date and Time values
- Apply Localization and Format Messages



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

Classes and Objects



ORACLE®

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to do the following:

- Model business problems using classes
- Define instance methods and variables
- Describe the "this" object reference
- Explain object instantiation
- Explain local variables and local variable type inference
- Explain static variables and methods
- Invoke methods and access variables
- Describe NetBeans IDE

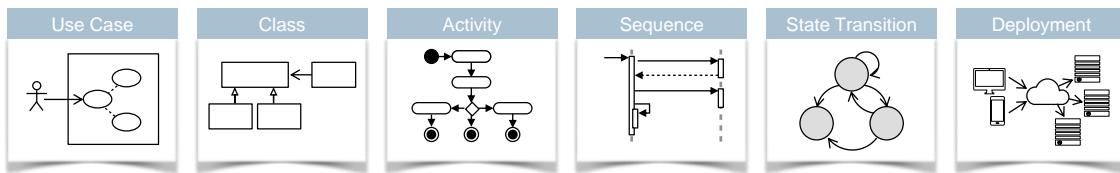


UML: Introduction

Unified Modelling Language (UML) diagrams:

- Are used to graphically represent business requirements and design software
- Facilitate communications between analysts, designers, and developers
- Clearly and concisely represent analyses, design, and implementation requirements
- Are eventually implemented in actual Java code

Examples of UML Diagrams:



✖ This course provides only very brief introduction to modelling. For more information, see the *Object-Oriented Analysis and Design Using UML* course



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

Snippets of diagram types shown in the slide:

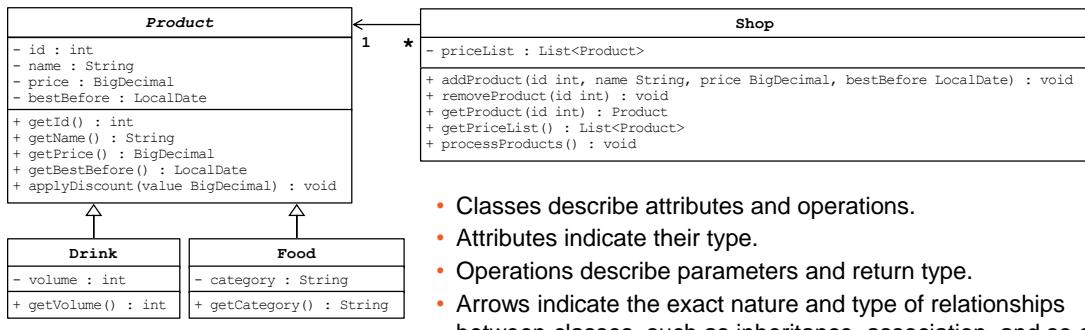
- Use Case Diagrams capture high-level business requirements.
- Class Diagrams represent classes and their relationships.
- Activity Diagrams describe program logic.
- Sequence Diagrams describe interactions between objects.
- State Transition Diagrams describe the life cycle of an object.
- Deployment Diagrams describe physical deployment topology.

Several other diagram types are available in UML, beyond the ones mentioned on this page.

This course provides only small examples of Class, Activity, and Sequence Diagrams.

Modeling Classes

This example shows a Class Diagram, which represents classes and their relationships.



- Classes describe attributes and operations.
- Attributes indicate their type.
- Operations describe parameters and return type.
- Arrows indicate the exact nature and type of relationships between classes, such as inheritance, association, and so on.

- ✿ The example provides a model of the set of classes required to implement product management system for a shop (see detailed description in notes).
- ✿ Also see notes for naming convention details.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

The example in the slide represents a set of Java classes **Shop**, **Product**, **Food**, and **Drink**. **Product** class name is displayed in italics, indicating that it is an abstract class. **Food** and **Drink** classes are subclasses of the **Product** class, which is indicated by their relationships.

According to Java naming convention, class names should be nouns. Model can look ambiguous, when a name can be misinterpreted as a verb. In this case, "Shop" is considered as noun, as it should be.

Each class has some attributes. For example, **Product** has:

- `id` type of `int`
- `name` type of `String`
- `price` type of `BigDecimal`
- `bestBefore` type of `LocalDate`

Each class has some operations. For example, **Product** has:

- `getId` that returns an `int`
- `getName` that returns a `String`
- `getPrice` that returns a `BigDecimal`
- `getBestBefore` that returns a `LocalDate`
- `applyDiscount` that accepts a value argument type of `BigDecimal` and is not designed to return any value. This is indicated by the `void` return type.

Naming conventions for classes, variables, and methods:

- Class names should be nouns with initial letter in uppercase and the rest of the letters in lowercase. If the class name comprises more than one word, the initial letter of every word should be capitalized:

Product, PurchaseOrder, and so on:

- Variable names should be nouns in lowercase. If the variable name comprises more than one word, the initial letter of every subsequent word should be capitalized:

bestBefore, totalDiscount, and so on.

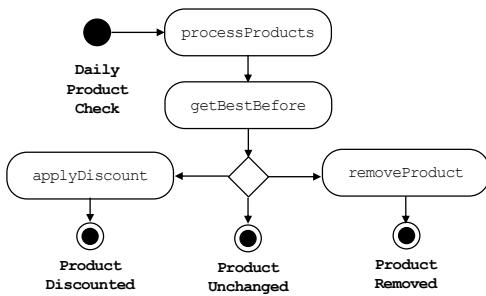
- Method names should be verbs in lowercase. If the method name comprises more than one word, the initial letter of every subsequent word should be capitalized.
- It is typical for methods that provide access to information to follow the getXXX/setXXX naming pattern. If a method provides access to boolean variable, it is usually named using isXXX pattern:

getBestBefore, setTotalDiscount, isFresh, and so on

Because of the way in which initial letters of words are capitalized, these naming conventions are sometimes referred to as "camel-case."

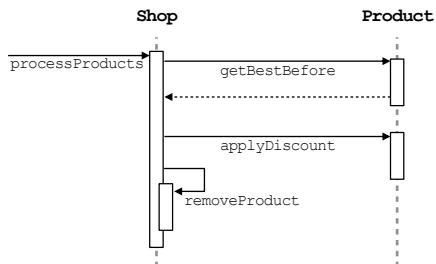
Modeling Interactions and Activities

Activity Diagram represents a flow of operations



- Flow of activities is triggered by an Event.
- Flow of activities may end in a number of Outcomes.
- Steps in a flow describe program Activities and Decisions.
- Lines represent the order of activities.

Sequence Diagram represents interactions between program objects



- Each object is represented by its Lifeline.
- Interactions between objects may include sending messages and returning values.
- Objects can invoke operations (send messages) upon other objects or recursively upon themselves.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

The examples in the slide represent different ways of modeling program logic. These diagrams model similar things, that is, invocations of operations, but focus on different aspects of the design.

The Activity diagram is focused on the way in which the activities are organized in relation to each other.

The Sequence diagram is focused on interactions between objects.

Designing Classes

Java class definition typically includes:

- Description of the **package** that this class is a member of
- Description of **imports** of classes from different packages that this class may need to reference
- This class **access modifier** (typically public)
- Keyword **class** followed by this **class name**
- Class and method bodies are enclosed with "{" and "}" symbols.
- A number of **variable** and **method** definitions within the class body

```
package <package name>;
import <package name>.<OtherClassName>;
<access modifier> class <ClassName> {
    // variables and methods
}
```

```
package demos.shop;
import java.math.BigDecimal;
public class Product {
    private BigDecimal price;
    public BigDecimal getPrice() {
        return price;
    }
    public void setPrice(double value) {
        price = BigDecimal.valueOf(value);
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Creating Objects

Java objects are instances of classes.

- The `new` operator **creates an Object** (an instance of a Class), allocating memory to store this object.
- Assign "**reference**" to the memory allocated for the object to be able to access it.
- Access variable or methods of the object using `".` operator.

```
Product p1 = new Product();    p1 → price=1.99  
p1.setPrice(1.99);  
BigDecimal price = p1.getPrice();
```

❖ Note: A **reference** is a typed variable that points to an **object** in memory.

```
package demos.shop;  
import java.math.BigDecimal;  
public class Product {  
    private BigDecimal price;  
    public BigDecimal getPrice() {  
        return price;  
    }  
    public void setPrice(double value) {  
        price = BigDecimal.valueOf(value);  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

In Java language, a "reference" is not actually a pointer to a specific memory address. Java run time automates management of memory allocation and cleanup. It is designed never to inform programmer about exact memory address where object is physically stored. Thus, Java reference should be considered as a purely logical way of pointing to an object.

Define Instance Variables

Classes may contain variable definitions to store state information about their instances (Objects).

- Variable is defined with its **type**, which can be one of the eight primitive types or any Class.
- Variable **name** is typically a noun written in lowercase.
- To protect data within the class, variables typically use **private access modifier**.
- Optionally, variables can be initialized (**assigned a default value**).
- Uninitialized primitives are defaulted to 0, except boolean, which defaults to **false**.
- Uninitialized object references are defaulted to **null**.

```
package <package name>;
import <package name>.<ClassName>;
<access modifier> class <ClassName> {
    <access modifier> <variable type> <variable name> = <variable value>;
}

package demos.shop;
import java.math.BigDecimal;
import java.time.LocalDate;
public class Product {
    private int id;
    private String name;
    private BigDecimal price;
    private LocalDate bestBefore = LocalDate.now().plusDays(3);
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Define Instance Methods

Classes may contain method definitions to implement behaviors of their instances (objects).

- Method must declare its **return type** or use the **void** keyword if a method does not need to return a value.
- Nonvoid methods must contain a **return statement**, which must return a value of the **corresponding type**.
- Method **name** is typically a verb (like get or set) written in lowercase, followed by descriptive nouns.
- Access modifiers** determine from where the method can be invoked.
- Methods may describe a comma-separated list of **parameters** enclosed in "()" symbols.
- Method body is enclosed with "{" and "}" symbols.

```
package <package name>;
<access modifier> class <ClassName> {
    <access modifier> <return type> <method name>(<ParameterType> <parameterName>,
                                                    <ParameterType> <parameterName>) {
        return <value>;
    }
}
```

Notes

- A return statement terminates the method.
- A void method may be terminated using empty `return;`.
- If no parameters are required, just put empty "()" brackets.

```
package demos.shop;
public class Product {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Object Creation and Access: Example

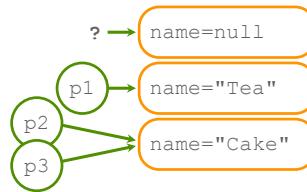
Object is an instance of a class.

- The `new` operator creates an object, allocating memory for it.
- You may assign an object reference to a variable of the appropriate type that (can be this class or any of its parents; details covered later).
- You can assign the same object reference to more than one variable. (This does not duplicate the object.)
- Invoke operations or access variables of the object using `."` operator, via object reference.
- Access modifiers may restrict an ability to access classes, variables, and methods from other classes.

```
package demos.shop;
public class Shop {
    public static void main(String[] args) {
        new Product();
        Product p1 = new Product();
        Product p2 = new Product();
        Product p3 = p2;
        p1.setName("Tea");
        p2.setName("Cake");
        System.out.println(p1.getName() + " in a cup");
        System.out.println(p2.getName() + " on a plate");
        System.out.println(p3.getName() + " to share");
        p1.name = "Coffee";
    }
}
```

```
package demos.shop;
public class Product {
    private String name;
    public void setName(String newName) {
        name = newName;
    }
    public String getName() {
        return name;
    }
}
```

```
>java demos.shop.Shop
Tea in a cup
Cake on a plate
Cake to share
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

You can create an object without assigning a reference to any variable.

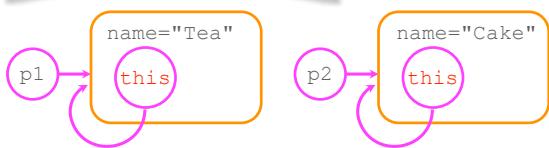
Essentially the `new` operator allocates memory to store the object even if you choose not to use it. You would not be able to access this object, so whatever logic the object is supposed to be doing should be triggered via this object's constructor. Coding constructors is covered later in this lesson.

Local Variables and Recursive Object Reference

Variables declared inside methods are known as Local.

- No access modifier can be applied to **local variables**. (They are not visible outside of the method anyway.)
- Method **parameters** are essentially local variables.
- The **local variable** can "shadow" the **instance variable** if their names coincide.
- Use the **this** keyword (recursive reference to current object) to refer to an instance, rather than local variable.
- Variables defined in inner blocks of code (delimited by " {}" symbols) are not visible outside of these blocks.

```
Product p1 = new Product();
Product p2 = new Product();
p1.setName("Tea");
p2.setName("Cake");
```



✿ Note: In the example, variables `p1` and `p2` are referencing different products, while `this` is referencing the current one.

```
public class Product {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        if (name == null) {
            String dummy = "Unknown";
            return dummy;
        }
        return name;
    }
    public String consume() {
        String feedback = "Good!";
        return feedback;
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

It may be best to avoid local and instance variable name clashes, by applying consistent naming conventions.

Note that the variable "`dummy`" in the example above is declared inside the `if` block. This makes it inaccessible to the code outside of this block.

In the example above, conditional logic inside the `getName` method is written to use an `if` statement without the `else` clause. This is because of the presence of the `return` statement inside the `if` block, which terminates this method.

Technically a method can return (terminate) at any point. However, it is usually considered to be a better coding style to write methods with a single `return` statement.

For example, same logic can be achieved like this:

```
public String getName() {  
    String dummy = "Unknown";  
    if (name == null) {  
        name = dummy;  
    }  
    return name;  
}
```

or may be even shorter using ternary (conditional assignment) operator:

```
public String getName() {  
    return (name == null) ? "Unknown" : name;  
}
```

Local Variable Type Inference

There is no need to describe a variable type if it can be unambiguously inferred from the context.

- Infer types of local variables with initializers.
- No need to explicitly declare local variable type if it can be inferred from the assigned value.
- Type cannot be reassigned and polymorphism is not supported.
- This feature is limited to:
 - Local variables with initializers
 - Indexes in the enhanced for-loops
 - Local variables declared in a traditional for-loops
- Overuse can reduce code readability.

```
public void someOperation(int param) {  
    var value1 = "Hello"; // infers String  
    var value2 = param; // infers int  
}
```

Notes

- ❖ Local variable type inference was introduced in Java 10.
- ❖ Polymorphism and loops are covered later.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Note that `var` is not a keyword, but a special identifier. You can create a variable called `var`, which is very confusing and therefore is something you would rather avoid.

Define Constants

Constants represent data that is assigned once and cannot be changed.

- The keyword `final` is used to mark a variable as a constant; once it is initialized, it cannot be changed.
- Instance final variables must be either initialized immediately or via all constructors.
- Local variables and parameters can also be marked as `final`.
- An attempt to reassign a final variable will result in compiler error.

```
public class Product {  
    private final String name = "Tea";  
    private final BigDecimal price = BigDecimal.ZERO;  
    public BigDecimal getDiscount(final BigDecimal discount) {  
        return price.multiply(discount);  
    }  
}  
  
public class Shop {  
    public static void main(String[] args) {  
        Product p = new Product();  
        BigDecimal percentage = BigDecimal.valueOf(0.2);  
        final BigDecimal amount = p.getDiscount(percentage);  
    }  
}
```

✿ Note: In this example, Product values are hardcoded. However, you can also dynamically assign them using a constructor, which is covered later.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

A final variable may reference an object that is not immutable. This would mean that although the variable itself cannot be reassigned, the content of the object that it references can still be modified.

```
final StringBuilder text = new StringBuilder();  
text.append("Hello");
```

In the very first version of Java, constants were described by using the keyword `const`. This is no longer the case, but `const` still remains a reserved word, although it is not actually used anymore.

Static Context

Each class has its own memory context.

- Class memory context (also known as static context) is shared by all instances of this class.
- The keyword `static` is used to mark variables or methods that belong to the class context.
- Objects can access shared static context.
- Current instance (`this`) is meaningless within the static context.
- Attempt to access current instance methods or variables from the static context will result in compiler error.

```
public class Product {
    private static Period defaultExpiryPeriod = Period.ofDays(3);
    private String name;
    private BigDecimal price;
    private LocalDate bestBefore;
    public static void setDefaultExpiryPeriod(int days) {
        defaultExpiryPeriod = Period.ofDays(days);
    }
    String name = this.name;
}
```

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

✖ Reminder: Java Classes are types and Objects are their instances.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

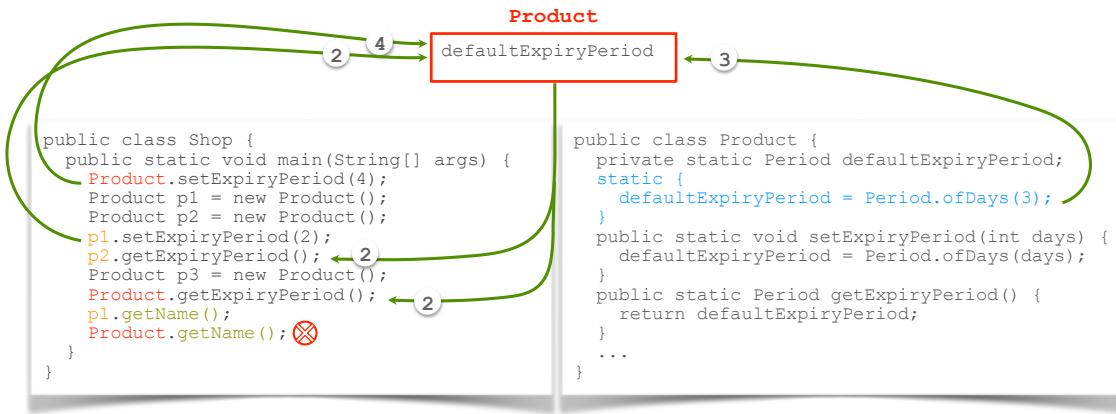
Although it is not possible to access the current instance context from static context, it is possible to access a specific instance:

```
public class Shop {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public static void main(String[] args) {
        Shop s = new Shop();
        s.setName("My Shop");
        s.name;
        /* referencing s.name or s.setName is ok,
           but this.name or this.setName(...) is not
           - because code is within the static method */
    }
}
```

Accessing Static Context

Rules for accessing static context:

- Object reference is not required (but can be used) to access static context.
- Static initializer runs once, before any other operation (when class is loaded).
- Instance variables and methods are not accessible through the static context.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

The example in this slide assumes that the `Product` class uses the `defaultExpiryPeriod` static variable to calculate the best before date.

Accessing static context via one of the instances using its reference may look ambiguous. In this example, `p1.setDefaultExpiryPeriod(2);` changes the value of a static variable, so all instances (not just `p2`) can now observe this changed value.

Accessing static methods or variables using object reference is considered to be a bad practice.

Combining Static and Final

Shared constants can be defined as `static` and `final` variables.

- It provides a simple way of defining globally visible constants.
- Encapsulation (private access modifier) is not required because value is read-only.

```
public class Product {  
    public static final int MAX_EXPIRY_PERIOD = 5;  
    ...  
}  
  
public class Shop {  
    public static void main(String[] args) {  
        Period expiry = Period.days(Product.MAX_EXPIRY_PERIOD);  
        ...  
    }  
}
```

- ❖ Reminder: Constants (final variables) could also be defined as instance and local variables and, therefore, could have a different initialization value for every instance or method context.
- ❖ Note: If only one such value is required (shared between all instances), then it can be defined in a class (static) context.



Other Static Context Use Cases

Static context is commonly used across many Java APIs.

Examples of static methods and variables encountered earlier in this course:

- The `main` method is static.
- All `Math` class operations are static.
- Factory methods** are static methods that create and return a new instance.
- Class `System` provides a static variable `out`. (Refer to `PrintStream` object for standard output.)

```
import static Math.random;
public class Shop {
    public static void main(String[] args) {
        Math.round(1.99);
        double value = random();
        BigDecimal.valueOf(1.99);
        LocalDateTime.now();
        ZoneId.of("Europe/London");
        ResourceBundle.getBundle("messages", Locale.UK);
        NumberFormat.getCurrencyInstance(Locale.UK);
        System.out.println("Hello World");
    }
}
```

✖ Note: `static import` enables referencing static variables and methods of another class as if they are in this class.

✖ The example shows the factory method creating a new big decimal object from the static method (fragment from the `BigDecimal` source code).

```
public class BigDecimal {
    public static BigDecimal valueOf(double val) {
        return new BigDecimal(Double.toString(val));
    }
    ...
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Class with static methods acts like a “function library” – supporting procedural programming paradigm. You do not need to instantiate `Math` class to use its methods.

Complete code example, that illustrates possible uses of static methods:

```
import static Math.random;
import java.time.*;
public class Shop {
    public static void main(String[] args) {
        double value1 = Math.round(1.99); // value1 is 2
        double value2 = random(); // value2 is a random number
        BigDecimal value3 = BigDecimal.valueOf(1.99); // value3 is reference to BigDecimal number 1.99
        LocalDateTime value4 = LocalDateTime.now(); //
```

```
value 4 is a reference to your current date and time:  
    ZoneId value5 = ZoneId.of("Europe/London");      //  
value5 is a reference to the timezone of London  
    ResourceBundle value6 =  
ResourceBundle.getBundle("messages", Locale.UK); // value6  
is a reference to a resource bundle for UK locale  
    NumberFormat value7 =  
NumberFormat.getCurrencyInstance(Locale.UK);          //  
value7 is a reference to NumberFormat object for UK  
locale  
    System.out.println("Hello World");  
}  
}
```

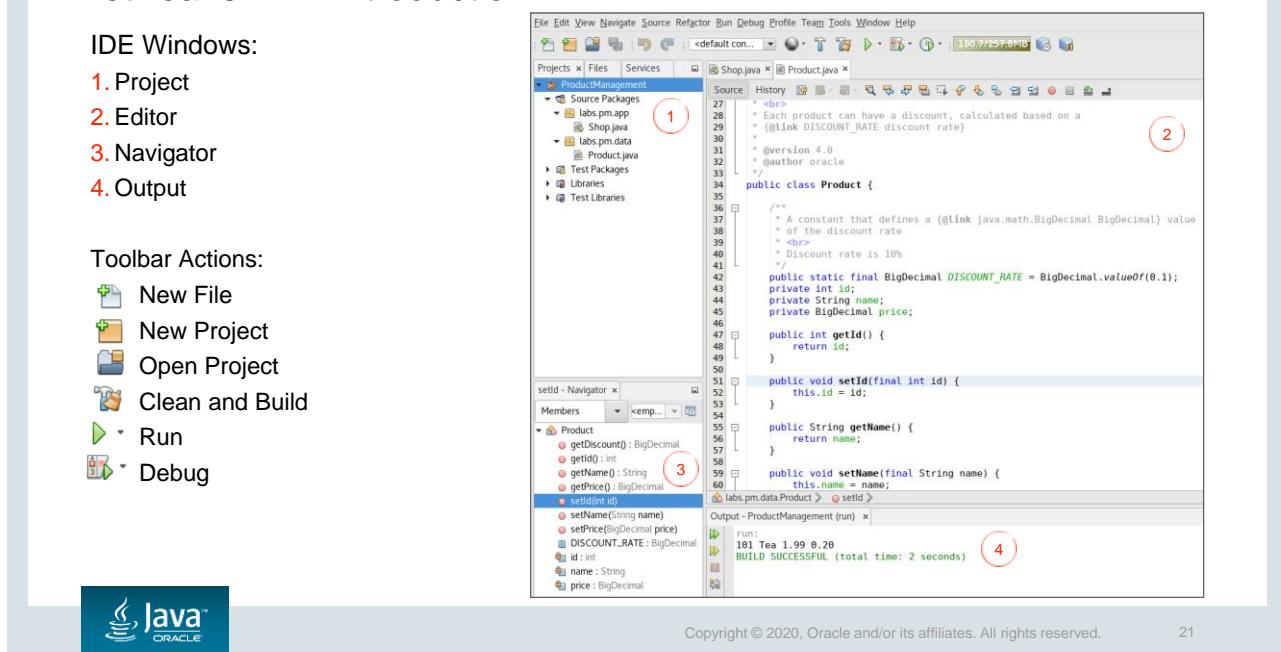
NetBeans IDE: Introduction

IDE Windows:

1. Project
2. Editor
3. Navigator
4. Output

Toolbar Actions:

- New File
- New Project
- Open Project
- Clean and Build
- Run
- Debug



The Projects window is the main entry point to your project sources. It shows a logical view of important project content. NetBeans 11 supports Maven, Gradle, and Ant project types.

The editor in NetBeans IDE is much more than a text editor. The NetBeans editor indents lines, matches words and brackets, and highlights source code syntactically and semantically.

Navigator enables you to quickly locate and navigate to specific parts of your source code.

The Output window displays anything printed on to the console, for example, compiler output or the output that your program produces.

Summary

In this lesson, you should have learned how to:

- Model business problems using classes
- Define instance methods and variables
- Describe the "this" object reference
- Explain object instantiation
- Explain local variables and local variable type inference
- Explain static variables and methods
- Invoke methods and access variables
- Describe NetBeans IDE



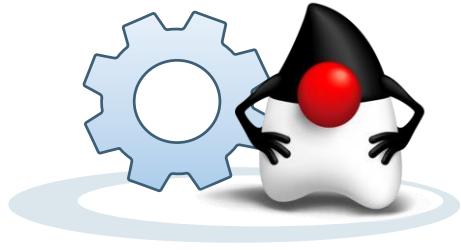
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

Practices

In this practice, you will:

- Use NetBeans to create a Product Management application project.
- Create classes Product and Shop.
- Documentation your code.
- Compile and execute your application.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

Improved Class Design



ORACLE®

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Use method overloading
- Create constructors
- Describe encapsulation and immutability
- Use enumerations
- Explain parameter passing
- Explain memory allocation and cleanup



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Overload Methods

Method overloading enables you to define more than one version of a method within a given class.

Overloaded methods follow these rules:

- Two or more methods, within the same class, that have **identical names**
- Must have **identical return type**
- Must have a **different number**, or **different types of parameters**, or both

```
public class Product {  
    private BigDecimal price;  
    private BigDecimal discount = BigDecimal.ZERO;  
    public void setPrice(double price) {  
        this.price = BigDecimal.valueOf(price);  
    }  
    public void setPrice(BigDecimal price) {  
        this.price = price;  
    }  
    public void setPrice(BigDecimal price,  
                         BigDecimal discount) {  
        this.price = price;  
        this.discount = discount;  
    }  
}
```

- Parameter names are irrelevant: You cannot define overloaded methods where only parameter names are different.
- Overloading is convenient: Invoker does not need to learn different method names, just set different parameters.

```
Product p = new Product();  
p.setPrice(1.99);  
p.setPrice(BigDecimal.valueOf(1.99));  
p.setPrice(BigDecimal.valueOf(1.99),  
           BigDecimal.valueOf(0.9));
```



Variable Number of Arguments

The vararg feature enables a **variable number of arguments** of the same type:

- It avoids creating too many overloaded versions of the same method.
- The vararg parameter is treated as an array, with the `length` constant indicating a number of values.
- An `int index` (starting at 0) is used to access array elements.
- In case where there are other parameters in the method, the vararg parameter must be defined last.
- Using varargs is convenient—invoker simply sets a required number of values.

```
Product p = new Product();
p.setFiscalDetails(1.99);
p.setFiscalDetails(1.99, 0.9, 0.1);
p.setFiscalDetails(1.99, 0.9);
```

```
public class Product {
    private BigDecimal price;
    private BigDecimal discount;
    private BigDecimal tax;
    public void setFiscalDetails(double... values) {
        switch (values.length) {
            case 3:
                tax = BigDecimal.valueOf(values[2]);
            case 2:
                discount = BigDecimal.valueOf(values[1]);
            case 1:
                price = BigDecimal.valueOf(values[0]);
        }
    }
}
```

✿ Note: Arrays are covered later in the course.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

Reminders:

- You have invoked a method with varargs earlier in this course, when formatting messages, with any number of substitution parameters:

```
MessageFormat.format("from {0} to {1}", "a", "b");
// produces text: from a to b

MessageFormat.format("from {0} to {1} via {2}", "a", "b",
"c"); // produces text: from a to b via c
```

- Another example for the vararg approach is the `main` method, which may be defined using varargs instead of array: (There is no difference in the way it is handled.)

```
public static void main(String... args) { }
```

Invoker can treat varargs as arrays as well:

```
Product p = new Product();
double[] fiscalValues = {1.99, 0.9, 0.1};
p.setFiscalDetails(fiscalValues);
```

Define Constructors

A Constructor is a special method that initializes the object:

- Is invoked using the `new` operator
- Must be **named after the class** and must not have a return type or be defined as `void`
- Usually public, **may have parameters**, and can be overloaded like any other method
- A default constructor **with no parameters** is implicitly added to the class, only if no other constructors were added.
- You may explicitly add the **no-arg constructor**, as yet another overloaded version of constructor, if you want to be able to instantiate your class with or without providing constructor arguments.

```
public class Product {
    private String name;
    private BigDecimal price;
}
```

`new Product();`

```
public class Product {
    private String name;
    private BigDecimal price;
    public Product() {
    }
    public Product(String name) {
        this.name = name;
        this.price = BigDecimal.ZERO;
    }
}
```

`new Product("Water");`
`new Product();`



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

In addition to constructors, the class may contain a simple block of code on a class level. This is known as instance initializer. Such a block is executed before any other constructor of this class.

This concept is similar to a static class initializer. However, the difference is that static initializer is only ever triggered once, when this class is loaded to memory, while the instance initializer is executed every time this class is instantiated.

```
public class Product {
    /* instance initialiser */
    static { /* class initialiser */ }
}
```

Reuse Constructors

A constructor can invoke another to reuse its logic.

- Invocation of another constructor is performed by using the following syntax:
`this(<other constructor parameters>);`
- Such a call must be the first line of code in the invoking constructor.
- A cycle (loop) of constructor invocations is not allowed.

```
public class Product {  
    private String name;  
    private BigDecimal price;  
    public Product(String name, double price) {  
        this(name);  
        this.price = BigDecimal.valueOf(price);  
    }  
    public Product(String name) {  
        this.name = name;  
        this.price = BigDecimal.ZERO;  
    }  
}  
  
new Product("Water"); ✓  
new Product("Tea", 1.99); ✓  
new Product(); ✗
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

Writing code that makes one constructor invoke another is sometimes referred to as constructor chaining.

You would be able to create an instance of a class from any other class if you mark constructor with a private access modifier.

Earlier in this course, you've seen an example of the Math class that provides a number of static methods to perform various mathematical functions.

You never actually needed to create an instance of Math. An attempt to create such an instance would not compile, because Math class constructor is marked as private.

Access Modifiers Summary

Access Modifiers control access to classes, methods, and variables:

- **public** - Visible to any other class
- **protected** - Visible to classes that are in the same package and to subclasses
- **<default>** - Visible to classes in the same package only
- **private** - Visible within the same class only

	any class	classes in the same package and subclasses	classes in the same package only	same class
public	✓	✓	✓	✓
protected	✗	✓	✓	✓
<default>	✗	✗	✓	✓
private	✗	✗	✗	✓

Notes

- ❖ <default> means that no access modifier is explicitly set.
- ❖ The Subclass-Superclass relationship (use of the `extends` keyword) is covered later in the course.



Define Encapsulation

Information contained within the object should normally be "hidden" inside it.

- Instance variables are usually declared with the `private` access modifier so that they are not visible outside of this class.
- To access this information from other classes, you may provide methods with less restrictive access modifier, for example, `public`, `<default>`, `protected`.
- This allows you to control access to information, validate data, or modify data format.
- There are also some `private` methods that can be invoked only from other methods of the same class.

```
public class Product {
    private BigDecimal price;
    private BigDecimal tax;
    private BigDecimal rate = BigDecimal.valueOf(0.1);
    public void setPrice(BigDecimal price) {
        this.price = price;
        calculateTax();
    }
    public BigDecimal getPrice() {
        return price.add(tax);
    }
    private void calculateTax() {
        tax = price.multiply(rate);
    }
}
```

```
public class Shop {
    public static void main(String[] args) {
        Product p1 = new Product();
        p1.setPrice(BigDecimal.valueOf(1.99));
        BigDecimal total = p1.getPrice();
        p1.tax = 3.20;
        p1.calculateTax();
    }
}
```

✿ Note: Attempt to access "hidden" variable or method from another class will result in a compilation error.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Reason for hiding information and behaviors within the class is to be able to provide a more suitable interface in a form of methods that are visible to outside callers. This approach allows operation to validate, calculate, and prepare values and prevents external observers from direct assessing internal object data or operations. Writing code that embraces encapsulation eliminates various side effects, such as shadowing of superclass variables and tight coupling between program units, and generally makes your code less brittle and easier to maintain.

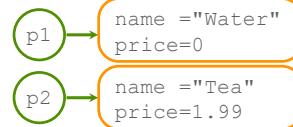
Define Immutability

Immutable objects present read-only data. (You cannot modify it after object construction.)

- Instance variables must be encapsulated (**private**) to prevent direct access.
- Instance variables are **initialized immediately** or via **constructors**.
- No setter methods are provided.
- Immutable objects are thread-safe, without an overhead cost of coordinating synchronized access.
- Many JDK classes are designed this way: primitive wrappers, local date and time, string, and so on.

```
public class Product {
    private String name;
    private BigDecimal price = BigDecimal.ZERO;
    public Product(String name) {
        this.name = name;
    }
    public Product(String name, BigDecimal price) {
        this.name;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public BigDecimal getPrice() {
        return price;
    }
}
```

```
public class Shop{
    public static void main(String[] args) {
        Product p1 = new Product("Water");
        String name = p1.getName();
        BigDecimal price = p1.getPrice();
        price = BigDecimal.valueOf(1.99);
        Product p2 = new Product("Tea", price);
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Constants and Immutability

Constants represent data that is assigned once and cannot be changed.

- The `final` keyword is used to mark a variable as a constant. Once it is initialized, it cannot be changed.
- Instance `final` variables must be either initialized immediately or using `instance initializer` or all constructors.

```
public class Product {  
    private static int maxId = 0;  
    private final int id;  
    private final String name;  
    private final BigDecimal price;  
    { id = ++maxId; }  
    public Product(String name) {  
        this.name = name;  
        this.price = BigDecimal.ZERO;  
    }  
    public Product(String name, BigDecimal price) {  
        this.name = name;  
        this.price = price;  
    }  
    public BigDecimal getDiscount(final BigDecimal discount) {  
        return price.multiply(discount);  
    }  
}
```

✖ Note: Instance initializer is a block of code that is triggered before the invocation of the constructor.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Note: Both discount and price are defined as final variables and cannot be changed. However, the multiply method is not changing either one of these variables but creates a new BigDecimal object instead. This is a typical coding technique for maintaining immutable objects.

Enumerations

Enumeration (enum) provides a fixed set of instances of a specific type.

- Enum values are **instances of this enum type**. (HOT, WARM, and COLD are instances of the Condition.)
- Enum values are implicitly public, static, and final.
- Enums can be used as:
 - Variable types
 - Cases in the switch constructs

```
package demos;
public enum Condition {
    HOT, WARM, COLD;
}
```

```
package demos;
import static Condition.*;
public class Shop {
    public static void main(String[] args) {
        Product tea = new Product("Tea", HOT);
        Person joe = new Person("Joe");
        joe.consume(tea.serve());
    }
}
```

```
package demos;
public class Product {
    private String name;
    private String caution;
    private Condition condition;
    public Product(String name, Condition condition) {
        this.name = name;
        this.condition = condition;
    }
    public Product serve() {
        switch(condition) {
            case Condition.COLD:
                this.addCaution("Warning COLD!");
                break;
            case Condition.WARM:
                this.addCaution("Just right");
                break;
            case Condition.HOT:
                this.addCaution("Warning HOT!");
                break;
        }
        return this;
    }
    ...
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

In the example in the slide, assume that the Person class has a consume method that accepts a Product object as an argument.

Enums enhance readability and reduce errors and maintenance of hard-coded values.

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

All the constants of an enum type can be obtained by calling the implicit `public static T[] values()` method of that type.

```
for (Condition c: Condition.values()) {
    System.out.println(c.ordinal()+" "+c.name());
}
```

The example prints the ordinal and name of every enum value:

```
0 HOT
1 WARM
2 COLD
```

Complex Enumerations

Enumerations can define instance **variables** and **methods**.

- A **Constructor** should be added to the enumeration to initialize its instance variables.
- The constructor for an enum type must have default or private access.
- Declaration of enum values **invokes the enum constructor**.
- The enum constructor can be invoked outside of enum.

```
package demos;
public enum Condition {
    HOT("Warning HOT!"),
    WARM("Just right"),
    COLD("Warning COLD!");
    private String caution;
    private Condition(String caution) {
        this.caution = caution;
    }
    public String getCaution() {
        return caution;
    }
}
```

Notes

- ❖ The restriction imposed on the enum constructor guarantees that enum represents a fixed set of values.
- ❖ Once **caution** becomes a property of the enum, the switch from the previous example becomes redundant.



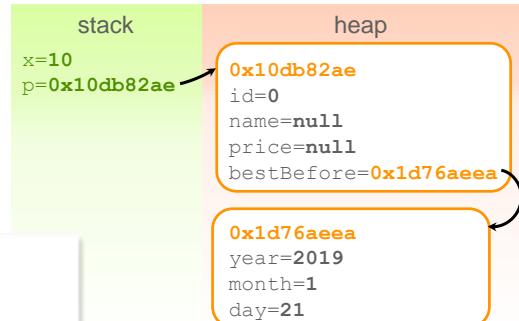
Java Memory Allocation

Java has the following memory contexts: stack and heap.

- Stack is a memory context of a thread, storing local method variables.
- Heap is a shared memory area, accessible from different methods and thread contexts.
- Stack can hold only primitives and object references.
- Classes and Objects are stored in the heap.

```
public class Shop {
    public static void main(String[] args) {
        int x = 10;
        Product p = new Product();
    }
}

public class Product {
    private int id;
    private String name;
    private BigDecimal price;
    private LocalDate bestBefore = LocalDate.of(2019,1,21);
}
```



✖ Note: Creation of a new object does not necessarily initialize all of its values.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

You may initialize instance variable within the object at the point when the new instance is created. This could be done by assigning default values to instance attributes immediately as they are declared, or via the use of constructors. Constructors are covered in the lesson titled “Inheritance.”

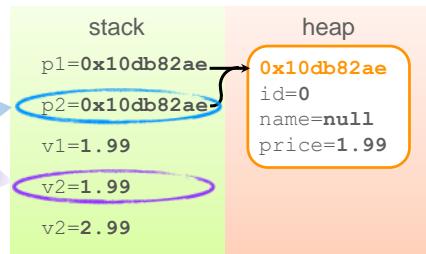
Parameter Passing

Just like any other local method variables, parameters are stored on a stack.

Passing parameters means copying stack values:

- A copy of an object reference value
- A copy of a primitive value

```
public void manageProduct() {
    Product p1 = new Product();
    orderProduct(p1);
    double v1 = p1.getPrice();
    changePrice(v1);
}
public void orderProduct(Product p2) {
    p2.setPrice(1.99);
}
public void changePrice(double v2) {
    v2 = 2.99;
}
```



- The `p2` parameter is initialized by copying the value of the `p1` variable, which is a reference pointing to the **same object** in the heap, making this object accessible to both methods.
- The `v2` parameter is initialized by copying the value of the `v1` variable. Modification of the `v2` variable has no effect on the `v1` variable.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

If your method operates on an immutable object, it cannot be modified.

However, your method can create another object with the required changes. In this case, such an object would also be invisible to the calling method, unless you explicitly return a reference to it.

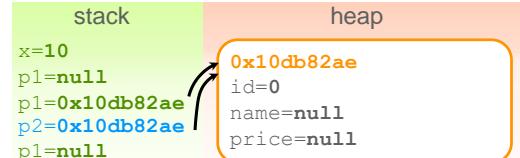
If you intend to inform the invoking method about value changes of locally scoped variables within your method, you can define such methods as returning a value, rather than void.

Java Memory Cleanup

Objects remain in the heap, so long as they are still referenced.

- An object reference is `null` until it is initialized.
- Assigning `null` does not destroy the object, just indicates the absence of reference.
- When a method returns, its local variables go out of scope and their values are destroyed.

```
public void manageProduct() {
    int x = 10;
    Product p1;
    p1 = new Product();
    orderProduct(p1);
    p1 = null;
}
public void orderProduct(Product p2) {
    // continue to use p2 reference
}
```



Garbage collection is a background process that cleans unused memory within Java run time.

- Garbage collection is deferred.
- Object becomes eligible for garbage collection when there are no references pointing to it.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Setting an object reference to `null`: `Product p1 = null;` will "disconnect" this variable from the specific object, but it does not mean that all other references to the object are lost too. It could be that this object reference has been previously copied into another variable (e.g. passed as parameter) and is still accessible, even though this particular variable (object reference) is no longer pointing to it.

Deferred Garbage Collection means that it is not immediately triggered when all object references to the object are lost.

You can prompt Java run time to run garbage collection using `System.gc();` or an equivalent `Runtime.getRuntime().gc();` operation. However, there is no guarantee that JVM would actually run garbage collection and clean memory at this point. This really depends on many factors, such as the amount of available memory versus used memory (called "memory pressure"). JVM is trying to achieve a good balance between housekeeping tasks such as memory management and the actual execution of the logic of your program. Deferred garbage collection is designed to achieve this balance.

Summary

In this lesson, you should have learned how to:

- Use method overloading
- Create constructors
- Describe encapsulation and immutability
- Use enumerations
- Explain parameter passing
- Explain memory allocation and cleanup



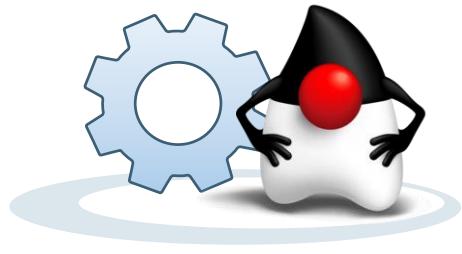
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

Practices

In this practice, you will:

- Create Enumeration to represent Product rating
- Add custom constructors to the Product class
- Make Product objects immutable



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Inheritance



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Extend classes
- Reuse code through inheritance
- Use the `instanceof` operator
- Describe the `super` object reference
- Define subclass constructors
- Override superclass methods
- Explain polymorphism
- Define abstract classes and methods
- Define final classes and methods
- Override methods of the `Object` class



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Extend Classes

Classes form a hierarchy descending from the `java.lang.Object` class.

- Class `Object` is an ultimate parent of any other class in Java.
- Class `Object` defines common, generic operations that all other Java classes inherit and reuse.
- There is no practical difference between:

```
public class Product { }  
and  
public class Product extends Object { }
```

because the `extends Object` clause is implied when an explicit `extends` clause is not present

- The explicit `extends` clause describes which specific class should be extended instead of the `Object` class.

```
public class Food extends Product { }
```

- Parent class (the one you extend) is known as the superclass.
- Child class (the one that extends the parent) is known as the subclass.
- A class can have only one immediate parent, as multiple inheritance is not allowed in Java

❖ Note: In the example, `Food` is a child of `Product` and a grandchild of `Object`.



Object Class

Object class contains generic behaviors that are inherited by all Java classes, such as:

- The `toString` method creates text value for the object.
- The `equals` method compares a pair of objects.
- The `hashCode` method generates int hash value for the object.
- The `clone` method produces a replica of the object.
- `wait`, `notify`, and `notifyAll` methods control threads.

Code of the `toString` operation of the Object class:

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Is instantly available for any Java class: `Product p = new Product(); String s = p.toString();`

Produces text: `demos.Product@6e8dacdf`

✖ Note: Operations of the Object class are covered later in more detail.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

Operations inherited from the Object class

- `protected Object clone()`: Creates and returns a copy of this object
- `public boolean equals(Object obj)`: Indicates whether some other object is "equal to" this one
- `protected void finalize()`: Deprecated, because the finalization mechanism is inherently problematic
- `public final Class<?> getClass()`: Returns the runtime class of this object
- `public int hashCode()`: Returns a hash code value for the object
- `public final void notify()`: Wakes up a single thread that is waiting on this object's monitor
- `public final void notifyAll()`: Wakes up all threads that are waiting on this object's monitor
- `public String toString()`: Returns a string representation of the object

- `public final void wait():` Causes the current thread to wait until it is awakened, typically by being notified or interrupted
- `public final void wait(long timeoutMillis):` Causes the current thread to wait until it is awakened, typically by being notified or interrupted or until a certain amount of real time has elapsed
- `public final void wait(long timeoutMillis, int nanos):` Causes the current thread to wait until it is awakened, typically by being notified or interrupted or until a certain amount of real time has elapsed

Reuse Parent Class Code Through Inheritance

The purpose of inheritance is to reuse generic superclass behaviors and state in subclasses.

