



# Java SE 21 Programming Complete

Student Guide

D1107035GC10

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



**Copyright © 2024, Oracle and/or its affiliates.**

## **Disclaimer**

This document contains proprietary information and is protected by copyright and other intellectual property laws. 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 and is not warranted to be error-free. If you find any errors, please report them to us in writing.

## **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 END USERS:** Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

## **Trademark Notice**

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

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

## **Third-Party Content, Products, and Services Disclaimer**

This documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

1006252024

## Contents

### I Course Introduction

Course Goals I-2

Audience I-3

Course Structure I-4

### 1 Introduction to Java

Objectives 1-2

What Is Java? 1-3

How Java Works 1-4

Object-Oriented Principles 1-5

Classes and Objects 1-6

Classes 1-7

Objects 1-8

Inheritance 1-9

Java APIs 1-10

Java Keywords, Reserved Words, and Special Identifiers 1-11

Java Naming Conventions 1-12

Java Basic Syntax Rules 1-13

Defining a Java Class 1-14

Accessing Classes Across Packages 1-15

Implementing Encapsulation with Access Modifiers 1-17

Creating a Main Application Class 1-18

Compiling a Java Program 1-19

Executing a Java Program 1-20

Comments and Documentation 1-21

Code Snippets in Javadoc 1-23

External Snippets 1-24

Summary 1-25

Practices for Lesson 1: Overview 1-26

### 2 Primitive Types, Operators, and Flow Control Statements

Objectives 2-2

Java Primitives 2-3

Declaring and Initializing Primitive Variables 2-4

Primitive Declarations and Initializations: Restrictions 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 Logical 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
Switch -> No Fall-Through Syntax	2-17
Switch Expressions yield a Value	2-18
Switch Statements and Expressions Summary	2-19
Using JShell (REPL Tool)	2-20
JShell Commands	2-21
Code Snippets	2-22
Summary	2-23
Practices for Lesson 2: Overview	2-24

### **3 Text, Date, Time, and Numeric Objects**

Objectives	3-2
String Initialization	3-3
String Operations	3-4
String Indexing	3-5
Mutable Text Objects	3-6
Text Blocks	3-7
Spaces, Lines, and Quotes	3-9
Wrapper Classes for Primitives	3-10
Representing Numbers Using BigDecimal Class	3-11
Method Chaining	3-12
Local Date and Time API	3-13
More Local Date and Time Operations	3-14
Instants, Durations, and Periods	3-16
Zoned Date and Time	3-18
Representing Languages and Countries	3-19
Formatting and Parsing Numeric Values	3-21
Formatting and Parsing Date and Time Values	3-23
Localizable Resources	3-26
Formatting Message Patterns	3-27
Formatting and Localization: Summary	3-29

Summary 3-31

Practices for Lesson 3: Overview 3-32

#### **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

Defining Instance Variables 4-9

Defining 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

Defining Constants 4-16

Static Context 4-17

Accessing Static Context 4-18

Combining Static and Final 4-19

Other Static Context Use Cases 4-20

IntelliJ IDE: Introduction 4-21

Summary 4-22

Practices for Lesson 4: Overview 4-23

#### **5 Improved Class Design**

Objectives 5-2

Overload Methods 5-3

Variable Number of Arguments 5-4

Defining Constructors 5-5

Reusing Constructors 5-6

Access Modifiers: Summary 5-7

Defining Encapsulation 5-8

Defining Immutability 5-9

Constants and Immutability 5-10

Enumerations 5-11

Complex Enumerations 5-13

Java Memory Allocation 5-14

Parameter Passing 5-15

Java Memory Cleanup 5-16

Summary 5-17

Practices for Lesson 5: Overview 5-18

## 6 Implement Inheritance and Use Records

- Objectives 6-2
- Extending Classes 6-3
- Object Class 6-4
- Reusing Parent Class Code Through Inheritance 6-5
- Instantiating Classes and Accessing Objects 6-6
- Rules of Reference Type Casting 6-7
- Verifying Object Type Before Casting the Reference 6-8
- Pattern Matching for instanceof 6-9
- Reference Code Within the Current or Parent Object 6-10
- Defining Subclass Constructors 6-11
- Class and Object Initialization: Summary 6-12
- Overriding Methods and Using Polymorphism 6-13
- Reusing Parent Class Logic in Overwritten Method 6-15
- Defining Abstract Classes and Methods 6-16
- Defining Final Classes and Methods 6-17
- Sealed Classes and Interfaces 6-18
- Overriding Object Class Operations: `toString` 6-20
- Overriding Object Class Operations: `equals` 6-21
- Override Object Class Operations: `hashCode` 6-22
- Comparing String Objects 6-23
- Java Records 6-24
- Custom Record Constructors 6-25
- Record Patterns 6-26
- Pattern Matching for switch 6-27
- Factory Methods 6-28
- Summary 6-29
- Practices for Lesson 6: Overview 6-30

## 7 Interfaces and Generics

- Objectives 7-2
- Java Interfaces 7-3
- Multiple Inheritance Problem 7-5
- Implement Interfaces 7-6
- Default, Private, and Static Methods in Interfaces 7-7
- Interface Hierarchy 7-8
- Default Methods Inheritance 7-9
- Interface Is a Type 7-10
- Functional Interfaces 7-11
- Generics 7-12
- Use Generics 7-13

Examples of Java Interfaces: java.lang.Comparable	7-15
Examples of Java Interfaces: java.util.Comparator	7-16
Examples of Java Interfaces: java.lang.Cloneable	7-17
Composition Pattern	7-18
Summary	7-19
Practices for Lesson 7: Overview	7-20

## 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 for Lesson 8: Overview	8-14

## 9 Collections

Objectives	9-2
Introduction to Java Collection API	9-3
Java Collection API Interfaces	9-4
Java Collection API Implementation Classes	9-5
Java Collection API Interfaces and Implementation Classes	9-6
Create List Object	9-7
Manage List Contents	9-8
Create Set Object	9-9
Manage Set Contents	9-11
Create Deque Object	9-12
Manage Deque Contents	9-13
Create HashMap Object	9-15
Manage HashMap Contents	9-16
Iterate Through Collections	9-17
Sequenced Collections	9-18
Other Collection Behaviors	9-20
Use java.util.Collections Class	9-21
Access Collections Concurrently	9-22

Prevent Collections Corruption 9-23

Legacy Collection Classes 9-25

Summary 9-26

Practices for Lesson 9: Overview 9-27

## 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 for Lesson 10: Overview 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
Spliterator	11-28
Spliterator Characteristics	11-29
Summary	11-30
Practices for Lesson 11: Overview	11-31

## **12 Exception Handling, Logging, and Debugging**

Objectives	12-2
Using Java Logging API	12-3
Logging Method Categories	12-4
Guarded Logging	12-5
Log Writing Handling	12-6
Logging Configuration	12-7
Describe Java Exceptions	12-8
Create Custom Exceptions	12-9
Throwing Exceptions	12-10
Catching Exceptions	12-11
Exceptions and the Execution Flow	12-12
Helpful NullPointerExceptions	12-13
Example Throwing an Unchecked Exception	12-14
Example Throwing a Checked Exception	12-15
Handling Exceptions	12-16
Resource Auto-Closure	12-17
Suppressed Exceptions	12-18
Handle Exception Cause	12-19
Java Debugger	12-20
Debugger Actions	12-21
Manipulate Program Data in Debug Mode	12-22
Validate Program Logic Using Assertions	12-23
Normal Program Flow with No Exceptions	12-24
Program Flow Producing a Runtime Exception	12-26
Program Flow Catching Specific Checked Exception	12-28
Program Flow Catching Any Exceptions	12-30
Summary	12-32
Practices for Lesson 12: Overview	12-33

## 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 13-7
- Basic Character Data Reading and Writing 13-8
- Connecting Streams 13-9
- Standard Input and Output 13-10
- Using Console 13-11
- Understand Serialization 13-13
- Serializable Object Graph 13-14
- Object Serialization 13-15
- Serialization of Sensitive Information 13-16
- Customize Serialization Process 13-17
- Serialization and Versioning 13-18
- Working with Filesystems 13-19
- Constructing Filesystem Paths 13-20
- Navigating the Filesystem 13-21
- Analyze Path Properties 13-22
- Set Path Properties 13-23
- Create Paths 13-24
- Create Temporary Files and Folders 13-25
- Copy and Move Paths 13-26
- Delete Paths 13-27
- Handle Zip Archives 13-28
- Represent Zip Archive as a FileSystem 13-29
- Access HTTP Resources 13-31
- Summary 13-32
- Practices for Lesson 13: Overview 13-33