- **Superclass** represents a more generic, parent type.
- Superclasses define common attributes and behaviors.
- **Subclass** represents a more specific, child type that **extends the parent type**.
- Subclasses inherit all attributes and behaviors from their parents.
- Subclasses may define subtype-specific attributes and behaviors.

```
public class Product {  
    private int id;  
    private String name;  
    private BigDecimal price;  
    public int getId() {  
        return id;  
    }  
    // other generic methods and variables  
}
```

```
public class Food extends Product {  
    private LocalDate bestBefore;  
    public LocalDate getBestBefore() {  
        return bestBefore;  
    }  
    // other Food specific methods and variables  
}  
  
public class Drink extends Product {  
    // other Drink specific methods and variables  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

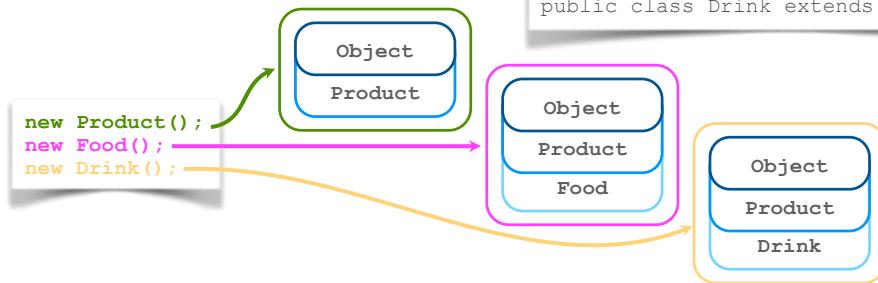
6

Instantiating Classes and Accessing Objects

Heap memory allocated to store object (class instance) that contains:

- Code of the specific subtype
- All of the parents up the class hierarchy

```
public class Product { }
public class Food extends Product { }
public class Drink extends Product { }
```



Object reference can be of generic or specific type, projecting the entire object or only some of its methods and variables. Example:

- **x1** accesses code declared on the Object class level only
- **x2** accesses code declared on the Object and Product class levels
- **x3** accesses code declared on the Object, Product, and Food class levels

```
Object x1 = new Food();
Product x2 = new Food();
Food x3 = new Food();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Rules of Reference Type Casting

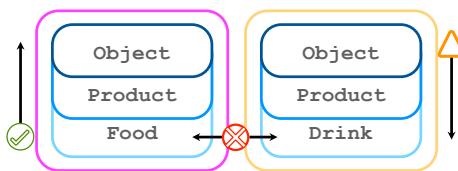
An object can be referenced using either of the following:

- Specific child subclass type
- Generic parent superclass types

To invoke an operation on the object, reference type has to be specific enough to be at least at the level in the class hierarchy where the operation was first declared.

Type casting rules:

- ⚠ Casting is required to assign parent to child reference type.
- ✓ No casting is required to assign child to parent reference type.
- ✗ Casting is not possible between objects of sibling types.



✳ Example assumes Product defines getName() and Food defines getBestBefore() operations

```

Food x1 = new Food();
Product x2 = new Drink();
x1.toString();
x1.getName();
x1.getBestBefore();
Product x3 = x1;
x3.toString();
x3.getName();
x3.getBestBefore();
Object x4 = x1;
x4.toString();
x4.getName();
x4.getBestBefore();
Product x5 = (Product)x4;
x5.toString();
x5.getName();
x5.getBestBefore();
Drink x6 = (Drink)x2;
Drink x6 = (Drink)x3;

```



Verify Object Type Before Casting the Reference

Methods should normally be defined using generic (superclass) parameter and return value types.

- You need not verify or cast the reference type to invoke **generic operations**.
- You should verify the object type by using the **instanceof** operator before **casting reference to a specific type**.
- Invoke **subtype specific operations** using a specific reference type.

```
public void order(Product p) {
    BigDecimal price = p.getPrice();
    BigDecimal discount = BigDecimal.ZERO;
    if (p instanceof Food) {
        discount = ((Food)p).getBestBefore().isEqual(LocalDate.now().plusDays(1))
            ? price.multiply(BigDecimal.valueOf(0.1))
            : BigDecimal.ZERO;
    }
    if (p instanceof Drink) {
        LocalTime now = LocalTime.now();
        discount = (now.isAfter(LocalTime.of(17,30)) && now.isBefore(LocalTime.of(18,30)))
            ? f.getPrice().multiply(BigDecimal.valueOf(0.2))
            : BigDecimal.ZERO;
    }
    price = price.subtract(f.getDiscount());
}
```

order(new Food("Cake",2.99));
order(new Drink("Tea",1.99));

 Note: If object is null, the instanceof operator returns a false value.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Using generic (superclass) types to define method parameters and return values help to promote better code reusability and extensibility. Operations would not be hard-coded to use specific subtypes and can still function even if extra subclasses would be added later.

Assumptions for the code example:

- Product is a superclass extended by classes Food and Drink.
- The getPrice method is a generic operation declared within the parent class Product.
- The getBestBefore method is specific to the Food class.
- The order method is designed to accept any Product as an argument.

There is no need to check which specific subtype this product is or cast the reference in order to access generic operations such as getPrice. However, because only those products that are of a Food subtype define the getBestBefore method, a type check and casting are required to access such subtype specific operations. A type check can also be used to alter program logic for specific subtype. In this example, products type of food get discount of 10% one day before the best-before date and drinks get 20% discount between 17:30 and 18:30.

Reference Code Within the Current or Parent Object

Reference the current object and the parent object.

- Use the `this` keyword to reference variables and methods of the current object.
- Use the `super` keyword to reference variables and methods of the parent object.
- The `this` or `super` keyword is not required when the reference is not ambiguous.

```
public class Product {  
    public BigDecimal discount;  
    public BigDecimal price;  
}  
  
public class Food extends Product {  
    private BigDecimal discount;  
    public BigDecimal getDiscount() {  
        return price.subtract(this.discount.add(super.discount));  
    }  
}
```

Reminders

- ❖ Access modifiers (default or private) would prevent subclass from accessing parent class variables and methods.
- ❖ Well-encapsulated code should only expose methods and should hide variables.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Variables that are not private are potentially visible within subclasses. This could create an ambiguity, when a subclass can create another variable with the same name, and both will be available to the subclass. You can resolve this ambiguity using the `super` and `this` keywords. However, it is easy to forget to put these keywords in front of a variable, which can result in bugs in the program logic. Therefore, it is strongly recommended to make all of your variables private so that no code outside of the given class, including subclasses, would never be put in this ambiguous situation.

Define Subclass Constructors

The subclass constructor must invoke the superclass constructor.

- Superclass contains the no-arg constructor (default or explicitly defined).
Subclass can implicitly invoke the superclass constructor.
- Superclass does not provide the no-arg constructor (only constructors with parameters are present).
Subclass constructor must explicitly invoke superclass constructor.
- Superclass constructor is invoked by using the `super` keyword
with **matching constructor signature**.
- Invocation of superclass constructor must be
the first line of code in the subclass constructor.

Must be
invoked
explicitly

```
public class Product {
    public Product(String name) {...}
}

public class Food extends Product {
    public Food(String name, LocalDate bestBefore) {
        super(name);
        ...
    }
}
```

no-arg constructor invocations can be implied

```
public Product() {
    super();
}
```

```
public Food() {
    super();
}
```

```
public class Product {}
```

```
public class Food extends Product {}
```

- ❖ Reminder: If no other constructors are present in the class, the no-arg constructor is provided by default.
- ❖ Note: Object class provides a no-arg constructor.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Reminder: Parameter names are irrelevant; you must match types and number of parameters when invoking the superclass constructor.

Class and Object Initialization Summary

Class loading and initialization execution order:

(The following code is executed only once.)

1. Object class static initializer
2. Shop class static initializer
3. Product class static initializer
4. Food class static initializer

❖ Notes

- All code of the class must be loaded into memory first.
- It needs to be loaded only once per class.

Object instantiation execution order:

(The following code is executed per each instantiation or relevant type of object.)

1. Object class constructor
2. Product instance initializer
3. Product constructor
4. Food instance initializer
5. Food constructor

❖ Notes

- Each object instance must be initialized together with all of its parents.
- Each object instance memory contains object data and references to the rest of the class code (shared between all instances).

```
public class Shop {
    static { }
    public static void main(String[] args) {
        Product p1 = new Food();
        Product p2 = new Food();
    }
}
```

```
public class Object {
    static { }
    public Object() { }
}

public class Product {
    static { }
    public Product() { }
}
```

```
public class Food extends Product {
    static { }
    public Food() { }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Class loading and initialization execution order:

1. Object class static initializer
 - Method main is triggered first
 - Class Shop must be loaded to memory
 - Shop extends Object, so Object must be loaded first anyway
2. Shop class static initializer
3. Product class static initializer
 - The main method creates a new instance of Food.
 - Food class must be loaded to memory.
 - Food extends Product, which extends Object, so they must be loaded first.
 - Object class has already been loaded to memory and its static initializer has already been executed.
4. Food class static initialiser

Object instantiation execution order

1. Object class constructor
 - The `main` method is trying to create an instance of `Food`, which requires initialization of `Product` and `Object` first.
 - `Object` class does have the no-arg constructor, but does not have instance initializer.
2. Product instance initializer
3. Product constructor
4. Food instance initializer
5. Food constructor

Instance and static initializers are optional, while constructors are not. However, default no-arg constructors are provided if no other constructor is available for a given class.

Override Methods and Use Polymorphism

The subclass can override parent class methods.

- The subclass defines the method whose signature matches parent class method.
- The subclass can widen, but cannot narrow access of methods it overrides.
- Polymorphism (many forms) in Java means when a method is declared in a superclass and is overridden in a subclass, the subclass method takes precedence without casting reference to a specific subclass type.

```
order(new Product("Something",1.5));
order(new Food("Cake",2.99));
order(new Drink("Tea",1.99));

public void order(Product p) {
    BigDecimal price = p.getPrice();
}
```

- ❖ Polymorphism simplifies coding (compare with the example from page 9).
- ❖ Annotation `@Override` is optional; it is used to ensure that subclass method signature matches the superclass method.

```
public class Product {
    public BigDecimal getPrice() {
        // generic product logic
    }
}
```

```
public class Food extends Product {
    @Override
    public BigDecimal getPrice() {
        // specific food logic
    }
}
```

```
public class Drink extends Product {
    @Override
    public BigDecimal getPrice() {
        // specific drink logic
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

When overriding a method, the subclass cannot narrow its access modifier.

For example, if the parent class defines a protected method, the subclass can override it using public access modifier, but not using default to private.

Overriding parent class methods allow Java classes to utilize benefits of polymorphism. In the example, all logic that is specific to the particular subtypes is defined within the overridden method of the corresponding subtype:

```
public class Food extends Product {
    // the rest of the Food class logic
    public BigDecimal getPrice() {
        BigDecimal discount =
            bestBefore.isEqual(LocalDate.now().plusDays(1))
                ?
                super.getPrice().multiply(BigDecimal.valueOf(0.1)) :
                BigDecimal.ZERO;
        return super.getPrice().subtract(discount);
    }
}
```

```
public class Drink extends Product {  
    // the rest of the Drink class logic  
    public BigDecimal getPrice() {  
        LocalTime now = LocalTime.now();  
        BigDecimal discount =  
            (now.isAfter(LocalTime.of(17, 30)) &&  
             now.isBefore(LocalTime.of(18, 30)))  
                ?  
                super.getPrice().multiply(BigDecimal.valueOf(0.2)) :  
                BigDecimal.ZERO;  
        return super.getPrice().subtract(discount);  
    }  
}
```

Any class that needs to operate with these operations can simply use the superclass, without a need to verify subclass types with the `instanceof` operator or cast reference types:

```
public void order(Product p) {  
    BigDecimal price = p.getPrice();  
}
```

Because of polymorphism, the appropriate subclass implementation of the operation will be automatically invoked. In the earlier example, these behaviors were not encapsulated by subclasses and, therefore, it has been the responsibility of the invoker to determine specific subtype logic, complicating the algorithm.

Another important benefit of polymorphism is that you can later create other subclasses and implement alternative subclass specific behaviors, without modifying any invoking operations that utilize the superclass type.

Reuse Parent Class Logic in Overwritten Method

The subclass may invoke a parent class method using the `super` reference to use parent logic.

```
public class Food extends Product {  
    private LocalDate bestBefore;  
    public BigDecimal getPrice() {  
        BigDecimal discount = (bestBefore.isEqual(LocalDate.now().plusDays(1))  
            ? super.getPrice().multiply(BigDecimal.valueOf(0.1))  
            : BigDecimal.ZERO;  
        return super.getPrice().subtract(discount);  
    }  
}
```

❖ Note: The subclass that overrides the parent method without invoking it via `super` reference essentially disables the parent method logic.

```
public class Product {  
    private BigDecimal price;  
    public BigDecimal getPrice() {  
        return price;  
    }  
}
```

```
public class Drink extends Product {  
    public BigDecimal getPrice() {  
        LocalTime now = LocalTime.now();  
        BigDecimal discount = (now.isAfter(LocalTime.of(17, 30))  
            && now.isBefore(LocalTime.of(18, 30))  
            ? f.getPrice().multiply(BigDecimal.valueOf(0.2))  
            : BigDecimal.ZERO;  
        return super.getPrice().subtract(discount);  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

Define Abstract Classes and Methods

The `abstract` keyword can be used to encourage class extensibility.

- Class cannot be directly instantiated if it is marked with the `abstract` keyword.
- The abstract class purpose is to be extended by one or more concrete subclasses.
- The abstract class can contain normal variables and methods, which are inherited by its subclasses as usual.
- It may also contain abstract methods that describe method signatures, without a method body.
- Concrete subclasses must override all abstract methods of their abstract parent.

```
order(new Product("???",0));
order(new Food("Cake",2.99));
order(new Drink("Tea",1.99));

public void order(Product p) {
    p.serve();
}
```

```
public abstract class Product {
    // any other code
    public abstract void serve();
}

public class Food extends Product {
    // any other code
    public void serve() {
        // put food on a plate
    }
}

public class Drink extends Product {
    // any other code
    public void serve() {
        // pour drink in a cup
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Abstract classes can be used to define generic methods that do not have any specific concrete implementation at the level in the hierarchy where this abstract class is defined. However, because abstract classes introduce method signatures at a very high level, they promote the use of polymorphism. Other classes and methods may use abstract type to define parameters and return types and can rely upon the fact that this type provides certain methods, even if the implementation of these methods is only available at the subtype level. Absence of concrete implementation for such methods in the abstract class itself is not an issue, because abstract classes cannot be directly instantiated, and any subclasses of the abstract class would have to either be abstract themselves or override all abstract methods inherited from their parent. Eventually, when you get to the level of the concrete class, all abstract methods would have subtype specific implementations. Because of Java polymorphism, subclass specific version of the abstract method will be automatically invoked, without a need to perform any type verification or type casting.

The following example demonstrates that an instance of subclass is also considered to be the instance of its parent, even if the parent is abstract and cannot be instantiated directly. (There is no need to actually verify object type to in this example.)

```
public void order(Product p) {  
    boolean x = (p instanceof Product);  
    boolean y = (p instanceof Food);  
    boolean z = (p instanceof Food);  
}
```

- Variable `x` is true so long as parameter is not null.
- Variable `y` is true when parameter is a not null and is of type Food.
- Variable `z` is true when parameter is a not null and is of type Drink.

Define Final Classes and Methods

The `final` keyword can be used to limit class extensibility.

- Class cannot extend a class that is marked with the `final` keyword.
- The subclass cannot override a superclass method that is marked with the `final` keyword.

```
public class Product {  
    public final void processPayment() {  
        // method can not be overridden by subclasses  
    }  
}  
  
public class Drink extends Product {  
    public void processPayment() {}  
}  
  
public final class Food extends Product {  
    // class can not be extended  
}  
  
public class JunkFood extends Food {  
          
}
```

✖ Note: Attempt to override a final method or extend a final class would result in compilation error.

✖ Reminder: A variable becomes a constant (cannot be reassigned) if it is marked with the `final` keyword.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Override Object Class Operations: `toString`

It is recommended that all Java classes override some operations defined by the `Object` class.

- The `toString` method produces text value for the object, which can be easily used in logging.
- Various operations in JDK classes use the `toString` operation. For example, the `println` method checks if the object is not null and invokes the `toString` method to produce text output.
- A specific subclass version of the operation automatically takes precedence due to Java polymorphism.
- The subclass version of the method may choose to **reuse** or redefine superclass operation logic.

```
public class Product {  
    // other methods and variables  
    public String toString() {  
        return id+" "+name+" "+price;  
    }  
}  
  
public class Food extends Product {  
    // other methods and variables  
    public String toString() {  
        return super.toString()+" "+bestBefore;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Product p = new Food(42,"Cake",2.99,LocalDate.now().plusDays(1));  
        System.out.println(p);  
    }  
}
```

`java Test
'42 Cake 2.99 2019-2-28'`



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

Override Object Class Operations: equals

It is recommended that all Java classes override some operations defined by the Object class.

- The `equals` method compares objects.
- This is used to identify if objects should be considered identical.

```
public class Product {  
    // other methods and variables  
    public boolean equals(Object obj) {  
        if (this == obj) {  
            return true;  
        }  
        if (!(obj instanceof Product)) {  
            return false;  
        }  
        Product other = (Product) obj;  
        return this.id == other.id;  
    }  
}
```

- The `==` operator compares values in the stack. It can be used to compare primitive values or to determine if two references are pointing to the same object.
- The overriding method `equals` enables you to compare object content in the heap.
- Java classes such as `String`, `Number`, `LocalDate`, and so on override the `equals` method.

```
public class Test {  
    public static void main(String[] args) {  
        Product p1 = new Product(42, "Cake", 2.99);  
        Product p2 = new Product(42, "Cake", 2.99);  
        boolean sameObject = (p1 == p2);           // false  
        boolean sameContent = (p1.equals(p2));      // true  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

21

The example in the slide assumes that `Product` does have int `id`, String `name`, `BigDecimal` `price`, and possibly other fields. The `equals` method is comparing product objects based on just the `Product id value`.

Override Object Class Operations: hashCode

It is recommended that all Java classes override some operations defined by the Object class.

- The `hashCode` method generates object identity as an `int` value.
 - Must consistently return the same `int` value for the same instance
 - Used for bucketing hashed collections, such as `HashSet`, `HashMap`, `Hashtable`
- The `hashCode` method should return the same `int` value for any pair of objects that are considered to be the same when compared with the `equals` method.
- The `Objects` class contains the `hash` method that generates a hash code value for a number of objects.

Assume equals method compares product ids:

```
public class Product {
    // other methods and variables
    public int hashCode() {
        return this.id;
    }
}
```

Assume equals method compares product names and prices:

```
public class Product {
    // other methods and variables
    public int hashCode() {
        return Objects.hash(name, price);
    }
}
```

- Hashed collections are covered in the Java Collection API lesson.
- A `java.security.MessageDigest` class should be used to generate secure hash values (see example in the Java I/O API lesson).



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

Many existing Java classes such as `String`, `Number`, `LocalDate`, and so on override the `hashCode` method. You can use the following to generate `hashCode` values for your own objects:

```
public class Payment {
    private LocalDate date;
    private BigDecimal amount;
    /* other methods and variables */
    public int hashCode() {
        return date.hashCode() + amount.hashCode();
    }
}
```

Class `Objects` provides convenience operation `hash` to generate such hash code for any number of objects.

Compare String Objects

Class String represents an unusual case

- String objects are interned; therefore, multiple references can point to the same String object.
- This makes the `==` operator work with Strings in a way that is similar to the primitives.
- To avoid confusion, compare Strings like any other objects by using the `equals` method.
- In addition to the `equals` method, the String class also provides an `equalsIgnoreCase` method.
- The String class is final and cannot be extended,

```
String a = "Hello";
String b = "Hello";
String c = new String("Hello");
String d = "heLLo";
boolean sameObjectAB = (a == b);           //true
boolean sameObjectAC = (a == c);           //false
boolean sameContentAB = (a.equals(b));      //true
boolean sameContentCA = (a.equals(c));      //true
boolean sameContentAD = (a.equals(d));      //false
boolean sameNoCaseAD = (a.equalsIgnoreCase(d)); //true
```

Reminder:

- ❖ Instances of String can and should be created without explicitly invoking a String constructor.
- ❖ String is the only Java object that allows simplified instantiation as just a text value enclosed within double quotes: "some text" and that is a recommended approach.
- ❖ Strings are interned objects; a single copy of a String literal is stored in a String Pool memory area.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

Factory Method Pattern

Factory methods can hide the use of a specific constructor.

- Dynamically choose the subtype instance to be created.
- Analyze conditions and produce an instance of a specific subtype.
- Invokers can remain subtype unaware.
- Later addition of extra subtypes may not affect such invokers.

```
public class ProductFactory {  
    public static Product createProduct(...) {  
        switch(productType) {  
            case FOOD:  
                return new Food(...);  
            case DRINK:  
                return new Drink(...);  
        }  
    }  
}  
  
public abstract class Product {  
    public Product(...) { }  
}  
  
public class Food extends Product {  
    public Food(...) { }  
}  
  
public class Drink extends Product {  
    public Drink(...) { }  
}
```

- ❖ Example shows pseudo-code. the actual decision can be based on factory method parameters or other dynamically determined factors.

```
Product product = ProductFactory.createProduct(...);
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

24

Summary

In this lesson, you should have learned how to:

- Extend classes
- Reuse code through inheritance
- Use the `instanceof` operator
- Describe the `super` object reference
- Define subclass constructors
- Override superclass methods
- Explain polymorphism
- Define abstract classes and methods
- Define final classes and methods
- Override methods of the `Object` class



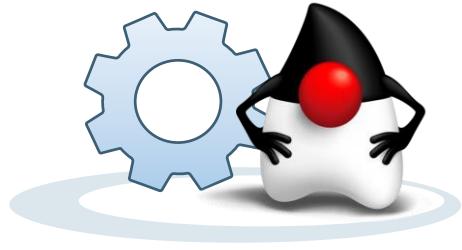
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

Practices

In this practice, you will:

- Create classes Food and Drink that extend class Product
- Make Product an abstract class
- Add subclass specific attributes and methods
- Override parent class methods
- Create class ProductManager to provide factory methods that create instances of Food and Drink



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

26

Interfaces



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Describe Java interfaces
- Implement an interface
- Describe nonabstract interface methods
- Explain generics
- Utilize some of the commonly used Java Interfaces
- Implement the Composition design pattern



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Java Interfaces

An interface defines a set of features that can be applied to various other classes.

- Instance methods are by default **public** and **abstract**.
- They can contain concrete methods **only** if they are either **default**, or **private**, or **static**.
- They can contain **constants**, but not variables.

```
public interface <InterfaceName> {  
    <constants>  
    <abstract methods>  
    <default methods>  
    <private methods>  
    <static methods>  
}  
  
public interface Perishable {  
    public static final Period MAX_PERIOD = Period.ofDays(5);  
    void perish();  
    boolean isPerished();  
    public default boolean verifyPeriod(Period p) {  
        return comparePeriod(p) < 0;  
    }  
    private int comparePeriod(Period p) {  
        return p.getDays() - MAX_PERIOD.getDays();  
    }  
    public static int getMaxPeriodDays() {  
        return MAX_PERIOD.getDays();  
    }  
}
```

❖ Note: Interface resembles an abstract class, except no variables or regular concrete methods are allowed.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

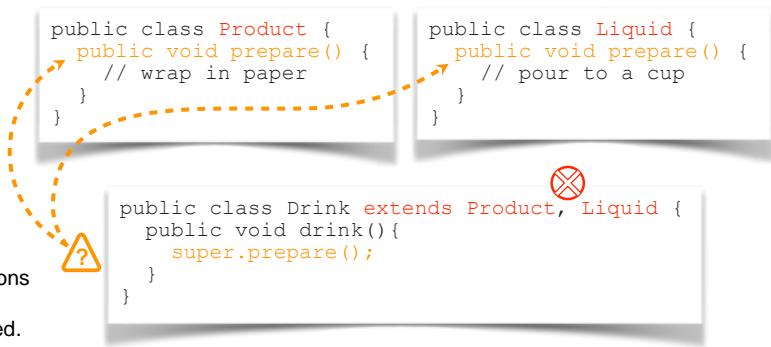
The default method is a special type of operation that allows provision of concrete implementation code within the interface; otherwise, public instance methods of an interface must all be abstract.

Multiple Inheritance Problem

Object-oriented inheritance is not very flexible.

- Depending on the point of view, a given class may require different parents (to be of more than one type).
- Different sets of features can be inherited through the extension of different types.
- Java class can only extend one immediate superclass. (**Multiple inheritance is not allowed in Java.**)
- Other object-oriented languages that do allow multiple inheritance face conflicts between parent types, when two or more parents provide concrete operations with identical signatures or variables with identical names.

- ❖ Note: Accessing superclass operations or variables could be ambiguous, if multiple inheritance would be allowed.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

There could be a requirement for the Drink class to be both a child of Product and a child of Liquid. Implementing such a requirement with multiple inheritance could result in a conflict between parent types, if they provide operations with identical signatures, but different implementations. An attempt to access such operations from the perspective of the child would be ambiguous.

Implement Interfaces

Interfaces solve multiple inheritance problem:

- Class can implement as many interfaces as needed (regular concrete methods are not allowed in interfaces).
- A concrete class must provide a **concrete method implementation** for each **abstract method signature** declared by interfaces that it implements.
- Default method can only be defined in an interface.
- A class must override **default interface method** only if it conflicts with another default method (implementing different interfaces that define default methods with the same signature).

```
public interface Consumable {
    int measure();
    void consume(int quantity);
}

public interface Liquid {
    public default void prepare() {
        // pour to a cup
    }
    int measure();
}

public class Drink extends Product
    implements Consumable, Liquid {
    public void consume(int quantity) {}
    public int measure() { return 0; }
}
```

✖ Reminder: Narrowing access to the concrete method, which implements an abstract method, is not allowed.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Interfaces provide solution for the multiple inheritance problem. There is no conflict between parent types due to the absence of regular concrete methods or variables in interfaces. Therefore, a given class can implement as many interfaces as required.

A class must provide only one implementation of a method, including the case when two interfaces define methods with identical signatures.

An abstract class does not have to provide implementations for all abstract methods. However, eventually such an abstract class is going to be extended by some concrete child class, which would have to provide implementations for all abstract methods that it has inherited via this abstract class and any interfaces it implements.

Default, Private, and Static Methods in Interfaces

Concrete code can be present in the interface only within the default, private, or static method.

- Private interface methods do not cause conflicts, because they are not visible outside of that interface.
- Static interface methods do not cause conflicts, because they are invoked via specific parent types and do not rely on the super reference.
- If there is a conflict between default methods, it must be resolved by overriding this default method within the implementation class.
- Otherwise, the default method implementation can be inherited.



```
public class W {
    public void e() {}
}

public interface X {
    void a();
    public default void b() {}
    private void c() {}
    public static void d() {}
}

public interface Y {
    void e();
    public default void b() {}
    private void c() {}
    public static void d() {}
}
```

```
Z z = new Z();
z.a();
z.b();
z.e(); ✓
X.d(); ✓
Y.d(); ✓
Z.d(); ✗
```

`public class Z extends W implements X, Y {`

```
* public void b() {}
* public void a() {}
}
```

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

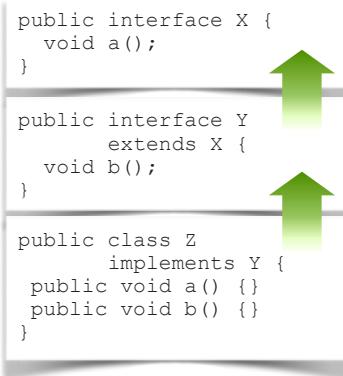
6

A class can inherit a concrete method that has a signature identical to the signature of a method defined by an interface that this class implements. In this case, the class does not have to override this method, as concrete implementation of it is already available for any given instance of this class.

Interface Hierarchy

An interface can extend another interface.

- Interfaces can form hierarchy in the same way as classes.
- A concrete class must override all abstract methods that it has inherited, regardless of where they were defined.



Interface Is a Type

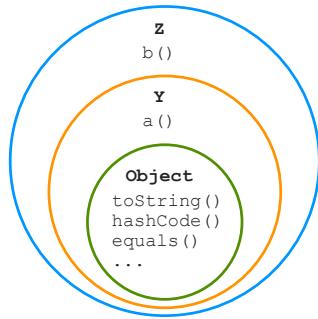
An interface is a type just like a class.

- Interface is a valid reference type, can be used in type casting, and works with the `instanceof` operator.
- The reference declared as a specific class type enables access to all publicly visible capabilities of this class, including methods defined by its parent classes, and interfaces it implements.
- The reference declared as interface type enables access only to those operations that are described by this interface and its parents.
- Object class is the ultimate parent type in Java. That is why methods of the Object class are accessible via any type of reference.

```
public interface Y {
    void a();
}

public class Z
    implements Y {
    public void a() {}
    public void b() {}
}

Z z = new Z();
z.toString();
z.a();
z.b();
if (z instanceof Y) {
    Y y = (Y)z;
    y.toString();
    y.a();
    y.b(); ✘
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Functional Interfaces

Functional Interface is an interface that defines a single abstract operation (function).

Problem:

- It is not possible to partially implement an interface. The concrete class must implement all abstract methods that it has inherited.
- An interface with many abstract methods is not convenient to use. A class may have to provide implementations for methods that it does not actually need.

Solution:

- A class may implement as many interfaces as needed.
- An interface can define just one abstract method; no extra operations that could be "unwanted baggage" has to be implemented.
- It is a flexible and recommended approach to designing interfaces.

```
public interface X {
    void a();
    void b();
    void c();
}
```

```
public class Y
    implements X {
        public void a() {}
        public void b() {}
        public void c() {}
    }
```

```
public interface A {
    void a();
}
```

```
public interface B {
    void b();
}
```

```
public class Y
    implements A, B {
        public void a() {}
        public void b() {}
    }
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

An interface is considered to be functional if it defines just one abstract method. However, you can indicate that specific interface is functional using the `@FunctionalInterface` annotation. See the appendix titled "Annotations" for more details.

Generics

Generics is a feature of Java language available since Java SE 5.

- It allows variables and methods to operate on objects of various types while providing compile-time type safety.
- Versions prior to Java SE 5 (**no generics style**) wrap values within the class using the type Object.
- Post Java SE 5 (**generics style**) versions wrap values within the class, but defer the exact type identification.
- Generic type avoids hard-coding the exact type as part of the class design.

Without Generics	With Generics
<pre>public class Some { private Object value; public Object getValue() { return value; } public void setValue(Object value) { this.value = value; } }</pre>	<pre>public class Some<T> { private T value; public T getValue() { return value; } public void setValue(T value) { this.value = value; } }</pre>

❖ Note: In the example, T is not a keyword, or class or interface name, but a generic type marker.
Other markers can be used: T (type), V (value), K (key), and any other marker you like, which could be a word or even a single letter.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Generics cannot be applied to:

- Descendants of class Throwable
- Anonymous classes
- Enumerations

Use Generics

The use of Generics helps to produce compact, type-safe code.

- Without generics:
 - Any type can be assigned to a variable or parameter whose type is Object
 - Programmatic type-check using the `instanceof` operator is required to ensure that you don't accidentally cast variable to the wrong type
- With generics:
 - Compiler checks that the type that is assigned, or passed as parameter corresponds to the generic type declaration, rejecting code that attempts to use types that don't match
 - No programmatic type-check or type-casting is required

Without Generics

```
Some some = new Some();
some.setValue(new Product("Tea", 1.99));
some.setValue("something");
Object value = some.getValue();
if (value instanceof Product) {
    Product product = (Product)value;
}
if (value instanceof String) {
    String text = (String)value;
}
```



With Generics

```
Some<Product> some = new Some<Product>();
some.setValue(new Product("Tea", 1.99));
some.setValue("something");
Product product = some.getValue();
```

- ✖ Note: Generics can be used with both classes and interfaces. Many existing Java interfaces utilize generics.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Ways of declaring initializing objects that utilize generics:

- This example works without any warnings; object is instantiated using the matching generic expression:

```
Some<Product> some3 = new Some<Product>();
```

- 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 (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond. This example works without any warnings, because compiler assumes that the instantiated object should use generics as declared on a variable to which the reference is assigned:

```
Some<Product> some2 = new Some<>();
```

- This example works, but produces a compiler warning that your class is using unchecked or unsafe operations, because the object is instantiated without specifying generic type that you use on a variable to which it is assigned:

```
Some<Product> some1 = new Some();
```

- This example would compile, but no information about the use of generics is going to be available via such reference. Therefore, it is no better than not using generics at all:

```
Some some4 = new Some<Product>();
```

- Code that does not use generics is sometimes described as using the "raw type" and essentially reverts to using Object as type:

```
Some some5 = new Some();
```

All examples above use the following class:

```
public class Some<T> {  
    private T value;  
    public T getValue() {  
        return value;  
    }  
    public void setValue(T value) {  
        this.value = value;  
    }  
}
```

Examples of Java Interfaces: `java.lang.Comparable`

Comparable interface describes a way of comparing current object (`this`) to another object.

- Defines single abstract method `int compareTo(T o)`
- Compares current object (`this`) with the specified object (parameter) to establish their order
- The `compareTo` method returns an `int` value.

current object	less than	equal to	greater than	parameter object
	negative	zero	positive	

```
public class Product
    implements Comparable<Product> {
    public int compareTo(Product p) {
        return this.name.compareTo(p.name);
    }
    // other variable and methods
}
```

```
Product[] products = {new Product("Tea"),
                      new Product("Coffee"),
                      new Product("Cake")};
Arrays.sort(products);
```

```
package java.lang;
public interface Comparable<T> {
    int compareTo(T o);
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

The example in the slide uses arrays, which are covered in the lesson titled "Arrays and Loops."

Existing Java classes, such as `String`, `LocalDateTime`, `BigDecimal`, and many others, already implement comparable interface, giving you a convenient natural way of sorting objects of different types.

Examples of Java Interfaces: `java.util.Comparator`

Comparator interface describes a way of comparing a pair of objects.

- Defines a single abstract method `int compare(T o1, T o2)`
- Compares one object with another to establish their order by returning an `int` value from the `compare` method

first object	less than	equal to	greater than	second object
	negative	zero	positive	

```
public class ProductNameSorter
    implements Comparator<Product> {
    public int compare(Product p1, Product p2) {
        return p1.getName().compareTo(p2.getName());
    }
}

Product[] products = {new Product("Tea"),
                     new Product("Coffee"),
                     new Product("Cake")};
Arrays.sort(products, new ProductNameSorter());
```

```
package java.lang;
public interface Comparator<T>{
    int compare(T o1, T o2);
}
```

```
public class Product {
    // variables and methods
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

The example in the slide uses arrays.

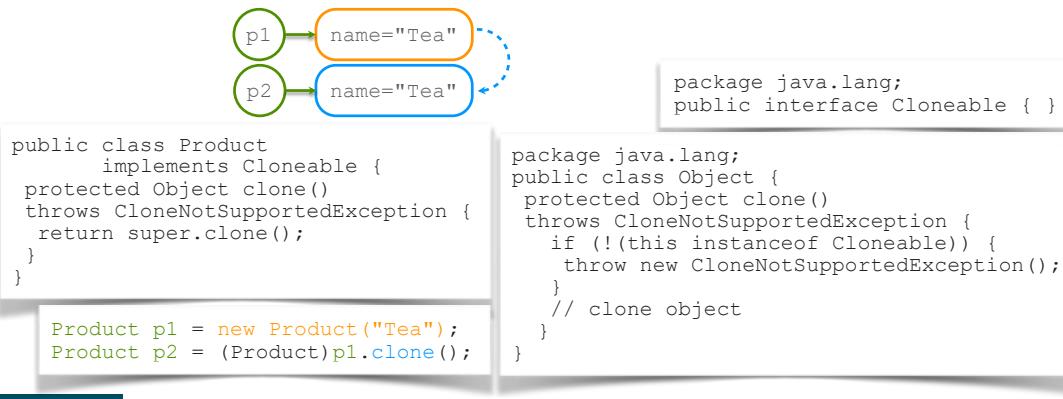
Comparable interface is implemented as a part of the class, comparing the current object to the parameter. The main advantage of using the Comparator interface is that it is implemented as a separate class. You can have as many implementations of it as required, thereby providing different types of comparison for the same object, for example, comparing products by name, or by date, or by price.

`java.util.Comparator` is a functional interface. It defines a single abstract method (function) that must be overridden when implementing this interface.

Examples of Java Interfaces: `java.lang.Cloneable`

`Cloneable` is an example of an interface used as a "type-marker" or "tag-interface."

- The interface does not have to define any methods.
- It can still be used with the `instanceof` operator to validate the object type.
- Cloning an object means creating a replica of the objects memory.
- The `java.lang.Cloneable` interface indicates a permission that an object can be cloned.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

There are many other type-marker interfaces available in Java, for example, `Serializable`, `Cloneable`, `Remote`, and so on.

Composition Pattern

A Class may represent a composition of features implemented by different other classes.

- Interfaces describe capabilities.
- Classes implement these capabilities.
- Capabilities are aggregated.

```
public class Bank
    implements Withdrawing,
               Depositing,
               Authentication {
    private Account a;
    private Security s;
    public BigDecimal withdraw() {
        authenticate();
        return a.withdraw();
    }
    public void deposit(BigDecimal amount) {
        authenticate();
        a.deposit(amount);
    }
    public void authenticate() {
        s.authenticate();
    }
}
```

```
public interface Withdrawing {
    BigDecimal withdraw();
}
```

```
public interface Depositing {
    void deposit(BigDecimal amount);
}
```

```
public interface Authentication {
    void authenticate();
}
```

```
public class Account
    implements Withdrawing,
               Depositing {
    public BigDecimal withdraw() { }
    public void deposit(BigDecimal amount) { }
}
```

```
public class Security
    implements Authentication {
    public void authenticate() { }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

The composition pattern is generally considered to be a better, more flexible design compared to the inheritance. It allows you to combine features from different classes together, without having to consider any restrictions imposed by the class hierarchy.

Using composition could also turn out to be much safer than inheritance, as your code is only relying upon stability of interface definitions, but not on actual inherited code, which, if changed, may lead to functional, security, and compatibility issues.

Summary

In this lesson, you should have learned how to:

- Describe Java interfaces
- Implement an interface
- Describe nonabstract interface methods
- Explain generics
- Implement the Composition design pattern



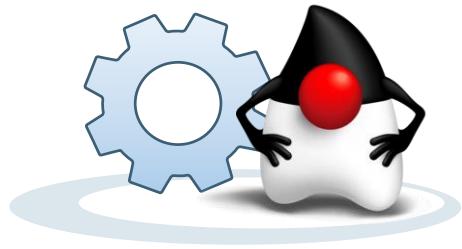
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Practices

In this practice, you will:

- Design Rateable interface to enable the rating of various objects.
- Design Review class to contain information about Rating and review comments.
- Add logic to the ProductManager class to handle product reviews, format and print reports.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

Arrays and Loops



ORACLE®

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Declare, initialize, and access arrays of Object and primitive types
- Use the `while`, `do/while`, `for`, and `forEach` loops
- Process arrays using the loop
- Use multidimensional arrays
- Use embedded loops
- Use break and continue operators



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Arrays

An array is a fixed-length collection of elements of the same type indexed by int.

- **Declare array object** - Determine the type of elements to be stored in the array.
- **Create array object** - Determine the length (number of elements) in the array.
 - Once an array is created, its length cannot be changed.
 - An array of object references is filled with null values.
 - An array of primitive values is filled with 0 values (false values if it is of boolean type).
- **Initialize array content** - Assign values to array positions in any order.
 - Index starts at 0.
 - Last valid index position is array.length-1.
- **Access elements** in the array using index.
 - No need to "extract" array element to operate on it.

✖ Access outside of the valid index boundaries produces `ArrayIndexOutOfBoundsException`.

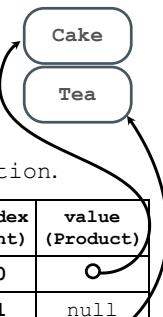
```
int[] primes;
// int primes[];
primes = new int[4];
primes[1] = 3;
primes[2] = 7;
primes[0] = primes[1]-1;
primes[4];
```



index (int)	value (int)
0	2
1	3
2	7
3	0

```
Product[] products;
// Product products[];
products = new Product[3];
products[0] = new Food("Cake");
products[2] = new Drink("Tea");
products[2].setPrice(1.99);
products[3];
```

index (int)	value (Product)
0	○
1	null
2	○



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

There are two alternative styles of declaring array object:

<type>[] <name> or <type> <name>[]

There is no practical difference between these alternatives.

Combined Declaration, Creation, and Initialization of Arrays

Array objects can be declared, created, and initialized at the same time.

- Combine declaration and creation of the array object:

```
int[] primes = new int[3];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
```

```
Product[] products = new Product[3];
products[0] = new Food("Cake");
products[1] = new Drink("Tea");
products[2] = new Food("Cookie");
```

- Combine creation of the array object and initialization of the array content:

```
int[] primes;
primes = new int[]{2,3,5};
```

```
Product[] products;
products = new Product[]{new Food("Cake"),
                        new Drink("Tea"),
                        new Food("Cookie")};
```

- Combine declaration and creation of the array object as well as the initialization of the array content:

```
int[] primes = {2,3,5};
```

```
Product[] products = {new Food("Cake"),
                      new Drink("Tea"),
                      new Food("Cookie")};
```

✿ Note: Array content can be provided as a comma-separated list, wrapped up in a block of code { , , , }



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

Multidimensional Arrays

- Java arrays can have multiple dimensions:
- Can use normal or short-hand initializations
 - Accessed by indicating each dimension index
 - Can be of nonsquare shapes
 - Can have more than two dimensions

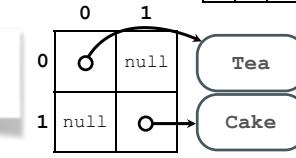
```
int[][] matrix = new int[2][3];
matrix[0][1] = 5;
matrix[1][2] = 7;
```

	0	1	2
0	0	5	0
1	0	0	7

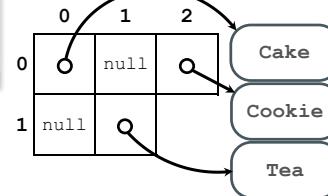
```
int[][] matrix = {{4,1},{2,0,5}};
```

	0	1	2
0	4	1	
1	2	0	5

```
Product[][] products =
matrix[1][1] = new Food("Cake");
matrix[0][0] = new Drink("Tea");
```



```
Product[][] products =
{{new Food("Cake"), null, new Food("Cookie")},
{null, new Drink("Tea")}};
```



Copying Array Content

Array content can be copied into another array.

- Java arrays are of fixed length (cannot be resized).
 - However, resize can be emulated by creating a new array with a different size and copying all or partial content from the original array to the new array object, using:
- ```
System.arraycopy(<source array>, <source position>,
 <destination array>, <destination position>,
 <length of content to copy from source>);
```
- or:
- ```
<new array> = Arrays.copyOf(<source array>, <new array length>);
<new array> = Arrays.copyOfRange(<source array>, <start position>, <end position>);
```

index int	value char
0	a
1	c
2	m
3	e

index int	value char
0	a
1	c
2	m
3	e
4	

```
char[] a1 = {'a','c','m','e'};
char[] a2 = {'t','o','',' ',' '};
System.arraycopy(a1, 2, a2, 3, 2);
```

```
char[] b1 = {'a','c','m','e'};
char[] b2 = Arrays.copyOf(b1,4);
```

index int	value char
0	a
1	c
2	m
3	e
4	

index int	value char
0	t
1	o
2	
3	m
4	e



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

The `Arrays.copy` and `Arrays.copyOfRange` methods are overloaded for each primitive datatype and also use generics.

The `Arrays.copyOfRange` method enables partial copying of content from the source array by providing the start position (inclusive) from which the content is taken from the original array. The end position parameter can be set to the value that is bigger than the source array length and thus allows creating a target array that is longer than the source.

Arrays Class

The `java.util.Arrays` class provides convenience methods for handling arrays, such as:

- Filling array with values
- Searching through the array
- Comparing content
- Sorting array content using:
 - `Comparable` (as implemented by objects within the array)
 - `Comparator` interfaces

```
String[] values = new String[5];
Arrays.fill(values, 2, 4, "aaa");
int x = Arrays.binarySearch(values, "aaa");
```

0	null
1	null
2	aaa
3	aaa
4	null

```
String[] names1 = {"Mary", "Ann", "Jane", "Tom"};
String[] names2 = {"Mary", "Ann", "John", "Tom"};
boolean isTheSame = Arrays.equals(names1, names2);
Arrays.sort(names2);
Arrays.sort(names2, new LengthCompare());
```

```
public class LengthCompare implements Comparator<String> {
    public int compare(String s1, String s2) {
        if (s1.length() > s2.length()) { return 1; }
        if (s1.length() < s2.length()) { return -1; }
        return 0;
    }
}
```

0	Mary
1	Ann
2	Jane
3	Tom

0	Mary	0	Ann	0	Ann
1	Ann	1	John	1	Tom
2	John	2	Mary	2	John
3	Tom	3	Tom	3	Mary



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Reminder: `Comparable` and `Comparator` interfaces were covered in the lesson titled “Interfaces.”

In the example in the slide, the array stores `String` values. `String` class already implements the `Comparable` interface overriding the `compareTo` method that performs lexicographical comparison. `Arrays.sort(<string array>)` will order array in alphabetical order.

Loops

Java loops contain the following parts:

- Declaration of the iterator
- Termination condition
- Iterator (typically increment or decrement)
- Loop body

Java loop constructs:

- while provides a simple iterator.
- do while guarantees to get through the loop body at least once.
- for keeps declaration, termination condition, and iterator together, on three positions separated with the ";" symbol.
- forEach provides a convenient way of iterating through the array or collection (see next page).

```
while (someMethod()) {
    // iterative logic
}
```

❖ Note: while or do/while loops are best used with a method call that returns boolean to control iterations.

```
int i = 0;
while (i < 10) {
    // iterative logic
    i++;
}
```

```
int i = 0;
do {
    // iterative logic
    i++;
} while (i < 10);
```

```
for (int i = 0; i < 10; i++) {
    // iterative logic
}
```

```
int i = 0;
for ( ; i < 10 ; ) {
    // iterative logic
    i++;
}
```

❖ Note: The example demonstrates the positional nature of the for loop construct (not a recommended way of writing a for loop.)



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

The benefit of the for loop is that it keeps all loop logic together within the same statement, making it easier to read code, especially when the loop body contains large amounts of complex logic. On the other hand, the while loop could be more convenient when loop progression and termination are controlled via a method call, eliminating the need to maintain your own iterator value:

```
while(someOperation()) {
    // loop body
}
```

The assumption in the example is that the someOperation() controls the iteration and returns boolean value to terminate the loop when required.

The positional nature of a for loop makes it possible to omit declaration or iterator parts, essentially "degrading" it to a while loop.

Processing Arrays by Using Loops

Processing **array** in a loop:

- Use `array.length` to determine the boundary for the **termination condition**.
- Access **array values** using an **index** and an **iterator**.
- Use the `forEach` loop to **auto-extract values**.

```
int[] values = {1,2,3};  
StringBuilder txt = new StringBuilder();  
for (int i = 0; i < values.length; i++) {  
    int value = values[i];  
    txt.append(value);  
}  
for (int value: values) {  
    txt.append(value);  
}
```

✿ Note: "foreach" is not an actual operator, or a reserved word, but simply a name that describes a `for` loop that iterates through the array or collection without a need to maintain an index and iterator logic.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

The example in the slide demonstrates the use of a `while` loop to iterate through the array. It is provided here just for completeness and comparison purposes, because it is more appropriate to use a `for` or a `forEach` loop construct to write such iterative logic.

The reason the `while` loop is not recommended to be used in such cases is because its iterative logic is scattered over several different places in code, making it harder to maintain, compared to a `for` or `forEach` loop, which keeps all such logic in the same place.

```
int[] values = {1,2,3};  
StringBuilder txt = new StringBuilder();  
int i = 0;  
while (i < values.length) {  
    int value = values[i];  
    txt.append(value);  
    i++;  
}
```

Complex for Loops

The positional nature of a `for` loop makes it possible to:

- Define multiple iterators separated by `" , "`
- Write a loop without a body if its `iterator` section also contains required `actions`

(However, mixing action and iteration logic can be confusing and thus is not a good coding practice.)

```
int[] values = {1,2,3,4,5,6,7,8,9};  
int sum = 0;  
for (int i = 0 ; i < values.length ; sum += i++) ;  
// sum is 36
```

```
int[][] matrix = {{1,2,3},{4,5,6},{7,8,9}};  
for (int i = 0, j = 2 ; !(i == 3 || j == -1) ; i++, j--) {  
    int value = matrix[i][j];  
}
```

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Syntactically, a `for` loop construct represents a very unusual case in Java language. It has positional structure, and it is using a `";"` symbol to separate positions and a `" , "` as a statement terminator in a multiple iterator scenario.

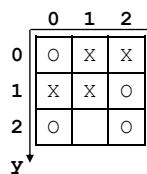
Embedded Loops

Loops can be placed inside other loops.

- Useful to process multidimensional arrays
- Can combine any loop types: while, do-while, for, and for-each

Result:

```
OXX
XXO
O O
OXX
XXO
O O
```



```
char[][] game = {{'O','X','X'},
                 {'X','X','O'},
                 {'O','','O'}};
StringBuilder txt = new StringBuilder();
for (int x = 0; x < game.length; x++) {
    int y = 0;
    while (y < game[x].length) {
        txt.append(game[x][y]);
        y++;
    }
    txt.append('\n');
}
for(char[] row : game) {
    for (char value: row) {
        txt.append(value);
    }
    txt.append('\n');
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Typical use case for the embedded loop is writing code to traverse master-detail data relations, for example, when you need to step through a number of orders and items for each one of these orders.

Break and Continue

Breaking out of loops and skipping loop cycles:

- `continue` operator skips the current loop cycle.
- `continue <label>` skips the labeled loop cycle.
- `break` terminates the current loop.
- `break <label>` terminates the labeled loop.

A	B	C	D	E
F	G	H	I	K
L	M	N	O	P
Q	R	S	T	U
V	W	X	Y	Z

Result:

```
ABDE
FGLM
QR
```

```
char[][] matrix = {{'A', 'B', 'C', 'D', 'E'},
                   {'F', 'G', 'H', 'I', 'K'},
                   {'L', 'M', 'N', 'O', 'P'},
                   {'Q', 'R', 'S', 'T', 'U'},
                   {'V', 'W', 'X', 'Y', 'Z'}};

StringBuilder txt = new StringBuilder();
outerLoopLabel:
for(char[] row : matrix) {
    for (char value: row) {
        if (value == 'C') { continue; }
        if (value == 'H') { continue outerLoopLabel; }
        if (value == 'N') { break; }
        if (value == 'S') { break outerLoopLabel; }
        txt.append(value);
    }
    txt.append('\n');
}
```



Summary

In this lesson, you should have learned how to:

- Declare, initialize, and access arrays of Object and primitive types
- Use the while, do/while, for, and forEach loops
- Process arrays using the loop
- Use multidimensional arrays
- Use embedded loops
- Use break and continue operators



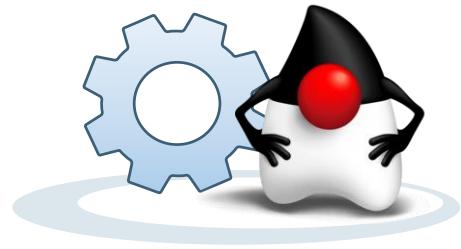
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

Practices

In this practice, you will:

- Add an array of Review objects to the ProductManager class.
- Compute Product Rating based on the average value of ratings in all reviews.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Collections



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Introduce Java Collection API interfaces and implementation classes
- Use List, Set, Deque, and Map collections
- Iterate through collection content
- Use Collections class
- Access collections concurrently



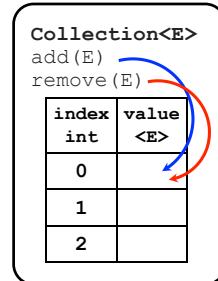
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Introduction to Java Collection API

Collection API presents a number of classes that manipulate groups of objects (collections).

- Collection classes may provide features such as:
 - use generics
 - dynamically expand as appropriate
 - provide alternative search and index capabilities
 - validate elements within the collection (for example, check uniqueness)
 - order elements
 - provide thread-safe operations (protect internal storage from corruption when it is accessed concurrently from multiple threads)
 - iterate through the collection
- Collection API defines a number of interfaces describing such capabilities.
- Collection API classes implement these interfaces and can provide different combinations of capabilities.



❖ Note: Collection API provides much better flexibility compared to just using arrays.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

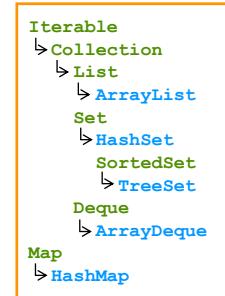
3

It is appropriate to compare collections to arrays, in a sense that either approach allows programs to store groups of objects in memory. In some cases, collection classes (`ArrayList` or `HashSet`) implement internal storage using arrays. However, in other collections such as `LinkedList`, this is not the case.

Java Collection API Interfaces and Implementation Classes

Java Collection API provides automation and convenience wrappers for array handling.

- Located in `java.util` package
- **Collection API interfaces:**
 - `Iterable<T>` is a top-level interface that allows any collection to be used in a `forEach` loop.
 - `Collection<E>` interface extends `Iterable` and describes generic collection capabilities, such as adding and removing elements.
 - `List<E>` interface extends `Collection` and describes a list of elements indexed by `int`.
 - `Set<E>` interface extends `Collection` and describes a set of unique elements.
 - `SortedSet<E>` interface extends `Set` to add ordering ability.
 - `Deque<E>` interface extends `Collection` and describes a double-ended queue, providing First-In-First-Out (FIFO) and Last-In-First-Out (LIFO) behaviors.
 - `Map<K, V>` interface contains a `Set` of unique keys and a `List` of values.
- **Collection API classes:**
 - `ArrayList<E>` implements `List` interface.
 - `HashSet<E>` implements `Set` interface and `TreeSet<E>` implements `SortedSet`.
 - `ArrayDeque<E>` implements `Deque` interface.
 - `HashMap<K, V>` implements `Map` interface.



✿ Note: This page describes frequently used collection types - more variants of collections are available.



Create List Object

Class `java.util.ArrayList` is an implementation of the `List` interface.

- Instances of `ArrayList` can be created using:
 - No-arg constructor, creating a list of initial capacity of 10 elements
 - Constructor with specific initial capacity
 - Any other `Collection` (for example, `Set`) to populate this list with initial values
- A fixed-sized `List` can be created from the array using var-arg method `Arrays.asList(<T>...)`
- A read-only instance of `List` can be created using a var-arg method `List.of(<T> ...)`
- `ArrayList` will auto-expand its internal storage, when more elements are added to it.

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Set<Product> set1 = new HashSet<>();
set1.add(p1);
set1.add(p2);

List<Product> list1 = new ArrayList<>();
List<Product> list2 = new ArrayList<>(20);
List<Product> list3 = new ArrayList<>(set1);
List<Product> list4 = Arrays.asList(p2,p2);
List<Product> list5 = List.of(p1,p2);
```

- ✿ Note: `HashSet` is explained later.
- ✿ Reminder: var-arg parameter accepts a number of parameters or an array.



Manage List Contents

List represents collection of elements index by int

- Insert boolean add(T)
- boolean add(int, T)
- Update boolean set(int, T)
- Delete boolean remove(int)
- boolean remove(T)
- Check Existence boolean contains(T)
- Find an index int indexOf(T)

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
List<Product> menu = new ArrayList();
menu.add(p1); //insert first element
menu.add(p2); //insert next element
menu.add(2, null); //insert null
menu.add(3, p1); //insert element
menu.add(2, p1); //insert element
menu.set(2, p2); //update element
menu.remove(0); //remove element
menu.remove(p2); //remove element
boolean hasTea = menu.contains(p2);
int index = menu.indexOf(p1);
menu.get(index).setName("Cookie");
menu.add(4,p2); //throws exception X
```

index int	value <Product>
0	Cake
1	Tea
2	Cake
3	null
4	Cake

index int	value <Product>
0	Cake
1	Tea
2	Tea
3	null
4	Cake

index int	value <Product>
0	Tea
1	null
2	Cake
3	Cookie



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

Internal storage within the list will be expanded as required. However, each newly added element can be either added in between other elements, pushing all further elements forward, or be added at the end of the list to the length+1 position. Likewise, when an element is removed from the middle of the list, all further elements are pushed to the front of the list.

Elements cannot be added beyond length+1 position.

A list may contain null elements.

List operations return boolean values that act as an indicator of the operations success. For example, if you attempt to remove an element that does not exist in the list the remove method would return false value.

Create Set Object

Class `java.util.HashSet` is an implementation of the `Set` interface.

- Instances of `HashSet` can be created using:
 - No-arg constructor, creating a set of initial capacity of 16 elements
 - Constructor with specific initial capacity
 - Constructor with specific initial capacity and load factor (default is 0.75)
 - Any other `Collection` (for example, `List`) to populate this set with initial values
- `HashSet` will auto-expand its internal storage when more elements are added to it.
- A read-only instance of `Set` can be created using a var-arg method `Set.of(<T> ...)`

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
List<Product> list = new ArrayList<>();
list.add(p1);
list.add(p2);

Set<Product> productSet1 = new HashSet<>();
Set<Product> productSet2 = new HashSet<>(20);
Set<Product> productSet3 = new HashSet<>(20, 0.85);
Set<Product> productSet4 = new HashSet<>(list);
Set<Product> productSet5 = Set.of(p1,p2);
```

- The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.
- List allows duplicate elements while Set does not; when Set is populated based on List, duplicate entries are discarded.
- Reminder: var-arg parameter accepts a number of parameters or an array.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

Manage Set Contents

Set represents collection of unique elements.

- Insert boolean add(T)
- Delete boolean remove(T)
- Check Existence boolean contains(T)
- Duplicate element cannot be added to the set:
 - Methods `add` and `remove` verify if the element exists in the set using `equals` method.
 - Methods `add` and `remove` will return `false` value when attempting to add a duplicate or remove an absent element.

value <Product>
Cake
Tea
Cookie

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Product p3 = new Food("Cookie");
Set<Product> menu = new HashSet<>();
menu.add(p1); //insert element
menu.add(p2); //insert element
menu.add(p2); //insert nothing
menu.add(p3); //insert element
menu.remove(p1); //remove element
menu.remove(p1); //remove nothing
boolean hasTea = menu.contains(p2);
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

In this pages example, values in the Set are references to same three product objects.

Create Deque Object

Class `java.util.ArrayDeque` is an implementation of the `Deque` interface.

- Instances of `ArrayDeque` can be created using:
 - No-arg constructor, creating a set of initial capacity of 16 elements
 - Constructor with specific initial capacity
 - Any other `Collection` (for example, `List`) to populate this deque with initial values
- `ArrayDeque` will auto-expand as more elements are added to it.

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
List<Product> list = new ArrayList<>();
list.add(p1);
list.add(p2);

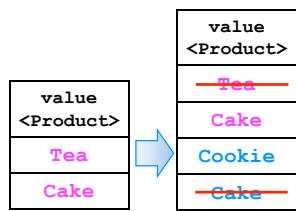
Deque<Product> productDeque1 = new ArrayDeque<>();
Deque<Product> productDeque2 = new ArrayDeque<>(20);
Deque<Product> productDeque3 = new ArrayDeque<>(list);
```



Manage Deque Contents

Deque represents collection that implements FIFO, LIFO behaviors.

- `offerFirst(T)` and `offerLast(T)` insert elements at the head and the tail of the deque.
- `T pollFirst()` and `T pollLast()` get and remove elements at the head and the tail of the deque.
- `T peekFirst()` and `T peekLast()` get elements at the head and the tail of the deque.
- Null values are not allowed in a deque.
- If deque is empty, poll and peek operations return null.



```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Product p3 = new Drink("Cookie");
Deque<Product> menu = new ArrayDeque<>();
Product nullProduct = menu.pollFirst();
menu.offerFirst(p1);
menu.offerFirst(p2);
Product tea = menu.pollFirst();
Product cake1 = menu.peekFirst();
menu.offerLast(p3);
menu.offerLast(p1);
Product cake2 = menu.pollLast();
Product cookie = menu.peekLast();
menu.offerLast(null); ❌
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

First In First Out (FIFO) and Last In First Out (LIFO) are typical deque handling mechanisms.

`boolean offerLast(E e)` inserts the specified element at the end of this deque.

`boolean offerFirst(E e)` inserts the specified element at the front of this deque.

`E pollFirst()` retrieves and removes the first element of this deque or returns null if this deque is empty.

`E removeFirst()` is an equivalent of the `pollFirst`, but it produces an exception instead of returning null when the deque is empty.

`E peekFirst()` retrieves, but does not remove, the first element of this deque or returns null if this deque is empty.

`E getFirst()` is an equivalent of the `peekFirst`, but it produces an exception instead of returning null when the deque is empty.

`E pollLast()` retrieves and removes the last element of this deque or returns null if this deque is empty.

`E removeLast()` is an equivalent of the `pollLast`, but it produces an exception instead of returning null when the deque is empty.

`E peekLast()` retrieves, but does not remove, the last element of this deque or returns null if this deque is empty.

`E getLast()` is an equivalent of the `peekLast`, but it produces an exception instead of returning null when the deque is empty.

Create HashMap Object

Class `java.util.HashMap` is an implementation of the `Map` interface.

Map is a composition of a Set of keys and a List of values.

- Instances of `HashMap` can be created using:
 - No-arg constructor, creating a set of initial capacity of 16 elements
 - Constructor with specific initial capacity
 - Constructor with specific initial capacity and load factor (default is 0.75)
 - Any other `Map` to populate this set with initial values
- `HashMap` will auto-expand its internal storage, when more elements are added to it.
- A read-only instance of `Map` can be created using
 - Method `of(<key>, <value>, ...)` overloaded for up to ten map entries
 - A var-arg method `ofEntries (Map.entry<key, value>... entries)`

```
Map<Product, Integer> items1 = new HashMap<>();  
Map<Product, Integer> items2 = new HashMap<>(20);  
Map<Product, Integer> items3 = new HashMap<>(items1);  
Map<Product, Integer> items4 = Map.of(new Food("Cake"), Integer.valueOf(2),  
                                         new Drink("Tea"), Integer.valueOf(3));
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

`Map.entry` is a wrapper for the key-value pairs. The example constructs `Map` object using such entries:

```
Map<Product, Integer> items = Map.ofEntries(Map.entry(new  
Food("Cake"), Integer.valueOf(2)),  
                                              Map.entry(new  
Drink("Tea"), Integer.valueOf(3)));
```

Manage HashMap Contents

Map represents collection of values with unique keys.

- `V put(K, V)` insert or update (when using an existing key) a key-value pair.
- `V remove(K)` delete a key-value pair.
- `V get(K)` return the value for a given key.
- `boolean containsKey(K)` check existence of a key.
- `boolean containsValue(V)` check existence of a value.

key <Product>	value <Integer>
Cake	2 \Rightarrow 5
Tea	2

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Map<Product, Integer> items = new HashMap<>();
items.put(p1, Integer.valueOf(2));
items.put(p2, Integer.valueOf(2));
Integer n1 = items.put(p1, Integer.valueOf(5));
Integer n2 = items.remove(p2);
boolean hasTea = items.containsKey(p2);
boolean hasTwo = items.containsValue(n1);
Integer quantity = items.get(p1);
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

You may also use `replace(<key>)` or `replace<key>, <value>` operations as an alternative to `put` operation to update existing entries.

Null key (just one) and null values are allowed in the HashMap.

Duplicate keys cannot be added to the map (uniqueness verified by the Set of keys):

- `put` returns old value when updating the map entry for existing key or `null` if inserting new value.
- `put` verifies if the element exists in the set using `equals` method.
- `remove` returns the old value or `null` if map entry with this key is not found.
- `get` returns the value for a key or `null` if map entry with this key is not found.

Iterate through Collections

Collections implement `Iterable` interface, allowing them to be used in a `forEach` loop.

- Any `List`, `Set`, or `Deque` can be directly used within a `forEach` loop.
- A more manual approach is to get an `Iterator` object from collection to step through the content.
- Get `Set of keys or List of values` from the `HashMap` to iterate through the content.
- Iterators also allow to `remove` content from the collection.

```
Map<Product, Integer> items = new HashMap<>();
Set<Product> keys = items.keySet();
Collection<Integer> values = items.values();
for (Product product : keys) {
    Integer quantity = items.get(product);
    // use product and quantity objects
}
for (Integer quantity : values) {
    // use quantity object
}
```

```
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake"));
menu.add(new Drink("Tea"));
for (Product product : menu) {
    // use product object
}

// less automated alternative:
Iterator<Product> iter = menu.iterator();
while (iter.hasNext()) {
    Product product = iter.next();
    // use product object
    iter.remove();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

Other Collection Behaviors

Some other examples of generic Collection behaviors include:

- Convert collection to an array using `toArray` method
 - Use array provided if collection fits into it
 - Otherwise create new array with the matching size
- Remove elements from collection based on a condition
 - Implement interface `Predicate<T>` overriding abstract method `boolean test(T);`
 - Use `removeIf(Predicate<T> p)` method to remove all matching elements from the collection

```
import java.util.function.Predicate;
public class LongProductsNames
    implements Predicate<Product> {
    public boolean test(Product product) {
        return product.getName().length() > 3;
    }
}
```

```
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake"));
menu.add(new Drink("Tea"));
menu.add(new Food("Cookie"));

Product[] array = new Product[2];
array = menu.toArray(array);

menu.removeIf(new LongProductsNames());
```

✿ In the example:

Product array is recreated to accommodate extra elements.

Method `removeIf` removes all products except tea.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

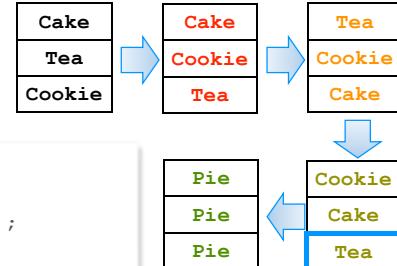
14

Use `java.util.Collections` Class

Class `java.util.Collections` provides convenience methods for handling collections.

- Filling collection with values
- Searching through the collection of Comparable objects or using Comparator
- Reordering collection content using:
 - Comparable (as implemented by objects within the array)
 - Comparator interface (see notes)
 - reverse method changes the order to opposite
 - shuffle method randomly reorders collection

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Product p3 = new Food("Cookie");
List<Product> menu = new ArrayList<>();
menu.add(p1);
menu.add(p2);
menu.add(p3);
Collections.sort(menu);
Collections.reverse(menu);
Collections.shuffle(menu);
Product x = Collections.binarySearch(menu, p2);
Collections.fill(menu, new Food("Pie"));
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

The example assumes that Product class implements Comparable interface and overrides compareTo method. Alternatively, an implementation of Comparator can be provided:

```
public class LengthCompare implements Comparator<Product>
{
    public int compare(Product p1, Product p2) {
        if (p1.getName().getLength() > p2.getName().length())
        { return 1; }

        if (p1.getName().getLength() < p2.getName().length())
        { return -1; }

        return 0;
    }
}

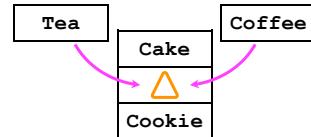
Collections.sort(menu, new LengthCompare());
```

Access Collections Concurrently

Collection can be corrupted if accessed concurrently from multiple threads.

- If two or more concurrent execution paths (**threads**) within your program try to access the collection at the same time, they can corrupt it, if this collection is not immutable.
- Any object in a heap is not thread-safe if it is not immutable. Any thread can be interrupted, even when it is modifying an object, making other threads observe incomplete modification state.
- Making collection thread-safe does not guarantee the thread safety of the objects it contains. Only immutable objects are automatically thread-safe.

```
List<Product> list = new ArrayList<>();
/*
    Attempt to insert, update or remove elements in the
    collection concurrently, may corrupt collection.
*/
```



❖ Details of how to write concurrent code are covered in a later lesson.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

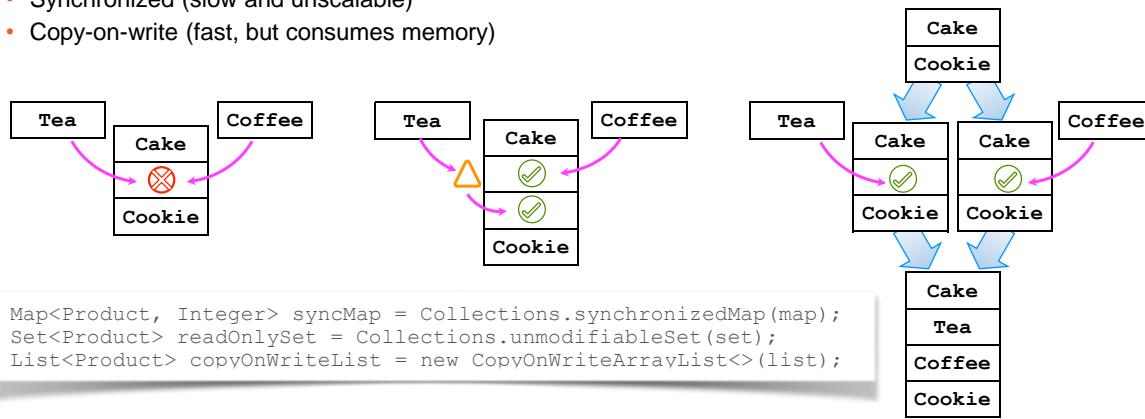
Making collection thread-safe is not the same as making objects inside this collection thread-safe as well:

- Collection itself can be modifiable yet contain immutable elements. Working with such elements is thread-safe, but adding and removing them from the collection is not.
- Collection itself can be unmodifiable yet contain mutable elements. Adding or removing elements in the collection is thread-safe, but working with the objects in it is not.

Prevent Collections Corruption

To prevent memory corruption in concurrently accessed collections, you can make collection:

- Unmodifiable (fast, but read-only)
- Synchronized (slow and unscalable)
- Copy-on-write (fast, but consumes memory)



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Read-only data is thread-safe - concurrent threads can read data at the same time, but they will not corrupt it because it is read only.

Unmodifiable collections can be created from regular collections using operations such as:

```
Collection<T> unmodifiableCollection(Collection<T>
collection)

List<T> unmodifiableList(List<T> list)

Set<T> unmodifiableSet(Set<T> set)

Map<K, V> unmodifiableMap(Map<K, V> map)
```

Synchronized collections allow only one thread to modify collection content at any given point in time and will block other threads, making them wait and take turns to access this collection.

This could cause a scalability problem, when you need to manage a significant number of concurrent calls that are trying to access same collection object and could wait for a long time for their turn.

Synchronized collections can be created from regular collections using operations such as:

```
Collection<T> synchronizedCollection(Collection<T>  
collection)  
  
List<T> synchronizedList(List<T> list)  
  
Set<T> synchronizedSet(Set<T> set)  
  
Map<K, V> synchronizedMap(Map<K, V> map)
```

Copy-on-write approach does not block concurrent callers, but instead creates a copy of a collection for each concurrent caller that attempts to modify its content. These copies are then merged. This approach could take more memory, but it is not blocking threads and thus is considered to be more scalable for a large number of concurrent callers.

Copy-on-write collections are available from `java.util.concurrent` package, for example:

```
CopyOnWriteArrayList<E>  
CopyOnWriteArraySet<E>
```

Legacy Collection Classes

You may still encounter earlier versions of Java Collection API classes.

- Legacy collections were all defined as synchronized, which is a performance issue.
- With new collections (covered in this lesson), you can choose the type of thread-safe implementation.
- Other behaviors of old collections are almost identical to the new classes.
- Examples of equivalent collection classes:
 - Class `ArrayList` is a new equivalent of the legacy class `Vector`.
 - Class `HashMap` is a new equivalent of the legacy class `Hashtable`.



Summary

In this lesson, you should have learned how to:

- Introduce Java Collection API interfaces and implementation classes
- Use List, Set, Deque, and Map collections
- Iterate through collection content
- Use Collections class
- Access collections concurrently



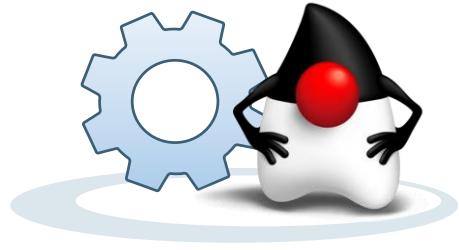
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

Practices

In this practice, you will:

- Use HashMap to store products and reviews
- Provide sorting mechanism for reviews and searching mechanism for products



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

21

Nested Classes and Lambda Expressions



ORACLE®

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



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

- Nested classes
 - Static
 - Member
 - Local
 - Anonymous
- Lambda expressions



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Types of Nested Classes

Classes can be defined inside other classes to encapsulate logic and constrain context of use.

- Type of the nested class depends on the context in which it is used.
 - Static nested class is associated with the static context of the outer class.
 - Member inner class is associated with the instance context of the outer class.
 - Local inner class is associated with the context of specific method.
 - Anonymous inner class is an inline implementation or extension of an interface or a class.
- Static and member nested classes can be defined as:
 - `public`, `protected`, or `default` - can be accessed externally
 - `private` - can be referenced only inside their outer class

```
public class Outer {
    public static class StaticNested {
        // code of the nested class
    }
}
Outer.StaticNested x = new Outer.StaticNested();
```

```
public class Outer {
    public static void createInstance() {
        new StaticNested();
    }
    private static class StaticNested {
        // code of the nested class
    }
}
Outer.createInstance();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

Nested classes implementation notes

- `this` reference in the context of the nested class points to the instance of itself and not the outer class.
- Outer class can access private variables and methods of any inner classes that it contains.

Nested classes access

Static nested classes are described as a distinct category (called nested, but not inner) because they could potentially be used as a top-level class. If the static class is not private, it can be accessed directly, just like any other top-level class, which could explain why this is considered to be a somewhat different case from any other inner class.

Note that top-level classes can only be described with either `public` or `default` access modifiers. Visibility of nested classes can be controlled more tightly using all access modifiers.

To create an instance of a member inner class, you have to create an instance of outer class first, even if the member inner class is defined as `public`.

Example of public member inner class (note that it could be accessed from outside the outer class):

```
public class Outer {  
    public MemberInner createMemberInner() {  
        return new MemberInner();  
    }  
    public class MemberInner {  
        // code of the member class  
    }  
}  
  
Outer x = new Outer();  
Outer.MemberInner y = x.createMemberInner();
```

Example of private member inner class (note that it is not visible outside the outer class):

```
public class Outer {  
    public void createMemberInner() {  
        new MemberInner();  
    }  
    private class MemberInner {  
        // code of the member class  
    }  
}  
  
Outer x = new Outer();  
x.createMemberInner();
```

Nested Classes' Design Considerations

Serialization of any nested classes is possible, but not recommended, as a probability of the serialized object across different implementations of Java runtimes is not guaranteed.

Nest class may "shadow" variables of the outer class, which is best to be avoided:

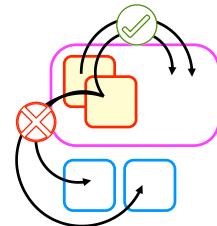
```
public class Outer {  
    private int x = 1;  
    public void outerMethod() {  
        int y = this.x; // y is 1  
    }  
    private class MemberInner {  
        private int x = 2;  
        public void outerMethod() {  
            int y = this.x; // y is 2  
        }  
    }  
}
```

Static Nested Classes

Static nested class is associated with the **static context of the outer class**.

- To create an **instance of a static nested class**, you do not need to create **instances of outer class**.
- Can access private variables and methods of the outer class
- Can only access static variables and methods of the outer class

```
public class Order {  
    public static void createShippingMode(String description) {  
        new ShippingMode(description);  
    }  
    private static class ShippingMode {  
        private String description;  
        public ShippingMode(String description) {  
            this.description = description;  
        }  
        // other methods and variables of the ShippingMode class  
    }  
}
```



```
Order.createShippingMode("Fast");  
Order.createShippingMode("Normal");  
Order order1 = new Order();  
Order order2 = new Order();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

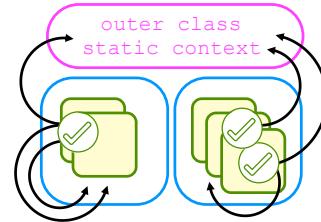
6

Member Inner Classes

Member inner class is associated with the instance context of the outer class.

- To create an **instance of a member inner class**, you must create **an instance of outer class** first.
- Can access private variables and methods of the outer class
- Can access both static and instance variables and methods of the outer class

```
public class Order {
    private Set<Item> items = new HashSet<>();
    public void addItem(Product product, int quantity) {
        items.add(new Item(product, quantity));
    }
    class Item {
        private Product product;
        private int quantity;
        private Item(Product product, int quantity) {
            this.product = product;
            this.quantity = quantity;
        }
        // other methods of the Item class
    }
}
```



```
Order order1 = new Order();
Order order2 = new Order();
order1.addItem(new Drink("Tea"), 2);
order1.addItem(new Food("Cake"), 1);
order2.addItem(new Drink("Tea"), 1);
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Outer class provides methods to operate on instances of the member inner class. Notice that in the code example, class Item has a default access modifier and therefore can be accessed from other classes in the same package, but its constructor is private; therefore, it can only be instantiated from the class Order.

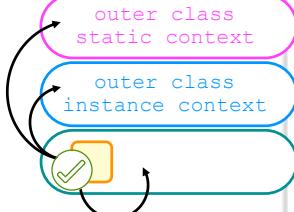
Code example of the member inner class `Item` and the outer containing class `Order` demonstrating the outer class managing the life cycle of the inner class:

```
public class Order {  
    private Map<Integer, Item> items = new HashMap();  
    public void addItem(Product product, int quantity) {  
        Item item = items.get(product.getId());  
        if (item != null) {  
            item.quantity += quantity;  
        } else {  
            items.put(product.getId(), new Item(product,  
                quantity));  
        }  
    }  
    public void removeItem(Product p) {  
        items.remove(p.getId());  
    }  
    public Item getItem(Product p) {  
        return items.get(p.getId());  
    }  
    // other methods of the Order class  
    class Item {  
        private Product product;  
        private int quantity;  
        private Item(Product product, int quantity) {  
            this.product = product;  
            this.quantity = quantity;  
        }  
        // other methods of the Item class  
    }  
}
```

Local Inner Classes

Local inner class is associated with the context of specific method.

- Instances of the local inner class can only be created within the outer method context.
- Contains logic complex enough to require the algorithms be wrapped up as a class
- Outer method local variables and parameters can only be accessed if they are final or effectively final.



```
public class Order {  
    private Map<Integer, Item> items = new HashMap<>();  
    public void manageTax(final String saleLocation) {  
        class OrderTaxManager {  
            private BigDecimal findRate(Product product) {  
                // use saleLocation and product to find the tax rate  
            }  
            BigDecimal calculateTax() {  
                // find tax rate in a given sale location for each product  
                // calculate tax value  
            }  
        }  
        OrderTaxManager taxManager = new OrderTaxManager();  
        BigDecimal taxTotal = taxManager.calculateTax();  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Anonymous Inner Classes

Anonymous inner class is an implementation of an interface or extension of a class.

- Extend a parent class or implement an interface to override operations
- Implemented inline and instantiated immediately
- Outer method local variables and parameters can only be accessed if they are final or effectively final.

```
public class Order {
    public BigDecimal getDiscount() {
        return BigDecimal.ZERO;
    }
}
```

Separate Class Implementation

```
public class OnlineOrder extends Order {
    @Override
    public BigDecimal getDiscount() {
        return BigDecimal.valueOf(0.1);
    }
}
```

Anonymous Inner Class Implementation

```
Order order = new Order() {
    @Override
    public BigDecimal getDiscount() {
        return BigDecimal.valueOf(0.1);
    }
};
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Anonymous inner class can be implemented inline. Its typical use case is to override methods of the class it extends or the interface that it implements:

```
processOrder(Order order) {
    order.getDiscount();
}

processOrder(new Order() {
    @Override
    public BigDecimal getDiscount() {
        return BigDecimal.valueOf(0.1);
    }
});
```

Anonymous classes also have the same restrictions as local classes with respect to their members:

- You cannot declare static initializers or member interfaces in an anonymous class.
- An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

- Fields
- Extra methods (even if they do not implement any methods of the supertype)
- Instance initializers
- Local classes

However, you cannot declare constructors in an anonymous class.

Anonymous inner class can invoke the constructor of the parent class (with parameters if necessary):

```
Product p = new Product("Cake") {  
    public String getName() {  
        return super.getName().toUpperCase();  
    }  
};  
String name = p.getName(); // name is in upper case
```

Anonymous Inner Classes and Functional Interfaces

Anonymous inner classes are typically used to provide inline interface implementations.

- Anonymous inner class can implement an interface inline and override as many methods as required.
- **Functional interfaces** define only one abstract method that must be overridden.
- Anonymous inner class that implements functional interface will only have to override one method.
- It could be more convenient to:
 - Use a regular class to override many methods
 - Use anonymous inner class to override a few methods (just one in case of a functional interface)

```
List<Product> products = ...  
Collections.sort(products, new Comparator<Product>() {  
    public int compare(Product p1, Product p2) {  
        return p1.getName().compareTo(p2.getName());  
    }  
});  
  
Collections.sort(products, new Comparator<Product>() {  
    public int compare(Product p1, Product p2) {  
        return p1.getPrice().compareTo(p2.getPrice());  
    }  
});
```



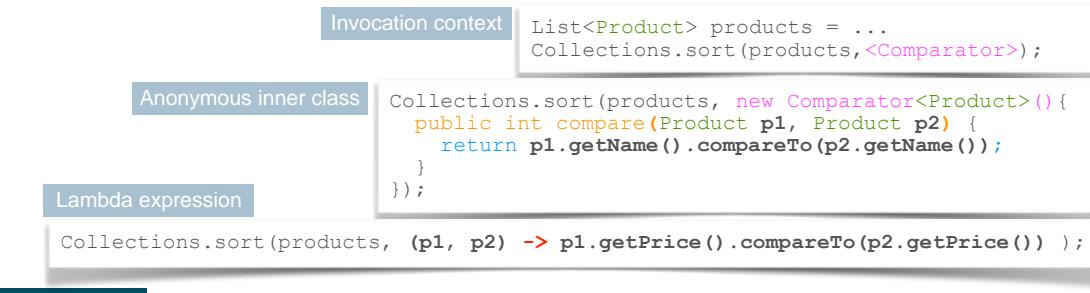
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Understand Lambda Expressions

Lambda expression is an inline implementation of a functional interface.

- Lambda expression infers its properties from the context:
 - Context of invocation infers which functional interface is implemented.
 - Functional interface's only abstract method infers the method that has to be overridden.
 - Generics infer which parameters the method should have.
 - Return type infers a return statement.
- Lambda expression may just contain parameter names and the desired algorithm.
- Lambda token `->` separates parameter list from the method body.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

You could think about lambda expressions as a much shorter way of writing anonymous inner classes that implement a functional interface.

Define Lambda Expression Parameters and Body

Parameter definitions of lambda expressions:

- To **apply modifiers** (annotations or keywords) to parameters, define them using:
 - Specific types
 - Locally inferred types
- When no modifier is required, you may just **infer types from the context**.
- Formal body { } and **return** statements are optional when using a simple expression.
- Round brackets for parameters are also optional.
- Expressions can be **predefined and reused**.

```
List<String> list = new ArrayList<>();
Comparator<String> sortText = (s1,s2) -> s1.compareTo(s2);
list.removeIf( final String s) -> s.equals("remove me") );
list.removeIf( final var s) -> s.equals("remove me") );
list.sort( s1,s2) -> { return s1.compareTo(s2); };
Collections.sort(list, sortText);
```

❖ Note: Annotations or modifiers can be used to impose restrictions upon parameters.
For example, making parameters `final` would prevent their reassignment within the expression.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

The example implemented using lambda expression is identical to this anonymous inner class implementation:

```
List<String> products = new ArrayList();
products.removeIf(new Predicate<String>() {
    public boolean test(String s) {
        return s.equals("remove me");
    }
});
```

Example uses: Collection class, method `removeIf(Predicate<T> p)` and Collections class, method `sort(collection, Comparator<T> c)`

Use Method References

Lambda expressions may use method referencing.

- Reference method is semantically identical to the method that lambda expression is implementing.
- `<Class>::<staticMethod>` - reference a static method
- `<object>::<instanceMethod>` - reference an instance method of a particular object
- `<Class>::<instanceMethod>` - reference an instance method of an arbitrary object of a particular type
- `<Class>::new` - reference a constructor (see notes)

```
public class TextFilter {
    public static boolean removeA(String s) {
        return s.equals("remove A");
    }
    public int sortText(String s1, String s2) {
        return s1.compareTo(s2);
    }
}
TextFilter filter = new TextFilter();
List<String> list = new ArrayList<>();
list.removeIf(s -> TextFilter.removeA(s));
list.removeIf(TextFilter::removeA); // same as the line above
Collections.sort(list, (s1,s2) -> filter.sortText(s1,s2));
Collections.sort(list, filter::sortText); // same as the line above
Collections.sort(list, (s1,s2) -> s1.compareToIgnoreCase(s2));
Collections.sort(list, String::compareToIgnoreCase); // same as the line above
```

✖ Class `String` has an instance method `compareToIgnoreCase` that implements not case-sensitive text sorting.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Using method references to invoke static methods or instance methods via specific object reference looks relatively simple.

The example of `String::compareToIgnoreCase` works because this lambda expression is translated to the specific `String` instance reference with the subsequent call to the instance `compareToIgnoreCase` method. This is probably not the best way of writing such expressions, because it is confusing to see the instance method referred to from a static context.

Example of using constructor referencing:

```
List<Product> menu = new ArrayList<>();
menu.add(null);
menu.add(null);

Collections.fill(menu, new Food()); // collection is
filled with new instances of Food

Collections.fill(menu, Food::new); // same as the line
above
```

Default and Static Methods in Functional Interfaces

Interfaces may provide additional non-abstract methods.

- Lambda expression implements the only **abstract method** provided by the functional interface.
- **default** and **static** methods may be defined by the interface to provide additional features.
- **private** methods could be present, but they are not visible outside of the interface.
- There is no requirement to override default methods unless there is a conflict between different interfaces that a given class implements.
- Lambda expressions cannot cause such conflicts, because each one is an inline implementation of exactly one interface.

```
public interface SomeInterface {  
    public static final int SOME_VALUE = 123;  
    void someAbstractMethod();  
    public default void someDefaultMethod() { }  
    private void somePrivateMethod() { }  
    public static void someStaticMethod() { }  
}
```



Use Default and Static Methods of the Comparator Interface

Examples of default methods provided by the `java.util.Comparator` interface:

- `thenComparing` adds additional comparators
- `reversed` reverses sorting order

Examples of static methods provided by the `Comparator` interface:

- `nullsFirst` and `nullsLast` return comparators that enable sorting collections with null values.

```
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake", BigDecimal.valueOf(1.99)));
menu.add(new Food("Cookie", BigDecimal.valueOf(2.99)));
menu.add(new Drink("Tea", BigDecimal.valueOf(2.99)));
menu.add(new Drink("Coffee", BigDecimal.valueOf(2.99)));
Comparator<Product> sortNames = (p1, p2) -> p1.getName().compareTo(p2.getName());
Comparator<Product> sortPrices = (p1, p2) -> p1.getPrice().compareTo(p2.getPrice());
Collections.sort(menu, sortNames.thenComparing(sortPrices).reversed());
menu.add(null);
Collections.sort(menu, Comparator.nullsFirst(sortNames));
```

✿ Note: Additional primitive variants of these methods are provided to avoid excessive boxing and unboxing.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Example of sorting the list of products by name and then by price and then reverse the order:

```
Tea      2.99
Cookie  2.99
Coffee   2.99
Cake     1.99
```

Static Comparator methods `nullsFirst` and `nullsLast` take a comparator object as an argument and return another comparator capable of handling null values in the collection.