## 14 Java Concurrency and Multithreading

- Objectives 14-2
- Java Concurrency Concepts 14-3
- Implement Threads 14-4
- Thread Life Cycle 14-5
- Interrupt Thread 14-6
- Block Thread 14-7
- Make Thread Wait Until Notified 14-8
- Common Thread Properties 14-10

Create Executor Service Objects	14-11
Manage Executor Service Life Cycle	14-14
Implementing Executor Service Tasks	14-16
Locking Problems	14-18
Writing Thread-Safe Code	14-19
Ensure Consistent Access to Shared Data	14-21
Nonblocking Atomic Actions	14-22
Ensure Exclusive Object Access Using Intrinsic Locks	14-23
Intrinsic Lock Automation	14-24
Nonblocking Concurrency Automation	14-25
Alternative Locking Mechanisms	14-26
CPU Versus IO Bound Concurrent Tasks	14-27
Virtual Threads API	14-28
Virtual Thread Operations	14-29
Summary	14-30
Practices for Lesson 14: Overview	14-31

## 15 Modules and Deployment

Objectives	15-2
Compile, Package, and Execute Nonmodular Java Applications	15-3
Nonmodular Java Characteristics	15-4
What Is a Module?	15-5
Java Modules	15-7
Java Module Categories	15-8
Define Module Dependencies	15-9
Export Module Content	15-10
Module 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 module	15-19
Create Custom Runtime Image	15-20
Execute Runtime Image	15-22
Optimize a Custom Runtime Image	15-23
Check Dependencies	15-24
Summary	15-25
Practices for Lesson 15: Overview	15-26

**A Annotations**

- Objectives A-2
- Annotations: Introduction A-3
- Design Annotations A-4
- Apply Annotations A-5
- Dynamically Discover Annotations A-6
- Document the Use of Annotations A-7
- Annotations that Validate Design A-8
- Deprecated Annotation A-9
- Suppress Compiler Warnings A-10
- Varargs and Heap Pollution A-11
- Summary A-12

**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 as Nonfunctional Requirement C-3
- Security Threats C-4
- Denial of Service (DoS) Attack C-5
- Define Security Policies C-6
- Changes in the Security API C-8
- Secure File System and I/O Operations C-9
- Best Practices for Protecting Your Code C-10
- Erroneous Value Guards C-11
- Protect Sensitive Data (Part 1) C-12
- Protect Sensitive Data (Part 2) C-13

Prevent SQL Injections	C-14
Prevent JavaScript Injections	C-15
Prevent XML Injections	C-16
Discover and Document Security Issues	C-17
Summary	C-18

## 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

## E Java Applications on Oracle Cloud

Objectives	E-2
Cloud Application Requirements	E-3
Cloud Application Runtime Infrastructure	E-4
Cloud Java Application Servers	E-5
Package and Deploy Cloud Application	E-6
Optimise deployment with GraalVM	E-7
HTTP Protocol Basics	E-9
REST Service Conventions and Resources	E-11
Configure and Launch REST Service Application Using Helidon SE	E-12
Summary	E-13

## F Miscellaneous Java Topics

Objectives	F-2
Builder Design Pattern	F-3
Singleton Design Pattern	F-4
Java Regular Expression API	F-5
Regular Expressions: Character Classes	F-6
Regular Expressions: Quantifiers	F-7
Regular Expressions: Boundaries	F-8
File IO Watch Service	F-9
Summary	F-11

Oracle Internal & Oracle Academy Use Only

# Course Introduction

---

This course provides an introduction to the Oracle Database 12c Release 2 (12.2) platform. It is designed for individuals who want to learn how to manage and administer Oracle Database 12c. The course covers the basic concepts of Oracle Database, including its architecture, data storage, and retrieval mechanisms. It also covers the various components of the Oracle Database system, such as the Oracle Database Server, Oracle Database Client, and Oracle Database Utilities.

## 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, I/O, and Concurrency
- Deploy Java SE applications



## 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 21 Certification exam



O

# Course Structure

**Lesson 1:** Introduction to Java

**Lesson 2:** Primitive Types, Operators, and Flow Control Statements

**Lesson 3:** Text, Date, Time, and Numeric Objects

**Lesson 4:** Classes and Objects

**Lesson 5:** Improved Class Design

**Lesson 6:** Implement Inheritance and Use Records

**Lesson 7:** Interfaces and Generics

**Lesson 8:** Arrays and Loops

**Lesson 9:** Collections

**Lesson 10:** Nested Classes and Lambda Expressions

**Lesson 11:** Java Streams API

**Lesson 12:** Exception Handling, Logging, and Debugging

**Lesson 13:** Java I/O API

**Lesson 14:** Java Concurrency and Multithreading

**Lesson 15:** Modules and Deployment

## Extras:

**Appendix A:**

Annotations

**Appendix B:**

Java Database Connectivity

**Appendix C:**

Java Security

**Appendix D:**

Advanced Generics

**Appendix E:**

Java Applications on Oracle Cloud

**Appendix F:**

Miscellaneous Java Topics



O

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

# Introduction to Java

---

# 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 a Java class with main method
- Compile and execute a Java application



# What Is Java?

- It is a general-purpose programming language similar to C and C++.
- It is object oriented and platform independent.
- It was 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 Standard Edition (SE) version 21.

❖ **Java Editions:**

**Java Card** - *Smart Card Edition*

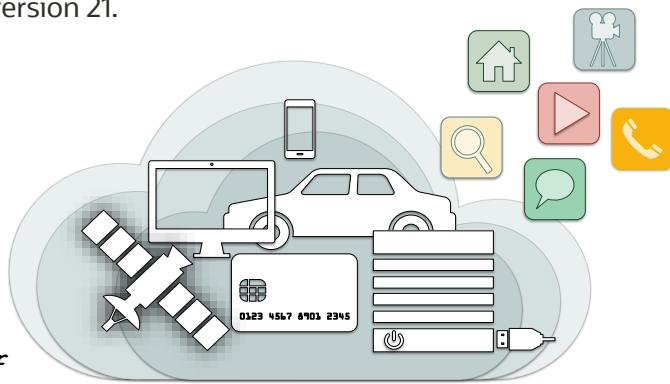
**Java ME** - *Micro Edition*

**Java SE** - *Standard Edition*

**Java MP** - *Micro Profile*

**Jakarta EE** - *(Java EE) Enterprise Edition*

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



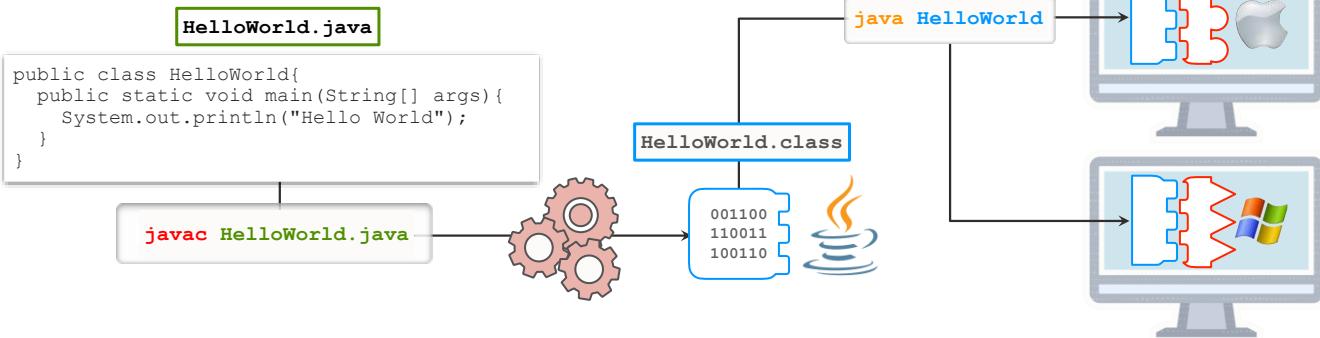
You start learning Java with Java SE because this is the base edition, representing the core of the Java language. All other Java editions represent more specialized use cases of Java that target particular environments, such as SIM cards (Java Card), mobile devices and smart TVs (Java ME), or application servers (Jakarta EE) previously known as a Java EE. The Micro Profile specification borrows many of its features from Jakarta EE, and is used in the production of microservices.

Starting from version 9, new versions of Java are released every 6 months. However, only some of these new versions are considered Long Term Support (LTS) versions. This course is based on Java SE version 21, which are the LTS versions.

# 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!

O

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 together with the Java run time for a given platform. Such a 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.