Sorting the list with null elements can result in an `NullPointerException` unless `nullsFirst` or `nullsLast` comparators are used.

Result after adding null value to the list and using `nullsFirst` method:

```
null
Cake 1.99
Coffee 1.99
Cookie 2.99
Tea 1.99
```

Use Default and Static Methods of the Predicate Interface

Default methods provided by the `java.util.function.Predicate` interface:

- `and` combines `predicates` like the `&&` operator
- `or` combines `predicates` like the `||` operator
- `negate` returns a `predicate` that represents the logical negation of this predicate

Static methods provided by the `Predicate` interface:

- `not` returns a `predicate` that is the negation of the supplied `predicate`
- `isEqual` returns a `predicate` that compares the supplied object with the contents of the collection

```
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake", BigDecimal.valueOf(1.99)));
menu.add(new Food("Cookie", BigDecimal.valueOf(2.99)));
menu.add(new Drink("Tea", BigDecimal.valueOf(1.99)));
menu.add(new Drink("Coffee", BigDecimal.valueOf(1.99)));
Predicate<Product> foodFilter = (p) -> p instanceof Food;
Predicate<Product> priceFilter = (p) -> p.getPrice().compareTo(BigDecimal.valueOf(2)) < 0;
menu.removeIf(foodFilter.negate().or(priceFilter));
menu.removeIf(Predicate isEqual(new Food("Cake", BigDecimal.valueOf(1.99))));
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

The example filters out all drinks and all products cheaper than 2, leaving only Cookie 2.99. The `isEqual` method creates the `predicate` that filters a specific object.

Summary

In this lesson, you should have learned how to use:

- Nested classes
 - Static
 - Member
 - Local
 - Anonymous
- Lambda expressions



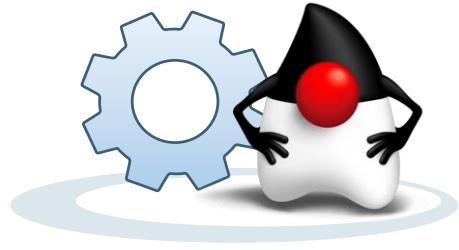
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Practices

In this practice, you will

- Create static nested helper class to encapsulate management of text resources and localization
- Provide Product sorting options with lambda expressions implementing Comparator interface



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

Java Streams API



ORACLE®



Copyright © 2020 Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

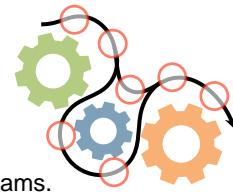
- Describe Java Streams API
- Process stream pipelines
- Implement functional interfaces using lambda expressions
- Understand parallel stream processing



Characteristics of Streams

Stream is an immutable flow of elements.

- Stream processing can be **sequential** (default) or **parallel**.
- Once an element is processed, it is no longer available from the stream.
- Stream pipeline traversal uses **method chaining** - intermediate operations return streams.
- Pipeline traversal is lazy.
 - Intermediate actions are deferred until stream is traversed by the terminal operation.
 - The chain of activities could be fused into a single pass on data.
 - Stream processing ends as soon as the result is determined; remaining stream data can be ignored.
- Stream operations use functional interfaces and can be implemented as **lambda expressions**.
- Stream may represent both finite and infinite flows of elements.



```
List<Product> list = new ArrayList();
    Stream:
list.stream() .parallel()
    .filter(p->p.getPrice()>10)
    .forEach(p->p.setDiscount(0.2));
```

```
Loop:
for (Product p: list) {
    if (p.getPrice()>10)) {
        p.setDiscount(0.2));
    }
}
```

✖ Examples assume that stream source is a collection of products.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

Processing of streams lazily allows for significant efficiencies; for example, if a pipeline needs to perform a chain of activities such as filtering, mapping, and summing, it can be fused into a single pass on the data, with minimal intermediate state. Laziness also allows avoiding examining all the data when it is not necessary; for operations such as "find the first string longer than 1000 characters", it is only necessary to examine just enough strings to find one that has the desired characteristics without examining all of the strings available from the source. (This behavior becomes even more important when the input stream is infinite and not merely large.)

Create Streams Using Stream API

Streams handling is described by the following interfaces:

- `BaseStream` - defines core stream behaviors, such as managing the stream in a parallel or sequential mode.
- `Stream`, `DoubleStream`, `IntStream`, `LongStream` interfaces extend the `BaseStream` and provide stream processing operations.
- Streams use generics.
- To avoid excessive boxing and unboxing, primitive stream variants are also provided.
- Stream can be obtained from any `collection` and `array` or by using `static methods` of the `Stream` class.
- Many other Java APIs can create and use streams (see notes).

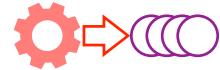
```
BaseStream
↳ Stream
↳ DoubleStream
↳ IntStream
↳ LongStream
```

```
IntStream.generate(() -> (int) (Math.random() * 10)).takeWhile(n -> n != 3).sum();

Stream.of(new Food(), new Drink()).forEach(p -> p.setPrice(1));

List<Product> list = new ArrayList();
Product[] array = {new Drink(), new Food()};

list.stream().parallel().mapToDouble(p -> p.getPrice()).sum();
Arrays.stream(array).filter(p -> p.getPrice() > 2).forEach(p -> p.setDiscount(0.1));
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

Examples of other Java APIs creating streams:

```
String s = "some text";
IntStream charCodes1 = s.chars();
IntStream charCodes2 = s.codePoints(); // handles unicode correctly
Stream<String> symbols = charCodes2.mapToObj(c ->
String.valueOf((char)c)); // convert int values from the stream into String objects
```

```
Random random = new Random();
DoubleStream randomNumbers = random.doubles(10);

Stream<String> textFileContent =
Files.lines(Paths.get("some.txt"));
```

Stream Pipeline Processing Operations

Stream handling operation categories:

- **Intermediate** - perform action and produce another stream
- **Terminal** - traverse stream pipeline and end the stream processing
- **Short-circuit** - produce finite result, even if presented with infinite input
- Use **functional interfaces** from `java.util.function` package
- Can be implemented using lambda expressions
- Basic function purposes:
 - `Predicate` performs tests
 - `Function` converts types
 - `UnaryOperator` (a variant of function) converts values
 - `Consumer` processes elements
 - `Supplier` produces elements

```
Stream.generate(<Supplier>)
    .filter(<Predicate>)
    .peek(<Consumer>)
    .map(<Function>/<UnaryOperator>)
    .forEach(<Consumer>);
```

Intermediate	Terminal
<code>filter</code>	<code>forEach</code>
<code>map</code>	<code>forEachOrdered</code>
<code>flatMap</code>	<code>count</code>
<code>peek</code>	<code>min</code>
<code>distinct</code>	<code>max</code>
<code>sorted</code>	<code>sum</code>
<code>dropWhile</code>	<code>average</code>
<code>skip</code>	<code>collect</code>
<code>limit</code>	<code>reduce</code>
<code>takeWhile</code>	<code>allMatch</code>
	<code>anyMatch</code>
	<code>noneMatch</code>
	<code>findAny</code>
	<code>findFirst</code>



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Operation `sum` and `average` are available only for primitive stream variants
`DoubleStream`, `IntStream`, `LongStream`.

Using Functional Interfaces

Stream operations use functional interfaces:

- Located in `java.util.function` package
- Can be implemented using lambda expressions
- Basic function shapes are:
 - `Predicate<T>` defines method `boolean test(T t)` to apply conditions to filter elements
 - `Function<T, R>` defines method `R apply(T t)` to convert types of elements
 - `UnaryOperator<T>` (variant of `Function`) defines method `T apply(T t)` to convert values
 - `Consumer<T>` defines method `void accept(T t)` to process elements
 - `Supplier<T>` defines method `T get()` to produce elements

❖ See notes for `Supplier` and `UnaryOperator` examples.

<code>Predicate</code>	<code>Function</code>	<code>UnaryOperator</code>	<code>Consumer</code>	<code>Supplier</code>
<code>Predicate<T></code>	<code>Function<T, R></code>	<code>UnaryOperator<T></code>	<code>Consumer<T></code>	<code>Supplier<T></code>

```
List<Product> list = new ArrayList();
list.stream()
    .filter(p -> p.getDiscount() == 0)
    .peek(p -> p.applyDiscount(0.1))
    .map(p -> p.getBestBefore())
    .forEach(p -> p.plusDays(1));
```

- ❖ Produce stream of products from the list
- ❖ Retain only products with no discount
- ❖ Apply discount of 10 percent
- ❖ Produce a stream of `LocalDate` objects
- ❖ Calculate the next day



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.



Example of Supplier and Consumer:

```
Supplier<String> textGenerator = () -> {
    Random random = new Random();
    StringBuilder txt = new StringBuilder(10);
    for (int i = 0; i < 10; i++) {
        txt.append((char)(random.nextInt(26) + 'a'));
    }
    return txt.toString();
}
Consumer<String> textPrinter = s ->
System.out.println(s);
UnaryOperator<String> textConverter = s ->
s.toUpperCase();
Stream.generate(textGenerator).limit(20).map(textConverter)
.forEach(textPrinter);
```

The above example generates a stream of random String objects of 10 characters each, limits the stream to 20 elements, converts values to upper case, and then prints them out.

Note: The example pre-creates `Supplier`, `UnaryOperator`, and `Consumer` objects, but they could have been written as inline lambda expressions.

Primitive Variants of Functional Interfaces

Improve stream pipeline handling performance by avoiding excessive auto-boxing-unboxing

	Predicate	Function	UnaryOperator	Consumer	Supplier
primitive output		ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>			IntSupplier LongSupplier DoubleSupplier
primitive input	IntPredicate LongPredicate DoublePredicate	IntFunction<R> LongFunction<R> DoubleFunction<R>		IntConsumer LongConsumer DoubleConsumer	
primitive input-output		IntToLongFunction IntToDoubleFunction LongToIntFunction LongtoDoubleFunction DoubleToIntFunction DoubleToLongFunction	IntUnaryOperator LongUnaryOperator DoubleUnaryOperator		

- ✖ The example maps strings to int values to filter and compute the result.
- ✖ See notes for more examples.

```
Stream.of("ONE", "TWO", "THREE", "FOUR")
    .mapToInt(s->s.length())
    .peek(i->System.out.println(i))
    .filter(i->i>3)
    .sum();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

The example demonstrates conversions from primitive to object and from object to primitive streams.

```
int x = DoubleStream.of(1.234, 1.0, 3.987, 0.321, 4.0) // create a stream of double primitive numbers
    .filter(n -> n != (int)n) // remove hole numbers using DoublePredicate
    .boxed() // convert primitive stream to object stream
    .map(n->BigDecimal.valueOf(n)) // convert Double to BigDecimal using Function
    .map(n->n.round(new MathContext(0, RoundingMode.HALF_UP))) // round BigDecimal value using UnaryOperator
    .mapToInt(n->n.intValue()) // convert object stream to int primitive stream using ToIntFunction
    .sum(); // compute sum of int values
```

The example creates an infinite stream of random numbers between 0 and 10, using `IntSupplier`. Stream processing is terminated when the supplier generates a value of 3, which is verified by the `IntPredicate`. This example also computes the sum of all numbers in the stream:

```
int x = IntStream.generate(() -> (int)(Math.random()*10))
// generate an infinite stream of random int values
between 0 and 10

        .takeWhile(n -> n !=3)
// terminate stream pipeline processing when value of 3
is generated

        .sum();
// compute the sum of int values
```

Class `Random` has various operations that generate a stream of random numbers. In the example, class `Random` uses `DoubleSupplier`, and `DoubleToIntFunction` is used in the mapping operation:

```
Random random = new Random();

int x = random.doubles(10)                                //
generate an stream of 10 random double values between 0
and 1

        .mapToInt(n->(int)(Math.round(n*100))) // 
produce stream of int values by multiplying each number
by 100 and rounding

        .sum();                                //
compute the sum of all int values in the stream
```

Bi-argument Variants of Functional Interfaces

Process more than one value at a time.

- Extra parameter is provided compared to basic function interfaces:
 - `BiPredicate<T, U>` defines method `boolean test(T t, U u)` to apply conditions.
 - `BiFunction<T, U, R>` defines method `R apply(T t, U u)` to convert two types into a single result.
 - `BinaryOperator<T>` (variant of `BiFunction`) defines method `T apply(T t1, T t2)` to convert two values.
 - `BiConsumer<T>` defines method `void accept(T t, U u)` to process a pair of elements.

	Predicate	Function	UnaryOperator	Consumer
primitive variants	<code>BiPredicate<T, U></code>	<code>BiFunction<T, U, R></code>	<code>BinaryOperator<T></code>	<code>BiConsumer<T, U></code>
		<code>ToIntBiFunction<T, U></code>	<code>IntBinaryOperator</code>	<code>ObjIntConsumer<T></code>
		<code>ToLongBiFunction<T, U></code>	<code>LongBinaryOperator</code>	<code>ObjLongConsumer<T></code>
		<code>ToDoubleBiFunction<T, U></code>		<code>ObjDoubleConsumer<T></code>

```
Product p1 = new Food("Cake", BigDecimal.valueOf(3.10));
Product p2 = new Drink("Tea", BigDecimal.valueOf(1.20));
Map<Product, Integer> items = new HashMap();
items.put(p1, Integer.valueOf(1));
items.put(p2, Integer.valueOf(3));
items.forEach((p, q) -> p.getPrice().multiply(BigDecimal.valueOf(q.intValue())));
```

❖ The example is processing product prices and quantities to produce totals.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Perform Actions with Stream Pipeline Elements

Intermediate or terminal actions are handled by `peek` and `forEach` operations.

- Operations `peek`, `forEach`, and `forEachOrdered` accept `Consumer<T>` interface.
- Lambda expression must implement abstract `void accept(T t)` method.
- Default `andThen` method provided by the `Consumer` interface combines consumers together.
- Difference between `forEachOrdered` and `forEach` is that the `forEach` operation does not guarantee respecting the order of elements, which is actually beneficial for parallel stream processing.



```
Consumer<Product> expireProduct = (p) -> p.setBestBefore(LocalDate.now());
Consumer<Product> discountProduct = (p) -> p.setDiscount(BigDecimal.valueOf(0.1));

list.stream().forEach(expireProduct.andThen(discountProduct));

list.stream().peek(expireProduct)
    .filter(p.getPrice().compareTo(BigDecimal.valueOf(10)) > 0)
    .forEach(discountProduct);
```

- ❖ Lambda expressions can be implemented inline.
- ❖ Consumer actions can be combined or applied at different points within the pipeline.
- ❖ Operation `filter` accepts a `Predicate` (details are covered next).



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

In the first example above, the current date is set as best before date for all products in the stream and then all products are discounted.

In the second example, the current date is set as best before date for all products, but discount is only applied to those which has a price greater than 10.

The first example could have been written as a single `Consumer` defined as an inline Lambda expression::

```
list.stream().forEach(p -> {
    p.setBestBefore(LocalDate.now());

    p.setDiscount(BigDecimal.valueOf(0.1)); }) ;
```

The reason behind having multiple `peek` handlers is to be able to use other intermediate operations, such as `map` or `filter` between such `peek` operations.

Perform Filtering of Stream Pipeline Elements

Filtering of the pipeline content is performed by the `filter` operation.

- Method `filter` accepts `Predicate<T>` interface and returns a stream comprising only elements that satisfy the filter criteria.
- Lambda expression must implement abstract `boolean test (T t)` method.
- Default methods provided by the `Predicate` interface:
 - `and` combines predicates like the `&&` operator.
 - `or` combines predicates like the `||` operator.
 - `negate` returns a predicate that represents the logical negation of this predicate.
- Static methods provided by the `Predicate` interface:
 - `not` returns a predicate that is the negation of the supplied predicate.
 - `isEqual` returns a predicate that compares the supplied object with the contents of the collection.



```
Predicate<Product> foodFilter = p -> p instanceof Food;
Predicate<Product> priceFilter = p -> p.getPrice().compareTo(BigDecimal.valueOf(2)) < 0;
list.stream().filter(foodFilter.negate().or(priceFilter))
    .forEach(p -> p.setDiscount(BigDecimal.valueOf(0.1)));
list.stream().filter(Predicate isEqual(new Food("Cake", BigDecimal.valueOf(1.99))))
    .forEach(p -> p.setDiscount(BigDecimal.valueOf(0.1)));
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

The first example could have been written as a single `Predicate` defined as an inline Lambda expression::

```
list.stream().filter((p) -> (! (p instanceof Food) || 
    (p.getPrice().compareTo(BigDecimal.valueOf(2)) < 0)))
    .forEach(p ->
    p.setDiscount(BigDecimal.valueOf(0.1));
```

The reason behind having multiple filters is to be able to use other intermediate operations, such as `map` or `peek` between these filters.

Perform Mapping of Stream Pipeline Elements

Map stream elements to a new stream of different content type using `map` operation.

- Method `map` accepts a `Function<T, R>` interface and returns a new stream comprising elements produced by this function based on the original stream content.
- Lambda expression must implement abstract `R apply(T t)` method.
- Default methods provided by the `Function`:
 - `andThen` and `compose` combine functions together.
- Static methods provided by the `Function` interface:
 - `identity` returns a function that always returns its input argument (equivalent of `t->t` function).
- Primitive variants of `map` are: `mapToInt`, `mapToLong`, and `mapToDouble`.
- Interface `UnaryOperator<T>` is a variant of a `Function` that maps values without changing the type.



```
Function<Product, String> nameMapper = p -> p.getName();
UnaryOperator<String> trimMapper = n -> n.trim();
ToIntFunction<String> lengthMapper = n -> n.length();
list.stream().map(nameMapper.andThen(trimMapper)).mapToInt(lengthMapper).sum();
```

- Difference between `andThen` and `compose` is the order in which functions are combined.
- In the example above, `nameMapper` is applied before `trimMapper`.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

The example could have been written as a single `Function` defined as an inline Lambda expression:

```
list.stream().mapToInt((p) ->
p.getName().trim().length()).sum();
```

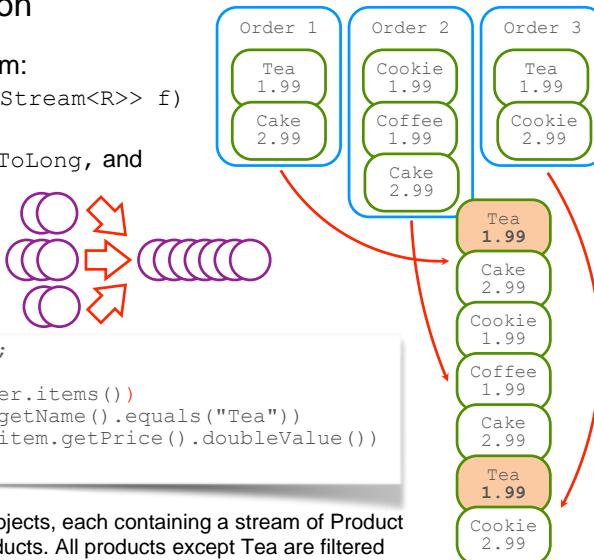
The reason behind having multiple maps is to be able to use other intermediate operations, such as `filter` or `peek` between these mappings.

Join Streams using flatMap Operation

Flatten a number of streams into a single stream:

- Operation Stream<R> `flatMap`(Function<T, Stream<R>> f) merges streams.
- Primitive variants are: `flatMapToInt`, `flatMapToLong`, and `flatMapToDouble` (see notes).

```
public class Order {
    private List<Product> items;
    public Stream<Product> items() {
        return items.stream();
    }
}
List<Order> orders = new ArrayList();
double x = orders.stream()
    .flatMap(order->order.items())
    .filter(item->item.getName().equals("Tea"))
    .mapToDouble(item->item.getPrice().doubleValue())
    .sum();
```



- ❖ The example processes a stream of Order objects, each containing a stream of Product objects, flattened into a single stream of Products. All products except Tea are filtered out, and a total sum of all Tea products sold in any of the orders is calculated.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

The example using `flatMapToDouble` operation (a primitive variant of the `flatMap`) calculates the sum of all products in all orders:

```
double x = orders.stream()
    .flatMapToDouble(order-
>order.items().mapToDouble(item-
>item.getPrice().doubleValue()))
    .sum();
```

Other Intermediate Stream Operations

Intermediate operations rearranging or reducing stream content:

- Operation `distinct()` returns a stream with no duplicates.
- Operations `sorted()` and `sorted(Comparator<T> t)` rearrange the order of elements.
- Operation `skip(long l)` skips a number of elements in the stream.
- Operation `takeWhile(Predicate p)` takes elements from the stream while they match the predicate.
- Operation `dropWhile(Predicate p)` removes elements from the stream while they match the predicate.
- Operation `limit(long l)` returns a stream of elements limited to given size.



```
Stream.of("A", "C", "B", "D", "B", "D")
    .distinct()
    .sorted()
    .skip(2)
    .forEach(s -> s.toLowerCase());
```



```
Stream.of("B", "C", "A", "E", "D", "F")
    .takeWhile(s->!s.equals("D"))
    .dropWhile(s->!s.equals("C"))
    .limit(2)
    .forEach(s -> s.toLowerCase());
```

- ❖ Operations `takeWhile` and `dropWhile` may produce different results if the stream is ordered.
- ❖ Their cost can be significantly increased for ordered and parallel streams.



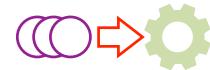
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Short-Circuit Terminal Operations

Short-circuit terminal operations produce finite result even if presented with infinite input.

- All short-circuit operations terminate stream pipeline processing as soon as result is computed.
- Operation `allMatch(Predicate p)` returns true if all elements in the stream match the predicate.
- Operation `anyMatch(Predicate p)` returns true if any elements in the stream match the predicate.
- Operation `noneMatch(Predicate p)` returns true if no elements in the stream match the predicate.
- Operation `findAny()` returns an element from the stream wrapped in `Optional` object.
- Operation `findFirst()` returns the first element from the stream wrapped in `Optional` object.



```
String[] values = {"RED", "GREEN", "BLUE"};
boolean allGreen = Arrays.stream(values).allMatch(s->s.equals("GREEN"));
boolean anyGreen = Arrays.stream(values).anyMatch(s->s.equals("GREEN"));
boolean noneGreen = Arrays.stream(values).noneMatch(s->s.equals("GREEN"));
Optional<String> anyColour = Arrays.stream(values).findAny();
Optional<String> firstColour = Arrays.stream(values).findFirst();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Process Stream Using count, min, max, sum, average Operations

Terminal operations calculate values from stream content.

- Method `count` returns a number of elements in the stream.
- Methods `sum` and `average` are available only for primitive stream variants (int, long, and double).
- Method `average` returns an `OptionalDouble` object (primitive variant for `Optional` class).
- Methods `min` and `max` return an `Optional` object wrapper for the minimum or maximum value from the stream, according to the `Comparator` supplied.



```
String[] values = {"RED", "GREEN", "BLUE"};
long v1 = Arrays.stream(values).filter(s->s.indexOf('R') != -1).count();           // 2
int v2 = Arrays.stream(values).mapToInt(v->v.length()).sum();                      // 12
OptionalDouble v3 = Arrays.stream(values).mapToInt(v->v.length()).average();
double avgValue = v3.isPresent() ? v3.getAsDouble() : 0;                           // 4
Optional<String> v4 = Arrays.stream(values).max((s1,s2) -> s1.compareTo(s2));
Optional<String> v5 = Arrays.stream(values).min((s1,s2) -> s1.compareTo(s2));
String maxValue = (v4.isPresent()) ? v4.get() : "no data";                         // RED
String minValue = (v5.isPresent()) ? v5.get() : "no data";                         // BLUE
```

❖ Notes on the use of Optional object:

Method `isPresent()` returns true if the value is present inside the `Optional` object.

Method `get()` retrieves the value.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

Implementations of `count` may not actually traverse the stream and return a number of elements in the source if that is faster to compute.

Finding `min` and `max` value is similar to taking the first or the last value after sorting.

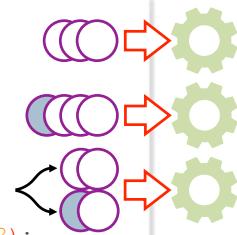
Aggregate Stream Data using `reduce` Operation

Produce a single result from the stream of values using `reduce` operation:

- `Optional<T> reduce(BinaryOperator<T> accumulator)` performs accumulation of elements.
- `T reduce(T identity, BinaryOperator<T> accumulator)` identity acts as initial (default) value.
- `<U> U reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)`
BiFunction performs both value mapping and accumulation of values.

`BinaryOperator` combines results produced by the BiFunction in parallel stream handling mode.

```
Optional<String> x1 = list.stream()
    .map(p->p.getName())
    .reduce((s1,s2)->s1+" "+s2);
    /*simple reduction*/
String x2 = list.stream()
    .map(p->p.getName())
    .reduce("",(s1,s2)->s1+" "+s2);
    /*reduction with initial(default) value*/
String x3 = list.stream()
    .parallel()
    .reduce("",(s,p)->p.getName()+" "+s, (s1,s2)->s1+s2);
    /*reduction with initial(default) value and a parallel combiner*/
```



❖ All examples perform stream reduction by concatenating product names into a single string.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Operations such as `min`, `max`, `count`, `average`, and `sum` can be considered as specific variants of what is generally a reduction - production of a single result from the stream of values. Operation `reduce` accumulates the result as it is processing the stream pipeline.

For example, the following three snippets of code produce the same result through different mechanics:

```
int x1 = IntStream.of(1, 2, 3, 4, 5).sum();
int x2 = IntStream.of(1, 2, 3, 4, 5).reduce(0,
Integer::sum);
int x3 = IntStream.of(1, 2, 3, 4, 5).reduce(0, (subtotal,
element) -> subtotal + element);
```

The example compares the use of `max` and `reduce` operations. Generally both examples are looking for the longest String in the stream. Once the maximum value is located, `max` function will not override it with another value that equals it. However, in the similar implementation provided by the `reduce` function, such a value will be overridden by the next matching `max` value. In this example, `reduce` function accepts `BinaryOperator<String>` as an argument.

```
Optional<String> x =
Stream.of("AAA", "B", "CCCC", "DD", "EEE", "FFFF", "GG", "HHHH")
        .reduce((s1,s2)-
>(s1.length()>s2.length())?s1:s2); // HHHH

Optional<String> x =
Stream.of("AAA", "B", "CCCC", "DD", "EEE", "FFFF", "GG", "HHHH")
        .max((s1,s2)-> s1.length() -
s2.length()); // CCCC
```

The example repeats the example of reduction with initial (default) value and parallel combiner, but also explicitly indicates all parameter types in lambda expressions. Notice that `BiFunction` accumulator accepts a `String` with an accumulated value and the `Product` as parameters and it is returning a `String`, and `BinaryOperator` combiner operates with the resulting type to which `BiFunction` has mapped `Products`, which is the `String`.

```
String x3 = list.stream()
        .reduce("",(String s, Product p)-
>p.getName()+" "+s,(String s1, String s2)->s1+" "+s2);
```

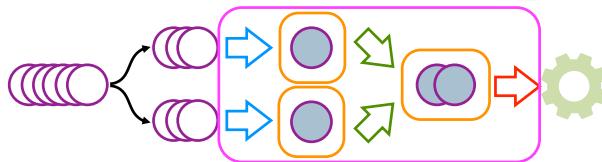
General Logic of the `collect` Operation

Perform a mutable reduction operation on the elements of the stream.

- Method `collect` accepts `Collector` interface implementation, which:
 - Produces new result containers using `Supplier`
 - Accumulates data elements into these result containers using `BiConsumer`
 - Combines result containers using `BinaryOperator`
 - Optionally performs a final transform of the processing result using the `Function`
- Class `Collectors` presents a number of predefined implementations of the `Collector` interface

```
stream.collect(<supplier>, <accumulator>, <combiner>)
stream.collect(<collector>)
stream.collect(Collectors.collectingAndThen(<collector>, <finisher>))
```

- Operations `collect` and `reduce` both perform reduction. However, `collect` operation accumulates results within intermediate containers, which may improve performance.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Method `collect` accepts either a `Collector` implementation (you may use existing implementations supplied by the `Collectors` class):

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

Or supplier, accumulator, and combiner provided separately (not wrapped into a `Collector` object):

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

Using Basic Collectors

Predefined implementations of the `Collector` interface supplied by `Collectors` class:

- Calculating summary values such as average, min, max, count, sum
- Mapping and joining stream elements
- Gathering stream elements into a collection such as list, set, or map

```
DoubleSummaryStatistics stats = 
    list.stream()
        .collect(Collectors.summarizingDouble(p->p.getPrice().doubleValue()));
```

```
String s1 = 
    list.stream()
        .collect(Collectors.mapping(p->p.getName(), Collectors.joining(", ")));
```

```
List<Product> drinks = 
    list.stream().filter(p->p instanceof Drink).collect(Collectors.toList());
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

Calculating and using summary statistics object:

```
DoubleSummaryStatistics stats =
list.stream().collect(Collectors.summarizingDouble(p-
>p.getPrice().doubleValue()));

long count = stats.getCount();
double avg = stats.getAverage();
double min = stats.getMin();
double max = stats.getMax();
double sum = stats.getSum();
```

Perform a Conversion of a Collector Result

Add finisher function to a collector to perform conversion of the collect result.

- Method `Collectors.collectingAndThen` appends a finishing `Function` to a `Collector`.



```
NumberFormat fmt = NumberFormat.getCurrencyInstance(Locale.UK);
String s2 = list.stream()
    .collect(Collectors.collectingAndThen(
        Collectors.averagingDouble(
            p->p.getPrice().doubleValue()),
        n->fmt.format(n)));
```

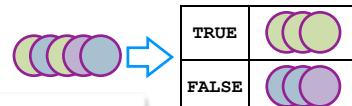


Perform Grouping or Partitioning of the Stream Content

Class Collectors provides Collector objects to subdivide stream elements into partitions or groups:

- **Partitioning** divides content into a map with two key values (boolean true/false) using `Predicate`.
- **Grouping** divides content into a map of multiple key values using `Function`.

```
Map<Boolean, List<Product>> productTypes =  
    list.stream()  
        .collect(Collectors.partitioningBy(p->p instanceof Drink));
```



```
Map<LocalDate, List<Product>> productGroups =  
    list.stream()  
        .collect(Collectors.groupingBy(p->p.getBestBefore()));
```

2019-03-07	Yellow circle
2019-03-14	Purple circle
2019-03-08	Blue circle



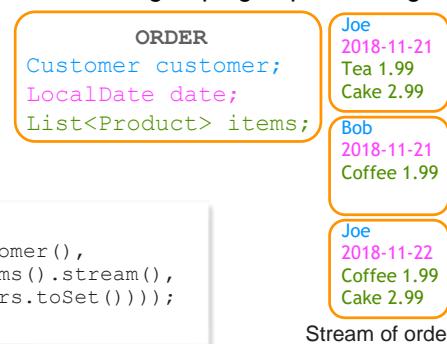
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

Mapping and Filtering with Respect to Groups or Partitions

Mapping and filtering in a multilevel reduction, using the downstream grouping or partitioning:

- flatMapping collector is applied to each input element in the stream before accumulation.
- filtering collector eliminates content from the stream without removing an entire group, if the group turns out to be empty.



```
Map<Customer, Set<Product>> customerProducts =
orders.collect(Collectors.groupingBy(o -> o.getCustomer(),
    Collectors.flatMapping(o -> o.getItems().stream(),
        Collectors.toSet())));
/*{Joe=[Tea, Coffee, Cake], Bob=[Coffee]}*/
```

```
Map<Customer, Set<Order>> customerOrdersOnDate =
orders.collect(Collectors.groupingBy(o -> o.getCustomer(),
    Collectors.filtering(o -> o.getDate().equals(LocalDate.of(2018,11,22))),
    Collectors.toSet()));
/*{Joe=[Order[date=2018-11-22, customer Joe, products=[Coffee, Cake]]], Bob=[]}*/
```

✖ See notes for comparing with flatMap and filter methods.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

This example demonstrates the effect of using mapping function on a stream compared to a flatMapping function:

```
Map<Customer, Set<Product>> customerItems =
orders.collect(Collectors.groupingBy(o ->
o.getCustomer(),
    Collectors.mapping(o -> o.getItems(),
        Collectors.toSet())));
{Joe=[[Tea, Cake], [Coffee, Cake]], Bob=[[Coffee]]}
```

Note that mapping function maps products per order, while the flatMapping function used at collect stage created a single set of products per customer.

This example demonstrates the effect of using `filter` function on a stream compared to a filtering function:

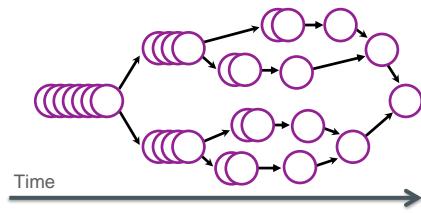
```
Map<Customer, Set<Order>> customerOrderOnDate =  
    orders.stream().filter(o ->  
        o.getDate().equals(LocalDate.of(2018,11,22)))  
            .collect(Collectors.groupingBy(o ->  
                o.getCustomer(),  
                Collectors.toSet()));  
  
{Joe=[Order[date=2018-11-22, customer Joe,  
products=[Coffee, Cake]]]}
```

Note that Bob is missing in this result, because he did not take any orders on a date specified in the filter condition. However, with the filtering performed at the collect stage, Bob will be included into the result, although without any orders.

Parallel Stream Processing

Parallel stream processing logic:

- Elements of the stream are subdivided into subsets.
- Subsets are processed in parallel.
- Subsets may be subdivided into further subsets.
- Processing order is stochastic (indeterminate).
- Subsets are then combined.
- Turn parallelism on or off using the `parallel` or `sequential` (default) methods.
- Entire stream processing will turn sequential or parallel depending on which method was invoked last.



```
list.stream().parallel().mapToDouble(p->p.getPrice().doubleValue()).sum();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

Parallel stream processing can be triggered using `parallelStream` operation available for any collection:

```
List<Product> list = new ArrayList();
list.parallelStream().mapToDouble(p-
>p.getPrice().doubleValue()).sum();
```

In this example, processing of the stream will be performed in parallel, since that was the last call:

```
list.stream().sequential().mapToDouble(p-
>p.getPrice().doubleValue()).parallel().sum();
```

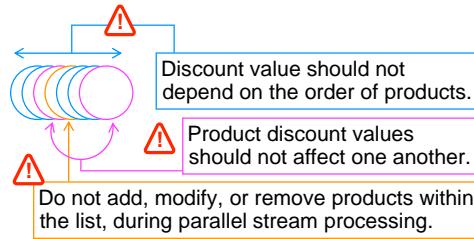
Term "Stochastic" describes the processing order of execution of which is randomly determined and cannot be predicted.

Parallel Stream Processing Guidelines

Parallel stream processing should observe the following guidelines:

- **Stateless** (state of one element must not affect another element)
- **Non-interfering** (data source must not be affected)
- **Associative** (result must not be affected by the order of operands)

```
List<Product> list = new ArrayList();
// populate list before processing starts
Double discount = list.stream().parallel()
    .mapToDouble(p -> p.getDiscount())
    .sum().get();
```



- ❖ The reason for these guidelines is stochastic nature of stream processing:
 - It is not possible to predict the order in which different CPU cores complete their processing.
 - Stream workload can be dynamically reassigned to different CPU cores to achieve better performance.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

26

Note: There are algorithms that cannot be correctly implemented in parallel mode, for example, if the value of specific discount depends on the overall amount of discounts.

Restrictions on Parallel Stream Processing

Incorrect handling of parallel streams can corrupt memory and slow down processing.

- Do not perform operations that require sequential access to a shared resource.
- Do not perform operations that modify shared resource.
- Use appropriate Collectors, such as:
 - `toMap` in sequential mode
 - `toConcurrentMap` in parallel mode

Parallel processing of the stream can only be beneficial if:

- Stream contains large number of elements
- Multiple CPU cores are available to physically parallelize computations
- Processing of stream elements requires significant CPU resources

```
List<BigDecimal> prices = new ArrayList<>();
list.stream()
    .parallel()
    .peek(p->System.out.println(p))
    .map(p->p.getPrice())
    .forEach(p->prices.add(p));
```



```
List<BigDecimal> prices = list.stream()
    .parallel()
    .map(p->p.getPrice())
    .collect(Collectors.toList());
```



```
Map<String, BigDecimal> namesAndPrices =
list.stream()
    .parallel()
    .collect(Collectors.toConcurrentMap(p->p.getName(),
                                         p->p.getPrice()));
```




Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

27

Factors that can degrade parallel stream processing performance:

- Triggering parallel stream processing on a system that does not have multi-core processing capabilities
- Operating on resources that cause parallel threads to be blocked, forcing sequential access
- Processing a small set of values likely to be faster in sequential mode

Operations that modify resources are considered not to be thread-safe, because concurrent calls may corrupt shared resource memory.

Parallel processing is unlikely to be efficient if:

- A system does not have at least four cores available to the JVM
- A data set is smaller than 10,000 items
- Processing uses any operations that cause threads to block

It is difficult to estimate efficiency of parallel processing, due to the number of factors that may affect the result, including factors outside your program control.

However, a rough estimate can be produced based on the following formula:

```
OverallProcessingCost = NumberOfStreamElements *  
IndividualElementProcessingCost
```

The higher the overall cost, the more likely that parallel processing will yield a performance benefit.

Processing cost is especially difficult to estimate, because an attempt to measure CPU processing time would distort the result. The best way to do such an estimation is to stress-load your program and collect overall processing time statistics on a number of test runs.

Summary

In this lesson, you should have learned how to:

- Describe Java Streams API
- Process stream pipelines
- Implement functional interfaces using lambda expressions
- Understand parallel stream processing



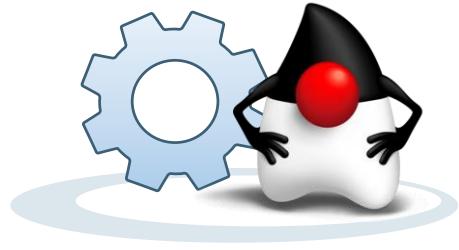
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

29

Practices

In this practice, you will:

- Replace all loops that process products and reviews collections with Streams
- Add a method that calculates a total discount per product rating



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

30

Handle Exceptions and Fix Bugs



ORACLE®

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Use Java Logging API
- Describe exception and error types
- Create custom exceptions
- Introduce try/catch/finally syntax
- Throw exception and pass exceptions to invokers
- Introduce try with parameters
- Use debugger
- Test code with assertions



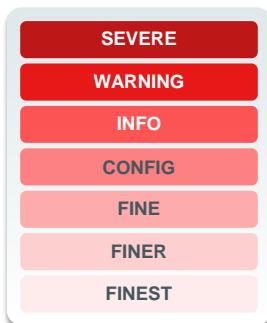
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Using Java Logging API

Use Logger class provided by Java Logging API to write logs.

- Each Logger is identified by a name.
- It is common practice to use class name as a [logger name](#).
- There are seven [levels of logging](#) that allow you to log different levels of severity



```
package demos;
import java.util.logging.*;
public class Test {
    private static Logger logger =
        Logger.getLogger(demos.Test.class.getName());
    public static void main(String[] args) {
        try {
            /* actions that can throw exceptions */
        }catch(Exception e) {
            logger.log(Level.ERROR,"Your error message",e);
        }
        logger.log(Level.INFO, "Your message");
        logger.info("Your message");
    }
}
```

```
module-info.java
module demos {
    requires java.logging;
}
```

❖ Use of the module-info class is covered in Modules lesson.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

Java Logging API is part of the `java.logging` module. Development and deployment of modularized Java applications are covered in Modules lesson.

The `java.util.logging.Level` class also defines `Level.ALL` and `Level.OFF`. These two levels are used for configuration and filtering purposes only.

Many logging frameworks exist, which provide similar APIs.

Logging libraries:

- log4j: A popular logging framework from Apache
- `java.util.logging` (Java Logging): Built-in logging framework included since JDK 1.4

Logging abstraction libraries:

- Commons Logging: An abstraction of logging frameworks provided by Apache
- Simple Logging Facade for Java (SLF4J): An alternative logging abstraction

Logging Method Categories

The logging methods are grouped into five main categories:

Method	Purpose
log	This is the most commonly used logging method. These overloaded methods have parameters for a level, a message, and optional additional parameters.
logp	Similar to the log methods, the "log precise" methods take additional parameters that specify a class and method name.
logrb	Similar to the logp methods, the "log with resource bundle" methods take a resource bundle name that is used to localize the message.
entering existing throwing	These are convenience methods used to log method entry, method exit, and exception throwing at the FINER level.
severe warning config info fine finer finest	These are convenience methods used in simple cases when a message should be logged at the level indicated by the method name.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

Example of using different logger methods:

```
public static void main(String[] args) {
    logger.log(Level.INFO, "Application Starting");
    try {
        readFile();
    } catch (IOException e) {
        logger.log(Level.SEVERE, "Failed to read
important file", e);
    }
}
```

```
public static void readFile() throws IOException {
    logger.entering("JavaLogging", "readFile");
    try {
        Path path = Paths.get("doesnotexist.txt");
        List<String> lines =
            Files.readAllLines(path,
Charset.defaultCharset());
    } catch (IOException e) {
        logger.throwing("JavaLogging", "readFile", e);
        throw e;
    } finally {
        logger.exiting("JavaLogging", "readFile");
    }
}
```

Guarded Logging

Use guarded logging to avoid processing messages that are due to be discarded.

1. Logging level can be set programmatically or via the configuration.
2. Message is concatenated, but is not recorded because it is below the logging-level threshold.
3. Message is not processed if it is below the logging-level threshold.
4. Object parameters can be used to avoid concatenating messages unnecessarily.

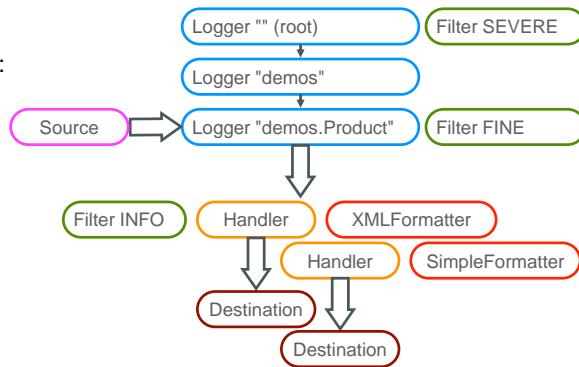
```
1 logger.setLevel(Level.INFO);
2 logger.log(Level.FINE, "Product "+id+" has been selected");
3 if(logger.isLoggable(Level.FINER)) {
4     logger.log(Level.FINE, "Product "+id+" has been selected");
}
logger.log(Level.FINE, "Product {0} has been selected", id);
```



Log Writing Handling

Log records can be discarded by one or more filters that may be attached to a logger or a log handler.

- Logger writes log messages with different log levels.
- Loggers form a hierarchy.
 - Child logger inherits log level from parent logger.
 - Child logger can override the log level.
- Log handler writes log messages to a log destination:
 - Console
 - File
 - Memory
 - Socket
 - Stream
- Filters can be set for both loggers and log handlers.
- Handlers use formatters to format the record.
 - SimpleFormatter
 - XMLFormatter



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Because the name of a logger defines its place in a hierarchy, the names you choose to give your loggers are important. The two most common names used are:

- The fully qualified classname of the class using logging statements
- The package name (not including the classname) of the class using logging statements

All loggers are registered with `LogManager`. You can use the `java.util.logging.LogManager` class to list all the registered logger names.