Java compiler, Java Virtual Machine together with many other essential tools and libraries are availed as part of the Java SE Development Kit (JDK). You can download JDK from here:

<https://www.oracle.com/javadownload>

# Object-Oriented Principles

- **Abstraction:** The software objects represent a subset of behaviors and information of the real-world objects.
- **Encapsulation:** The software object hides its implementation details and internal state.
- **Inheritance:** A specialized type (child) inherits code from the more general type (parent).
- **Polymorphism:** Objects of different types respond differently to the same message.

O

Abstraction is a fundamental OO principal because finding the right abstractions enables you to design only those features defined by the system requirements and to ignore all other details that are irrelevant to the problem being solved. The abstraction object is a representation of the real-world object with currently irrelevant behavior and data hidden.

Encapsulation describes the ability of an object to hide its data and operations from the rest of the world and is one of the fundamental principles of object-oriented programming. In Java, a class encapsulates the **fields** (properties that represent data which hold the state of an object), and the **methods** (operations that define the actions of the object). Encapsulation enables you to write reusable programs. It also enables you to restrict access only to those features of an object that are declared public. All other fields and methods are private and can be used for internal object processing. Objects may respond to messages from other objects via publicly exposed operations, and forbid direct manipulation with their internal state.

Inheritance enables a specialized type (child) to inherit properties and behaviors from the corresponding general type (parent). This can improve code reusability.

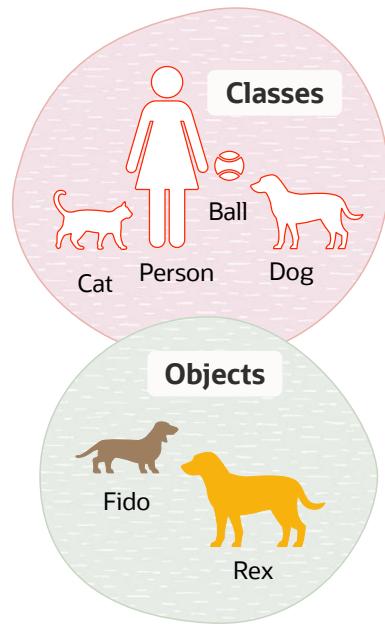
# Classes and Objects

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.
- Class is an abstraction of a real-world thing or concept.
- An object is a specific **instance** (example of) a class.

```
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();
    }
}
```



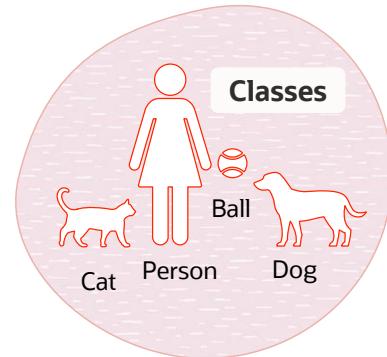
The term **class** means type of thing and **object** means instance (example) of that type. However, sometimes people use these terms interchangeably, which can be quite confusing.

# Classes

- 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**) it is capable of.
- Operations implement program logic (**algorithms**):
  - A Dog could bark and fetch a Ball.
  - A Cat could meow but is not likely to play fetch.

```
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();
    }
}
```



O

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," "property," and "field" to be synonymous. They all represent the same concept—a way of defining information and data storage within the class.

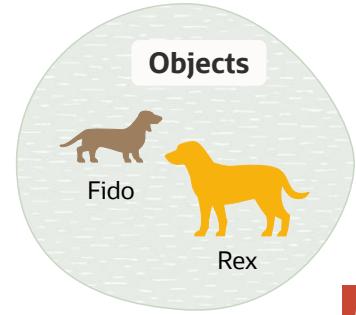
Java has procedural language capabilities; that is, you write algorithms contained within functions defined by your classes.

# Objects

- Each object would be capable of having **specific values** for each attribute defined by a class that represents its type.
- Examples:
  - 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 by 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.

```
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();
    }
}
```



Reference is a way of pointing to a given object. A computer needs to place objects into memory and references are eventually resolved as memory addresses where these objects were placed. A reference can be used to access object properties and invoke operations on the object. Also, references can be passed to operations as parameters. This allows invokers of the operation to let it know which specific object it should work with.

# Inheritance

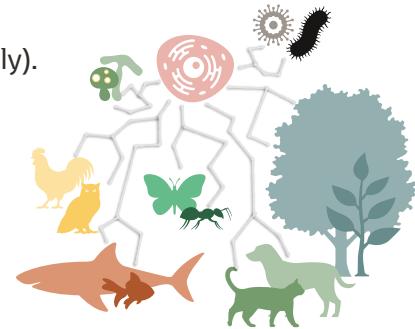
You can reuse (inherit) attributes and behaviors across class hierarchy.

- 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  
}
```

Inherited

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



O

## 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, and 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. Java SE documentation can be found at:  
<https://docs.oracle.com/en/java/javase/index.html>



O

# Java Keywords, Reserved Words, and Special Identifiers

available since 1.0

reserved words for literal values

no longer in use

added in 1.4

added in 5.0

added in 9.0

added in 10

added in 19

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

## 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.



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 (this is not a good idea, though, because the code looks very confusing), but not as a class name.

The keyword `strictfp` has been introduced in Java 1.2, but since Java SE 17 it is considered redundant.

# 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 onwards.

O

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. Where possible, try to use whole words; avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

Java packages serve as a namespace identifiers, which group Java classes together.

## 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:** The example shows some constructs such as `if/else` and `while` that are covered later in the course.

O

## Defining a Java Class

- **Class name** is typically represented by one or more nouns: Dog, GreatCat.
- Class must be saved into a file with the same name and the **.java** extension.
- Classes are grouped into packages, 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: com.oracle.demos, org.acme.something.
- Package and class name must form a unique combination.

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

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

```
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.

O

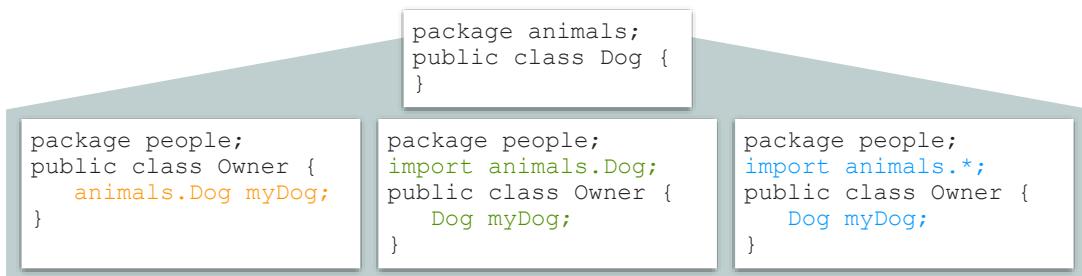
Typically, each **.java** contains only one class definition. However, in some cases, for example with Nested and Inner classes, it is possible to define more than one class in a single **.java** file. These cases are covered in later lessons.

# Accessing Classes Across Packages

To access a class in another package, do one of the following:

- 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 on the 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.



Modularity adds a higher level of aggregation above packages. The key new language element is the module—a uniquely named, reusable group of related packages, as well as resources (such as images and XML files) and a module descriptor specifying:

- Module name
- Module dependencies (that is, other modules this module depends on)
- The packages it explicitly makes available to other modules (all other packages in the module are considered concealed packages internal to the module; they can be used by code inside the module but not by code outside the module)
- Services it offers
- Services it consumes
- To what other modules it allows reflection

The key goals of modularizing the Java SE platform are:

- **Reliable configuration:** Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner that's recognized both at compile time and execution time.
- **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.

Core Java language classes are grouped into packages that are under the “java” package. For example, `java.lang`, `java.math`, `java.time`, `java.io` and so on.

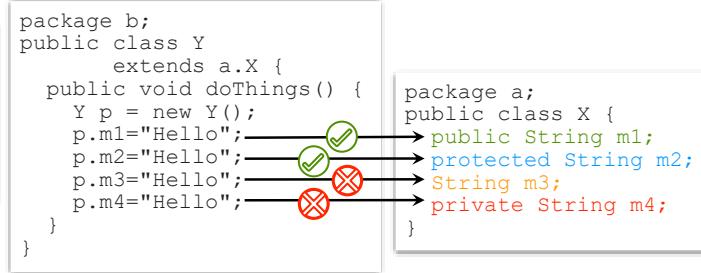
- **Scalable Java platform:** Previously, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain, and evolve. The platform is now modularized into. You can create custom runtimes consisting of only modules you need for your apps or the devices you're targeting.
- **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.
- **Improved performance:** The JVM uses various optimization techniques to improve application performance.
- Modules are covered in later lessons.

# Implementing Encapsulation with 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
- `no access modifier (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>
}
```



## Notes:

- ❖ 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 to such code may adversely affect several other classes that may be using your code.

O

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.

The subclass-superclass relationship (use of the `extends` keyword) is covered in the lesson titled "Implement Inheritance and Use Records".

The default access modifier makes an element visible only to other classes in the same package. Unlike protected access modifier which also makes an element visible to subclasses in other packages.

# Creating a Main Application Class

The main method is the entry point into your application.

- It is the starting point of program execution.
- The method must be called **main**.
- It must be **public**. You intend to invoke this method from outside 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.

O

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

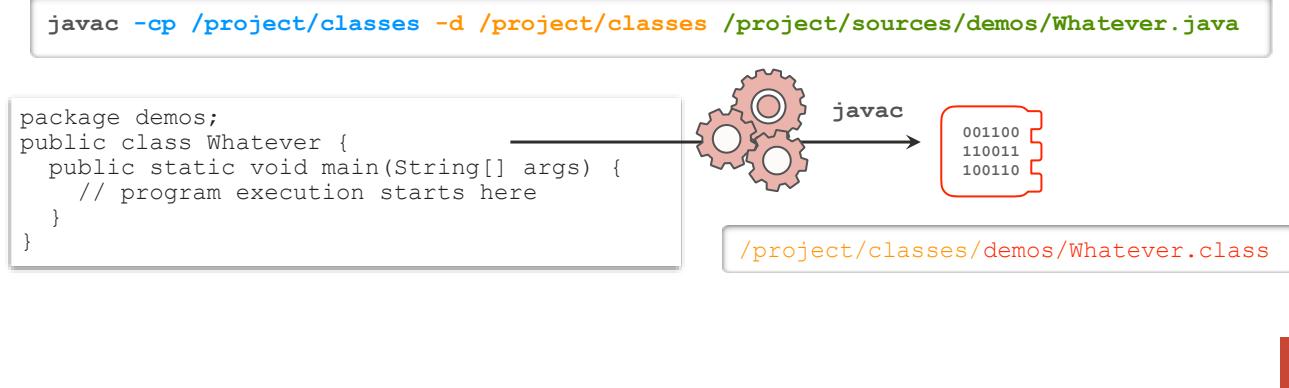
```
public static void main (String[] args) { }  
public static void main (String args[]) { }  
public static void main (String... args) { }
```

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

# Compiling a 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 the compilation result**.  
(The compiler creates **package subfolders** with **compiled class files** in this path.)
- Provide **path to source code**.



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.

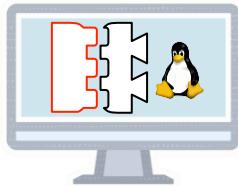
# Executing a Java Program

Execute a program using the `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 a package prefix; do not use the `.class` extension.
- Provide a **space-separated list of parameters** after the class name.

Access command-line parameters:

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



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

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

- 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
```

O

Also, consider this example:

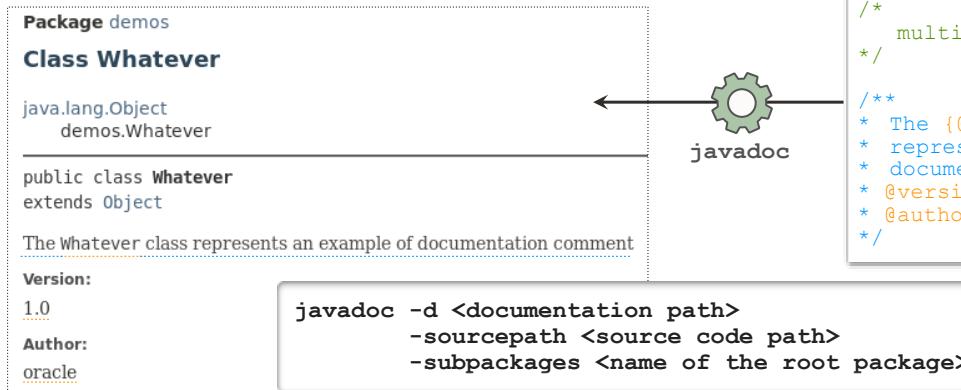
```
cd /projects/classes
javac -d /project/classes /project/sources/demos/Whatever.java
java demos.Whatever
```

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 the @ sign
  - Are used by the Javadoc tool to generate documentation



```

// single-line comment
/*
   multi-line comment
*/
/**
 * The {@code Whatever} class
 * represents an example of
 * documentation comment
 * @version 1.0
 * @author oracle
 */
  
```

❖ **Note:** All APIs in the Java Development Kit are documented using the Javadoc utility.



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

- <https://docs.oracle.com/en/java/javase/21/javadoc/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 Application Programming Interface (API) 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) file names.

**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 `ObjectStreamField` 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>`

## Code Snippets in Javadoc

- A @snippet tag simplifies the inclusion of example source code in documentation.
- Snippet features include highlighting, replacing, and linking code fragments.

```
/**  
 * The following code shows how to use{@code Purchase.isComplete}:  
 * {@snippet :  
 * if (!p.isComplete()) {  
 * LocalDate d; // @link substring="LocalDate" target="java.time.LocalDate"  
 * d = p.completeNow(); // @highlight substring="completeNow"  
 * }  
 * }  
 */
```

✿**Note:** Code snippets were introduced in Java SE 18.

O

Starting from JDK 18, JEP 413: add a @snippet tag for Javadoc's Standard Doclet, to simplify the inclusion of example source code in API documentation.

Also, the standard doclet now supports an --add-script option used to include a reference to an external script file in the file for each page of generated documentation.

It is also possible to use @SuppressWarnings annotation for DocLint messages to suppress messages about issues in documentation comments, when it is not possible or practical to fix the issues that were found.

# External Snippets

- An external snippet refers to a separate file that contains the content of the snippet.
- Snippets may be used in various files, such as Java sources, property bundles.

```
/**  
 * The following code shows how to use{@code Purchase.isComplete}:  
 * {@snippet class="Shop.java" region="example"}  
 */
```

```
public class Shop {  
    void buyProducts(Purchase p) {  
        // @start region="example"  
        if (!p.isComplete()) {  
            p.completeNow();  
        }  
        // @end  
    }  
}
```

O

## Snippet tag reference

Attributes are name-value pairs that provide arguments for snippet tags and markup tags. A value can be an identifier or a string enclosed in either single or double quotes. Escape sequences in strings are not supported. For some attributes, the value is optional and can be omitted.

Attributes for the {@snippet} tag:

- **class** — Class containing the content for the snippet
- **file** — File containing the content for the snippet
- **id** — Identifier for the snippet, to identify the snippet in the generated documentation
- **lang** — Language or format for the snippet
- **region** — Name of a region in the content to be displayed

## 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



## Practices for Lesson 1: Overview

In this practice you will:

- Verify the JDK installation
- Create a `HelloWorld` Java application class with main method
- Compile and execute this application



# Primitive Types, Operators, and Flow Control Statements

---

# Objectives

After completing this lesson, you should be able to:

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



# Java Primitives

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

Whole numbers				Floating point numbers			
<b>byte</b>	<b>short</b>	<b>int</b>	<b>long</b>	<b>float</b>	<b>double</b>		
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			
<b>Boolean</b>		Character (represents a single character value)					
<b>boolean</b>		<b>char</b>					
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 '\\'							

O

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, which has a code of '`\u0020`'.

# Declaring and Initializing Primitive Variables

- Variable declaration and initialization syntax:  
`<type> <variable name> = <value>;`
- A variable can be declared with no immediate initialization, as 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 a = 5, b = 3;
float c = a;
char d = 'A';
char e = '\u0041', f = '\101';
int g;
g = 77;
```

O

In the example in the slide, variables a, b, c, and 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).

In Java SE 7 and later, any number of underscore characters (\_) can appear between digits in a numerical literal. This feature enables you to separate groups of digits in numeric literals, which can improve the readability of your code.

For example:

```
float x = 12_345.98_76f;
```

Note that the underscore characters (\_) can not appear at the beginning or end of a numeric value, including prior to an F or L suffix, as well as adjacent to a decimal point in a floating point literal.

Following example shows illegal positions for the \_ symbol:

```
float x = _12345_.9876_f; // This is not allowed!
```

## Primitive Declarations and Initializations: 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 to **not** compile.

O

# Java Operators

Java operators in the order of precedence:

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

O

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 have 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--; // 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--; // decrement postfix (b is 3, a is 2)
b = --a; // decrement prefix (b is 1, a is 1)
```

O

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.