```

LogManager lm = LogManager.getLogManager();
Enumeration<String> nameEnum =
lm.getLoggerNames();
while(nameEnum.hasMoreElements()) {
    String loggerName = nameEnum.nextElement();
    Logger lgr = Logger.getLogger(loggerName);
}
    
```

In addition to existing log value filters, custom filters can also be created.
Example of the custom log filter:

```
package demos;  
import java.util.logging.Filter;  
public class CustomFilter implements Filter {  
    public boolean isLoggable(LogRecord record) {  
        // analyse log record and return true if record  
        should be logged or false if it should be discarded  
        return false;  
    }  
}
```

Logging Configuration

Logging can be configured using `logging.properties`:

```
handlers=java.util.logging.ConsoleHandler
demos.handlers=java.util.logging.FileHandler

.level=INFO
demos.level=FINE

java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

java.util.logging.FileHandler.pattern=%h/java%u.log
java.util.logging.FileHandler.limit=50000
java.util.logging.FileHandler.count=1
java.util.logging.FileHandler.formatter=java.util.logging.XMLFormatter
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

By default, the `logging.properties` file located in `JAVA_HOME/jre/lib` is used to configure Java Logging. WebLogic Server can be either Java Logging (the default) or Log4J Logging.

The following example shows passing the `logging.properties` file in the `-Djava.util.logging.config.file` argument to the `weblogic.Server` startup command:

```
java -
-Djava.util.logging.config.file=C:\mydomain\logging.properties weblogic.Server
```

Programmers can also update the logging configuration programmatically at run time in several ways:

- FileHandlers, MemoryHandlers, and PrintHandlers can all be created with various attributes.
- New handlers can be added and old ones can be removed.
- New loggers can be created and can be supplied with specific handlers.
- Levels can be set on target handlers.

Describe Java Exceptions

Exception is an unexpected event that occurs within a program.

Exceptions interrupt normal execution flow.

All exceptions descend from the class `Throwable`.

Types of Java Exceptions:

- **Checked Exceptions**
 - Must be caught or
 - Must be explicitly propagated
 - **Unchecked (Runtime) Exceptions**
 - May be caught and
 - Do not have to be explicitly propagated
- ❖ Unchecked exceptions are often an evidence of a bug in your code, which should be fixed rather than caught.
- ❖ Checked exceptions usually represent genuine problems that a normal functioning program may encounter and thus must be caught.

Exception type examples:

```
java.lang.Throwable
java.lang.Error
java.lang.AssertionError
java.lang.VirtualMachineError
java.lang.OutOfMemoryError
java.lang.Exception
java.sql.SQLException
java.io.IOException
java.io.FileNotFoundException
java.nio.file.FileSystemException
java.nio.file.NoSuchFileException
java.lang.RuntimeException
java.lang.NullPointerException
java.lang.ArithmetricException
java.lang.IndexOutOfBoundsException
java.lang.StringIndexOutOfBoundsException
java.lang.ArrayIndexOutOfBoundsException
java.lang.IllegalArgumentException
java.lang.NumberFormatException
java.util.IllegalFormatException
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Java program does not have to explicitly handle exceptions that are represented by classes `Error` and `RuntimeException` and their descendants. All other exceptions must be handled explicitly.

Create Custom Exceptions

Custom exception class characteristics:

- Must extend class `Exception` or another more specific descendant of `Throwable`
- May provide constructors that utilize superclass constructor abilities such as
 - Provide an error message
 - Wrap another exception indicating a cause of this exception

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

- ❖ Ability to wrap one exception inside another is often used when you want to catch one exception and throw another, but not lose information about the original cause of the error



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Typical use cases for custom exceptions are:

- Use custom exceptions to achieve a cleaner program interface that is less cluttered with implementation specific details. This means catching a low-level API exception and wrapping it up with a custom exception before throwing this custom exception to the invoker, for example, catching `FileNotFoundException` and wrapping it up into `OrderException` (custom exception).
- Use custom execution to report violations of business rules. This means checking if the program achieved the required business condition and throwing custom exception in case it did not, for example, verify the product discounted price to be within certain limits.

Remember, the list of exceptions that an operation can throw forms a part of the operation's signature and thus your program's contract. This means that each exception that a program can throw must be documented, with exact details that point out which exact circumstances may result in your program throwing this exception.

Throwing Exceptions

To produce an exception:

- Instantiate exception of the required type using any of the available constructors
- Use `throw` operator to interrupt the flow and trigger exception propagation process

When exception is raised:

- Normal program flow is terminated
- Control is passed to the nearest available exception handler (covered next)

If exception handler is not available within this method:

- Unchecked exceptions are automatically propagated to the invoker
- Checked exceptions must be explicitly listed within the `throws` clause

```
public void doThings() throws IOException, CustomException {  
    /* actions that may produce one of the following exceptions:  
     * throw new IOException();  
     * throw new CustomException();  
     * throw new NullPointerException();  
     * or any other runtime exceptions that do not require  
     * to be explicitly declared by the throws clause */  
}
```

- ✖ It is technically sufficient to declare that a method throws just a generic exception in order to propagate any checked exceptions. However, this is not a good practice, because it obscures which specific exceptions this method can produce.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

When instantiating an exception, you can use one of the following constructors:

- With no parameters
- With a String argument to supply custom error message
- With a String argument to supply custom error message and another exception that you want to carry inside the exception you are constructing

This last constructor is usually used when you want to intercept some exception and then throw another exception, but you do not want to lose information about the origin of the error.

Catching Exceptions

To catch an exception:

- Surround **code that can produce exceptions** with the **try** block.
- Place one or more **catch** blocks or **finally** block or both after that try block.
- Specific exception handlers (catching exception subtypes) must be placed before generic handlers.
- Unchecked (runtime exception) handlers are optional.
- When exception occurs within the try block, program flow is interrupted, and the control is passed to the nearest catch that matches the exception type.

- ❖ Unrelated exceptions can be handled by the same catch block.
- ❖ It is technically sufficient to provide a single **generic exception handler**. However, this is not a good practice, because you would struggle to distinguish specific reasons for why the program failed.

```
try {
    doThings();
} catch(NullPointerException |  
        ArithmeticException e){  
    /* exception handler actions */  
} catch(NoSuchFileException e){  
    /* exception handler actions */  
} catch(IOException e){  
    /* exception handler actions */  
} catch(Exception e){  
    /* exception handler actions */  
} finally{  
    /* actions that will be executed  
       regardless if exceptions occur or not */  
}
```



Exceptions and the Execution Flow

When a **normal execution path** encounters a runtime exception:

- Starting from the current statement the program flow is interrupted
- Control is passed to the closest matching exception handler, or if none is available, the program will exit.

```

1  public class Test {
2      public static void main(String[] args) {
3          int x = 5;
4          int y = 0;
5          int z = divide(x,y);
6          System.out.println(z);
7      }
8      public static int divide(int x, int y) {
9          int z = x/y;
10         return z;
11     }
12 }
```

In the example:

- Exception has not been intercepted
- Method that caused the exception was interrupted, so was the invoking method
- Program has exited
- The information about the exception and the stack trace is printed to the console
- Exception stack trace showed the path of the exception propagation

```

java Test
Exception in thread "main" java.lang.ArithmetricException: / by zero
        at Test.divide(Test.java:9)
        at Test.main(Test.java:5)
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

No explicit exception handler is required in this example, because `ArithmetricException` is a subclass of a `RuntimeException` and thus represents an unchecked exception.

Example Throwing an Unchecked Exception

To produce the exception:

- Create an instance of the relevant type of the exception
- Use `throw` operator to interrupt the flow and trigger exception propagation process

```
public class Test {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 0;  
        int z = divide(x,y);  
        System.out.println(z);  
    }  
    public static int divide(int x, int y) {  
        if (y == 0) {  
            throw new ArithmeticException("Error: "+x+"/"+y);  
        }  
        int z = x/y;  
        return z;  
    }  
}
```

- ❖ You may (but don't have to) provide a handler for catching unchecked exceptions.
- ❖ Unchecked exceptions often indicate a bug in a code that should be fixed, not caught.

```
java Test  
Exception in thread "main" java.lang.ArithmetricException: Error: 5/0  
at Test.divide(Test.java:10)  
at Test.main(Test.java:5)
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Example Throwing a Checked Exception

Checked exceptions must be explicitly caught or propagated.

- To catch an exception:
 - Surround code that can produce exceptions with the `try` block
 - Place one or more `catch` blocks or `finally` block or both after that try block
- To propagate checked exceptions to the invoker, add `throws` clause to the method definition listing exception that this method may potentially produce.

```
public static void main(String[] args) {  
    try {  
        openFile(null);  
    } catch(IOException e) {  
        e.printStackTrace();  
    }  
}  
public static void openFile(String fileName) throws IOException {  
    if (name == null) {  
         throw new NoSuchElementException("Filename must be set");  
    }  
}
```

- ✖ You may catch unchecked exceptions.
- ✖ You must catch checked exceptions (unless you listed them in a `throws` clause to be propagated further).



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

Throws clause can list generic exception types when the method throws an exception of a specific subtype. However, a throws clause that is too vague is not a good coding practice, since it makes it less clear which specific exceptions are to be expected to be produced by a method.

It's better to use Logger API than the `printStackTrace` method, to avoid directly writing to the console. Logger can also be configured to write to console if required, but logger will perform slow output operations on a separate thread, thus not blocking your thread.

Handling Exceptions

Exception handling structures:

- try block contains logic that may throw exceptions.
 - Exceptions within the try block interrupt the rest of the block.
 - Transfer control to the nearest matching catch block.
- catch blocks contain exception handling logic:
 - Writing logs
 - Throwing other exceptions
 - Terminating the rest of the method
 - Any other corrective actions
- finally block
 - Is executed no matter if errors occur or not within the try block
 - Performs resource cleanup and closure
- Further, try blocks may be embedded inside catch or finally blocks.

```
BufferedReader in = null;
try {
    in = new BufferedReader(new FileReader("some.txt"));
    String text = in.readLine();
} catch (FileNotFoundException ex) {
    logger.log(Level.SEVERE, "Error opening file", ex);
    return;
} catch (IOException ex) {
    logger.log(Level.SEVERE, "Error reading file", ex);
    throw new CustomException("Failed to read text",ex);
} finally {
    try {
        in.close();
    } catch (IOException ex) {
        logger.log(Level.SEVERE, "Error closing file", ex);
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Resource Auto-Closure

Try-with-parameters syntax provides auto-closure of resource.

- Classes that implement `AutoCloseable` interface can be instantiated using try-with-parameters syntax.
- Multiple resources may be initialized inside try-with-parameters construct.
- Automatic closure of such resources is provided by an implicitly formed finally block.

```
try /*initialise autocloseable resources*/ {
    /* use resources */
} catch (Exception e) {
    /* handle exceptions */
}
/* finally block invoking close method on every
autocloseable resource is formed implicitly */
```

```
try (BufferedReader in = new BufferedReader(new FileReader("some.txt"));
        BufferedWriter out = new BufferedWriter(new FileWriter("other.txt"))) {
    String text = in.readLine();
    out.writeLine(text);
} catch (FileNotFoundException ex) {
    logger.log(Level.SEVERE, "Opening file error", ex);
} catch (IOException ex) {
    logger.log(Level.SEVERE, "Read-write error", ex);
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

JDK 9 and later versions allow resources declared outside as final or effectively final to be used within the try-with-resources statement:

```
BufferedReader in = new BufferedReader(new
FileReader("some.txt"));

BufferedWriter out = new BufferedWriter(new
FileWriter("other.txt"));

try (in ; out) {
    String text = null;
    while ((text = in.readLine()) != null) {
        out.writeLine(text);
    }
} catch (FileNotFoundException ex) {
    logger.log(Level.SEVERE, "Opening file error", ex);
} catch (IOException ex) {
    logger.log(Level.SEVERE, "Read-write error", ex);
}
```

Suppressed Exceptions

Auto-closure of a resource may produce suppressed exceptions.

Consider the following example:

- One exception is produced in the try block.
- Another exception is produced in the implicitly formed finally block (thrown by the close method).
- Method getSuppressed returns a list of suppressed exceptions.

```
public class SomeResource implements AutoCloseable {
    public void doThings(boolean error) throws Exception {
        if (error) { throw new Exception("Action failed"); }
    }
    @Override
    public void close() throws Exception {
        throw new Exception("Closure failed");
    }
}
try (SomeResource r = new SomeResource()) {
    r.doThings(true);
} catch (Exception ex) {
    logger.log(Level.SEVERE, "Exception encountered:", ex);
    Throwable[] suppressedExceptions = ex.getSuppressed();
    for (final Throwable exception : suppressedExceptions) {
        logger.log(Level.SEVERE, "Suppressed Exception:", exception);
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Handle Exception Cause

Exception can be wrapped up by another exception.

Consider the following example:

- **Catch exception** and **throw new exception of different type**.
- **Wrap original exception into the new exception that you throw**.
- Method `getCause` retrieves the original exception.

```
public void doThings() throws CustomException {
    try (Reader in = new FileReader("some.txt")) {
        char[] buffer = new char[1024];
        in.read(buffer);
    }catch(IOException e){
        logger.log(Level.SEVERE, "File error", e);
        throw new CustomException("Failed to read text",e);
    }
}
try {
    doThings();
}catch(CustomException e) {
    logger.log(Level.SEVERE, "Exception encountered:", e);
    Throwable cause = e.getCause();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

Java Debugger

Debugger is used to find and fix bugs in Java platform programs.

- Command-line debugger tool `jdb` provided with JDK
- Java IDEs provide visual debug capabilities:
 - Toggle breakpoints in your source code



```
1 import demos.Drink;
2 import demos.Product;
3 import java.math.BigDecimal;
4 public class Shop {
5     public static void main(String[] args){
6         Product p1 = new Drink("Tea",BigDecimal.valueOf(1.99));
7         BigDecimal discount = p1.getDiscount();
8     }
9 }
```

- Launch program in debug mode

- ❖ Debugger actions are covered next.
- ❖ JDB tool documentation is available at <https://docs.oracle.com/en/java/javase/11/tools/jdb.html>.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

21

To set breakpoints, click in the margin of a line of code.
You can set multiple breakpoints in multiple classes.

Debugger Actions

Once debugger is running, you can execute the following debug actions:

1. finish debugger session
2. pause
3. continue
4. step over
5. step over expression
6. step into
7. step out
8. run to cursor
9. apply code changes



```

1 import demos.Drink;
2 import demos.Product;
3 import java.math.BigDecimal;
4 public class Shop {
5     public static void main(String[] args){
6         Product p1 = new Drink("Tea",BigDecimal.valueOf(1.99));
7         BigDecimal discount = p1.getDiscount();
8     }
9 }
16 public BigDecimal getDiscount() {
17     return discount;
18 }
19

```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

Action available in debug mode allows you to do the following:

1. Exit the debug session by clicking the button.
2. Pause the session.
3. Continue running until the next breakpoint or the end of the program.
4. If execution has stopped just before a method invocation, you may want to fast-forward to the next line after the method.
5. If execution has stopped just before an expression, you may want to fast-forward to the next line after the expression to see the final result.
6. You may prefer to step into an expression or method so that you can see how it functions at run time. You can also use this button to step into another class that is being instantiated.
7. If you have stepped into a method or another class, use the last button to step back out into the original code block.
8. Position cursor in a source code and request program to continue until it reaches this point, treating this line of code as an implicit breakpoint.
9. Make modifications to the source code and apply these changes.

Manipulate Program Data in Debug Mode

When your IDE is executing program in debug mode, you can

- Study all the available variables
- Modify the values
- Watch the specific variables

Name	Type	Value
<Enter new watch>		
Static		...
args	String[]	#118(length=0)
p1	Drink	#138
Inherited		
name	String	"Tea"
price	BigDecimal	#146
intVal	...	null
scale	int	2
precision	int	3
stringCache	...	null
intCompact	long	199
Static		...
discount	BigDecimal	#148



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

To focus on a specific variable, right-click on it and invoke the "Create Fixed Watch" menu. This would display the variable at the top of the list, so you would not have to scroll down in order to find it.

Validate Program Logic Using Assertions

Assertions can be used to verify the application is executing as expected.

- Assertions test for failure of various conditions.

```
Set<String> values = new HashSet();
String value = "acme";
boolean existingValue = values.add(value);
assert existingValue: "Value "+value+" already exists in the set";
```

- Assertions are disabled by default:

- Never assume they'll be executed (not used in production code).
- Do not use assertion to validate parameters or user input.
- Do not create assertions that cause any state changes or other side effects in the program flow.

- To enable assertions, use `-ea` or `-enableassertions` command-line option

```
java -ea <package>.<MainClass>
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

24

Assertions are designed for code testing purposes. If you need to validate validate parameters or user input, it is better to achieve this by other means, such as use BeanValidation API or throw exceptions such as an `IllegalArgumentException`.

Assertion can also be enabled or disabled for specific classes or packages:

```
java -ea:<package> -da <package>.<Class>
```

Enabling or disable assertions in Netbeans:

1. In Netbeans, right-click the project and select Properties.
2. In the window that appears, select Run.
3. Enter `-ea` or `-enableassertions` in VM Options to enable assertions.
4. Enter `-da` or `-disableassertions` in VM Options to disable assertions.

Input parameter validation should normally be considered as part of a program contract, and therefore your program should be throwing exception (other than assertion) to indicate any such validation issues.

Normal Program Flow with No Exceptions

Which operations are going to be invoked in this scenario?



```
public void doThings() {  
    try {  
        a();  
        b();  
    } catch(NoSuchFileException x) {  
        c();  
    } catch(IOException y) {  
        d();  
    } finally{  
        e();  
    }  
    f();  
}  
public void a() throws IOException {  
    if (false) {  
        throw new IOException();  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

Normal Program Flow with No Exceptions

When no exception is produced:

- Execution path invokes methods a, b, e, and f
- No catch block is triggered
- Finally block is executed
- Program continues normally

```
public void doThings() {  
    try {  
        a();  
        b();  
    }catch(NoSuchFileException x){  
        c();  
    }catch(IOException y){  
        d();  
    }finally{  
        e();  
    }  
    f();  
}  
  
public void a() throws IOException {  
    if (false) {  
        throw new IOException();  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

26

Program Flow Producing a Runtime Exception

Which operations are going to be invoked in this scenario?



```
public void doThings() {  
    try {  
        a();  
        b();  
    } catch(NoSuchFileException x) {  
        c();  
    } catch(IOException y) {  
        d();  
    } finally{  
        e();  
    }  
    f();  
}  
public void a() throws IOException {  
    if (true){  
        throw new NullPointerException();  
    }  
    throw new IOException();  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

27

Program Flow Producing a Runtime Exception

When `NullPointerException` is produced:

- Execution path invokes methods a and e
- No catch block is triggered
- Finally block is executed
- Normal program flow does not resume because this exception is left unhandled

- ✖ Because this is an unchecked exception, it does not have to:
 - Be listed by the `throws` clause
 - Have a `catch` block accosted with it

```
public void doThings() {
    try {
        a();
        b();
    }catch(NoSuchFileException x) {
        c();
    }catch(IOException y) {
        d();
    }finally{
        e();
    }
    f();
}
public void a() throws IOException {
    if (true){
        throw new NullPointerException();
    }
    throw new IOException();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

28

You could also add a catch clause to catch unchecked exceptions, in which case the handling would be no different from any other check exception scenario.

Program Flow Catching Specific Checked Exception

Which operations are going to be invoked in this scenario?



```
public void doThings() {  
    try {  
        a();  
        b();  
    } catch(NoSuchFileException x) {  
        c();  
    } catch(IOException y) {  
        d();  
    } finally{  
        e();  
    }  
    f();  
}  
public void a() throws IOException {  
    if (true) {  
        throw new NoSuchFileException();  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

29

Program Flow Catching Specific Checked Exception

When `NoSuchFileNotFoundException` is produced:

- Execution path invokes methods `a`, `c`, `e`, and `f`
- `NoSuchFileNotFoundException` catch block is triggered
- Finally block is executed
- The rest of the program continues normally because exception was intercepted

- NoSuchFileNotFoundException is a subclass of `IOException`, so the `throws` clause may (but does not have to) explicitly list it.
- `IOException` catch block could intercept the `NoSuchFileNotFoundException`, unless a more specific catch is provided.

```
public void doThings() {
    try {
        a();
        b();
    }catch(NoSuchFileNotFoundException x) {
        c();
    }catch(IOException y) {
        d();
    }finally{
        e();
    }
    f();
}

public void a() throws IOException {
    if(true) {
        throw new NoSuchFileNotFoundException();
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

30

When exception occurs, a `try` block is terminated, and control is passed to the nearest matching exception handler. If exception has been successfully handled, normal program flow can resume, passing control to the rest of the code after the `try/catch/finally` construct.

Program Flow Catching Any Exceptions

Which operations are going to be invoked in this scenario?



```
public void doThings() {  
    try {  
        a();  
        b();  
    } catch(NoSuchFileException x) {  
        c();  
    } catch(Exception y) {  
        d();  
        return;  
    } finally{  
        e();  
    }  
    f();  
}  
public void a() throws IOException {  
    if (true) {  
        throw new IOException();  
    }  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

31

Program Flow Catching Any Exceptions

When `IOException` (or any other) is produced:

- Execution path invokes methods `a`, `d`, and `e`
- Exception catch block is triggered
- Finally block is executed
- Although exception was intercepted, the normal program flow does not resume, because the exception handler has terminated the method.

- ❖ A generic exception handler would catch any checked or unchecked exceptions.
- ❖ Exception handler may terminate the rest of the method using `return` statement or even terminate Java runtime using `System.exit(0)`.

```
public void doThings() {
    try {
        a();
        b();
    }catch(NoSuchFileException x) {
        c();
    }catch(Exception y){
        d();
        return;△
    }finally{
        e();
    }
    f();
}

public void a() throws IOException {
    if(true) {
        ✗ throw new IOException();⊗
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

32

Summary

In this lesson, you should have learned how to:

- Use Java Logging API
- Describe exception and error types
- Create custom exceptions
- Introduce try/catch/finally syntax
- Throw exception and pass exceptions to invokers
- Introduce try with parameters
- Use debugger
- Test code with assertions



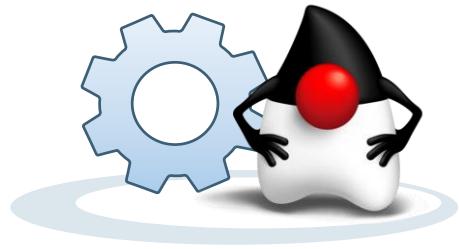
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

33

Practices

In this practice, you will:

- Test circumstances in which exceptions will be thrown from ProductManagement
- Write exception handling and propagation code to mitigate program errors
- Parse text, numeric, and date values and handle related exceptions



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

34

Java IO API



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Describe Java IO
- Read and write text and binary data
- Use Java Serialization
- Work with the file system



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

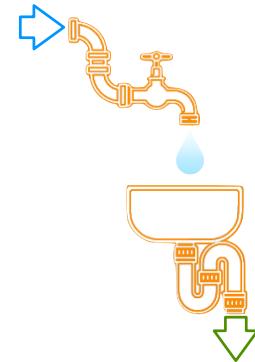
Java Input-Output Principles

Characteristics of Java Input-Output (IO):

- Read information from various **sources** - input direction
- Write information to various **destinations** - output direction
- Information is transferred through a series of **interconnected streams** (pipes)

Streams are categorized based on:

- The type of data that stream can carry, for example, text or binary
- Direction of the stream - input or output
- Type of the source or destination to which this stream is connected
- Additional features, such as filtering or transformation of data



❖ Note: Sources and destinations could be Files, Network Sockets, Console, Memory, etc.



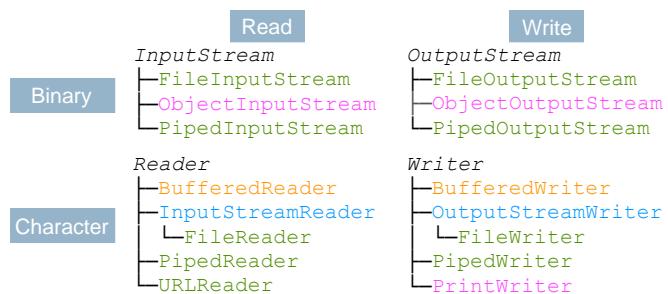
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

Java Input-Output API

IO classes are located in `java.io` and `java.nio` packages.

- *Abstract* classes define general text and binary data read and write abilities.
- Concrete classes descend from these parents to provide different types of IO stream handlers:
 - Connect to different sources and destinations
 - Transform stream content
 - Perform content buffering
 - Provide convenience methods



- ❖ Many other types of IO stream handling classes are available.
- ❖ This taxonomy is a bit artificial as many IO streams provide combinations of abilities.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

The examples show the following types of streams:

Connect to different sources and destinations: `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`, `PipedInputStream`, `PipedOutputStream`, `PipedReader`, `PipedWriter`, `URLReader`

The purpose of these groups of streams is to provide connectivity to data sources or destinations, such as files and URLs, or to provide stream connectivity between threads.

Transform stream content: `InputStreamReader`, `InputStreamWriter`

The purpose of these groups of streams is to provide text to binary and binary to text conversions.

Perform content buffering: Gather characters into `String` objects or write characters from `String` objects

Provide convenience methods: Printing text, object, and primitive values; serialize or deserialize objects

Many other types of specialized IO streams are available.

Reading and Writing Binary Data

Basic binary data reading and writing capabilities are defined by the pair of abstract classes:

InputStream	
<code>int read(byte[] buffer, int offset, int length)</code>	read binary data from the stream
<code>void mark(int position)</code>	mark position in the stream
<code>boolean markSupported()</code>	
<code>long transferTo(OutputStream out)</code>	transfer all data from input to output
<code>int available()</code>	check the amount of available data
<code>void skip(long length)</code>	skip data
<code>void reset()</code>	reset the stream
<code>void close()</code>	close the stream
OutputStream	
<code>void write(byte[] buffer, int offset, int length)</code>	write binary data to the stream
<code>void flush()</code>	flush the stream
<code>void close()</code>	close the stream

✖ See notes for overloaded versions of read and write methods.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Operation read is overloaded:

```
int read()                                read
one byte at a time

int read(byte[] buffer)
    read complete buffer

int read(byte[] buffer, int offset, int length)
    read a portion of the buffer
```

Operation write is overloaded:

```
void write(int charCode)
    write single byte

void write(byte[] buffer)
    write a complete buffer

void write(byte[] buffer, int offset, int length)
    write a portion of buffer
```

Basic Binary Data Reading and Writing

Basic Binary data read and write capabilities are provided by Input and Output streams.

- Input and Output streams both implement `AutoCloseable` interface.
 - Can be used in the try-with-parameters construct
 - Closed within the implicit finally block
 - Otherwise must be closed explicitly within the finally block
- Method `read` populates the `buffer` with portions of binary data and returns an int `length` indicator.
 - On intermediate reads, this indicator is equal to the buffer length.
 - On the read before last, it is equal to how much data remains in the stream.
 - On the last read, it equals to -1, which indicates the end of the stream.

buffer							length
80	75	...	0	0	28		1024
0	1	...	15	0	4		1024
5	67	...	3				1022
...							-1

```
try (InputStream in = new FileInputStream("some.xyz");
     OutputStream out = new FileOutputStream("other.xyz")) {
    byte[] buffer = new byte[1024];
    int length = 0;
    while((length = in.read(buffer)) != -1) {
        out.write(buffer, 0, length);
    }
} catch(IOException e){ /* exception handling logic */ }
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

The example reads and writes file data, but it could have been any other source or destination. `FileInputStream` and `FileOutputStream` classes are just two examples of any different types of source and destination streams available.

No actual data handling is applied, which means you could have just copied the file:

```
try {
    Path sourceFile = Paths.get("some.xyz");
    Path destinationFile = Paths.get("other.xyz");
    Files.copy(sourceFile, destinationFile);
} catch(IOException e) {
/* exception handling logic */
}
```

Generally for any source and destination, data could be transferred in a more automated way:

```
try (InputStream in = new FileInputStream("some.xyz");
     OutputStream out = new FileOutputStream("other.xyz")) {
    int amountOfData = in.transfer(out);
} catch (IOException e) {
/* exception handling logic */
}
```

Reading and Writing Character Data

Basic character data reading and writing capabilities are defined by the pair of abstract classes:

Reader	
int read(char[] buffer, int offset, int length)	read character data from the stream
boolean ready()	check if the stream is ready
void mark(int position)	mark position in the stream
boolean markSupported()	
long transferTo(Writer out)	transfer all data from input to output
void close()	close the stream
Writer	
void write(char[] buffer, int offset, int length)	write character data to the stream
void flush()	flush the stream
void close()	close the stream

❖ See notes for overloaded versions of read and write methods.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Operation read is overloaded:

```
int read()                                read
one character at a time
int read(char[] buffer)
    read complete buffer
int read(char[] buffer, int offset, int length)
    read a portion of the buffer
int read(CharBuffer buffer)                use
CharBuffer object as a buffer wrapper
```

Operation write is overloaded:

```
void write(int charCode)
    write single character
void write(char[] buffer)
    write a complete buffer
void write(char[] buffer, int offset, int length)
    write a portion of buffer
void write(String buffer)
    use String as a buffer
void write(String buffer, int offset, int length)
    use portion of a String as a buffer
```

Basic Character Data Reading and Writing

Basic text read and write capabilities provided by Reader and Writer:

- Reader and Writer both implement `AutoCloseable` interface.
 - Can be used in the try-with-parameters construct
 - Closed within the implicit finally block
 - Otherwise must be closed explicitly within the finally block
- Method `read` populates the `buffer` with portions of character data and returns an int `length` indicator.
 - On intermediate reads, this indicator is equal to the buffer length.
 - On the read before last, it is equal to how much data remains in the stream.
 - On the last read, it equals to -1, which indicates the end of the stream.

buffer						length
T	h	...	p	l	a	1024
n	t	...	a	m		1024
a	s	...	d			1022
						-1

```
Charset utf8 = Charset.forName("UTF-8");
try (Reader in = new FileReader("some.txt", utf8);
     Writer out = new FileWriter("other.txt", utf8)) {
    char[] buffer = new char[1024];
    int length = 0;
    while((length = in.read(buffer)) != -1) {
        out.write(buffer, 0, length);
    }
} catch(IOException e){ /* exception handling logic */ }
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

The example reads and writes file data, but it could have been any other source or destination.

No actual data handling is applied, which means you could have just copied the file:

```
try {
    Path sourceFile = Paths.get("some.xyz");
    Path destinationFile = Paths.get("other.xyz");
    Files.copy(sourceFile, destinationFile);
} catch(IOException e){
    /* exception handling logic */
}
```

Generally for any source and destination, data could be transferred in a more automated way:

```
try (Reader in = new FileReader("some.txt");
     Writer out = new FileWriter("other.txt")) {
    int amountOfData = in.transfer(out);
} catch(IOException e){
    /* exception handling logic */
}
```

Connecting Streams

Streams can be connected to one another to apply features.

- Connecting streams apply transform, filter, and buffer data capabilities.
- Connect stream into a chain until you get the required and convenient way of handling content.



```
Charset utf8 = Charset.forName("UTF-8");
try (BufferedReader in =
      new BufferedReader(new InputStreamReader(new FileInputStream("some.txt"), utf8));
     PrintWriter out =
      new PrintWriter(new OutputStreamWriter(new FileOutputStream("other.txt"), utf8))) {
    String line = null;
    while((line = in.readLine()) != null){ // null indicates the end of stream
      out.println(line);
    }
} catch(IOException e){ /* exception handling logic */ }
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Class BufferedWriter is a "mirrored" counterpart of **BufferedReader**. It provides `write` method to write String values. **Class PrintWriter** is similar to **BufferedWriter**, but it also provides printing capability (`println` method), which can be convenient when you program needs to print lines.

Standard Input and Output

Accept input and produce output using the standard input and output.

- Class `System` provides references to standard input, output, and error output using the following constants:
 - `System.in` references an instance of `InputStream` reading data from the standard input.
 - `System.out` references an instance of `PrintStream` writing data to the standard output.
 - `System.err` references an instance of `PrintStream` writing data to the standard error output.
- Class `java.util.Scanner` provides convenient ways of parsing input.

```
public class Echo {  
    public static void main(String[] args) {  
        Scanner s = new Scanner(System.in);  
        String txt = null;  
        System.out.println("To quit type: exit");  
        System.out.println("Type value and press enter:");  
        while (!(txt = s.nextLine()).equals("exit")) {  
            System.out.println("Echo: "+txt);  
        }  
    }  
}
```

- ❖ It is not advisable to overuse standard input and output, as they represent sequential access to resources and may significantly degrade performance and scalability of the program.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Using Console

Class `java.io.Console` provides access to the system console with operations such as:

- `readLine()` reads user input
- `readPassword()` reads user input suppressing the display of input characters
- `reader()` - retrieves `Reader` object associated with the console
- `writer()` - retrieves `PrintWriter` object associated with the console

```
public class Echo {
    public static void main(String[] args) {
        Console c = System.console();
        if (c == null) {
            System.out.println("Console is not supported");
            return;
        }
        PrintWriter out = c.writer();
        out.println("To quite type: exit");
        out.println("Type value and press enter:");
        String txt = null;
        while (!(txt = c.readLine()).equals("exit")) {
            out.println("Echo: "+txt);
        }
    }
}
```

See notes for:

- ❖ The difference between the `PrintWriter` and `PrintStream`
- ❖ Password handling example



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Compare characteristics of `PrintStream` and `PrintWriter`:

- Generally provide very similar capabilities
- `PrintStream` outputs bytes rather than characters
- `PrintWriter` outputs characters and automatically flushes the stream

Both catch `IOExceptions` and set an internal flag that can be tested via the `checkError` method:

```
if (out.checkError()) {
    throw new IOException("Error using the output");
}
```

The `Charset` can be specified when more control over the encoding process is required. The example wraps `PrintStream` pointing to standard output with `PrintWriter` that performs auto-flush of the buffer and converts characters using UTF-8 encoding:

```
new PrintWriter(System.out, true, forName("UTF-8"));
```

The example processes console entry of secure data, such as username and password. Notice that operation `readPassword` suppresses the display of user input characters.

The example below also calculates a password digest using `java.security.MessageDigest` class, which supports different digest algorithms, such as MD-5 or SHA-256.

```
public class SecureInput {  
    public static void main(String[] args) {  
        Console c = System.console();  
        if (c == null) {  
            System.out.println("Console is not supported");  
            return;  
        }  
        PrintWriter out = c.writer();  
        out.println("Enter username:");  
        String username = c.readLine();  
        try {  
            MessageDigest md = MessageDigest.getInstance("SHA-  
256");  
            out.println("Enter password:");  
            byte[] digest =  
                md.digest(String.valueOf(c.readPassword()).getBytes());  
            String hash = (new BigInteger(1,  
                digest)).toString(16);  
        } catch(NoSuchAlgorithmException ex) {  
            out.println("Unable to create password hash");  
        }  
    }  
}
```

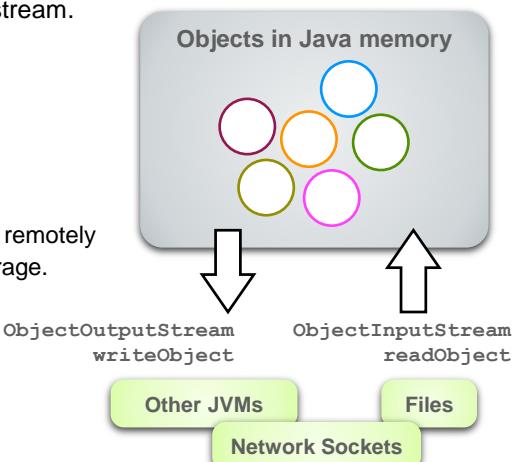
Understand Serialization

Serialization Purpose:

- Serialization is a process of writing objects from memory into a stream.
- Deserialization is a process of reading objects from the stream.
- Data is serialized in a binary form.

Serialization use cases:

- Swapping objects to avoid running out of memory
- Sending objects across network:
 - replicate data between nodes in a cluster
 - pass parameters and return values when calling methods remotely
- Serialization is not a suitable solution for long-term data storage.
 - Serialized value is specific to the compiled code version.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Serialization allows direct read-write operations on memory to extract values and place them into streams or restore values directly to memory from streams.

It provides a good mechanism to replicate values across network or perform memory swap.

The reason serialization is not suitable for storing program data long term is because of the fact that serialization writes the actual binary compiled version of your code directly from JVM memory. If you recompile your class, previously serialized objects of that class would become invalid, as they would no longer match new class definition.

Serializable Object Graph

An object can be written to or be read from a stream.

- `java.io.Serializable` interface is used to indicate permission to serialize instances of a class.
- You can serialize all primitives as well as many other classes such as strings, numbers, dates, and collections.
- Serialization includes the entire object graph, except transient variables.
- `SerializationException` is produced upon an attempt to serialize a non-transient variable that is of non-serializable type.

```
public class PriceList implements Serializable {
    private LocalDate date;
    private Set<Product> items = new HashSet<>();
    private transient String hash = generateHash();
    // ...
}
public class Product implements Serializable {
    private String name;
    private BigDecimal price;
    // ...
}
```

```
PriceList list = new PriceList(LocalDate.now());
list.addItem(new Drink("Tea", 1.9));
list.addItem(new Food("Cake", 3.5));
```

- ❖ Keyword `transient` indicates that a field should exist only in memory and is not going to be written into an `ObjectOutputStream`.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Object Serialization

Java IO provides classes to read and write objects in and out of streams:

- `ObjectOutputStream` writes serializable object to a stream.
- `ObjectInputStream` reads serializable object from a stream.

```
PriceList list = new PriceList(LocalDate.now());
list.addItem(new Drink("Tea", 1.99));
list.addItem(new Food("Cake", 3.5));
try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("swap"))) {
    out.writeObject(list);
    list = null;
} catch (IOException e) {
    logger.log(Level.SEVERE, "Failed write object into a file", e);
}
try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("swap"))) {
    list = (PriceList)in.readObject();
} catch (FileNotFoundException e) {
    logger.log(Level.SEVERE, "File not found", e);
} catch (IOException e) {
    logger.log(Level.SEVERE, "Failed to read object from file", e);
} catch (ClassNotFoundException e) {
    logger.log(Level.SEVERE, "Unknown serialised type", e);
}
```

Example:

- ❖ Writes object into a file
- ❖ Cleans object reference
- ❖ Reads object from a file



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

Cleaning the existing object reference is not an important part of the example. However, it demonstrates that serialization can be used to swap objects out of memory and restore them back when required.

Serialization of Sensitive Information

Serialization can write data outside of the secure environment of your program.

- This could present a security concern for dealing with sensitive information.
- Consider protecting information by generating secure object hash or using encryption.

```
public String generateHash(Object obj) { NoSuchAlgorithmException, IOException {
    String hash = null;
    try (ByteArrayOutputStream byteArrayStream = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(byteArrayStream)) {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        out.writeObject(obj);
        hash = new BigInteger(1, md.digest(byteArrayStream.toByteArray())).toString(16);
    }
    return hash;
}
```

Example:

- ❖ Serialize object
- ❖ Get serialized object data as byte array
- ❖ Generate SHA-256 digest out of this array
- ❖ Convert digest into String

- ❖ The example scrambles data, generating a secure hash value.
- ❖ More information on security and encryption can be found in Security Appendix.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

A cryptographic hash function is a hash function that is suitable for use in cryptography.

- Map data of arbitrary size (aka the "message") to a binary value of a fixed size (the "hash value", "hash", or "message digest").
- It is a one-way function, which is practically infeasible to invert.

Customize Serialization Process

Serialization-Deserialization processes can be customized.

- Method `writeObject` performs custom actions when serializing this object.
- Method `readObject` performs custom actions when deserializing this object.
- Note that these methods must use `private` access modifiers.
- Invocation of `defaultWriteObject` and `defaultReadObject` methods allows normal operations.

```
public class Product implements Serializable {
    private LocalDate date;
    private Set<Product> items = new HashSet();
    private transient byte[] hash = new byte[32];
    private void writeObject(ObjectOutputStream out)
        throws IOException {
        out.defaultWriteObject();
        out.writeObject(Instant.now());
    }
    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        hash = generateHash();
    }
    // ...
}
```

Example:

- Add current timestamp to the output
- Recalculate transient values



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

You can decide which fields to serialize using `serialPersistentFields` static array of fields instead of using a `transient` keyword.

```
public class Product implements Serializable {
    private String name;
    private BigDecimal price;
    // private transient BigDecimal discount;
    private BigDecimal discount;
    private static final ObjectStreamField[]
    serialPersistentFields = {
        new ObjectStreamField("name", String.class),
        new ObjectStreamField("price", BigDecimal.class),
    };
}
```

Serialization and Versioning

When designing Serializable classes, consider the class life cycle.

- Serialization is not a suitable solution for long-term data storage.
- A modified class definition is produced when:
 - Source code is changed
 - Class is recompiled with a different version of JDK
- Mismatch of the class definition could result in unpredictable program behavior.
- `ObjectInputStream` checks the `serialVersionUID` indicator to ensure that the class definition you are using is the same as the class definition used by Java runtime at the time when the object was serialized and throws an `InvalidClassException` in case of a mismatch.

```
public class Product implements Serializable {  
    private static final long serialVersionUID = 1L;  
    // original code  
}  
  
public class Product implements Serializable {  
    private static final long serialVersionUID = 2L;  
    // next version code  
}
```

new version

- ❖ For long-term data storage, consider saving data in XML or JSON formats using JAXB and JSON-P APIs or saving data to databases using JPA API.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Working with Filesystems

Package `java.nio.file` contains classes that handle filesystem interactions:

- Class `Path` represents files and folders.
- Class `Files` provides operations that handle path objects.
- Class `FileSystem` describes available filesystems and their properties:
 - Access properties of file stores using a stream of `FileStore` objects
 - Discover filesystem roots using a stream of `Path` objects "/" or "C:" etc.
 - Get path node separator for the given OS "/" or "\".

```
FileSystem fs = FileSystems.getDefault();
fs.getFileStores().forEach(s->System.out.println(s.type()+' '+s.name()));
fs.getRootDirectories().forEach(p->System.out.println(p));
String separator = fs.getSeparator();
```

- ❖ Class `java.io.File` is a legacy API class whose functionalities are now distributed between `Path`, `Files` and `FileSystem` classes of a new IO API.
- ❖ Operation `toFile()` of the `Path` class converts path object to file object.
- ❖ Operation `toPath()` of the `File` class converts file object to path object.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

Examples of functionalities provided by the `FileSystem` class:

```
FileSystem fs = FileSystems.getDefault();

// locating users or groups

UserPrincipalLookupService ul =
fs.getUserPrincipalLookupService();

GroupPrincipal group =
ul.lookupPrincipalByGroupName("staff");

UserPrincipal user = ul.lookupPrincipalByName("joe");
if (user != null ) { String name = user.getName(); }

if (group != null ) { String name = group.getName(); }
```

```
// finding file store space
fs.getFileStores().forEach(s-> {
    try {
        System.out.println(s.name() +" " +s.getTotalSpace() +" "
" +s.getUnallocatedSpace() +" " +s.getUsableSpace());
    } catch (IOException ex) {
        System.out.println(ex);
    } });
}
```

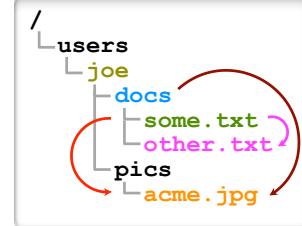
Constructing Filesystem Paths

Class `Path` represents files and folders as immutable objects.

- A var-arg method `of` constructs new path objects.
- Path objects may represent an absolute or a relative path (can convert between path representations).
- You may create non-existent path objects without causing exceptions until you try to use them.

```
Path someFile = Path.of("/", "users", "joe", "docs", "some.txt");
Path justSomeFile = someFile.getFileName();
Path docsFolder = someFile.getParent();
Path currentFolder = Path.of(".");
Path acmeFile = docsFolder.resolve("../pics/acme.jpg");
Path otherFile = someFile.resolveSibling("other.txt");
Path normalisedAcmeFile = acmeFile.normalize();
Path verifiedPath = acmeFile.toRealPath();
Path betweenSomeAndAcme = someFile.relativize(acmeFile);
```

- ❖ Absolute path starts from the filesystem root: "/" or "C:"
- ❖ Path nodes are separated with "/" or "\" character
- ❖ Relative path starts at the current folder.
- ❖ Relative path "." references the current folder.
- ❖ Relative path ".." references the parent folder.
- ❖ Method `toRealPath` validates existence of the path.