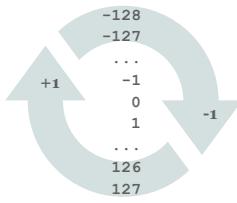
Operator + can be used to indicate a positive number; however, it is considered excessive, because numbers are assumed to be positive anyway.

# 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;           // compilation fails
✗ byte c = a+b;               int
✓ 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
```



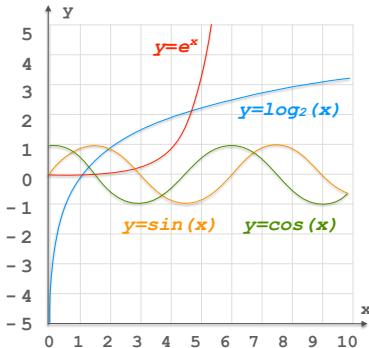
Type casting makes sense if you think about similarities between different types. For example, you could say that a whole number is a variant of a floating point number. For example, an int can be imagined as a double, which has no actual floating point part. Therefore, if you assign an int value to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

- Exponential such as `ex`
- Logarithmic such as `log2(x)`
- Trigonometric such as `sin(x)` `cos(x)`



## 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); //  $e^a$ 
c = Math.max(a,b); // greater of two values
c = Math.min(a,b); // smaller of two values
c = Math.pow(a,b); //  $a^b$ 
c = Math.sqrt(a); // square root
c = Math.random(); // random number  $0.0 \geq c < 1.0$ 
```

## 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
```

O

For more examples of Math class functions, see the Java documentation at:

<https://docs.oracle.com/en/java/javase/21/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.

The 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 will be explained later in the course.

# Binary Number Representation

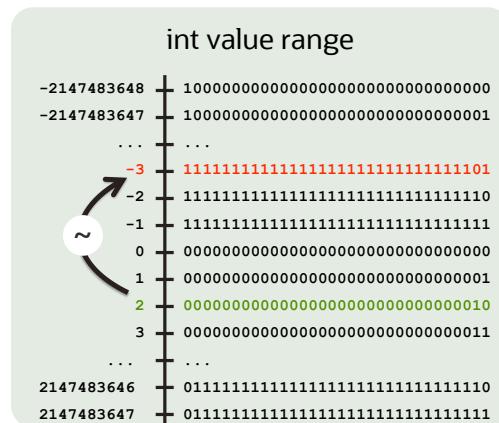
All Java numeric primitives are signed (*that is, could represent positive and negative values*).

- 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.

The **bitwise complement** operator inverts all bits of a number:  
The result of the bitwise complement 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.



O

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; they use high-level APIs to program. However, under the hood, these high-level APIs would use these kinds of operations anyway.

Bitwise logical operators *AND* `&`, *OR* `|`, *Exclusive OR* `^`, and more are covered in detail in Appendix F titled: Miscellaneous Java Topics.

# Bitwise Logical 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;           // 00000101
byte b = 3;           // 00000011
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** >>> is the 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;    // 1111111111111111111111111111111101100 (d is -20)
int e = a >> 2;    // 000000000000000000000000000000000001 (e is 1)
int f = b >> 2;    // 111111111111111111111111111111111110 (f is -2)
int g = a >>> 2;  // 000000000000000000000000000000000000000001 (e is 1)
int h = b >>> 2;  // 001111111111111111111111111111111110 (h is 1073741822)
```

O

- Bit manipulation operations, including logical and shift operations, perform low-level operations directly on the binary representations used in integers.
- These operations are not often used in enterprise-type systems but might be critical in graphical, scientific, or control systems.
- The ability to operate directly on binary might save large amounts of memory, might enable certain computations to be performed very efficiently, and can greatly simplify operations on collections of bits, such as data read from or written to parallel I/O ports.
- The Java programming language supports bitwise operations on integral data types. These are represented as the operators ~, &, ^, and | for the bitwise operations of NOT (bitwise complement), bitwise AND, bitwise XOR, and bitwise OR, respectively.

## Right-Shift Operators >> and >>>

- The operator >> performs an arithmetic or signed right shift. The result of this shift is that the first operand is divided by 2 raised to the number of times specified by the second operand.
- The logical or unsigned right shift operator >>> works on the bit pattern rather than the arithmetic meaning of a value and always places 0s in the most significant bits.
- The operator << performs a left shift. The result of this shift is that the first operand is multiplied by two raised to the number specified by the second operand.

# 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 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.

O

# Short-Circuit Evaluation

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

O

In a short-circuit `&&` evaluation, when the left side of an 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 side of the expression. This is precisely what is going to happen—Java would not attempt to evaluate the right 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 braces {} 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.

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

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



# 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 need to only assign a value based on a condition.
- When the Boolean expression yields true, the value after the ? is assigned. When the Boolean expression yields false, the 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, because it can make your code less readable.

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



O

The if/else example works fine, but it 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.

- The switch statement expression must be of one of the following types:  
**byte, short, int, char, String, enum, Object**
- Case **labels** must match the expression type.
- Execution flow proceeds to the case in which the label matches the switch statement expression value.
- Execution flow continues until it reaches the end of the switch or encounters an optional **break** statement.
- If the switch statement 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)

✿ Note: The use of switch with Strings, enums and Objects cases is covered later in the course.

```
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
```

O

The ability to use Objects in a switch cases is a new feature of the Java language, known as "Pattern matching for switch", which became production in Java SE 21. This capability is covered later in the course in Lesson 6 "Implement Inheritance and use Records".

## Switch -> No Fall-Through Syntax

- Cases labeled with the arrow symbol `case <label>->`
- Just like a `case <label>:` (Fall-Through syntax) may execute a **single line**, or a **block of code**.
- Matched case **does not fall through to the next case**, without a need to use a `break` statement.

```
char status = 'N';
double price = 10;
switch (status) {
    case 'S', 'N' -> price += 1;
    case 'D' -> {
        double discount = price*0.2;
        price -= discount;
    }
    case 'E' -> price = 0;
    default -> price = 3;
}
```

✿ Note: A `case` in both switch expressions and statements may list several comma-separated label values.

O

Cases are matched to any of the labels to the left of the arrow, and then execute the code to the right of the arrow. Switch expression do not fall through to the next case, they do not run any other code in the switch expression (or statement), except the case that was matched.

Unlike switch statements, the cases of switch expressions must be exhaustive, which means that for all possible values, there must be a matching switch label.

## Switch Expressions yield a Value

A switch expressions are used to evaluate a value.

- Contain exhaustive set of cases that yields a value.
- A single-line expression does not require explicit use of a `yield` statement.
- A block requires a `yield` statement to return a value.
- An overall statement must be terminated with a `;` symbol.

```
char status = 'N';
double cost = 10;
double price = switch (status) {
    case 'S', 'N' -> cost += 1;
    case 'D' -> {
        double discount = cost*0.2;
        cost -= discount;
        yield cost;
    }
    case 'E' -> 0;
    default -> 3;
};
```

### Notes:

- `yield` has special meaning only within switch expressions.
- If a variable called `yield` exists elsewhere, that code won't break.

O

The `yield` statement makes it easier for you to differentiate between switch statements and switch expressions. A switch statement, but not a switch expression, can use a `break` statement. Conversely, a switch expression, but not a switch statement, can use a `yield` statement. A switch expression must either complete normally with a value or complete abruptly by throwing an exception.

# Switch Statements and Expressions Summary

## Switch statement using : syntax

```
switch(...) {
    case <label> : {
        // May use break otherwise will
        // fall-through to the next case.
    }
    ...
}
```

## Switch expression using : syntax

```
x = switch(...) {
    case <label> : {
        // Must produce the value using yield or
        // fall-through to the next case, but eventually
        // must yield a value, or throw an exception.
        // Must not use break.
    }
    ...
};
```

## Switch statement using -> syntax

```
switch(...) {
    case <label> -> {
        // May use break.
        // No fall-through to the next case.
    }
    ...
}
```

## Switch expression using -> syntax

```
x = switch(...) {
    case <label> -> {
        // Must either produce result from a simple expression,
        // or return a value using yield, or throw an exception.
        // Must not use break.
        // No fall-through to the next case.
    }
    ...
};
```

O

Switch statement does not return a value.

Switch expression returns a value, or throws an exception.

Both could use : or -> syntax.

Switch expressions action can be a simple statement, a block of code, or it can throw an exception.

When switch that uses a : notation returns a value it is using **yield** not just to return a value, but also to prevent fall-through to the next case.