```
/users/joe/docs/some.txt
some.txt
/users/joe/docs
assuming "." is /users/joe
/users/joe/docs/../pics/acme.jpg
/users/joe/docs/other.txt
/users/joe/pics/acme.jpg
../../pics/acme.jpg
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

Other ways of constructing `Path` objects are:

```
Path someFile1 = Path.of("/users/joe/docs/some.txt");
Path someFile2 =
Paths.get("/", "users", "joe", "docs", "some.txt");
Path someFile3 = Paths.get("/users/joe/docs/some.txt");
FileSystem fs = FileSystems.getDefault();
Path otherFile =
fs.getPath("/", "users", "joe", "pics", "acme.jpg");
```

Methods `resolve` and `resolveSibling` accept either `String` or `Path` parameter and return another path that is constructed in relation to the original path object.

Method `normalize` removes redundant elements from the path.

Method `toRealPath()` allows removal of redundant elements from the path, just like the `normalize` method, but it also verifies that such a path exists and throws `IOException` if it does not. When verifying the path, you may request not to follow symbolic links

`toRealPath(LinkOption.NOFOLLOW_LINKS)`.

Converting paths to canonical form is considered to be a security improvement, to prevent potential directory traversal attack, i.e., an attempt to guess the directory structure on a computer by using ".../somepath" relative paths.

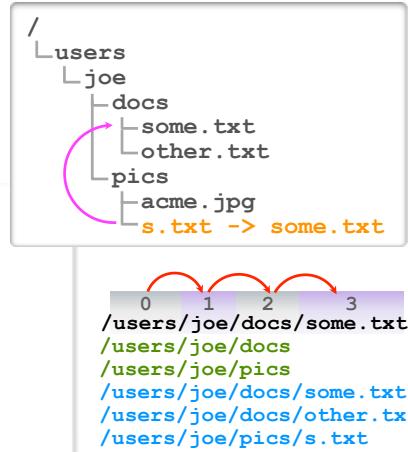
Method `relativize` constructs a relative path between two other path objects.

Navigating the Filesystem

Class `Files` provides operations that handle path objects.

- Path object can be represented as a **sequence of path elements**.
- Class `Files` provides operations to navigate the filesystem:
 - List folder content
 - Walk down filesystem path
- Symbolic links represent "shortcuts" to other paths.
- Class `Files` can **create** and **read** symbolic links.

```
Path joe = Path.of("/", "users", "joe");
Path p1 = Path.of("/", "users", "joe", "docs", "some.txt");
for(int i=0; i<p1.getNameCount(); i++){
    Path p = p1.getName(i);
}
Path p2 = Path.of("./pics/s.txt");
Files.createSymbolicLink(p2, p1);
Files.list(joe).forEach(p -> System.out.println(p));
Files.walk(joe).map(p -> p.toString())
    .filter(s -> s.endsWith("txt"))
    .forEach(p -> System.out.println(p));
Path p3 = Files.readSymbolicLink(p2);
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

24

Both `list` and `walk` operations return `Stream` objects, to which `map`, `peek`, `filter`, `forEach`, etc. operations could be applied.

Operation `list` returns a stream of immediate child path objects of a given path (contents of a specific directory).

Operation `walk` returns a stream of immediate child path objects and their descendants (contents of a specific directory and any further subdirectories).

Optionally, you can limit the directory depth when using method `walk`:

```
Path startPath = Path.of(".");
int depth = 10;
Files.walk(startPath, depth);
```

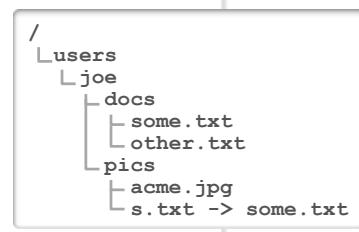
Many operations of the `Files` class can be augmented to follow or not to follow symbolic links:

```
Files.walk(startPath, FileVisitOption.FOLLOW_LINKS);
or
Files.exists(Path.of("acme"), LinkOption.NOFOLLOW_LINKS);
```

Analyse Path Properties

Class `Files` provides operations to retrieve path object properties.

```
Path p1 = Path.of("/users/joe/docs/some.txt"); // absolute path
Path p2 = Path.of("./docs/some.txt"); // relative path
Path p3 = Path.of("./pics/s.txt"); // symbolic link
Files.isDirectory(p1); // false
Files.isExecutable(p1); // false
Files.isHidden(p1); // false
Files.isReadable(p1); // true
Files.isWritable(p1); // true
Files.isRegularFile(p1); // true
Files.isSymbolicLink(p3); // true
Files.isSameFile(p1, p2); // true
Files.isSameFile(p1, p3); // true
Files.probeContentType(p1); // text/plain
PosixFileAttributes fa = Files.readAttributes(p1, PosixFileAttributes.class);
long size = fa.size(); // 640
FileTime t1 = fa.creationTime(); // 2019-01-24T14:23:40Z
FileTime t2 = fa.lastModifiedTime(); // 2019-05-09T20:47:54.438626Z
FileTime t3 = fa.lastAccessTime(); // 2019-05-10T10:16:18.715692Z
UserPrincipal user = fa.owner(); // joe
GroupPrincipal group = fa.group(); // staff
Set<PosixFilePermission> permissions = fa.permissions();
String t = PosixFilePermissions.toString(permissions); // rwxr-xr-x
```



✖ See notes on attribute views.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

Depending on a type, filesystems may present different sets of path properties:

```
Path path = Path.of("some.txt");
BasicFileAttributes a1 = Files.readAttributes(path,
BasicFileAttributes.class); // generic attributes
PosixFileAttributes a2 = Files.readAttributes(path,
PosixFileAttributes.class); // posix attributes
DosFileAttributes a3 = Files.readAttributes(path,
DosFileAttributes.class); // DOS attributes
```

Example of setting dos file attribute:

```
Files.setAttribute(path, "dos:hidden", true);
```

It is a good practice to check type of the file system before operating with system specific attributes:

```
FileSystem fs = path.getFileSystem();
Set<String> fsTypes = fs.supportedFileAttributeViews();
Conversions between java.nio.file.attribute.FileTime and
java.time.Instant and visa versa are provided:
FileTime t1 = a1.creationTime();
Instant timeStamp = t1.toInstant();
FileTime t2 = FileTime.from(Instant.now());
```

Set Path Properties

Modify path object properties using class `Files`.

- Set last modified time
- Set permissions
- Look up users and groups using `FileSystem UserPrincipalLookupService`
- Set owner and group

```
Path path = Path.of("/users/joe/docs/some.txt");
Files.setLastModifiedTime(path, FileTime.from(Instant.now()));
Set<PosixFilePermission> perms = PosixFilePermissions.fromString("rw-rw-r--");
Files.setPosixFilePermissions(path, perms);
FileSystem fs = path.getFileSystem();
UserPrincipalLookupService uls = fs.getUserPrincipalLookupService();
UserPrincipal user = uls.lookupPrincipalByName("joe");
GroupPrincipal group = uls.lookupPrincipalByGroupName("staff");
Files.setOwner(path, user);
Files.getFileAttributeView(path, PosixFileAttributeView.class).setGroup(group);
```

-rw-rw-r-- joe staff 25 10 May 11:59 some.txt

✖ See notes for explanation of POSIX permissions.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

26

Posix Permissions Symbolic Notation Examples:

```
----- no permissions
-rwx----- read, write, & execute only for owner
-rwxrwx--- read, write, & execute for owner and group
-rwxrwxrwx read, write, & execute for owner, group and others
---x---x-- execute
---w---w--w write
---wx-wx-wx write & execute
-r--r--r-- read
-r-xr-xr-x read & execute
-rw-rw-rw- read & write
-rwxr----- owner can read, write, & execute; group can only read; others have no permissions
```

Alternatively, Posix file permissions can be constructed explicitly as a Set object:

```
Set<PosixFilePermission> perms =  
Set.of(PosixFilePermission.OWNER_READ,  
  
PosixFilePermission.OWNER_WRITE,  
  
PosixFilePermission.GROUP_READ,  
  
PosixFilePermission.GROUP_WRITE,  
  
PosixFilePermission.OTHERS_READ);
```

Create Paths

Class Files provides operations to create folders and files:

- `nonExists` and `exists` verify existence of the path.
- `createDirectory` creates immediate child subfolder, implying that the parent folders exist.
- `createDirectories` creates a chain of subfolders.
- `createFile` creates a new file object.

```
Path source = Path.of("/users/joe/docs");
Path backup = Path.of("/users/joe/backup/docs");
if (!Files.notExists(backup)) {
    Files.createDirectories(backup);
}
Path readme = backup.resolve("../readme.txt").normalize();
Files.createFile(readme);
Files.writeString(readme, "Backup time: "+Instant.now());
Files.lines(readme, Charset.forName("UTF-8"))
    .forEach(line->System.out.println(line));
```



Backup time: 2019-05-10T14:43:32.017926Z

❖ Operations `lines` and `writeString` enable instantaneous text file read and write capabilities.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

28

`createDirectory` operation throws an exception when trying to create a subfolder within a non-existent parent folder.

`createDirectories` operation creates parent folders before creating child folders.

Create Temporary Files and Folders

Class `Files` provides operations to create temporary files and folders.

- Temporary directories or files are created inside the default temporary file directory.
- Operations `createDirectory`, `createDirectories`, `createFile`, `createTempDirectory`, `createTempFile` all accept an optional var-arg `FileAttribute` parameter to describe any required file or folder properties, such as times, access permissions, or ownership.
- You are supposed to delete temporary files and folders once they are no longer required.

```
Path p1 = Files.createTempDirectory("TEMP");
Path p2 = Files.createTempFile(p1, "TEMP", ".tmp");
Files.deleteIfExists(p2);
Files.deleteIfExists(p1);
```

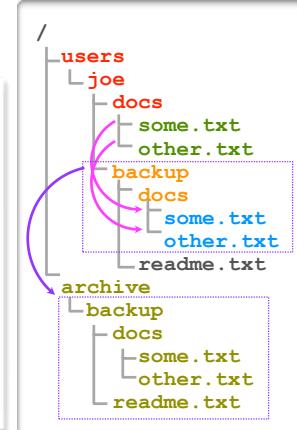


Copy and Move Paths

Class `Files` provides operations to copy and move folders and files:

- `copy` creates a replica of the path object.
- `move` deletes a path after copying it.

```
Path source = Path.of("docs");
Path backup = Path.of("backup");
Path archive = Path.of("/archive");
Files.list(source).forEach(file -> {
    try {
        Files.copy(file, backup.resolve(file),
                   StandardCopyOption.COPY_ATTRIBUTES,
                   StandardCopyOption.REPLACE_EXISTING);
    } catch (IOException ex) {
        logger.log(Level.SEVERE, "Error copying file", ex);
    }
});
Files.move(backup, archive, StandardCopyOption.COPY_ATTRIBUTES,
          StandardCopyOption.REPLACE_EXISTING,
          StandardCopyOption.ATOMIC_MOVE);
```



- ❖ Copying a folder does not copy its subfolder and files.
- ❖ Moving a folder does move all of its subfolder and files and auto-creates a target folder.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

30

Optionally when copying or moving path object, you may set the following options:

`StandardCopyOption.COPY_ATTRIBUTES` copies all attributes, such as permissions.

`StandardCopyOption.REPLACE_EXISTING` replaces existing paths rather than throw an exception.

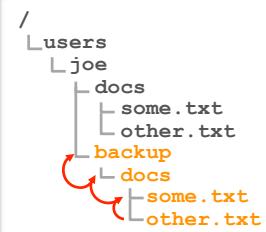
`StandardCopyOption.ATOMIC_MOVE` treats the entire move operations as an atomic action, which may not partially succeed or fail.

Delete Paths

Class `Files` provides operations to delete folders and files:

- `delete` and `deleteIfExists` delete files and folders.
- Attempting to delete a non-empty directory throws an exception.

```
Path backup = Path.of("backup");
Files.walk(backup)
    .sorted(Comparator.reverseOrder())
    .forEach(path->{
        try {
            Files.delete(path);
        } catch (IOException ex) {
            logger.log(Level.SEVERE, "Error deleting file", ex);
        }
    });
});
```

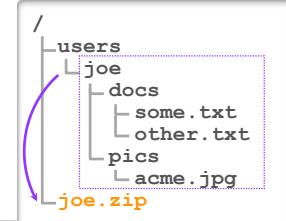


Handle Zip Archives

`ZipInputStream` and `ZipOutputStream` enable reading and writing zip files.

- You may create and extract `ZipEntry` objects using these streams.

```
Path joe = Path.of(".");
Path zip = Path.of("/joe.zip");
Files.createFile(zip);
try (ZipOutputStream out = new ZipOutputStream(Files.newOutputStream(zip))) {
    out.setLevel(Deflater.DEFAULT_COMPRESSION);
    Files.walk(joe.filter(p -> !Files.isDirectory(p)))
        .forEach(p -> {
            ZipEntry zipEntry = new ZipEntry(source.relativize(p).toString());
            try {
                out.putNextEntry(zipEntry);
                out.write(Files.readAllBytes(p));
                out.closeEntry();
            } catch (Exception e) {
                logger.log(Level.SEVERE, "Error creating zip entry", e);
            }
        });
} catch(IOException e) {
    logger.log(Level.SEVERE, "Error creating zip archive", e);
}
```



✿ The example shows all files and folders are copied from `joe` to the `joe.zip` file.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

32

The example shows extracting content from zip archive:

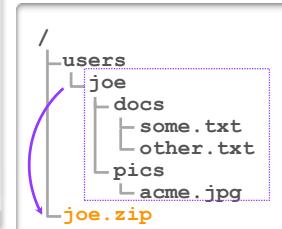
```
Path target = Path.of("extract_to");
Path zip = Path.of("/joe.zip");
try (ZipInputStream in = new
ZipInputStream(Files.newInputStream(zip))) {
    ZipEntry e = null;
    while((e = in.getNextEntry()) != null) {
        Path p = Paths.get(target.toString(), e.getName());
        if (e.isDirectory()) {
            Files.createDirectories(p);
        } else {
            Files.copy(in, p,
StandardCopyOption.REPLACE_EXISTING);
        }
        in.closeEntry();
    }
} catch (IOException e) {
    logger.log(Level.SEVERE, "Error extracting zip
archive", e);
}
```

Represent Zip Archive as a FileSystem

FileSystem mechanism can be used to represent archives.

- You may create, copy, move, delete, and navigate archived paths just like any other paths.

```
Path joe = Path.of(".");
Path zip = Path.of("/joe.zip");
Files.create(zip);
try (FileSystem fs = FileSystems.newFileSystem(zip, null)) {
    Files.walk(joe).forEach(source -> {
        try {
            Path target = fs.getPath("/"+source.toString());
            Files.copy(source, target);
        } catch (Exception e) {
            logger.log(Level.SEVERE, "Error archiving file", ex);
        }
    });
} catch (IOException e) {
    logger.log(Level.SEVERE, "Error creating archive", ex);
}
```



- In the example, all files and folders are copied from joe to the joe.zip file.
- Unfortunately, the zip file system provider is not implemented to compress files as they are when added to the archive.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

33

There are different ways of creating a new zip file system:

By using the JAR URL syntax:

```
Map<String, String> env = new HashMap<>();
env.put("create", "true"); // create zip file if it does not exist
URI uri = URI.create("jar:file:/joe.zip");
FileSystem fs = FileSystems.newFileSystem(uri, env);
```

By specifying a path and using automatic file type detection:

```
Path zip= Paths.get("/joe.zip");
Files.create(zip);
FileSystem fs = FileSystems.newFileSystem(zip, null); // second parameter (null) allows to set class loader for the jar file
```

The example shows files extracted from zip archive into a target folder:

```
Path zip = Path.of("/joe.zip");
Path target = Path.of("extract_here");
try ( FileSystem fs = FileSystems.newFileSystem(zip,
null)) {
    Files.walk(fs.getPath("/")).forEach(source -> {
        try {
            Files.copy(source,
Path.of(target.toString() + source.toString()),
StandardCopyOption.COPY_ATTRIBUTES,
StandardCopyOption.REPLACE_EXISTING);
        } catch (Exception e) {
            logger.log(Level.SEVERE, "Error extracting zip
entry", e);
        }
    });
} catch (Exception e) {
    logger.log(Level.SEVERE, "Error extracting zip
archive", e);
}
```

Access HTTP Resources

Java IO is a foundation of many other APIs, such as HTTP Client API.

- Classes in `java.net.http` package provide HTTP client functionalities.
- Handle HTTP methods: GET/POST/PUT/DELETE/HEAD/OPTIONS
- Supports authentication, encryption, and proxies
- Supports both synchronous and asynchronous modes
- Supports HTTP protocol versions 1.1 and 2

`module-info.java`

```
module demos {
    requires java.net.http;
}
```

```
Path path = Path.of("docs", "index.html");
URI = URI.create("http://openjdk.java.net");
HttpRequest req = HttpRequest.newBuilder(uri).GET().build();
HttpClient client = HttpClient.newHttpClient();
HttpResponse<Path> res = client.send(req, HttpResponse.BodyHandlers.ofFile(path));
```

- ❖ The example shows an html document downloaded into a file.
- ❖ Use of the `module-info` class is covered in a Modules lesson.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

35

Java HTTP access API is part of the `java.net.http` module.
Development and deployment of modularized Java applications are covered in a Modules lesson.

Legacy http connectivity API available from `java.net` package:

```
Path p1 = Path.of("docs", "index.txt");
URL url = URI.create("http://openjdk.java.net").toURL();
Files.copy(url.openStream(), p1);
```

Summary

In this lesson, you should have learned how to:

- Describe Java I/O
- Read and write text and binary data
- Use Java serialization
- Work with the file system



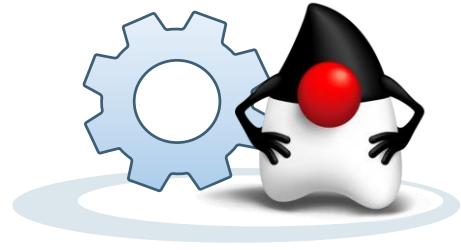
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

36

Practices

In this practice, you will:

- Write products and reviews reports to text files
- Bulk-load data from comma-delimited files
- Serialize and deserialize Java objects using temporary files



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

37

Java Concurrency and Multithreading



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Describe multithreading
- Manage thread life cycle and execution order
- Automate management and execution of concurrent tasks
- Ensure thread-safety using volatile variables, atomic actions, and locks



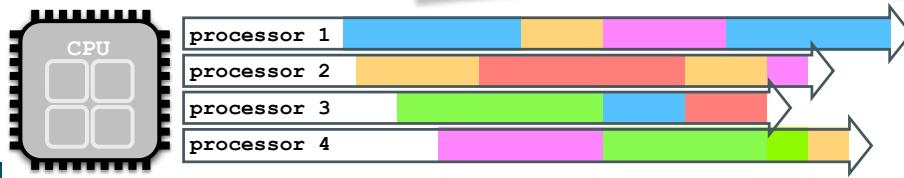
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Java Concurrency Concepts

Java thread is an execution path.

- Thread actions are implemented using method `run` of a `Runnable` interface.
- Thread is scheduled to run using `start` method of a `Thread` class.
- Thread scheduler will allocate portions of CPU time (time-slice) to execute thread actions.
- Java threads time-slice hardware threads (processors) provided by the CPU cores and can be interrupted at any time to give way to another thread, making the order of actions performed by different threads stochastic.
- The return of method `main` or `run` terminates the thread.



```
public class Test {
    public static void main(String[] args) {
        Lateral la = new Lateral();
        new Thread(la).start();
        new Thread(la).start();
        new Thread(la).start();
        new Thread(la).start();
        // main thread actions
    }
}
```

```
public class Lateral implements Runnable {
    public void run() {
        // thread actions
    }
}
```

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

The term "Parallelism" describes different execution paths that run simultaneously, such as when executed using multiple CPU cores.

The term "Concurrency" describes different execution paths that look like they run simultaneously, which may or may not be really the case.

Java thread is an execution path that feels like it is performed in parallel with other execution paths. It may or may not be really parallel, depending on how many physical hardware threads your computer provides or how busy your CPU is.

When Java program starts, at least one thread is created. This thread executes method `main`. Other threads within your java runtime include a garbage collector and any other threads that you have started.

Modern processors provide a number of CPU cores, with one or more hardware threads per core, which represent your machine's physical capacity to execute code in parallel. You may create more threads in your program, but they will have to "time-share" physical hardware threads.

In the example, you can find out how many hardware threads your JVM has access to:

```
Runtime r = Runtime.getRuntime();
int numHardwareThreads = r.availableProcessors();
```

Note that to simplify hardware representation, no distinction is made between different physical CPUs and individual CPU cores. Instead, "hardware threads" are represented just as "processors". For example, running `availableProcessors` method on a single i5-7360U CPU machine, which has 2 cores and 2 hardware threads per core, yields the result of 4.

Implement Threads

- Describe thread actions
- Instantiate thread object
- Schedule thread to run

✖ Common practice

```
// Create class that implements Runnable
public class Lateral implements Runnable {
    public void run() {...}
}
// Pass Runnable object to Thread constructor
Thread t = new Thread(new Lateral());
// Schedule thread to run
t.start();
```

```
// Create class that extends class Thread
// (which already implements Runnable)
public class Lateral extends Thread {
    public void run() {...}
}
// Instantiate your class
Thread t = new Lateral();
// Schedule thread to run
t.start();
```

✖ Not a flexible design and thus not recommended

✖ Good for small amount of actions

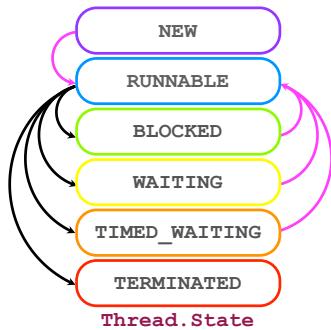
```
// Implement Runnable using a Lambda Expression
Runnable r = () ->{...};
// Pass Runnable object to Thread constructor
Thread t = new Thread(r);
// Schedule thread to run
t.start();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Thread Life Cycle



Thread has not yet started.

Thread is executing.

Thread is blocked, waiting for a monitor lock to be released.

Thread is waiting as long as it takes for another thread's signal.

Thread is waiting for a specified period of time.

Thread has exited.

Transitions to the running state are not immediate
- thread scheduler needs to allocate next available CPU time slot for this thread.

- ❖ The same Runnable instance can be wrapped up by more than one Thread instance and stated as different threads.
- ❖ You can verify if thread has not yet terminated and its life cycle phase.
- ❖ Same Thread instance cannot be started twice.

```

Runnable r = () -> {
    /* run method implementing thread logic */
};

Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
t1.start();
t2.start();
boolean x = t2.isAlive();
Thread.State phase = t1.getState();
x t1.start();
    
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

An attempt to start the same thread object twice will throw a `java.lang.IllegalThreadStateException`.

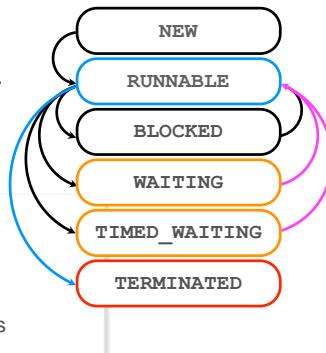
Method `getState` is designed for use in monitoring of the system state, not for synchronization control.

Interrupt Thread

Logic of a `run` method is in charge of making life cycle decisions.

- A thread in a `Runnable` state may check if it has received an interrupt signal.
- A thread that has entered a `Waiting` or `timed waiting` state must catch `InterruptedException`, which puts it back to `Runnable` state, and then decide what it should do.

```
Runnable r = () -> {
    Thread ct = Thread.currentThread(); // locate current thread object
    while(!ct.isInterrupted()) { // check interrupt signal when running
        // perform thread actions
        try {
            Thread.sleep(1000); // enter timed waiting state for 1000 milliseconds
        }catch(InterruptedException ex) {
            // perform interrupted when waiting actions
            return;
        }
    } // getting to the end of the run method terminates the thread
Thread t = new Thread(r);
t.start();
t.interrupt(); // when thread is in the running state it is not forced to check this signal
```



- When handling an interrupt signal, thread may choose to terminate (get to the end of run method) or it may decide to continue.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

If thread is in running state, it is its own responsibility to check the interrupt signal and decide whether it should terminate or not. Threads that enter a waiting or timed waiting state must catch `InterruptedException`. When exception is thrown and a control is passed to the exception handler, thread goes back to running state and again can make a decision whether to terminate or not.

After the timed wait expires, thread goes back to the running state. However, it would not happen at that very moment, but rather when the thread scheduler would allocate the next variable CPU time slot for this thread.

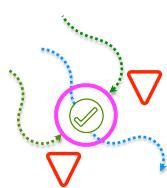
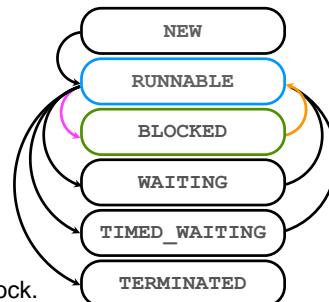
Block Thread

Monitor object helps to coordinate order of execution of threads.

- Any object or a class can be used as a monitor.
- It allows threads to enter blocked or waiting states.
- It enables mutual exclusion of threads and signaling mechanisms.

Keyword `synchronized` enforces exclusive access to the block of code.

- **Thread that first enters the synchronized block remains in runnable state.**
- **All other threads accessing the same block enter the blocked state.**
- When a runnable thread exits the synchronized block, the lock is released.
- **Another thread is now allowed to enter the runnable state and place a new lock.**



Monitors in the example are:

- Current object (`this`) for the `a` method
- `Some.class` for the `b` method
- Object `s` for the synchronized block invoking `c` method

```
public class Some {
    public void synchronized a() { }
    public static void synchronized b() { }
    public void c() { }
}
```

```
Some s = new Some();
Runnable r = () -> {
    s.a();
    Some.b();
    synchronized (s) {
        s.c();
    }
};
new Thread(r).start();
new Thread(r).start();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Keyword `synchronized` enforces exclusive access to an object or a class.

Establishes the order of threads accessing a locked object (monitor object) known as intrinsic lock

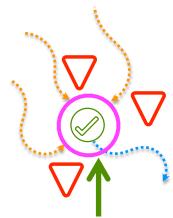
Ensures that a thread completes the sequence of actions within the synchronized block (even if it was interrupted and then resumed) before another thread is allowed to enter this block of code

Make Thread Wait Until Notified

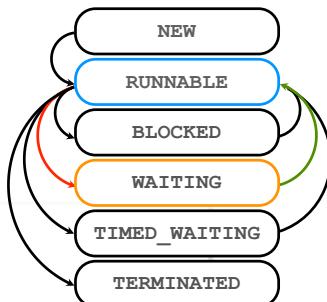
Suspend a thread waiting indefinitely.

- Method `wait` puts a thread into **waiting state** against **specific monitor**.
- Any number of threads can be waiting against the same monitor.
- Method `notify` wakes up one of the waiting threads (stochastic).
- Method `notifyAll` wakes up all waiting threads.

❖ Methods `wait/notify/notifyAll` must be invoked within synchronized blocks against the same monitor.



```
Object obj = new Object();
Runnable r = () -> {
    try {
        synchronized (obj) {
            obj.wait();
        }
    } catch (InterruptedException ex) { ... }
};
Thread t = new Thread(r);
t.start();
try {
    Thread.sleep(1000);
} catch (InterruptedException ex) { ... }
synchronized (obj) {
    obj.notify();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Common Thread Properties

- Thread could be given a **custom name** using constructor and set/get name methods.
- Thread has a **unique id**.
- Thread can be marked as a **daemon** or a **user** (default) thread.
- Thread may **wait for another thread to terminate**.
- Thread could be assigned a **priority**.

<code>Thread.MAX_PRIORITY</code>	10	
<code>Thread.MIN_PRIORITY</code>	1	
<code>Thread.NORM_PRIORITY</code>	5	(default)

- Priority determines the number of CPU time slots the thread scheduler allocates to this thread, but it cannot guarantee the order of execution.

```
Runnable r = () -> { /* do work */ };
Thread t = new Thread(r, "My Thread");
t.setDaemon(true);
t.start();
long id = t.getId();
if (t.isDaemon()) {
    /* it will auto-terminate once all user threads have terminated */
}
t.setPriority(3);
try {
    t.join(); // wait for the thread to terminate
} catch (InterruptedException ex) { }
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

The thread ID is a positive long number generated when this thread was created. The thread ID is unique and remains unchanged during its lifetime. When a thread is terminated, this thread ID may be reused.

Create Executor Service Objects

Class `java.util.concurrent.Executors` provides a number of thread management automations using different `ExecutorService` objects:

- **Fixed Thread Pool** reuses a fixed number of threads.
- **Work Stealing Pool** maintains enough threads to support the given parallelism level.
- **Single Thread Executor** uses a single worker thread.
- **Cached Thread Pool** creates new threads as needed or reuses existing threads.
- **Scheduled Thread Pool** schedules tasks to execute with a delay and/or periodically.
- **Single Thread Scheduled Executor** schedules tasks to execute with a delay using a single worker thread.
- **Unconfigurable Executor Service** provides a way to "freeze" another Executor Service configuration.

```
Runnable task = () -> { /* perform concurrent actions */ };
ScheduledExecutorService ses = Executors.newScheduledThreadPool(3);
ses.scheduleAtFixedRate(task, 10, 5, TimeUnit.SECONDS);
ExecutorService es = Executors.unconfigurableExecutorService(ses);
```

Example

- ❖ Creates a `ScheduledExecutorService` with a pool of 3 threads
- ❖ Schedules a `Runnable` task to be executed very 5 seconds with initial delay of 10 seconds
- ❖ Freezes the configuration of the executor service to prevent any changes

❖ Schedule one or more tasks, with different delays and periods using the same thread pool.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Fixed Thread Pool reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most `nThreads` threads will be active processing the tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly shut down.

For example, creating a fixed size pool of 10 threads:

```
ExecutorService es = Executors.newFixedThreadPool(10);
```

Work Stealing Pool maintains enough threads to support the given parallelism level and may use multiple queues to reduce contention. The parallelism level corresponds to the maximum number of threads actively engaged in, or available to engage in, task processing. The actual number of threads may grow and shrink dynamically. A work-stealing pool makes no guarantees about the order in which the submitted tasks are executed.

For example, creating a work stealing thread pool with parallelism level of up to four threads:

```
ExecutorService es = Executors.newWorkStealingPool(4);
```

Single Thread Executor uses a single worker thread operating off an unbounded queue. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent newFixedThreadPool(1), the returned executor is guaranteed not to be reconfigurable to use additional threads.

For example, creating a single thread executor:

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

Cached Thread Pool creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using ThreadPoolExecutor constructors.

For example, creating a cached thread pool:

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

Single Thread Scheduled Executor schedules commands to run after a given delay or to execute periodically. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent newScheduledThreadPool(1), the returned executor is guaranteed not to be reconfigurable to use additional threads.

For example, creating a single thread scheduled executor service:

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

Scheduled Thread Pool creates a thread pool that can schedule commands to run after a given delay or to execute periodically.

For example, creating a scheduled executor thread pool:

```
ScheduledExecutorService es =  
Executors.newScheduledThreadPool(2);
```

Scheduled executor service allows to schedule an execution of one or more tasks using the following methods:

- Submit a one-shot task that becomes enabled after the given delay:

```
ScheduledFuture<?> schedule(Runnable command, long delay,  
TimeUnit unit)
```

- Submit a value-returning one-shot task that becomes enabled after the given delay:

```
<V> ScheduledFuture<V> schedule(Callable<V> callable,  
long delay, TimeUnit unit)
```

- Submit a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is, executions will commence after initialDelay, then initialDelay + period, then initialDelay + 2 * period, and so on:

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,  
long initialDelay, long period, TimeUnit unit)
```

- Submit a periodic action that becomes enabled first after the given initial delay and subsequently with the given delay between the termination of one execution and the commencement of the next:

```
ScheduledFuture<?> scheduleWithFixedDelay(Runnable  
command, long initialDelay, long delay, TimeUnit unit)
```

ScheduledFuture represents delayed result-bearing action that can be canceled. For example:

```
Runnable task = () -> { /* perform concurrent actions */  
};  
  
ScheduledExecutorService es =  
Executors.newScheduledThreadPool(2); /* create pool of 2  
threads */  
  
ScheduledFuture result = es.schedule(task, 10,  
TimeUnit.SECONDS); /* schedule a runnable task to be executed in 10  
seconds */  
  
long delay = result.getDelay(TimeUnit.MILLISECONDS); /* get  
delay amount in milliseconds */  
  
Thread.sleep(delay*3); /* wait for 3 times as long as the delay amount  
for the task to complete its actions */  
  
if (!(result.isDone() || result.isCancelled())) { /* check if  
the task has not yet been completed or canceled */  
  
    result.cancel(true); /* request the task to be interrupted */  
}
```

Unconfigurable Executor Service delegates all defined ExecutorService methods to the given executor, but not any other methods that might otherwise be accessible using casts. This provides a way to safely "freeze" configuration and disallow tuning of a given concrete implementation.

Unconfigurable executor produces a wrapper around the existing executor service that preserves the configuration:

```
ExecutorService es =  
Executors.unconfigurableExecutorService(<other executor  
service>);
```

Manage Executor Service Life Cycle

Use an `ExecutorService` to `start`, `stop` accepting new, `wait` for completion, and `stop` concurrent tasks.

```
/* create pool of 3 threads */
ExecutorService es = Executors.newFixedThreadPool(3);
/* launch 10 Runnable tasks with up to 3 running at any time */
for (int i = 0; i < 10; i++) {
    es.execute(()->{
        /* perform concurrent actions and check for interruption */
    });
}
es.shutdown(); /* stop accepting new tasks */
try {
    /* wait for existing tasks to terminate and check if they all have actually stopped */
    if (!es.awaitTermination(30, TimeUnit.SECONDS)){
        /* request cancelation of tasks that are still running */
        es.shutdownNow();
    }
} catch(InterruptedException e){
    /* request cancelation of runnings tasks when launcher thread was interrupted */
    es.shutdownNow();
    Thread.currentThread().interrupt(); /* continue launcher thread interrupt process */
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via `Thread.interrupt()`, so any task that fails to respond to interrupts may never terminate.

Example of controlling Runnable tasks assigned to be performed by a fixed size pool of threads:

```

Runnable task = () -> {
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 10; i++) {
        if(t.isInterrupted()) { return; } // check if thread has
        received an interrupt signal
        System.out.println(name+' '+i+' '+Instant.now()); // perform actions
    }
}

ExecutorService es = Executors.newFixedThreadPool(3); // Create a pool of 3 threads
for (int i = 0; i < 10; i++){
    es.execute(task); // Use threads from a pool to execute a runnable task
}
es.shutdown(); // Disable new tasks from being submitted
boolean allTasksTerminated = false; // Indicator showing if all Runnable tasks have completed or not
try {
    allTasksTerminated = es.awaitTermination(10,
    TimeUnit.SECONDS); // wait for existing tasks to terminate
} catch (InterruptedException ex) {
    System.out.println("Thread main was interrupted");
} finally{
    if(!allTasksTerminated) {
        es.shutdownNow(); // Cancel any tasks that are still running
    }
}
System.out.println("All thread terminated "+allTasksTerminated);

```

Implementing Executor Service Tasks

ExecutorService supports two types of task implementations:

- Runnable objects
 - Implementing `public void run();` method of `Runnable` interface
 - Launched using `execute` or `submit` methods of the `ExecutorService`
- Callable objects
 - Implementing `public <T> call()` throws `Exception`; method of `Callable` interface
 - Launched using `submit` method of the `ExecutorService`
 - Returned value is wrapped into the `Future` object, which is returned immediately.
 - Method `get` blocks invoking thread until timeout or when the value within the `Future` object becomes available.

```
Callable<String> t = new Callable<>() {
    public String call() throws Exception {
        /* perform concurrent actions */
        return "some value";
    }
}
ExecutorService es = Executors.newFixedThreadPool(10);
Future<String> result = es.submit(t);
try {
    String value = result.get(10, TimeUnit.SECONDS);
} catch(Exception e) { /* see notes for details */ }
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

When a `Runnable` object is passed to `ExecutorService submit` method, the returned `Future` object is not expected to actually contain any value, but can be used to find out when the `run` method completes processing.

When getting a value from the `Future` result object, the following exception can occur:

```
Callable<String> t = new Callable<>() {
    public String call() throws Exception {
        /*concurrent actions*/
        return "some value";
    }
}
ExecutorService es = Executors.newFixedThreadPool(10);
Future<String> result = es.submit(t);
try {
    /* Method get return the value immediately if it was already produced by the
       call method
}
```

```
otherwise it blocks the invoker until the value would be produced */

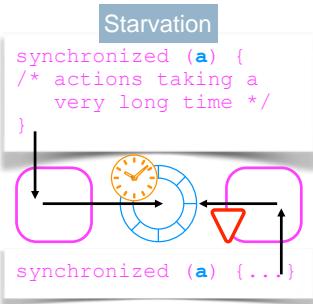
String value = result.get();

//String value = result.get(10, TimeUnit.SECONDS); // 
wait duration can be limited

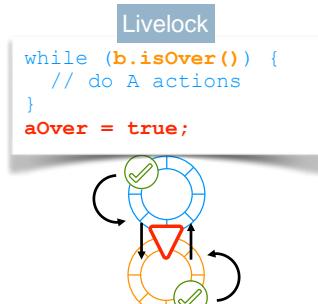
} catch(InterruptedException e) {
    /* exception indicates that the current thread was
interrupted while waiting */
} catch(ExecutionException e) {
    /* exception indicates that the call method threw an
exception*/
} catch(TimeoutException e) {
    /* exception indicates that the wait timed out */
}
```

You may also catch a runtime `CancellationException`, which indicates that the computation was canceled.

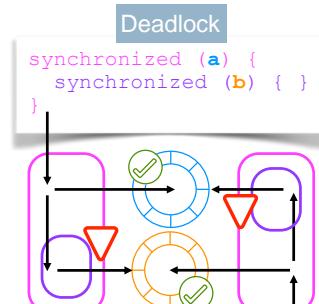
Locking Problems



Thread is waiting for a resource blocked by another busy thread.



Threads form an indefinite loop, expecting confirmation of completion from each other.



Two or more threads are blocked forever, waiting for each other.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Well-designed software should embrace the stochastic nature of the concurrently executed code. This means that it's best to write code that is not dependent upon the order in which concurrent actions are executed, since such an order is inherently unpredictable. The order of executions should not really be a problem, or cause any issues, as long as a thread does not try to access shared mutable objects. Threads that attempt to write data outside their own context become dependent on the order of execution, or timing, or on other uncontrollable events that may occur concurrently. Such dependency is called a "race condition" and should be avoided.

It is theoretically possible to try to force the order of concurrent actions using synchronization and locks. However, this approach is inherently problematic, due to increased code complexity, performance, and scalability issues.

Writing Thread-Safe Code

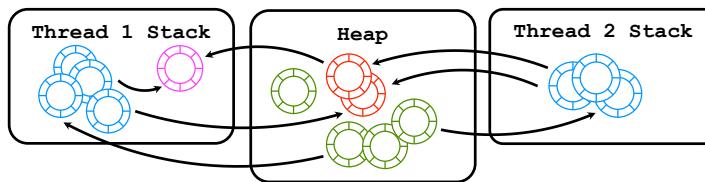
Stack values such as **local variables and method arguments are thread-safe**.

- Each thread operates with its own stack.
- No other thread can see this portion of memory.

Immutable objects in a shared heap memory are thread-safe because they cannot be changed at all.

Mutable objects in a shared heap memory are thread-unsafe.

- Heap memory is shared between all threads.
- Heap values undergoing modifications may be
 - Inconsistent - observed by other threads before modification is complete
 - Corrupted - partially changed by another thread writing to memory at the same time
- Compiler may choose **cache heap value locally** within a thread, causing a thread not to notice that data has been changed by another thread.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

Memory inconsistencies and corruption may occur as a result of different threads writing to the same heap object at the same time or partially completing write actions because of thread interruptions.

Example of shared memory write problem:

```
List<String> list = new ArrayList<>(); // this list is shared
between a number of threads and it is not immutable

Runnable r = () -> {
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 5; i++) {
        list.add(name+' '+i); // adding elements to the list may corrupt
        memory when performed by different threads
    }
};

for (int i = 0; i < 10; i++) {
    new Thread(r).start();
}
```

Due to code optimizations, thread can read the value from the local cache rather than from the main memory, resulting in it missing changes that another thread could have applied to the main memory.

Example of locally cached value problem:

```
public class Shared {  
    public int x;  
    public int y;  
}  
  
Shared s = new Shared();  
  
new Thread(() -> {  
    while(s.y < 1) { // compiler may choose to cache s.y  
        as a local value within the thread memory context  
        int x = s.x;  
    }  
}).start();  
  
new Thread(() -> {  
    s.x = 2;  
    s.y = 2; // when s.y is updated another thread may  
    not detect this change, if it has cached the old value  
}).start();
```

Ensure Consistent Access to Shared Data

Disable compiler optimization that is caching the shared value locally within a thread.

Keyword `volatile` instructs Java compiler:

- Not to cache the variable value locally
- Always read it from the main memory
- Applies all changes to the main memory that occurred in a thread before the update of the volatile variable

```
public class Shared {  
    public int x;  
    public volatile int y;  
}
```

```
Shared s = new Shared();  
new Thread(() -> {  
    while(s.y < 1){  
        int x = s.x;  
    }  
}).start();  
new Thread(() -> {  
    s.x = 2;  
    s.y = 2;  
}).start();
```

- ❖ The while loop in the example could become indefinite without the `volatile` instruction if compiler chooses to cache variable `y` locally.
- ❖ Even with the `volatile` keyword, it is not really possible to predict how many iterations this while loop is going to perform, because there is no way to tell the order in which these threads would get CPU time to execute their instructions.

- ❖ The example uses lambda expressions to implement method `run` of the `Runnable` interface.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

Runnable interface has only one abstract method that you must implement; therefore, it can be implemented using lambda expression:

`new Thread(() ->{/*thread actions*/}).start();`

instead of writing verbose interface implementation code:

```
public class X implements Runnable {  
  
    public void run() {  
        /*thread actions*/  
    }  
  
    X = new X();  
    Thread t = new Thread(x);  
    t.start();
```

Non-Blocking Atomic Actions

Action is atomic if it is guaranteed to be performed by a thread without an interruption.

- Atomic actions cannot be interleaved.
- Only actions performed by a CPU in a single cycle are by default atomic.
- Variable assignments are atomic actions, except `long` and `double`; these are 64-bit values, and it takes more than a single step to assign these on a 32-bit platform.
- Other operations such as `+` `-` `*` `%` `++` `--` etc. are not atomic.
- Package `java.util.concurrent.atomic` provides classes that implement lock-free thread-safe programming of atomic behaviors on single variables, for example:
 - `AtomicBoolean`
 - `AtomicInteger`
 - `AtomicLong`
 - `AtomicReference<V>`
- Atomic variables also behave as if they are volatile.

```
public class Shared {
    public AtomicInteger x = new AtomicInteger(0);
}
```

```
Shared s = new Shared();
Runnable r = () -> {
    int y = 0;
    while(y < 10) {
        y = s.x.incrementAndGet();
    }
};
new Thread(r).start();
new Thread(r).start();
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

The example demonstrates non-atomic actions that can be interrupted, resulting in memory inconsistencies in a multithreaded environment.

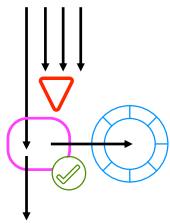
```
public class Shared {
    public int x;
}

Shared s = new Shared();
Runnable r1 = () -> {
    int z = 0;
    while(z < 10) {
        z = ++s.x; // this is not an atomic action
    }
};
```

Ensure Exclusive Object Access Using Intrinsic Locks

Use intrinsic lock to enforce an exclusive access to a shared object.

- Order of execution and object consistency are ensured.
- Synchronized logic creates a bottleneck in a multithreaded application.
- Performance and scalability can be significantly degraded.



```
List<String> list = new ArrayList<>();  
Runnable r = () -> {  
    String name = Thread.currentThread().getName();  
    for (int i = 0; i < 10; i++) {  
        synchronized(list){  
            list.add(name+' '+i);  
        }  
    }  
    for (int i = 0; i < 10; i++) {  
        new Thread(r).start();  
    }  
}
```

✖ The example ensures that only one thread at a time is allowed to add elements to the list.



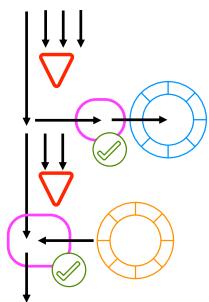
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

24

Intrinsic Lock Automation

Some Java APIs provide synchronized versions of objects, for example:

- Class `Collections` provides synchronized wrappers for Collection, List, Set, and Map objects.
- Operations such as add and remove are already synchronized to ensure consistent access to the collection content.



```
List<String> list = new ArrayList<>();
List<String> sList = Collections.synchronizedList(list);
Runnable r = () -> {
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 10; i++) {
        sList.add(name+ ' +i);
    }
}
/* start threads and wait for their completions (see full code in notes) */
synchronized (sList) {
    Iterator i = sList.iterator();
    while (i.hasNext())
        System.out.println(i.next());
}
```

✖ Iterating through the synchronized collection would still require an explicit synchronized block.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

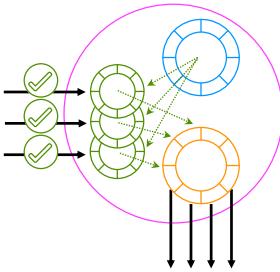
The example shows operating with a synchronized list:

```
List<String> list = new ArrayList<>();
List<String> sList = Collections.synchronizedList(list);
Runnable r = () -> {
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 10; i++) {
        sList.add(name+ ' +i);
    }
};
Thread[] threads = new Thread[10];
for (int i = 0; i < threads.length; i++) {
    threads[i] = new Thread(r);
    threads[i].start();
}
```

```
for (Thread t : threads) {  
    try {  
        t.join();  
    } catch (InterruptedException ex) { }  
}  
  
synchronized (sList) {  
    Iterator i = sList.iterator();  
    while (i.hasNext())  
        System.out.println(i.next());  
}
```

Non-Blocking Concurrency Automation

- Package `java.util.concurrent` provides classes to manage concurrency.
- For example, classes such as `CopyOnWriteArrayList` or `CopyOnWriteArraySet` provide thread-safe variants of `List` and `Set`.
 - All mutative operations (add, remove, and so on) make fresh copies of the underlying collection.
 - The read-only snapshot of merge content is used for traversal.



```
List<String> list = new ArrayList<>();
List<String> copyOnWriteList = new CopyOnWriteArrayList<(list>;
Runnable r = () -> {
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 10; i++) {
        copyOnWriteList.add(name + ' ' + i);
    }
};
/* start threads and wait for their completions (see full code in notes) */
Iterator i = copyOnWriteList.iterator();
while (i.hasNext())
    System.out.println(i.next());
}
```

- ❖ It is best suited for small collections, where read-only operations vastly outnumber mutative operations and prevent interference among threads during traversal.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

27

The example shows operating with a copy-on-write list:

```
List<String> list = new ArrayList<>();
List<String> copyOnWriteList = new
CopyOnWriteArrayList<(list);
Runnable r = () -> {
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 10; i++) {
        copyOnWriteList.add(name + ' ' + i);
    }
};
Thread[] threads = new Thread[10];
for (int i = 0; i < threads.length; i++) {
    threads[i] = new Thread(r);
    threads[i].start();
}
for (Thread t : threads) {
    try {
        t.join();
    } catch (InterruptedException ex) { }
}
Iterator i = sList.iterator();
while (i.hasNext())
    System.out.println(i.next());
}
```

Alternative Locking Mechanisms

Locking API provides more flexible programmatic concurrency control mechanisms.

- Allows actions to be performed on an object, without interference from other threads
- Available from the `java.util.concurrent.locks` package
- Write lock prevents other threads from concurrently modifying the object
- Read lock can be acquired if Write lock is not held by another thread, allowing concurrent read actions

```
public class PriceList {
    private List<Product> menu = new ArrayList<>();
    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private Lock rl = rwl.readLock();
    private Lock wl = rwl.writeLock();
    public Product get(int id) {
        rl.lock();
        try { return menu.stream().findFirst(p->p.getId()==id); }
        finally { rl.unlock(); }
    }
    public void add(Product product) {
        wl.lock();
        try { return menu.add(product); }
        finally { wl.unlock(); }
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

28

Only one Write lock at a time can be acquired for the given object. Read locks do not interfere with each other.

It is possible to allow a program to fail rather than be stuck if the lock cannot be acquired within a reasonable time frame.

```
public List<Product> getAll() {
    if (!(r.tryLock() || r.tryLock(10,
    TimeUnit.MILLISECONDS)) {
        return null;
    }
    try {
        return Collections.unmodifiableList(menu);
    }finally{
        r.unlock();
    }
}
```

Summary

In this lesson, you should have learned how to:

- Describe multithreading
- Manage thread life cycle and execution order
- Automate management and execution of concurrent tasks
- Ensure thread-safety using volatile variables, atomic actions, and locks



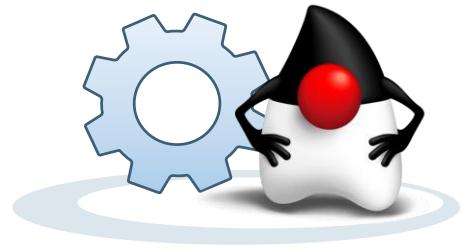
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

29

Practices

In this practice, you will:

- Simulate multiple concurrent callers that are going to share a single instance of the ProductManager
- Decouple locale management from the instance of the ProductManager to allow different concurrent callers to set their own locales without affecting others
- Protect your data cache (map of products and reviews) from corruption using appropriate concurrent collections and locking strategy
- Resolve filesystem path clashes that may occur when concurrent callers attempt to write same files (such as a report for the same product) at the same time



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

30

Java Modules



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Compare modular and non-modular Java applications
- Describe Java Platform Module System (JPMS)
- Create modules dependencies
- Define and use module services
- Create runtime images
- Deploy and execute modular Java applications



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Compile, Package, and Execute Non-Modular Java Applications

Legacy (non-modular) Java application deployment and execution:

- Compile Java classes using **javac** utility

```
javac -cp /project/classes:other.jar:some.jar  
      -d /project/classes  
      -sourcepath /project/sources
```

- Package Java class into Java Archive (JAR) using **jar** utility
- (Optionally) provide JAR descriptor MANIFEST.MF file
 - May contain name of the main application class
 - May set up class path to reference other archives containing classes required by this application

MANIFEST.MF

```
Manifest-Version: 1.0  
Main-Class: demos.shop.ShopApp  
Class-Path: other.jar some.jar
```

```
jar --create --file shop.jar  
      --manifest=META-INF/MANIFEST.MF  
      -C /project/classes .
```

- Execute Java Application using **java** runtime

```
java -jar shop.jar
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

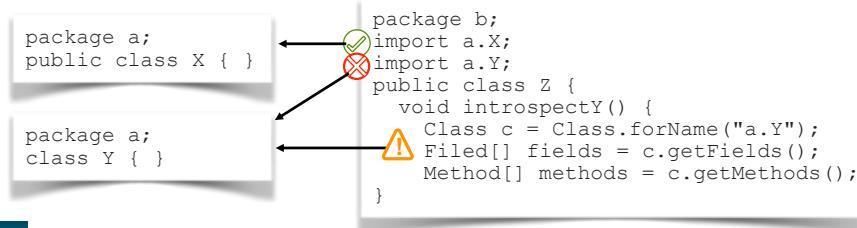
3

MANIFEST.MF file is placed in the META-INF folder inside JAR root directory.

Non-Modular Java Characteristics

There were no modules in Java before version 9.

- ✓ Packages provided logical grouping of classes.
- ⚠ Packages did not impose physical restrictions on how they are used.
- ✓ Classes are packaged into jar files and accessed via classpath.
- ⚠ Common deployment of related classes is not enforced.
- ✓ Visibility of classes is controlled with public and default access modifiers.
- ⚠ Encapsulation can always be bypassed using reflection.
- ⚠ Impossible to restrict in which exact other packages your code can be used



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

Prior to the introduction of modules, it was not possible to make some class visible to members of specific packages, but not others. Using default access modifier also does not prevent other classes from accessing this class and even its private members via reflection. Reflection allows access to object fields and methods, regardless of their access modifiers. This really means that no Java code is, strictly speaking, really encapsulated.

Not enforcing related classes to be deployed together can lead to maintenance issues, such as missing classes in the classpath or partial redeployment, when only some of the archives are redeployed, potentially rendering application unusable.

Note: Non-modular deployments (as simple jar files accessed via classpath) are still possible in Java.

What Is a Module?

Module is high-level code aggregation.

- It comprises one or more closely related packages and other resources such as images or xml files.
- Module descriptor `module-info.class` stored in the module's root folder contains:
 - Unique module name (recommended reverse-dns convention `com.yourcompany.whatever`)
 - Required module dependencies (other modules that this module depends on)
 - Packages that this module exports, making them available to other modules (all other packages contained within the module are unavailable to other modules)
 - Permissions to open content of this module to other modules using reflection
 - Services this module offers to other modules
 - Services this module consumes
 - Modules do not allow splitting java packages even when they are not exported (private).

```
module <this module name> {  
    requires <other module names>;  
    exports <packages of this module to other modules that require them>;  
    opens <packages of this module to other modules via reflection>;  
    uses <services provided by other modules>;  
    provides <services to other modules> with <service implementations>;  
    version <value>;  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Historically, Java allowed code to be grouped into packages and achieve encapsulation using access modifiers. This is a simple and time-proven method of achieving encapsulation. However, this approach has certain limitations. For example, it does not allow to limit visibility of a public class to just some other classes, but not others, or to limit access to an entire package.

Java SE 9 has introduced a new module system governed by a number of specifications:

- JEP 200 The Modular JDK
- JEP 201 Modular Source Code
- JEP 220 Modular Run-Time Images
- JEP 260 Encapsulate Most Internal APIs
- JEP 261 Module System
- JEP 275 Modular Java Application Packaging
- JEP 282 JLINK: The Java Linker
- JSR 376 Java Platform Module System
- JSR 379 Java SE 9

A module is a group of Java packages and resources such as images or xml files that can explicitly list which of its content it wants to make available to classes located in other modules. Also, a module can explicitly define what other modules it wants to use.

Modules help you to achieve:

Reliable configuration—Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner that's recognized both at compile time and execution time. The system can walk through these dependencies to determine the subset of all modules required to support your app.

Strong encapsulation—The packages in a module are accessible to other modules only if the module explicitly exports them. Even then, another module cannot use those packages unless it explicitly states that it requires the other module's capabilities. This improves platform security because fewer classes are accessible to potential attackers. You may find that considering modularity helps you come up with cleaner, more logical designs.

Scalable Java platform—Previously, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain, and evolve. It couldn't be easily subsetted. The platform is now modularized into 95 modules (this number might change as Java evolves). You can create custom run times consisting of only modules you need for your apps or the devices you're targeting. For example, if a device does not support GUIs, you could create a run time that does not include the GUI modules, significantly reducing the run time's size.

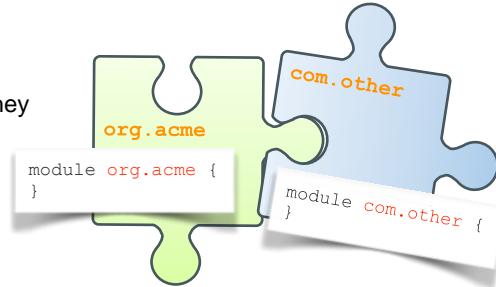
Greater platform integrity—Before Java 9, it was possible to use many classes in the platform that were not meant for use by an app's classes. With strong encapsulation, these internal APIs are truly encapsulated and hidden from apps using the platform. This can make migrating legacy code to modularized Java 9 problematic if your code depends on internal APIs.

Improved performance—The JVM uses various optimization techniques to improve application performance. JSR 376 indicates that these techniques are more effective when it's known in advance that the required types are located only in specific modules.

Java Platform Module System (JPMS)

Module is a group of packages.

- Starting from version 9, all JDK APIs were repackaged as modules.
- You can describe your own modules in the same way, using `module-info.java` descriptor file, placed into the `module root directory`, named after the `module name`.
- Module name must be unique and is typically named after the primary package contained within this module.
- Packages within the module are hidden by default unless they are exposed to specific modules or to all other modules.
- Module declares which other modules it uses:
 - Enables smaller application deployment footprint
 - Circular module dependencies are not allowed.
 - Dependencies between modules can be verified at application startup.
- Classes of the modularized applications are loaded using module-path, not class-path.
 - Deployment errors, such as missing modules, are detected at startup.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

The Java Platform Module System is also known as Project Jigsaw.

There is no need to deploy an entire set of JDK APIs with your application - only required modules have to be deployed.

Module root directory should be named after the module, i.e., `org.acme` folder contains `module-info.java` file that describes `org.acme` module

Alternatively, the `module-info` file can be placed in the root of the archive that contains the module.

JPMS Module Categories

Java SE Modules

- Core Java platform for general-purpose APIs
- Module names start with "java"
- Examples: `java.base`, `java.se`, `java.logging`

JDK Modules

- Additional implementation-specific modules
- Module names start with "jdk"
- Examples: `jdk.httpserver`, `jdk.jconsole`, `jdk.jshell`

Other Modules

- Create your own and use third-party modules.

```
java --list-modules
```

- ❖ Example: List the existing set of modules, supplied by the Java SE environment.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

When listing the existing modules, notice that they have a version indicator, for example, "@9" for Java 9.

Define Module Dependencies

Module defines which other modules it needs

- **requires <modules>** directive specifies a normal module dependency.

Example: Module "com.some" needs access to content provided by "java.logging" module.

- **requires transitive <modules>** directive makes dependent module available to other modules.

Example: Any module that uses "com.some" will also use "org.acme", without having to declare explicit dependency.

- **requires static <modules>** directive indicates module dependency at compile time only.

Example: Module "com.some" optionally uses module "com.foo"; this usage is not required at run time.

```
module com.some {  
    requires java.logging;  
    requires transitive org.acme;  
    requires static com.foo;  
}
```

❖ Note: These instructions accept comma-separated lists of module names.

❖ Note: Directive **requires java.base** is implied for all modules, but any other module has to be referenced explicitly.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

A **requires** module directive specifies that this module depends on another module; this relationship is called a module dependency. Each module must explicitly state its dependencies. When module A requires module B, module A is said to read module B and module B is read by module A.

A **requires transitive** directive specifies a dependency on another module and ensures that other modules reading your module also reads that dependency. This is known as implied readability.

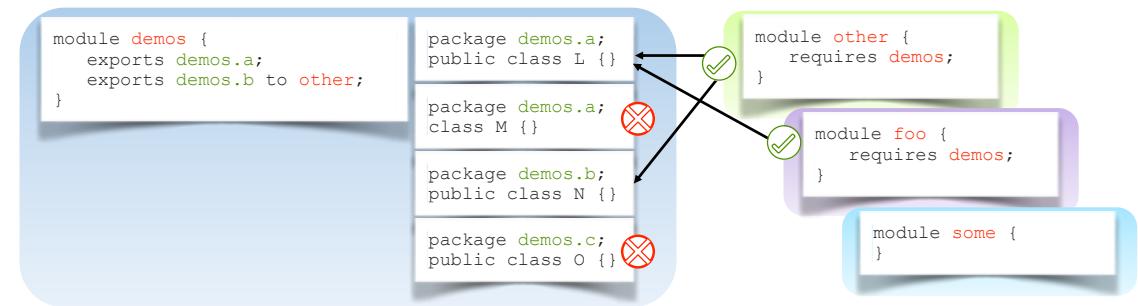
A **requires static** directive indicates that a module is required at compile time, but is optional at run time. This is known as an optional dependency.

This page explains how one module can express an "interest" in the content from another module. The next slide will explain what content that other module may provide.

Export Module Content

Modules define dependencies by exporting packages and requiring other modules.

- Exporting a package means making all of its public types (and their nested public and protected types) available to other modules.
- **exports <packages>** directive specifies packages whose public types should be accessible to all other modules.
- **exports <packages> to <other modules>** restricts exported packages to a list of specific modules.



- ❖ Note: These instructions accept comma-separated lists of package and module names.
- ❖ Reminder: Modules register their interest in other modules using **requires <modules>** directive.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

An exports module directive specifies one of the module's packages, whose public types (and their nested public and protected types) should be accessible to code in all other modules.

An exports...to directive enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package—this is known as a qualified export.

In the example above, module "demos" exports all public classes and their nested public and protected members from package "demos.a" to all other interested modules and from package "demos.b" just to module "other".

Module expresses an interest in accessing content from another module with the help of the requires directive.

Class "L" is visible to modules "other" and "foo" because it is a public class in the package exported to all other modules.

Class "M" is not visible to other modules because it is not public.

Class "N" is not visible to module "foo" because its package has only been exposed to module "other."

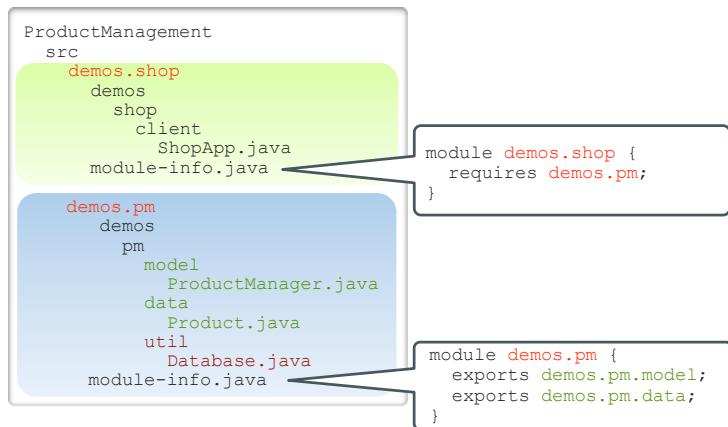
Class "O" is not visible to anyone outside of the module "demos" because this module never exported package "demos.c" thus making it essentially private.

Code in the module "some" cannot access anything in module "demos" because it did not declare such a requirement.

Modules Example

- This example presents two modules: `demos.shop` and `demos.pm`
- There are three packages in the `demos.pm` module, but only two of these (`demos.pm.model` and `demos.pm.data`) are exposed to other modules.
- Package `demos.pm.util` remains private to this module.
- In this example, both modules share a common application folder. However, modules could be created by different developers and are likely to originate from different folders.

✿ Note: In this example, modules provide an ability to create different client applications using business logic exposed by the `demos.pm.model` module, yet encapsulating its implementation.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

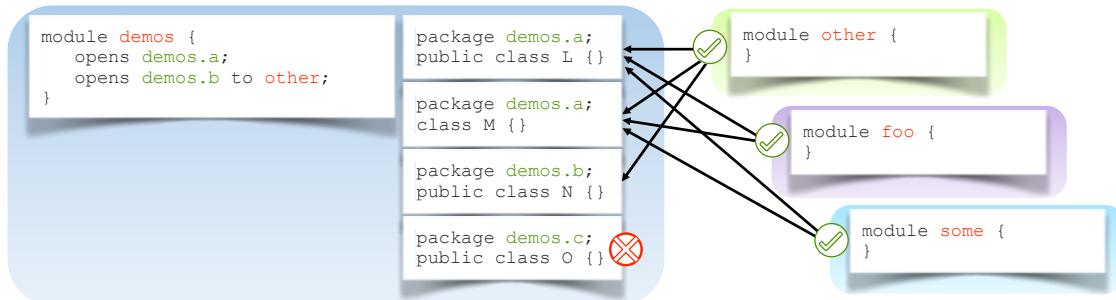
11

Module names are derived from primary package names of respected modules. They do not have to coincide, but they often do to ensure that module names are unique.

Open Module Content

Module may allow runtime-only access to a package using "opens" directive.

- `opens <packages>` directive specifies package whose entire content is accessible to all other modules at run time.
- `opens <packages> to <modules>` restricts opened package to a list of specific modules.
- Opening a package works similar to export, but also makes all of its non-public types available via reflection.
- Modules that contain injectable code should use "opens" directive, because injections work via reflection.



✖ Note: These instructions accept comma-separated lists of package and module names.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Before Java 9, reflection could be used to learn about all types in a package and all members of a type, even if they are 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, this also applies to reflection.

An `opens <packages>` directive allows you to specify a package whose content is accessible to code in other modules at run time only; it makes all the types in the specified package (and all of the types' members) accessible via reflection.

An `opens <packages> to <modules>` directive indicates that a specific package's public types (and their nested public and protected types) are accessible to code in the listed modules at run time only. All of the types in the specified package (and all of the types' members) are accessible via reflection to code in the specified modules.

Open an Entire Module

Module may allow runtime-only access to all of its content.

- `open module` specifies that this module's entire content is accessible to all other modules at run time via reflection.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

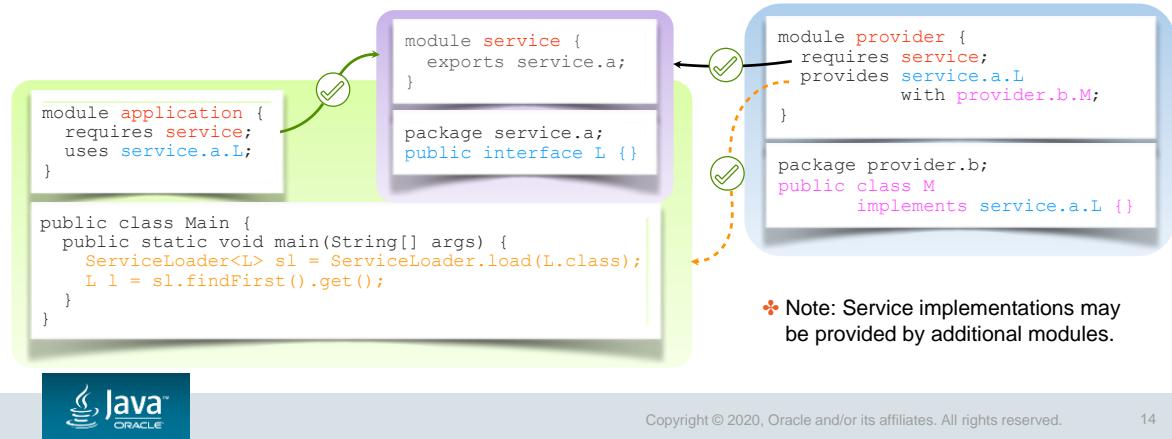
13

All classes in all packages of the module `demos` are accessible at run time to any module via reflection.

Produce and Consume Services

Modules can produce and consume services, rather than just expose all public classes in selected packages.

- Service comprises an interface or abstract class and one or more implementation classes.
- **provides <service interface> with <classes>** directive specifies that module provides one or more service implementations that can be **dynamically discovered by service consumer**.
- **uses <service interface>** directive specifies an interface or an abstract class that defines a service that this module would like to consume.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

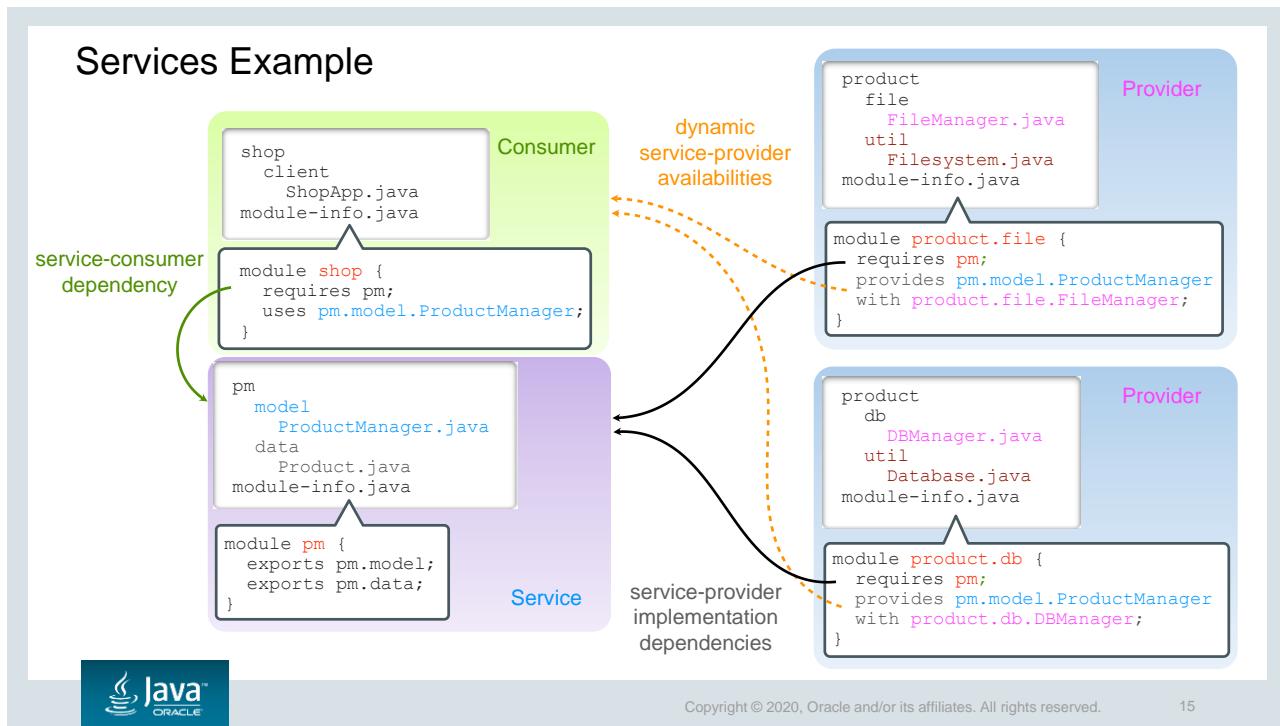
14

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

A **provides <service> with <implementation classes>** directive specifies that a module provides a service implementation, making the module a service provider. The "provides" part of the directive specifies an interface or abstract class listed in a module's **uses** directive and the "with" part of the directive specifies the names of the service provider classes that implements the interface or extends the abstract class.

Your code can list available service implementations using `java.util.ServiceLoader` class:

```
ServiceLoader<L> sl = ServiceLoader.load(L.class);
sl.stream().forEach(s -> s.get().getClass().getName());
```



In this example, **pm** module presents a service with two alternative implementations, provided by **product.db** and **product.file** modules. This service is used by the **shop** module. Content of the **pm** module contains classes and interfaces required by both service consumers and service providers.

Consumer module can dynamically discover available service providers, based on which modules are added to the module path.

Multi-Release Module Archives

Only one copy of a module can be placed into a module-path.

- Multi-Release JAR can be used to support different versions of code for different versions of Java.
- Module root directory may contain either a **default version of the module** or a **non-modularized version of code** to be used by Java versions prior to 9.
- **Specific version** of code may be provided for each version of Java.
- **Versioned descriptors** (`module-info`) are optional and must be identical to the **root module descriptor**, with two exceptions:
 - Can have different non-transitive requires clauses of `java.*` and `jdk.*` modules
 - Can have different use clauses



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

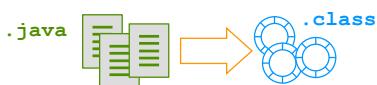
Root folder of this archive may contain a non-modular version of the application, to be used for versions of Java before 9. In this case, there would be no module-module-info descriptor in this root folder.

Before the introduction of modules, java runtime searched for classes by scanning the entire classpath. Module path search is conditioned by module dependencies and thus works faster. Classpath can be set on a system level as well as for a specific application, and sometimes this could result in applications breaking at run time, for example, if you have installed a Java application that used a particular API and then you install another application that used a newer version of the same API. On executing these applications with common classpath definition, it would potentially result in `NoClassDefFound` or `NoSuchMethod` errors depending on the order in which the different versions of the same API occurred in your classpath. When designing modules, you may include explicit version identity to avoid such errors.

Compile and Package a Module

Compile Module

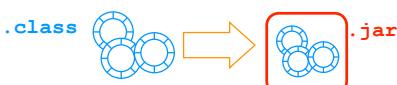
- Specify all of your Java sources from various packages that you want this module to contain
- Include packages that are exported by this module to other modules and a module-info
- Reference other modules required for this module to compile



```
javac --module-path <paths to other modules>
      -d <compiled output folder>
      -sourcepath <path to source code>
```

Package module into a JAR file

- You may describe a main class for this module, if appropriate.
- ".." indicates inclusion of all files from a compiled code folder.



```
jar --create -f <path and name of the jar file>
      --main-class <package name>.<main class name>
      -C <path to compiled module code> .
```

Verify packaged module

- Get description of the compiled module to find out which modules it contains, exports, requires, etc.

```
java --module-path <path to compiled module>
      --describe-module <module name>
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

--create option instructs jar utility to create new jar file. -f option sets path to the car file. -C option sets path to compiled code of the module.

If module has not explicitly defined that it requires java.base and relied upon an implicit inclusion, then --describe-module command would display this as requires java.base mandated.

Existing modules can be updated using --update option.

Execute a Modularized Application

Execute Modular Application Using Module Path

- Classes are located using `-p` or `--module-path` option
- Reference the main class using `-m` or `--module` option
- Non-modular JARs are treated as Automatic Modules (described in the next slide).

```
java -p <path to modules>
      -m <module name>/<package name>. <main class name> <arguments>
```

Execute Non-Modular Application (reminder)

- Classes are located using `-cp` or `--class-path` option
- Modular JARs located via class path are treated as non-modular for backward compatibility.

```
java -cp <path to jars including modularised jars>
      <package name>. <main class name> <arguments>
```

- ❖ Note: In Java 9 onward, any classes accessed via `classapth` are assigned to the "unnamed module" as a temporary solution for the period of migration to the modular application structure.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

Unnamed module is created for each class loader for all classes that are loaded via class-path.

Unnamed module can read all other modules, and all modules can read content of the unnamed module.

Migrating Legacy Java Applications Using Automatic Modules

Legacy JAR files, which are placed into the module-path, are treated as Automatic Modules.

- Such JARs can be referenced as modules by other modules using `requires` declaration.
- By default, `JAR file name` is used instead of module name for such referencing.
- To avoid deployment and maintenance issues, such as naming clashes, specify `automatic module name` using MANIFEST.MF file placed into the legacy JAR as a temporary measure.



❖ Note: To complete migration, create module-info descriptor and repackage legacy JAR as a module.



Create Custom Runtime Image

Use `jlink` utility to create custom runtime image:

```
jlink --module-path <paths to compiled modules and %JAVA_HOME/jmods folder>
      --add-modules <list of module names>
      --bind-services
      --launcher <command name>=<module name>
      --output <name of the runtime image>
```

Structure of the runtime JIMAGE:

- Optimized for space and speed
- Enables faster search and class loading compared to traditional classpath JARs

```
bin
  java executable
  launcher
code
conf
  configuration
  files
include
  C/C++ header files
legal
  legal notices
lib
modules
```

❖ Note: `--module-path` may optionally reference `%JAVA_HOME/jmods folder` when creating a module for the JDK of another platform.

❖ Note: Launcher option creates an alias for running Java with a specific main executable class.

Examine JIMAGE:

```
<image>/bin/java -version
<image>/bin/java --list-modules
```

Only modules used by the application are included



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

For more information about `jlink` option, refer to:

<https://docs.oracle.com/javase/9/tools/jlink.htm>

When adding modules to an image, transitive dependencies are automatically resolved and added.

Launcher is an optional feature that can be used to create platform-specific executable in the bin directory.

If your module has been packaged with a designated main class, simply associate a command with the module name:

```
--launcher <command name>=<module name>
```

Otherwise, you need to specify a main class within a module:

```
--launcher <command name>=<module name>/<package
name><main class name>
```

Before binding a service to a runtime image, you can check which modules contain provider implementations for this service:

```
jlink --module-path <paths to compiled modules and
%JAVA_HOME/jmods folder> --suggest-providers=<name of the
service interface or class>
```

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

Option `--bind services` is used to bind any available service providers. You don't have to use this option if the modules that contain required service providers are already included in the `--add-modules` list.

Execute Runtime Image

Runtime image contains all required modules and actual java runtime to execute application.

- Use `-m` or `--module` when the module does not define launcher command shortcut.

```
<image>/bin/java -m <module name>
```

- Use command name when it is provided by the module.

```
<image>/bin/<command name>
```

❖ Note: There is no need to separately install Java runtime, because it is already contained within the image.



Optimize a Custom Runtime Image

Runtime image optimization options:

```
jlink <options that were discussed earlier>
      --limit-modules <List of module names>
      --endian {little|big}
      --compress={0|1|2}[filter:<pattern list>]
      --no-header-files
      --no-man-pages
      --strip-debug
      --strip-native-commands
      --vm={client|server|minimal|all}
      --class-for-name
      --exclude-files=<pattern list>
      --exclude-resources=<pattern list>
      --include-locales=<list of locales>
      --save-opts <file name>
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

--limit-modules mod [,mod...]: Limits the universe of observable modules to those in the transitive closure of the named modules, mod, plus the main module, if any, plus any further modules specified in the --add-modules option.

--endian: Specifies the byte order of the generated image. The default value is the format of your system's architecture.

--compress: Enables compression with level options: 0 - no compression, 1 - enable constant string sharing, 2 - use ZIP. Filter is an optional parameter that allows one to specify which files to compress.

--no-header-files: Excludes header files

--no-man-pages: Excludes man pages

--strip-debug: Strips debug information from the output image

--exclude-native-commands: Excludes native commands (such as java/java.exe) from the image

--vm: Selects the HotSpot VM in the output image. Default is all.

--class-for-name: Enables class optimization, converting Class.forName calls to constant loads

--exclude-files: Specifies files to exclude

--exclude-resources: Specifies files to exclude
--include-locales: Includes the list of locales where langtag is a BCP 47 language tag. This option supports locale matching as defined in RFC 4647. Ensure that you add the module jdk.localedata when using this option.
--save-opt filename: Saves jlink options in the specified file

Summary

In this lesson, you should have learned how to:

- Compare modular and non-modular Java applications
- Describe modularization
- Create modules dependences
- Define and use module services
- Create Runtime Images
- Deploy and execute modular Java applications



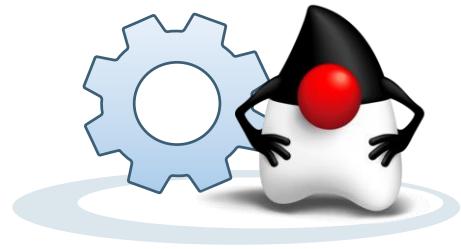
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

25

Practices

In this practice, you will:

- Compare modular and non-modular deployment formats
- Migrate application to modular Java format
- Restructure application modules to achieve more flexible design
- Create and execute application using runtime image



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

26

Annotations



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Understand annotations
- Create custom annotations
- Dynamically discover annotations
- Study frequently used Java annotations



Introduction to Annotations

Annotations are a form of metadata.

- Provide information about a program that's not part of the program itself
- **Can be retained at different levels:**
 - SOURCE is retained in the source code, but discarded by the compiler.
 - CLASS is retained by the compiler, but ignored by the JVM.
 - RUNTIME is retained by the JVM and readable at run time.
- **Applicable to different type of targets:**
 - ANNOTATION_TYPE Annotation type declaration
 - CONSTRUCTOR Constructor declaration
 - FIELD Field declaration (includes enum constants)
 - LOCAL_VARIABLE Local variable declaration
 - METHOD Method declaration
 - MODULE Module declaration
 - PACKAGE Package declaration
 - PARAMETER Formal parameter declaration
 - TYPE Class, interface (including annotation type), or enum declaration
 - TYPE_PARAMETER Type parameter declaration
 - TYPE_USE Use of a type
- You can use existing annotations or **construct your own annotations.**



```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD,
        ElementType.FIELD)
public @interface SomeAnnotation { }
```

✿ Note: Annotation can be applied
to more than one type of target.



Design Annotations

- Annotations can have **attributes**.
- Attributes could be of the following types:
 - primitive
 - String
 - Class
 - Enum
 - Annotation
 - An array of any of the above
- Attribute can have a **default value**, which could be a constant expression.
- Attribute can be an **array of values**, including other repeatable annotations.
- Repeatable annotation is allowed to be **used more than once** within a given **container**.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface BusinessPolicies {
    BusinessPolicy[] value();
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Repeatable(BusinessPolicies.class)
public @interface BusinessPolicy {
    String name() default "default policy";
    String[] countries();
    String value();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

Repeating annotations are stored in a container annotation that's automatically generated by the Java compiler. For the compiler to do this, two declarations are required in your code.

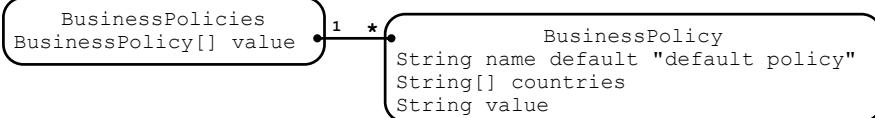
- First, mark your annotation with the meta annotation `@Repeatable`. The container annotation type that the Java compiler generates to store repeating annotations is in parentheses. In this example, the container is `BusinessPolicies`.
- Second, declare the container annotation type. It must have an element that's an array type of your repeatable annotation. In this example, it's `BusinessPolicy[]`.

Apply Annotations

Apply annotations to the appropriate type of target.

- Attributes are set as a list of name-value pairs `@Annotation(name=value, arrayName={e1,e2})`.
- When no attributes are used, `()` can be omitted.
- Attribute `value` does not have to be specified by name, when it's the only attribute that needs to be set.
- If array has only one value, `{}` can be omitted.
- Attributes with default values can be omitted.

```
@BusinessPolicies({
    @BusinessPolicy(name="Returns Policy", countries="GB", value="4 weeks")
    @BusinessPolicy(countries={"GB","FR"}, value="Ship via Dover-Calais")
})
public class Shop { }
```



❖ Note: Annotation applied to a class is not inherited by its subclasses, unless it is marked as `@Inherited`.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

An element named `value` gets special treatment. If you name an element `value`, you can omit its name later when you apply the annotation and assign its value. When assigning a value to an array element, you may omit the curly braces if the array contains only one value.

Compiler rejects code that tries to set annotation attribute to a `null` value.

See the verbose example of applying annotations that explicitly use all attribute names, including attribute "value" and override default values:

```
@BusinessPolicies(value={
    @BusinessPolicy(name="Returns Policy", countries="GB",
    value="4 weeks")
    @BusinessPolicy(name="Shipping Policy"
    countries={"GB","FR"}, value="Ship via Dover-Calais"))
}
public class Shop { }
```

Example illustrating the process of inheriting annotations from the parent class:

Given annotation X:

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.TYPE)
```

```
@Inherited
```

```
public @interface X { }
```

and class A annotated with X:

```
@X
```

```
public class A {}
```

class B that extends A implicitly inherits A class annotation X

```
public class B extends A {}
```

Dynamically Discover Annotations

Java reflection API allows dynamic discovery of class structures, including annotations.

- Get array of annotations for the class, methods, or fields (see more examples in notes).
- Discover annotation type.
- Get annotations by type.
- Invoke operations upon annotation to retrieve its attributes.

```
// All class level annotations and their types:  
Stream.of(Shop.class.getAnnotations())  
    .forEach(t->t);  
// Type of the first class level annotation:  
Class annotationType = Shop.class.getAnnotations()[0].getAnnotationType();  
// Class level annotations of BusinessPolicy type:  
BusinessPolicy[] policyAnnotations =  
    Shop.class.getAnnotationsByType(BusinessPolicy.class);  
// Retrieve values of annotation attributes:  
for(BusinessPolicy policy: policyAnnotations) {  
    System.out.println(policy.name());  
    System.out.println(policy.value());  
    for (String country: policy.countries()) {  
        System.out.println(country);  
    }  
}
```

✿ Note: Source-level annotations cannot be dynamically discovered, because they are not present in the compiled code.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Get all method-level annotations:

```
Stream.of(Shop.class.getMethods())  
    .flatMap(m->Stream.of(m.getDeclaredAnnotations()))  
    .forEach(a->System.out.println(a));
```

// Get all field level annotations:

```
Stream.of(Shop.class.getFields())  
    .flatMap(f->Stream.of(f.getDeclaredAnnotations()))  
    .forEach(a->System.out.println(a));
```

Example assumes that class Shop has been annotated with class-level annotations:

```
@demos.BusinessPolicies(value={  
    @demos.BusinessPolicy(name="Returns Policy",  
        countries={"GB"}, value="4 weeks"),  
    @demos.BusinessPolicy(name="default policy",  
        countries={"GB", "FR"}, value="Ship via Dover-Calais") })  
BusinessPolicy annotations:
```

```
@demos.BusinessPolicy(name="Returns Policy",  
    countries={"GB"}, value="4 weeks")  
@demos.BusinessPolicy(name="default policy",  
    countries={"GB", "FR"}, value="Ship via Dover-Calais")
```

Vales of BusinessPolicy annotation attributes:

```
Returns Policy 4 weeks GB  
default policy Ship via Dover-Calais GB FR
```

Document the Use of Annotations

Annotation can be marked with the `@Documented` annotation.

- Class documentation would include a reference to annotations that are marked as documented.

```
package demos.annotations;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface X { }
```

```
package demos.annotations;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Y { }
```

```
javadoc -d project/docs
         -sourcepath project/src
         -subpackages demos
```

Package demos.api
Class Some
Example Class
Version: 1.0
Author: John Doe
Method Detail
• a
 → `@X` public void a()
• b
 public void b()

```
package demos.api;
/**
 * Example Class
 * @author John Doe
 * @version 1.0
 */
public class Some {
    @X
    public void a() {}
    @Y
    public void b() {}
}
```



Annotations that Validate Design

Annotation may be used to validate class or interface design.

- Enforce the definition of a functional interface
 - Only one abstract method should be present in a functional interface.
 - The `FunctionalInterface` annotation prevents interface from compiling if this rule is broken.

```
@FunctionalInterface
public interface Perishable {
    boolean perish(LocalDate expiryDate);
    int getDays(LocalDate expiryDate);
}
```

- Verify that a method actually overrides a parent operation
 - Subclass must match the signature of the parent class method.
 - The `Override` annotation prevents subclass from compiling if this rule is broken.

```
public class Product {
    @Override
    public boolean equals(Product product) {
        return this.id == product.id;
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

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

The `@Override` annotation makes compiler generate an error if the annotated method fails to correctly override an inherited method.

Deprecated Annotation

Code marked with the `@Deprecated annotation` should no longer be used.

- Attribute `since` indicates a version after which this code should no longer be used.
- Attribute `forRemoval` is a boolean indicator:
 - true indicates intent to remove the annotated program element in a future version.
 - false indicates that the use of the annotated program element is discouraged, but at the time the program element was annotated; there was no specific intent to remove it.
- Documentation should describe:
 - Reason the code is deprecated
 - Alternative API to use instead
 - Section of documentation that describes such a code is marked with Javadoc `@deprecated tag`

```
public class ProductManager {
    /**
     * @deprecated This method has been deprecated,
     * as it is inherently deadlock-prone.
     * Please use {@link #commit} method instead
     */
    @Deprecated(since="11", forRemoval=true)
    public synchronized void save() {
        ...
    }
    public void commit() {
        ...
    }
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

The `@Deprecated annotation` discourages code use. This annotation may be applied to constructors, fields, local variables, methods, packages, module, parameters, classes, interfaces, and enums. Code may be deprecated for several reasons; for example, its usage likely leads to errors. It may be changed incompatibly or removed in a future version; it's superseded by a more preferable alternative or it's obsolete. The compiler generates a warning when deprecated code is used or overridden in non-deprecated code.

It's strongly recommended that you document the reason for deprecating a program element using the `@deprecated Javadoc tag`. Documentation should also suggest and link to any recommended replacement API. A replacement API often has subtly different semantics, which should also be noted. The use of the at sign (@) in both Javadoc comments and annotations is not coincidental: they are related conceptually. Also, note the Javadoc tag starts with a lowercase d and the annotation starts with an uppercase D.

`@Deprecated` has a String element `since`; its value indicates the version when deprecation first occurred. In the example, method `save` is marked as deprecated in version 11. It's recommended you specify a `since` value for any newly deprecated program elements.