Compiler checks exhaustiveness for the switch expressions.

# Using JShell (REPL Tool)

JShell is an interactive Read-Evaluate-Print Loop (**REPL**) command-line tool.

- Launch it via the terminal by using the `jshell` command.
- 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 presents a **Loop** of the following actions:

- **R**ead the expression
- **E**valuate the result (creating a "**scratch**" variable if required)
- **P**rint the result

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

O

## JShell Commands

```
/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
```

O

## Code Snippets

The term **snippet** refers to the code you enter in a single JShell loop, such as:

- Declarations
- Expressions
- Statements

JShell code snippets are not allowed to:

- Declare packages
- Use the public, protected, private, static, final, break, continue, and return keywords in a top-level constructs

```
float price = 12.99f;
price ==> 12.99
float discount = 0.1f
discount ==> 0.1
float adjustPrice(float price,
                  float discount) {
    return price - price * discount;
}
created method adjustPrice(float, float)
import java.nio.file.*
class Product { }
created class Product
Math.pow(2, 7)
$4 ==> 128.0
new Product()
$5 ==> Product@12edcd21
if (discount < 1) {
    adjustPrice(price, discount);
}
```

 **Note:** The ; statement terminator is optional only in simple top-level snippets.

O

A snippet can be one line. A snippet can be many lines including a loop, a method, a class and so on.... This slide shows examples of declarations, expressions, and statements.

What's not allowed is syntax that would seem illogical.

A package statement doesn't make sense because your goal in JShell is to test code, not to develop libraries.

The break keyword wouldn't make sense unless there was a loop to break.

## Summary

In this lesson, you should have learned how to:

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



## Practices for Lesson 2: Overview

In this practice, you will:

- Use the JShell tool
- Declare, initialize, and perform operations on primitives
- Use the `if/else` and `switch` constructs and use a ternary operator



## Text, Date, Time, and Numeric Objects

---

# Objectives

After completing this lesson, you should be able to:

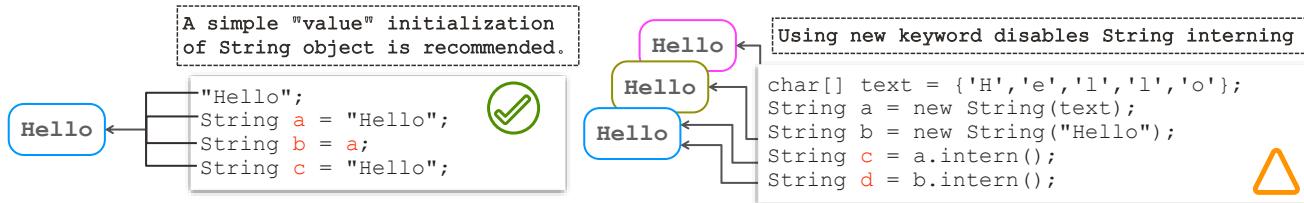
- Manipulate text values by using String, Text Blocks, and StringBuilder classes
- Describe primitive wrapper classes
- Perform string and primitive conversions
- Handle decimal numbers by using the BigDecimal class
- Handle date and time values
- Describe localization and formatting classes



# String Initialization



- The `java.lang.String` class represents a sequence of characters.
- String objects can be instantiated, with the `new` keyword, or with simplified instantiation as a text value enclosed with double quotation marks: "some text".
- JVM can automatically `intern` 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**.
- The `intern()` method returns a reference to an interned (single) copy of a string literal.



## Notes:

- A primitive `char` represents a single character. Its values are enclosed in single quotation marks: 'a'.
- The example uses a `char` array `char[]`. – The use of arrays is covered later in the course.

O

String is a class, not a primitive like `char`. Instances of `String` represent sequences of `char` values.

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 quotation marks: "some text" and that is a recommended approach.

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 in Java 9 onwards `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. After 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.



✿ **Note:** The `+` is both a string **concatenation** as well as an **arithmetic** operator.

O

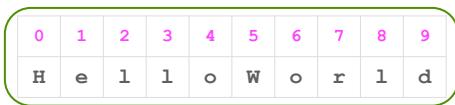
There are three more methods that allow removing leading and trailing spaces from the string: `strip()`, `stripLeading()` and `stripTrailing()` (available since version 11), in addition to the `trim()` method.

For more information about String class and its methods, see Java documentation:  
<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html>

# String Indexing

String contains a sequence character indexed by integer.

- The string index starts from 0.
- When getting a substring of a string, the **begin index** is inclusive of the result, but the **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 starting index.



## Notes:

- ❖ An attempt to access text beyond the last valid index position `length-1` will produce an exception.
- ❖ Exception handling is covered later in the course.

```
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); // throws StringIndexOutOfBoundsException
```

O

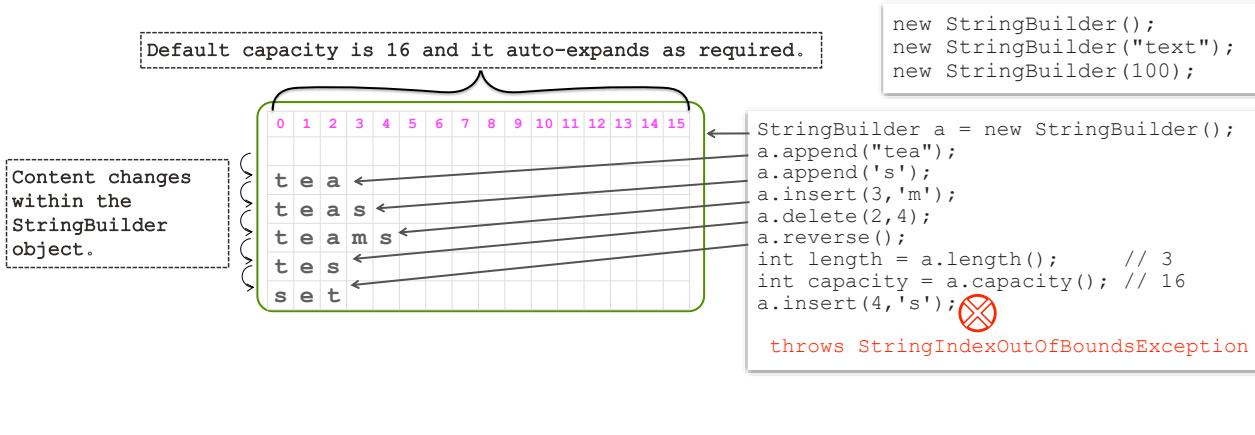
The purpose of a `substring` method is to return a portion of a string between a given starting and ending position.

The purpose of an `indexOf` method is to return an index that indicates a starting position of a substring within a given string.

# Mutable Text Objects

`java.lang.StringBuilder` objects are mutable, allowing modifications of the text they represent.

- Some methods such as `substring`, `indexOf`, `charAt` are identical to that of a `String` class.
- Extra methods are available: `append`, `insert`, `delete`, `reverse`.
- The sequence of characters must be continuous. It may contain spaces, but you may not leave gaps of no characters at all.
- Are instantiated using the `new` keyword and be initialized with a predefined content or capacity.



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. Handling text modifications with `StringBuilder` reduces the number of string objects you need to create. `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. `StringBuilder` operations such as `append`, `insert`, `delete` accept string or char parameters.

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

<https://docs.oracle.com/en/java/javase/21/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 detrimental effect on a 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.

## Text Blocks

A text block is a preformatted multiline string literal:

- Begins with """ three double-quote characters followed by a line terminator
- Does not require an explicit escape sequences such as \n or \t
- Does not require you to escape the \" and \' embedded double quotation marks
- Is otherwise treated like any other string object



```
String basicText = "Tea\tprice 1.99 quantity 2\n\"English Breakfast\"\n";  
  
String textBlock = """  
    Tea      price 1.99 quantity 2  
    "English Breakfast"  
""";  
  
// All String operations apply as usual:  
int p1 = basicText.indexOf("price");  
int p2 = textBlock.indexOf("price");
```

O

Text blocks simplify the task of writing Java programs by making it easy to express strings that span several lines of source code, while avoiding escape sequences in common cases.

Text blocks enhance the readability of strings in Java programs that denote code written in non-Java languages.

A text block is an alternative form of Java string representation that can be used anywhere a traditional double quoted string literal can be used. The object produced from a text block is a `java.lang.String` with the same characteristics as a traditional double quoted string, and all `String` methods are applicable to a text blocks as usual.

Just like any other `String` objects text blocks may be used as a method argument.

Text block initialization returns a string with incidental white spaces removed from the beginning and end of every line in the original text.