`@Deprecated` has a boolean element `forRemoval`. A true value indicates intent to remove the deprecated element in a future version. A false value indicates that although use of the deprecated element is discouraged, there's no intent to remove it yet.

Suppress Compiler Warnings

Indicate that compiler warnings should be suppressed for the annotated element

- Warning can be suppressed on a class or specific method level.
- Unchecked warnings are caused by assignment of raw-type object to generic-type variable.
- Deprecated warnings are caused by the use of out-of-date APIs.

```
@SuppressWarnings({"unchecked", "deprecation"})
public void processProducts(ProductManager pm) {
    List<Product> products = pm.find();
    pm.save();
}

public class ProductManager {
    public List find() {
        return List.of(new Food("Cake", 2.99),
                     new Drink("Tea", 1.99));
    }
    @Deprecated
    public synchronized void save() { }
}
```

- ❖ Unchecked warning indicates potential heap pollution.
- ❖ Non-parameterized (raw) object may allow values that contradict parameterized (generic) type.
- ❖ Compiler is unable to perform type safety check on the raw object.
- ❖ Suppressing compiler warnings could be dangerous. When compiling code, you may not notice that program is using out-of-date APIs, such as pre-generics or deprecated code.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Unchecked conversion is used to enable a smooth interoperation of legacy code, written before the introduction of generic types, with libraries that have undergone a conversion to use genericity (a process we call generification). In such circumstances (most notably, clients of the Collections Framework in `java.util`), legacy code uses raw types (e.g. `Collection` instead of `Collection<String>`). Expressions of raw types are passed as arguments to library methods that use parameterized versions of those same types as the types of their corresponding formal parameters.

Var-args and Heap Pollution

Incorrect use of var-args with generics can lead to **heap pollution**.

- Var-arg is essentially an array, so it can be assigned to an array of objects (arrays are covariant).
- Array of objects allows adding of elements that are not of a type expected by a generic declaration.
- `ClassCastException` may occur when trying to get elements from the collection.

Annotation `@SafeVarargs` suppresses heap-pollution warning when using var-args.

- Ensure that your code does not produce actual heap pollution.
- Annotated method must be **private** or **final** to maintain type safety guarantee.

```
public class Some {
    @SafeVarargs
    public final void some(List<String>... values) {
        // Object[] objectArray = values;
        // objectArray[0] = Arrays.asList(1,2,3);
        for (List<String> value: values) {
            value.stream().forEach(v->System.out.print(v));
            System.out.println(--);
        }
    }
}

@SuppressWarnings("unchecked")
public static void main(String[] args) {
    Some s = new Some();
    s.some(Arrays.asList("A", "B", "C"),
           Arrays.asList("X", "Y", "Z"));
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

The `@SafeVarargs` annotation is applicable to methods and constructors. It asserts that your code doesn't perform potentially unsafe operations on its varargs parameter. These unsafe operations may result in heap pollution. Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. Because this is difficult for the compiler to verify, a warning is issued instead. Your code may be perfectly safe and free of heap pollution issues. If this is true and you are very sure of your assertions, you may apply this annotation to let the compiler know there is no reason to warn you about this issue. The `@SafeVarargs` annotation lets you suppress these unchecked warnings.

Summary

In this lesson, you should have learned how to:

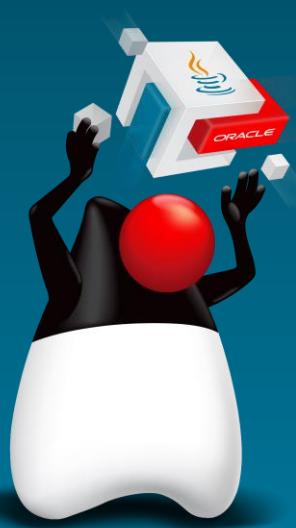
- Understand Annotations
- Create custom annotations
- Dynamically discover annotations
- Study frequently used Java annotations



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Java Database Connectivity



ORACLE®

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Learn about Java Database Connectivity (JDBC) API Structure
- Manage database connections
- Execute SQL statements
- Process query results
- Manage transactions
- Discover metadata
- Customize query results processing

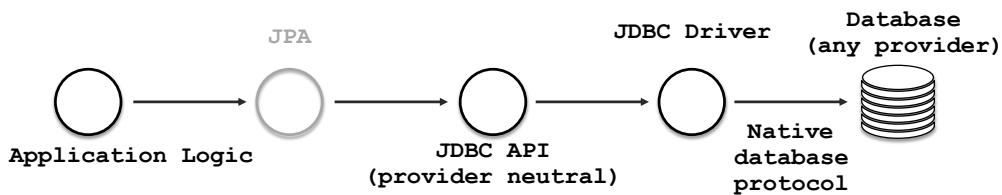


Java Database Connectivity (JDBC)

JDBC API provides an interface to interact with databases.

- It's defined in the `java.sql` package.
- It's not specific to any particular database provider.
- JDBC drivers provide database specific implementations of the JDBC API.
 - Driver implementation libraries must be made available to the application class loader (via class path or module path).
 - Driver is selected using JDBC connection url:

<code>jdbc:<provider>:<driver type>:<connection details></code>	General JDBC url pattern
<code>jdbc:oracle:thin:@<host>:<port>:<database name></code>	Oracle thin driver example
<code>jdbc:derby:<host>:<port>:<database name></code>	Java DB (Derby) driver example



✿ Note: Java Persistence API (JPA) is covered by the "Developing Application for the Java EE 7 Platform" course



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

3

Provider-specific JDBC driver implementation libraries must be made available to the application class loader using either class path or module path.

You need to have the JDBC driver for your database to connect to the database using JDBC.

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

Oracle provides several different implementations of JDBC driver, such as `thin` (generally recommended driver), `OCI` (platform specific driver), `default`, or `kprb` (server-side internal database driver).

JDBC API Structure

- `DriverManager` class manages JDBC drivers and creates connection objects.
- `Connection` interface represents a session with a specific database and creates all types of statements.
- `Statement` interface represents a basic SQL statement.
- `PreparedStatement` interface represents a precompiled SQL statement.
- `CallableStatement` interface represents calls to SQL-stored procedures or functions.
- `ResultSet` interface represents a set of records returned by the execution of a SQL query statement.
- All of the above interfaces implement an `AutoCloseable` interface.
 - can be used with the try-with-resources construct
 - otherwise must be explicitly closed in a `finally` block
- JDBC API operations can throw `SQLException`, which provides database-specific error code and SQL status information in addition to the usual Java exception details.

```
try ( Connection c = DriverManager.getConnection(...);
      PreparedStatement s = c.prepareStatement(...);
      ResultSet rs = s.executeQuery(); ) {
    /* handle statement execution results */
} catch(SQLException e) {
    String state = e.getSQLState();
    int code = e.getErrorCode();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

If you provide explicit `finally` block, you must close objects in specific order:

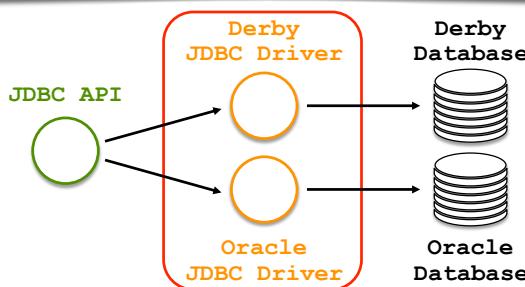
```
Connection c = null;
Statement s = null;
ResultSet rs = null;
try {
    c = DriverManager.getConnection(...);
    s = c.createStatement(...);
    rs = s.executeQuery();
    /* process statement execution results */
} catch (SQLException e) {
    /* handle exceptions */
} finally{
    rs.close(); // first close ResultSet
    s.close(); // then close Statement, PreparedStatement
    or CallableStatement objects
    c.close(); // last close Connection
}
```

Manage Database Connections

Interface `java.sql.Connection` represents JDBC connection to any database.

- From JDBC 4.0, driver is auto-selected based on a JDBC url.
- Method `getConnection` of the `java.sql.DriverManager` class returns provider-specific JDBC driver implementation of the `Connection` interface.

```
String url = "jdbc:derby:localhost:1527:productDB";
// String url = "jdbc:oracle:thin:@localhost:1521:orcl";
String username = "pm";
String password = "welcome1";
Connection connection = DriverManager.getConnection(url, username, password);
/* use connection to execute SQL statements */
```



- See notes for the loading of the pre-JDBC 4.0 drivers.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Provider-specific JDBC driver implementation libraries must be made available to the application class loader using either class path or module path.

Oracle provides several different implementations of JDBC driver, such as `thin` (generally recommended driver), `OCI` (platform specific driver), and `default` or `kprb` (server-side internal database driver).

Method `getConnection` of the `java.sql.DriverManager` class is overloaded to use alternative forms of authentication, instead of just username and password.

Earlier, JDBC 4.0 drivers had to be loaded explicitly in one of the following ways:

- Instantiating relevant driver class and registering the driver with the `DriverManager`:

```
DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
```

- Loading driver class:

```
try {
java.lang.Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (ClassNotFoundException c) { }
```

- Using command-line option:

```
java -djdbc.drivers=oracle.jdbc.driver.OracleDriver
```

From JDBC 4.0 onward, JDBC drivers are automatically loaded, and no explicit driver loading or registration is required.

Create and Execute Basic SQL Statements

Interface `java.sql.Statement` represents all types of **SQL statements**.

All types of statements, (`Statement`, `PreparedStatement`, `CallableStatement`) can execute:

- A query (`select`) operation to get a `ResultSet` object
- Any other sql operation, such as `insert`, `update`, `delete`, `create`, `alter`, `drop`
- Dynamically determine a type of SQL operation

Disadvantages of the basic statement compared to prepared and callable statements:

- Parameter concatenation presents a security risk of a **SQL injection**.
- Basic statements have to be parsed and recompiled before every execution.

```
Connection connection = DriverManager.getConnection(...);
Statement statement = connection.createStatement();
String productQuery = "select name from products where price > "+price;
String productUpdate = "update products set price = "+price+" where id = "+id;
// ResultSet results = statement.executeQuery(productQuery);
// int rowCount = statement.executeUpdate(productUpdate);
boolean isQuery = statement.execute(productQuery);
if (isQuery) {
    ResultSet results = statement.getResultSet();
} else{
    int rowCount = statement.getUpdateCount();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

Examples executing select, insert, update, and delete operations using basic statement:

```
statement.executeUpdate("select id, name from
products where price > "+price);

statement.executeUpdate("insert into products
values ("+id+","+name+","+price+");

statement.executeUpdate("update products set price
= "+price+" where id = "+id);

statement.executeUpdate("delete from products
where id ="+id);
```

The use of parameter concatenation by basic statements presents a risk of SQL injections:

```
ResultSet rs = s.executeQuery("select id, price from
products where name like '" +param+"');
```

This example can be exploited if user submits an arbitrary SQL statement as a param value: "Tea'; drop table products;"

To address this issue, use either `PreparedStatement` object or basic statement `s.enquoteLiteral(param)` method to sanitize parameter value.

Create and Execute Prepared SQL Statements

Interface `java.sql.PreparedStatement` represents precompiled SQL statements:

- Execute any SQL operations
- Use positional substitution parameters, marked with the ? symbol
- Set parameters using:
 - `setXXX(<position>, <value>)` operations (where XXX is a value type)
 - `setObject(<position>, <value>, <type>)` operation

```
Connection connection = DriverManager.getConnection(...);
String productQuery = "select id, name from products where price > ?";
String productUpdate = "update products set price = ? where id = ?";
PreparedStatement findProduct = connection.prepareStatement(productQuery);
PreparedStatement updatePrice = connection.prepareStatement(productUpdate);
findProduct.setObject(1, price, Types.NUMERIC); // findProduct.setBigDecimal(1, price);
updatePrice.setObject(1, price, Types.NUMERIC); // updatePrice.setBigDecimal(1, price);
updatePrice.setObject(2, id, Types.INTEGER); // updatePrice.setInt(2, id);
ResultSet results = findProduct.executeQuery();
int rowCount = updatePrice.executeUpdate();
// boolean isQuery = findProduct.execute();
```

✿ Note: Parameter position index starts from 1.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Examples preparing select, insert, update, and delete operations using prepared statement:

```
connection.prepareStatement("select id, name from
products where price > ?");

connection.prepareStatement("insert into products values
(?, ?, ?)");

connection.prepareStatement("update products set price =
? where id = ?");

connection.prepareStatement("delete from products where
id = ?");
```

Create and Execute Callable SQL Statements

Interface `java.sql.CallableStatement` is used to **invoke** stored procedures or functions.

- Use **positional substitution parameters**, marked with the `?` symbol
- Before statement execution:
 - Register the type of each output parameter or returned value
`registerOutParameter(<position>, <type>)`
 - Set each input parameter value using either `setXXX(<position>, <value>)` operations or `setObject(<position>, <value>, <type>)` operation
- After statement execution:
 - Retrieve each output parameter or returned value using either `getXXX(<position>)` operations or `getObject(<position>, <type>)` operation

```
Connection connection = DriverManager.getConnection(...);
String functionCall = "? = { call some_function(?) }";
CallableStatement some = connection.prepareCall(functionCall);
some.registerOutParameter(1, Types.VARCHAR);
some.setObject(2, date, Types.DATE); // some.setDate(2, date);
some.execute();
String value = some.getObject(1, Types.VARCHAR); // some.getString(1);
```

❖ Note: In/out parameter **position index** starts from 1.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Parameter can be both input and output:

```
Connection = DriverManager.getConnection(...);

String functionCall = "{ call some_function(?) }";

CallableStatement some =
connection.prepareCall(functionCall);

some.registerOutParameter(1, Types.NUMERIC);

some.setObject(1, value, Types.NUMERIC);

some.execute();

String value = some.getObject(1, Types.NUMERIC);
```

Process Query Results

Interface `java.sql.ResultSet` is used to traverse a set of records returned by the query.

- Method `next()`
 - Returns `true` if the next record exists and moves the pointer to point to the next record
 - Returns `false` if there are no more records in this `ResultSet`
- Retrieve each column value using one of the following operations:
 - `getXXX(<column position>)` (first column position index is 1)
 - `getXXX(<column name>)`
 - `getObject(<column position>, <class>)`
 - `getObject(<column name>, <class>)`

1	Tea
2	Coffee
3	Cake
4	Cookie

```
Connection connection = DriverManager.getConnection(...);
String productQuery = "select id, name from products where price > ?";
PreparedStatement findProduct = connection.prepareStatement(productQuery);
findProduct.setObject(1, price, Types.NUMERIC);
ResultSet results = findProduct.executeQuery();
while(results.next()) {
    int id = results.getObject(1, Integer.class); // results.getInt(1);
    String name = results.getObject("name", String.class); // results.getString("name");
}
```

❖ Note: Default `ResultSet` behavior is forward only; the other options are covered later.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Always check for the next record existence even if the query is supposed to return exactly one record. For example, a query selecting records using primary key value can return no results if the record with the specified key does not exist:

```
Connection = DriverManager.getConnection(...);

String productQuery = "select name, price from products
where id = ?";

PreparedStatement findProduct =
connection.prepareStatement(productQuery);

findProduct.setInt(1, id);

ResultSet results = findProduct.executeQuery();

if(results.next()) {

    String name = results.getString(1);
    BigDecimal price = results.getBigDecimal(2);
}
```

You may improve performance of your program by controlling fetch size (number of records downloaded from the database in one go).

Fetch size gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed for ResultSet objects generated by this statement. If the value specified is zero, then the hint is ignored. The default value is zero.

Default fetch size can be set at the Statement level, and it can also be changed at the ResultSet level:

```
String productQuerySQL = "select * from products where
price > ?";

PreparedStatement productQuery =
conn.prepareStatement(productQuerySQL);

productQuery.setObject(1, price, Types.NUMERIC);
productQuery.setFetchSize(10); // records will be
downloaded in chance of 10

ResultSet results = productQuery.executeQuery();
while(results.next()) {
    /* process records one at the time */
    if (<some condition>) {
        results.setFetchSize(10);
    }
}
```

Control Transactions

By default, connections are in auto-commit mode.

- Commit occurs automatically when the statement processing successfully completes.
- Method `setAutoCommit` changes commit mode behavior.
- Method `getAutoCommit` returns current commit mode behavior.

Transaction must end in commit or rollback.

- Use `commit` method to make all pending transaction changes permanent.
- Use `rollback` method to discard pending transaction changes.
- If required, use method `setSavepoint` to create savepoints.
- Rollback to savepoint discards pending changes that occurred in the transaction after this savepoint.
- Normal connection closure also causes transaction to commit.

```
try ( Connection connection = DriverManager.getConnection(...) ) {
    connection.setAutoCommit(false); // switch auto-commit mode off
    /* execute SQL statements */
    Savepoint sp1 = connection.setSavepoint();
    /* execute SQL statements */
    connection.rollback(sp1);        // rollback to savepoint
    connection.commit();            // commit transaction
} catch (SQLException e) {
    connection.rollback();          // rollback transaction
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Connection is closed implicitly if it was initialized using try-with-resources construct, so it is important to remember to roll back when exceptions occur.

Discover Metadata

Metadata objects are available for entire database and for each result set.

- DatabaseMetaData object contains comprehensive information about the database as a whole, such as database provider, SQL capabilities, supported features.

Examples of database metadata properties:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
String dbProductName = dbMetaData.getDatabaseProductName();
String dbProductVersion = dbMetaData.getDatabaseProductVersion();
String supportedSQLKeywords = dbMetaData.getSQLKeywords();
boolean outerJoins = dbMetaData.supportsOuterJoins();
boolean savepoints = dbMetaData.supportsSavepoints();
```

- ResultSetMetaData object contains information about types and properties of the columns in a ResultSet object.

Examples of result set metadata properties:

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
for (int i = 1; i <= rsMetaData.getColumnCount(); i++) {
    String name = rsMetaData.getColumnName(i);
    int type = rsMetaData.getColumnType(i);
}
```

❖ Note: Later pages contain more examples of metadata properties.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

JPA API implementations such as Hibernate and EclipseLink are fully aware of database and result set metadata and can produce database provider-specific SQL code, taking advantage of specific database implementation capabilities.

Customize ResultSet

Both basic and prepared statements allow customizations of a result set they can produce.

- **ResultSetType**
 - Direction of the result set traversal
 - Reflection of changes made to the underlying data source while this result set remains open
- **ResultSetConcurrency**
 - Updatability or the result set
- **ResultSetHoldability**
 - Retention or closure of a result set when transaction is committed

```
Statement s = conn.createStatement(<ResultSetType>,
                                  <ResultSetConcurrency>,
                                  <ResultSetHoldability>);
PreparedStatement ps = conn.prepareStatement(<SQL>,
                                         <ResultSetType>,
                                         <ResultSetConcurrency>,
                                         <ResultSetHoldability>);
```

✿ Note: Next two pages explain these options in more details.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

CallableStatement can also present a ResultSet object, for example, when stored procedure returns an opened cursor reference.

Set Up ResultSet Type

ResultSetType controls

- Direction of the result set traversal
- Reflection of changes made to the underlying data source while this result set remains open
- Possible values are:
 - `ResultSet.TYPE_FORWARD_ONLY`: (default)
 - Only method `next()` is enabled.
 - Reflection of changes in the underlying datasource is database specific.
 - Other types enable the use of `previous()`, `next()`, `first()`, `last()`, `absolute(int row)`, `relative(int row)` methods.
 - `ResultSet.TYPE_SCROLL_INSENSITIVE`:
 - Changes are not reflected.
 - `ResultSet.TYPE_SCROLL_SENSITIVE`:
 - Changes are reflected.

❖ Note: Not all databases and JDBC drivers support all ResultSet types.
The method `DatabaseMetaData.supportsResultSetType` returns true if the specified ResultSet type is supported and false otherwise.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

TYPE_FORWARD_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

TYPE_SCROLL_INSENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is not sensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

TYPE_SCROLL_SENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

Method `absolute(int row)` moves the cursor to the given row number in this `ResultSet` object.

- If the row number is positive, the cursor moves to the given row number with respect to the beginning of the result set. The first row is row 1, the second is row 2, and so on.
- If the given row number is negative, the cursor moves to an absolute row position with respect to the end of the result set. For example, calling the method `absolute(-1)` positions the cursor on the last row; calling the method `absolute(-2)` moves the cursor to the next-to-last row, and so on.
- If the row number specified is zero, the cursor is moved to before the first row.
- An attempt to position the cursor beyond the first/last row in the result set leaves the cursor before the first row or after the last row.

Note: Calling `absolute(1)` is the same as calling `first()`. Calling `absolute(-1)` is the same as calling `last()`.

Method `relative(int rows)` moves the cursor a relative number of rows, either positive or negative.

- Calling `relative(0)` is valid but does not change the cursor position.
- Attempting to move beyond the first/last row in the result set positions the cursor before/after the first/last row.

Note: Calling the method `relative(1)` is identical to calling the method `next()` and calling the method `relative(-1)` is identical to calling the method `previous()`.

Set Up ResultSet Concurrency and Holdability

ResultSetConcurrency controls the updatability of the result set.

Possible values are:

- CONCUR_READ_ONLY, which indicates that a ResultSet object may NOT be updated (default)
- CONCUR_UPDATABLE, which indicates that a ResultSet object may be updated

ResultSetHoldability controls a retention or closure of a result set when transaction is committed.

Possible values are (default is database dependent):

- HOLD_CURSORS_OVER_COMMIT, which indicates a holdable result set. Holdable cursors are best used with read-only result set objects.
- CLOSE_CURSORS_AT_COMMIT, which indicates ResultSet objects (cursors) are closed upon the commit. Closing cursors can result in better performance for some applications.

❖ Note: Not all JDBC drivers and databases support concurrent and holdable/non-holdable cursors.

Methods of DatabaseMetaData object indicate specific database support levels:

`supportsResultSetConcurrency` returns true if the specified concurrency level is supported.

`getResultSetHoldability` returns default holdability value for the current driver.

`supportsResultSetHoldability ResultSet.HOLD_CURSORS_OVER_COMMIT` returns true if applicable.

`supportsResultSetHoldability ResultSet.CLOSE_CURSORS_AT_COMMIT` returns true if applicable.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

The example demonstrates the handling for the scrollable and updatable result set:

```
float price = 2.99F;
PreparedStatement productQuery =
    connection.prepareStatement("select id, price from
products where price = ?",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
rs.setFloat(1, price);
ResultSet rs = productQuery.executeQuery();
rs.absolute(3); // moves the cursor to the third row of
the result set
price+=0.99F; // change price
rs.updateFloat(2,price); // apply price change
rs.updateRow(); // updates the row in the data source
```

Summary

In this lesson, you should have learned how to:

- Learn about Java Database Connectivity (JDBC) API Structure
- Manage database connections
- Execute SQL statements
- Process query results
- Manage transactions
- Discover metadata
- Customize query results processing



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Java Security



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



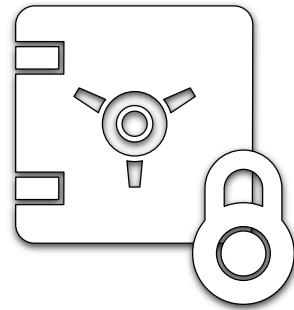
After completing this lesson, you should be able to:

- Apply restriction using security policies
- Protect code
- Validate values
- Protect sensitive data
- Prevent injections
- Discover and document security issues



Security Threats

- Denial of Service (DoS) Attacks
 - Caused by unchecked and unrestricted resource utilizations
- Sensitive data leaks
 - Caused by lack of encryption or information reduction
- Code corruption
 - Caused by lack of encapsulation and immutability
- Code injections
 - Caused by lack of input value validation and sanitation



❖ See notes for DoS attack example



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Characteristics of DoS Attacks

Symptoms

- Legitimate users are unable to access resources and services.
- There is excessive resource consumption. No resources remain to do legitimate work.

Causes

- A file or code construct grows too large.
- A service or connection is overwhelmed with bogus requests.

Prevention

- Use permissions to restrict access to code that consumes vulnerable resources.
- Validate all application inputs.
- Release resources in all cases.
- Monitor excessive resource consumption disproportionate to that used to request a service.

Denial of Service (DoS) Attack

Unchecked and unrestricted resource utilizations can be exploited.

Symptoms:

- Legitimate users are unable to access resources and services.
- There is excessive resource consumption. No resources remain to do legitimate work.

Causes:

- A file or code construct grows too large.
- A service or connection is overwhelmed with bogus requests.

Prevention:

- Use permissions to restrict access to code that consumes vulnerable resources.
- Validate all application inputs.
- Release resources in all cases.
- Monitor excessive resource consumption disproportionate to that used to request a service.

❖ See notes for DoS attack example



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

The example shows a DoS attack that creates infinite number of http requests that are downloaded one byte at a time.

Warning: Do not actually try this outside of a controlled environment.

The server should try to detect excessive resource consumption and block this client.

```
ExecutorService es = Executors.newFixedThreadPool(10000);
String url = <site to be attacked>;
while (true) {
    es.execute(() -> {
        try ( InputStream in =
URI.create(url).toURL().openStream() ) {
            int value = 0;
            while ((value = in.read()) != -1) {
                Thread.sleep(20000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
}
```

Define Security Policies

Security policies can be used to restrict access to code and resources.

- Java security descriptor file is used to configure
 - General security settings
 - References to certificate keystore files
 - References to security policies files

```
${java.home}/conf/security/java.security
```

```
policy.url.1=file:${java.home}/conf/security/java.policy
policy.url.2=file:/somepath/java.policy
```

- Java security policy descriptor file configures security permissions.
 - Code from specific codebase (location of Java code, which could be a folder or URL)
 - Code signed with specific digital signatures
 - Access to classes and resources

```
java.policy
```

```
grant codeBase "file:/some.jar" signedBy "Jane, John" {
    permission java.net.SocketPermission "localhost:7777", "listen";
    permission java.io.FilePermission "/someFile", "read, write";
};
```

- See <https://docs.oracle.com/en/java/javase/11/security/java-se-platform-security-architecture.html> for more information



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Java policy file describes permissions for different security principals to access resources and execute restricted actions.

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
      [, Principal [principal_class_name]
"principal_name"]
      [, Principal [principal_class_name]
"principal_name"] ... {
    permission permission_class_name [ "target_name" ]
      [, "action"] [, SignedBy "signer_names"];
    permission ...
};
```

Control Access Using Permissions

Java security API enables verification of permissions.

- There are many types of permissions available.
- All permissions descend from `java.security.Permission` class.
- Permissions are set using `java.policy` files.
- Check permissions before performing an action that accesses resources.
- If access is denied, an `AccessControlException` is thrown.

```
SocketPermission =  
    new SocketPermission("localhost:7777", "accept, connect, listen");  
FilePermission =  
    new FilePermission("/someFile", "read, write");  
try {  
    AccessController.checkPermission(socketPermission);  
    AccessController.checkPermission(filePermission);  
} catch(AccessControlException e) {  
    /* access denied by policies */  
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

If a requested access is allowed, `checkPermission` returns quietly. If denied, an `AccessControlException` is thrown.

`AccessControlException` can also be thrown if the requested permission is of an incorrect type or contains an invalid value.

Execute Privileged Code

Policies can be used to permit access to resources to execute privileged actions.

- Method `doPrivileged` performs the specified `PrivilegedAction` with privileges enabled.
- When making access control decisions, the `checkPermission` method stops checking if it reaches a caller that was marked as "privileged" via a `doPrivileged` call.

```
String file = "/someFile";
String text = AccessController.doPrivileged(
    new PrivilegedAction<String>() {
        public String run() {
            try {
                return Files.readString(Path.of(file));
            } catch (IOException ex) {
                logger.log(Level.SEVERE, "Error reading text", ex);
            }
        }
    });
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Secure Flesystem and IO Operations

Protect against directory traversal attacks:

(attempts to guess directory structure by using ".../somepath" relative paths)

- Remove redundant elements from the path and convert it to canonical form using methods
 - normalize
 - toRealPath(LinkOption)

Protect against DoS attack:

(attempts to make your program process too much data or take too long to process)

- Verify expected sizes of files and lengths of streams
- Monitor IO operations to detect excessive use
- Terminate operations that process excessive amount of data
- Time out lengthy operations to release resources



Deserialize cautiously:

(Deserialization constructs object bypassing normal constructor behaviors)

- It effectively creates an object that may sidestep security checks.
- Deserialization of untrusted data is dangerous.
- Like other means of setting state, you should first validate values.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Best Practices for Protecting your Code

Enforce tight encapsulation:

- Use Java Platform Module System to protect classes from reflection.
- Encapsulate with the most restrictive permissions possible.

Make objects as immutable as possible:

- Consider that a final variable can reference an object that is itself mutable
- Operate on cloned replicas of such objects to protect an otherwise immutable class design

Do not break subclass assumptions about inherited code:

- Beware of changes to superclass method implementations.
- Beware of introduction of new methods by a superclass.

Design classes and methods for inheritance or declare them final or private:

- Non-final and non-private classes and methods can be maliciously overridden by an attacker.
- Use factory methods to perform validations before invoking constructors.
- Don't invoke overridable methods from constructors.

Protect byte-code against tampering and dangerous behavior:

- Many environments may generate or modify Java byte-codes.
- Beware of the command-line arguments `-Xverify:none` or `-noverify`, which disable byte-code verification. This is almost always a very bad idea.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Consider this example of ensuring that mutable `Date` object does not compromise immutable design of a `Product` class:

```
public class Product {
    private final Date date;
    public Product(Date date) {
        this.date = new Date(date.getTime());
    }
    public Date getDate() {
        return (Date)date.clone();
    }
}
```

Alternatively use immutable alternatives to `java.util.Date`, such as `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, or `Instant` classes.

Subclass may assume that certain behavior in a superclass is safe and secure, but a change to this superclass implementation can break these assumptions.

By disabling byte-code verification, you are trusting all these components in your entire stack are completely safe and bug-free.

Erroneous Value Guards

Guard against overflowed number space.

- Methods `xxxExact` of the `Math` class provide mathematical operations.
- Throw `java.lang.ArithmaticException` in case of the value overflow.

Guard against bad floating point values.

- Method `isInfinite` guards against positive or negative floating point number infinity.
- Method `isNaN` guards against division by 0.0 or infinities minus infinities.

Guard against null references.

- Class `Optional` provides a wrapper for object references.

```
final int MAX = Integer.MAX_VALUE;
int value = Math.addExact(MAX, 1);
double badValue = 1/Double.MIN_VALUE; // or 1/-Double.MIN_VALUE
boolean infinite = Double.isInfinite(badValue);
boolean NaN = Double.NaN(untrusted_double);
Optional<Product> o = someDatabase.findProduct(101);
if (o.isPresent()) {
    Product p = o.get();
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

The space reserved to store an integer value is -2^{15} to $2^{15} - 1$. If you exceed the lower or upper bound, you'll wrap around the value space. This is known as a silent overflow. There are methods that guard against this by producing exceptions. In this example, `addExact` produces an `ArithmaticException` when attempting to add 1 to the highest possible integer value.

The result of a floating point operation may be too high or low to be represented by the memory space that backs a primitive floating point value. These values may be considered positive or negative infinities. Similarly, you may have issues dealing with operations that divide by zero, which are not a number (NaN). There are methods to check for these scenarios. These two examples show boolean methods for checking that a floating point is infinite or NaN.

Protect Sensitive Data (Part 1)

Protect sensitive data such as user's ID, address, or credit card numbers.

- Consider scrambling such data.
- Clean it out of memory as soon as possible.
- Remove it from exceptions.
- Do not serialize it or write to log files.
- This information could be used to commit fraud and identity theft.

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] digest = md.digest(value.getBytes());
String hash = (new BigInteger(1, digest)).toString(16);
```

```
try {
    /* actions that may result in errors
       containing sensitive information */
} catch(SQLException e) {
    logger.error("Reconstructed Message");
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Protect Sensitive Data (Part 2)

Encrypt and Decrypt Values

- Use `javax.crypto` API
- Support different types of encryption algorithms

```
String text = "Value that requires encryption";
SecretKey key = KeyGenerator.getInstance("AES").generateKey();
Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] value = text.getBytes();
byte[] encryptedValue = cipher.doFinal(value);
GCMParameterSpec ps = cipher.getParameters().getParameterSpec(GCMParameterSpec.class);
cipher.init(Cipher.DECRYPT_MODE, key, ps);
byte[] decryptedValue = cipher.doFinal(encryptedValue);
```

- ❖ Refer to <https://docs.oracle.com/en/java/javase/11/docs/specs/security/standard-names.html> for the list of supported standard security algorithm names.



Prevent SQL Injections

Basic Statement object use of parameter concatenation presents a risk of SQL injections.

- Problem: User can submit an arbitrary SQL statement as a parameter value.
- Solution:
 - Use PreparedStatement object with substitution parameters.
 - Or use basic Statement method `enquoteLiteral(param)` to sanitize parameter value.

```
"Tea'; drop table products;"
```

```
s.executeQuery("select id, price from products where name like '"+param+"'");
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

Prevent JavaScript Injections

Web UI applications can suffer from JavaScript code injections.

- Problem: Arbitrary code (including JavaScript) can be passed as parameter, cookie, header, or url value.
- Solution: Validate and sanitize every value processed by the browser.

Java Script Code

❖ Note: This is easier said than done; consider using open-source libraries such as OWASP to automate this process.

<input type='text' name='param'>

```
import org.owasp.html.Sanitizers;
import org.owasp.html.PolicyFactory;
PolicyFactory sanitizer = Sanitizers.FORMATTING.and(Sanitizers.BLOCKS);
String cleanResults = sanitizer.sanitize(param);
```

❖ Note: Construction of Java web applications and the use of JavaScript are covered by other courses:

[Developing Applications for the Java EE 7 Platform](#)

[JavaScript and HTML5 Develop Web Applications](#)



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Prevent XML Injections

XML documents may contain entities or namespaces with code injections.

- Problems:
 - Large entities that overwhelm parsing program
 - Recursive references that send parser into an indefinite loop
 - External malicious URL references
 - Inclusions of sensitive files
- Solutions:
 - Instruct XML Parser to stop processing documents when unsafe constructs are detected.
 - Use open-source libraries such as OWASP to apply XML data sanitation.

```
<?xml version="1.0"?>
<!DOCTYPE some [
    <!ENTITY x "&y;">
    <!ENTITY y "&x;">
]>
<some>&x;</some>
```

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

- ❖ Note: Consider using JAXB API to map XML documents to Java Classes.
- ❖ JAXB API is covered by the [Developing Applications for the Java EE 7 Platform](#) course.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Discover and Document Security Issues

Test software for possible vulnerabilities:

- Security/penetration testing is different from normal function or system testing.
- Your goal is to try to break software, extract sensitive information, or perform unauthorized actions.

Document discovered security bugs:

- Reports let others know what to be aware of and plan accordingly until a fix is ready.
- Customers may not appreciate you keeping secrets when they're at risk.
- It's dangerous to mess with code you don't understand.
- Reports help the right people fix issues and anticipate/prevent similar issues in the future.

Inject SQL	Failed
Attack network socket	Succeed
Intercept sensitive data	Failed
Gain unauthorised access	Failed



Summary

In this lesson, you should have learned how to:

- Apply restriction using security policies
- Protect code
- Validate values
- Protect sensitive data
- Prevent injections
- Discover and document security issues



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Advanced Generics



ORACLE®



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives



After completing this lesson, you should be able to:

- Understand Generics erasure
- Use Generics with wildcards



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

2

Compiler Erases Information About Generics

Generics information can be erased from the compiled code.

- Compiler verifies type-safety of your code before erasing generics.
- Compiler adds relevant type-casting operations.

```
public class Some<T> {  
    public T apply(T t) {  
        return t;  
    }  
}
```

original source

compiler →

```
public class Some {  
    public Object apply(Object t) {  
        return t;  
    }  
}
```

generated code

```
Some<String> s = new Some<>();  
String v = s.apply("");
```

original source

compiler →

```
Some<String> s = new Some<>();  
String v = (String)s.apply("");
```

generated code



Generic and Raw Type Compatibility

When applying generics to override methods:

- Compiler verifies type-safety of your code before erasing generics
- Adds a synthetic (compiler generated) bridge method
- Bridge method compiles with the non-generics signature of the method that your code is overriding
- Bridge method invokes your non-synthetic method, applying type-casting to the generic type
- No type-casting needs to be applied to the code that invokes such an operation

```

public class Some
    implements UnaryOperator<String> {
    public String apply(String s) {
        return s;
    }
}

Some<String> s = new Some<>();
String v = s.apply("");

```

original source

compiler →

```

public class Some
    implements UnaryOperator<String> {
    public String apply(final String s) {
        return s;
    }
    /* bridge synthetic method
     * generated by compiler */
    public Object apply(final Object o) {
        return this.apply((String)o);
    }
}

```

generated code



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

4

You can use reflection API to dynamically discover the actual code that the compiler would generate:

```

Stream.of(Some.class.getDeclaredMethods())
    .forEach(x-> {
        boolean isBridge = x.isBridge();           // true
        for bridge methods or false otherwise
        boolean isSynthetic = x.isSynthetic(); // true
        for synthetic (compiler-generated) methods false
        otherwise
        String methodSignature = x.toString(); // actual
        signature of a method
    });

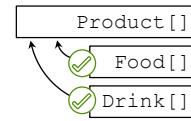
```

Generics and Type Hierarchy

Generics are invariant to enforce compile-time verification of types.

- Java arrays are **covariant**, which can result in runtime executions

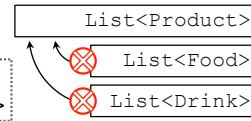
```
Product[] products = new Food[10];
products[0] = new Drink("Tea");
```



- Collection API uses generics that are **invariant**; code is validated at compile time.

```
List<Product> products = new ArrayList<Food>();
```

Compiler error indicating incompatible types:
ArrayList<Food> cannot be converted to List<Product>



- Generics compiler checks are not performed for raw types, which can result in runtime exceptions.

```
List<Food> foods = new ArrayList<Food>();
List values = foods;
List<Product> products = values;
products.add(new Drink("Tea"));
Drink x1 = (Drink)values.get(0);
Food x2 = foods.get(0);
```

Compiler warns about unchecked or unsafe operations
No compiler warning
java.lang.ClassCastException: class demos.Drink
cannot be cast to class demos.Food



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Invariant behavior is needed to guarantee compiler time-type safety checks.
Use of raw types circumvents generics mechanism, so compiler can no longer tell if your assignees are type-safe or not. Therefore, compiler will produce a warning telling that your code cannot be verified for type-safety.

Wildcard Generics

Generics are used to enforce compile-time verification of a type.

Generics are often used with Collection API. Consider the following collection examples:

- When **generics are not applied**, code defaults to use type Object.
 - Only Object class operation can be safely used.
 - Type-check and type-casting must be applied to access any sub-type specific operations.
- When **specific type is applied**:
 - Any operations declared for this type or its parents can be safely used
 - Type-check and type-casting must be applied to access any sub-type specific operations
- When **wildcard <?> is applied** (representing an unknown type):
 - Elements are accessed just like in a collection of Objects
 - Effectively it's a read-only collection
 - No object in Java is of unknown type, so no values (except null) can be added to such a collection

```
// Add, remove and manipulate instances of Object class or its descendants:
List listOfAnyObjects1 = new ArrayList();
List<Object> listOfAnyObjects2 = new ArrayList<>();
// Add, remove and manipulate instances of Product class or its descendants:
List<Product> listOfProducts = new ArrayList<>();
// Nothing could be added to this list, except null:
List<?> listOfUnknownType = listOfProducts;
```

 Note: Such an assignment is covariant.



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

In generic code, the question mark (?), called the wildcard, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable, sometimes as a return type (though it is a better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Unbounded generics type is used:

- If you are writing a method that can be implemented using functionality provided in the Object class
- When the code is using methods in the generic class that don't depend on the type parameter, for example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Upper Bound Wildcard

Upper bounded wildcard `<? extends ParentType>` allows use of subtype collections.

- A list of specific type `List<Product>`
 - Is writable - you can add instances of `Product`, `Food` and `Drink` to such a list
 - Is **invariant** - you cannot assign a `List<Drink>` or `List<Food>` to such a list
- A list of super type and descendants `List<? extends Product>`
 - Is read only - no values (except null) can be added to such a list
 - Is **covariant** - you can assign a `List<Drink>` or `List<Food>` to such a list

```
public void setProducts(List<Product> products) { }
public void setProductAndSubtypes(List<? extends Product> products) { }
```



```
Product p1 = new Food("Cake",2.99);
Product p2 = new Drink("Tea",1.99);
Product p3 = new Food("Cookie",2.99);
List<Products> products = List.of(p1,p2,p3);
List<Food> foods = List.of((Food)p1,(Food)p3);
setProducts(products);
setProductAndSubtypes(products);
setProductAndSubtypes(foods);
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Collection of an unknown type is covariant:

```
List<?> anyValues = new ArrayList<Product>();
List<? extends Object> anyObjects = new
ArrayList<Product>();
```

These two collections essentially work in the same way, because everything in Java no matter the type is eventually type of Object.

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on `List<Integer>`, `List<Double>`, and `List<Number>`; you can achieve this by using an upper bounded wildcard. To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the `extends` keyword, followed by its upper bound. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

Covariant means that specific type can be assigned to its ancestor generic type

Invariant means that specific type can be assigned to its ancestor generic type

Lower Bound Wildcard

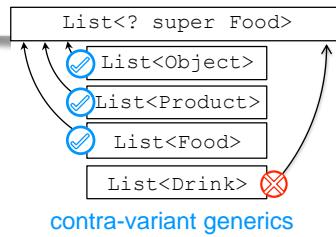
Lower bounded wildcard `<? super Type>` allows to use this type and its parents.

- A list of specific type `List<Food>`
 - Is **invariant** - you can only assign a `List<Food>` to such a list
- A list of type and its parents `List<? super Food>`
 - Is **writable** - you can add instances of `Object`, `Product` and `Food` to such a list
 - Is **contra-variant** - you can **assign** a `List<Food>` or `List<Product>` or `List<Object>` to such a list

```
public void addFoodToFoods(List<Food> order, Food food) {
    order.add(food);
}
public void addFoodToFoodParents(List<? super Food> order, Food food) {
    order.add(food);
}
```

```
List<Product> products = ...
List<Food> foods = ...
Food f = new Food("Cake", 2.99);
addFoodToFoods(foods, f);
addFoodToFoods(products, f);
addFoodToFoodParents(foods, f);
addFoodToFoodParents(products, f);
```

✖ List of products may contain any products, including food and drink types, but methods above only allow to add food to such a list.



Collections and Generics Best Practices

Collections and generics wildcard best practices:

- When class hierarchy (super/sub types) is irrelevant
 - Use specific type `<SpecificType>` **invariant, read-write** generics
 - This allows type-safe read-write access to the collection.
- When collection is a consumer of values and your code needs to be type-hierarchy aware
 - Use `<? super LowerBoundType>` **contravariant, writable** generics
 - This allows type-safe addition of new values to the collection.
- When collection is a producer of values and your code needs to be type-hierarchy aware
 - Use `<? extends UpperBoundType>` **covariant, read-only** generics
 - This allows type-safe retrieval of values from the collection.
- Avoid using raw types

```
public void addFood(List<? super Food> o, Food i) {o.add(i);}
public void addDrink(List<? super Drink> o, Drink i) {o.add(i);}
public void processOrder(List<? extends Product> o) {o.stream().forEach(p->p.prepare());}
public void addProductAndProcessOrder(List<Product> o, Product i) {
    o.add(f);
    o.stream().forEach(p->p.prepare());
}
```



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

```
public void addFood(List<? super Food> order, Food food)
{
    // use Food specific behaviours
    order.add(food);
}

public void addDrink(List<? super Drink> order, Drink drink) {
    // use Drink specific behaviours
    order.add(drink);
}

public void processOrder(List<? extends Product> order) {
    // use any Product behaviours
    order.stream().forEach(p->p.prepare());
}

public void addProductAndProcessOrder(List<Product> order, Product product) {
    // use any Product behaviours
    order.add(product);
    order.stream().forEach(p->p.prepare());
}
```

Summary

After completing this lesson, you should be able to:

- Understand Generics erasure
- Use Generics with wildcards



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10