**Example 1:**

```
String textBlock1 = """
    Tea "English Breakfast"
        price 1.99
        quantity 2
""";
```

When printed will result in the following text, where first two spaces are considered incidental and thus removed):

0123456789012345678901234 (character index for reference)

```
Tea "English Breakfast"
    price 1.99
    quantity 2
```

**Example 2** (notice that closing "" now set the outmost left-hand side part of this text, thus no spaces on a left are considered incidental):

```
String textBlock2 = """
    Tea "English Breakfast"
        price 1.99
        quantity 2
""";
```

When printed will result in the following text:

0123456789012345678901234 (character index for reference)

```
Tea "English Breakfast"
    price 1.99
    quantity 2
```

A text block requires a line terminator after the starting thee double-quote characters and must be closed with thee double-quote characters.

The \ character must still be escaped, just like in any other string.

Because of this, the following examples will result in a compilation error:

**Missing line terminator after opening delimiter:**

```
String textBlock = """Tea price 1.99
                        quantity 2
                        "English Breakfast"
"""
String a = """"";
String b = """ """;
```

**Missing closing delimiter (text block continues to EOF):**

```
String c = """
    ";
";
```

**Unescaped backslash character:**

```
String d = """
    price \ quantity
""";
```

## Spaces, Lines, and Quotes

Text blocks retain left-side spaces, but by default remove line-end spaces.

Extra escape sequences exist for text blocks:

- `\s` inserts a space (for example for padding spaces before the end of the line).
- `\<line terminator>` prevents automatic line break.
- `\"""` represents three quotation marks.

```
String textBlock = """"
    This game is a square:
X|0|0
|0|\s
X|X|0
    Text on the\ same line
    These are \""" three quotes\ """
""";
```



```
This game is a square:
X|0|0
|0|\s
X|X|0
    Text on the same line
    These are """ three quotes
```

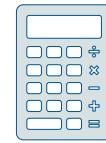
❖ **Note:** Avoid overusing escape sequences to improve readability.

0

In fact, it's only the first `\"` escape sequence, that is actually making the three sequential quote symbols possible within the text block. After the first quote symbol is escaped, the other two quotes are not treated as a text block delimiter.

It is typical to add a `\` symbol to the end of the block to prevent the line terminator at the end in case a closing delimiter is on the next line.

# 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 a wrapper object out of primitive or string by using the `valueOf()` methods.
- Extract primitive values out of the wrapper by 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 a wrapper or primitive out of the string by using the `parseXXX()` methods.
- Convert a primitive to a string by using the `String.valueOf()` method.
- Wrapper classes provide constants, such as **min** and **max values** for every type.

## Notes:

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

```
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 = Float.parseFloat(d);
String g = String.valueOf(a);
Short.MIN_VALUE;
Short.MAX_VALUE;
```

## Primitive Wrapper

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

O

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.

The 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 "Improved 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.
- However, unlike other numeric wrapper classes, `BigDecimal` has arbitrary precision.
- 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 numbers 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
```



O

Just like all other primitives and wrapper objects, `BigDecimal` object are immutable (cannot be modified) and signed (can be positive or negative). `BigDecimal` objects may represent numeric values with arbitrary precision, for example a `Double` object has a specific precision as a binary 64 bit number, but a `BigDecimal` can be set to any precision you need.

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, because 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 the next operation on this object immediately.
- It is possible to use the **chain method invocations** technique with any operation that returns an object.

## Examples:

- ❖ Text manipulating operations of String return String objects.
- ❖ Arithmetic operations of BigDecimal return BigDecimal 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
```

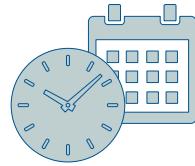
## Notes:

- ❖ Method chaining is a common coding technique used when an intermediate object returned by a method is not required, so code immediately invokes of next method 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);
```

O

# Local Date and Time API



Handle date and time values with the API provided with the `java.time` package.

- Implemented with `LocalDate`, `LocalTime`, and `LocalDateTime` classes.
- Date and time objects can be created using the following 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
  - Combine other date and time objects by using `atTime()` and `of(localDate, localTime)` or
  - Extract date and time portions from `LocalDateTime` by 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();
```

## Notes:

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

O

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.

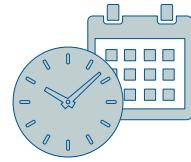
The example on this page uses an enum `java.time.Month`, which represents 12 months. Another useful enum is `java.time.DayOfWeek`, which represents seven days of the week. Enums are covered later in the course.

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

O

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.LocalDate`, `java.time.LocalDateTime`, and `java.time.LocalTime` classes.

**Methods to create new date and time by changing specific portions 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:**

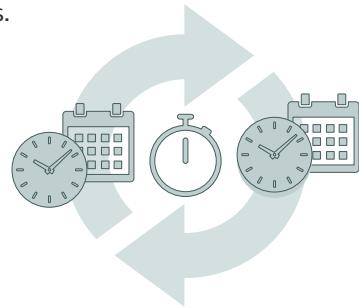
```
getYear()
getDayOfYear()
getMonth()
getDayOfMonth()
getDayOfWeek()
getHour()
getMinute()
getSecond()
getNano()
```

# Instants, Durations, and Periods

Handle periods and durations of time with the following classes from the `java.time` package:

- `Duration`: Represents an amount of time in nanoseconds
- `Period`: Represents an amount of time in units such as years or days
- `Instant`: Represents an instantaneous point on the timeline (time stamp)
- Produce with `now()`, `ofXXX()`, `plusXXX()`, `minusXXX()`, `withXXX()` methods.
- Specific time values can be retrieved with `getXXX()` methods.
- Measure distance between points in time with `between()`, `isNegative()` methods.
- 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();
```



## Notes:

- ❖ Instant and duration are more suitable for implementing system tasks, such as using a time stamp when writing logs.
- ❖ Period is more suitable for implementing business logic.
- ❖ Just like the rest of the local date and time API, duration, period, and instant objects are immutable.

O 1  
6

## `toInstant()` method:

`FileTime` represents the value of a file's timestamp attribute. For example, it may represent the time that the file was last modified, accessed, or created.

`toInstant()` converts this `FileTime` object to an instant.

The conversion creates an instant that represents the same point on the timeline as this `FileTime`.

Example from Oracle/Helidon:

```
private Instant getLastModified(String path) throws IOException {
    Path file = Paths.get(path);
    if (Files.exists(file) && Files.isRegularFile(file)) {
        return Files.getLastModifiedTime(file).toInstant();
    } else {
        return null;
    }
}
```

## `ofInstant()` method:

`public static LocalTime ofInstant(Instant instant, ZoneId zone)`

Obtains an instance of `LocalTime` from an instant and zone ID

`public static LocalDate ofInstant(Instant instant, ZoneId zone)`

Obtains an instance of `LocalDate` from an instant and zone ID

`public static LocalDateTime ofInstant(Instant instant, ZoneId zone)`

Obtains an instance of `LocalDateTime` from an instant and zone ID

```
public static ZonedDateTime ofInstant(Instant instant, ZoneId zone)
```

Obtains an instance of `ZonedDateTime` from an instant

### Parameters

`instant`: The instant to create the date from, not null

`zone`: The time-zone, which may be an offset, not null

First, the offset from UTC/Greenwich is obtained using the zone ID and instant, which is simple as there is only one valid offset for each instant. Then, the instant and offset are used to calculate the local time, or local date or local date-time.

### Example

```
LocalDateTime mydatetime = LocalDateTime.ofInstant(Instant.now(),  
zoneOffset.UTC);
```

```
System.out.println("Date is : " + mydatetime);
```

# Zoned Date and Time

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

`java.time.ZonedDateTime` class:

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

```
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:

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



# Representing Languages and Countries

Make your application adjustable to different languages and locations around the world.

- `java.util.Locale` class represents languages and countries.
- 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.

Language	Country	Variant
<code>Locale uk = Locale.of("en", "GB");</code>	<code>// English</code>	<code>Britain</code>
<code>Locale th = Locale.of("th", "TH", "TH");</code>	<code>// Thai</code>	<code>Thailand (Thai digits variant)</code>
<code>Locale us = Locale.of("en", "US");</code>	<code>// English</code>	<code>America</code>
<code>Locale fr = Locale.of("fr", "FR");</code>	<code>// French</code>	<code>France</code>
<code>Locale cf = Locale.of("fr", "CA");</code>	<code>// French</code>	<code>Canada</code>
<code>Locale fr = Locale.of("fr", "029");</code>	<code>// French</code>	<code>Caribbean</code>
<code>Locale es = Locale.of("fr");</code>	<code>// French</code>	
<code>Locale current = Locale.getDefault();</code>	<code>// current default locale</code>	
<code>// Example constructing locale that uses Thai digits and Buddhist calendar:</code>		
<code>Locale th = Locale.forLanguageTag("th-TH-u-ca-buddhist-nu-thai");</code>		



O

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

```
Locale us = Locale.US;
```

is the same as:

```
Locale us = Locale.of("en", "US");
```

Historically, locales were constructed using variants of the `new Locale()` constructor. However, since Java SE 19 all variants of `Locale` constructors are deprecated and replaced with the equivalent variants of the `Locale.of()` method.

This is a pre Java SE 19 equivalent:

```
Locale us = Locale.of("en", "US");
```

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)

Instances of the locale can also be constructed with a variant argument (custom variant), for example to use different locale-specific system of digits:

```
Locale th1 = Locale.of("th", "TH");
Locale th2 = Locale.of("th", "TH", "TH");
NumberFormat.getCurrencyInstance(th1).format(12345.67); // produces ₩12,345.67
NumberFormat.getCurrencyInstance(th2).format(12345.67); // produces
฿๑๒,๓๔๕.๖๗
```

You may change the default locale for this instance of the Java Virtual Machine.

```
Locale.setDefault(Locale.FRANCE);
```

Or just for a specific category (display or format) for this instance of the Java Virtual Machine:

```
Locale.setDefault(Locale.Category.FORMAT, Locale.US);
```

Prior to Java SE 17 before the default locale could be changed a security manager permission verification using `checkPermission` method with a `PropertyPermission("user.language", "write")` was required. However, this is no longer the case, since Java SE 17 the Security Manager class is deprecated and subject to removal in a future release.

For more information, see Java Locale documentation:

<https://docs.oracle.com/en/java/javase/21/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/java/technologies/javase/jdk21-supported-locales.html>

# Formatting and Parsing 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 = Locale.of("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*

£2.99 20% 12,345

*formatted results*

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

```
BigDecimal p = BigDecimal.valueOf((Double)currencyFormat.parse("£1.7"));
Double t = (Double)percentageFormat.parse("12%");
int q = numberFormat.parse("54,321").intValue();
```

*parsing values*

1.75 0.12 54321

*parsed values*

O

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 a 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.

`CompactNumberFormat` is a concrete subclass of `NumberFormat` that formats a decimal number in its compact form. The compact number formatting is designed for the environment where space is limited, and the formatted string can be displayed in that limited space. It is defined by LDML's specification for Compact Number Formats. A compact number formatting refers to the representation of a number in a shorter form, based on the patterns provided for a given locale.

Example:

In the US locale, 1000 can be formatted as "1K", and 1000000 as "1M", depending upon the style used.

In the "hi\_IN" locale, 1000 can be formatted as "1 हजार", and 5000000 as "5 क.", depending upon the style used.

To obtain a `CompactNumberFormat` for a locale, use one of the factory methods given by `NumberFormat` for compact number formatting. For example, `NumberFormat.getCompactNumberInstance(Locale, Style)`.

```
NumberFormat fmt = NumberFormat.getCompactNumberInstance (Locale.of("hi", "IN"),  
NumberFormat.Style.SHORT);  
  
String result = fmt.format(1000);
```

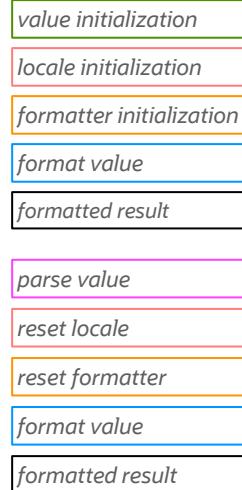
A number can be formatted in compact forms with two different styles, `SHORT` and `LONG`. Use `NumberFormat.getCompactNumberInstance(Locale, Style)` for formatting and parsing a number in `SHORT` or `LONG` compact form, where the given style parameter requests the desired format. A `SHORT` style compact number instance in the US locale formats 10000 as "10K". However, a `LONG` style instance in same locale formats 10000 as "10 thousand".

## Formatting and Parsing Date and Time Values

- `java.time.format.DateTimeFormatter` parses and formats date and time values.
- `java.time.format.FormatStyle` enum defines standard format patterns.
- Alternatively, set up custom format patterns.

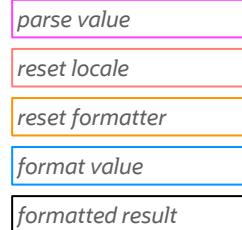
```
LocalDate date = LocalDate.of(2019, Month.APRIL, 1);
Locale locale = Locale.of("en", "GB");
DateTimeFormatter dateTimeFormat =
    DateTimeFormatter.ofPattern("EEEE dd MMM yyyy", locale);
String result = date.format(dateTimeFormat);
// dateTimeFormat.format(date);
```

Monday 01 Apr 2019



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

31 Mar. 2020 r.



O

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	3; 03; Q3; 3rd quarter

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", Locale.of("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

The `java.util.ResourceBundle` class represents bundles of localizable resources.

- Resources may be stored as classes or plain text files with the `.properties` extension.
- Resources are placed into resource bundles as `<key>=<value>` pairs.
- They may represent user-visible messages or message patterns with **substitution parameters**.
- A **default bundle** can be used if no locale is specified, or if the resource you're trying to get is not present in a **language and country-specific bundle**.

```
Locale locale = Locale.of("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=こんにちは {0}  
 product={0}, 価格 {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}

❖ Note: Value substitutions are explained later.

O

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 a package prefix to load the bundle file.

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

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

```
Locale locale = Locale.of("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.

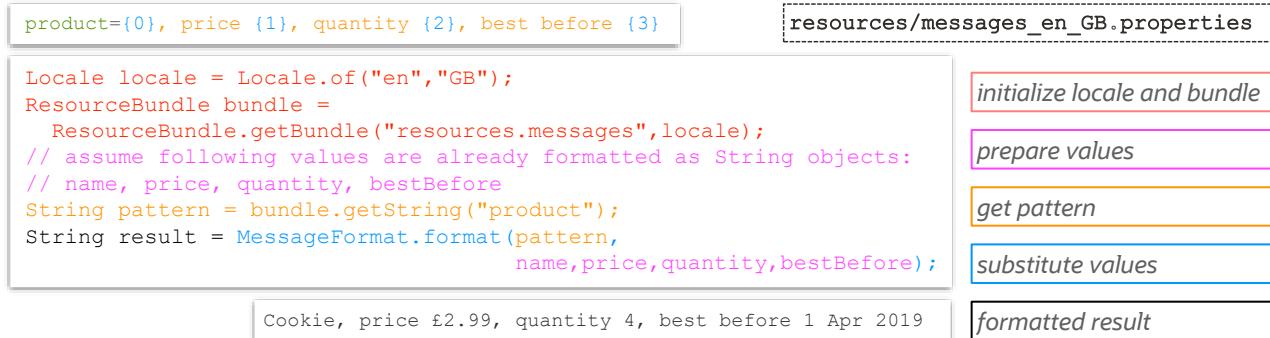
A root resource file, `MessageBundle.properties`, is the default file for the application. For alternative languages, a special naming convention of a default locale bundle is used.

Its naming format is as follows: `MessageBundle_xx_YY.properties`, where `xx` is the language code and `YY` is the country code.

# Formatting Message Patterns

`java.text.MessageFormat` class is used to:

- Format messages by substituting values into text patterns
  - Parse messages by extracting values out of the text based on a pattern
- Message patterns are typically stored in resource bundles.



✿ **Note:** See notes for more examples and use cases.

O

There are two ways of formatting messages with the `MessageFormat` class.

- Use static format method as shown in the example on the screen. This way, a pattern and corresponding values are supplied as method arguments every time there is a need to perform formatting.
- Produce a `MessageFormat` object with a predefined pattern, and then using this object to format same pattern, but with different values as many times as required.

```

MessageFormat formatter = new MessageFormat(pattern, locale);
Object[] values = {name, price, quantity, bestBefore};
String result = formatter.format(values);
  
```

Notice that in this example, values are supplied in the form of an array. Arrays are covered in another lesson of this course.

The example on this screen assumes that all substituted values are supplied in the form of preformatted string objects. However, this does not have to be the case, because format method of the `MessageFormat` can also perform date and number formatting.

```

String pattern = "{0} price {1,number,currency} "+
    "quantity {2,number,integer} best before {3,date}";
String name = "Tea";
int quantity = 2;
BigDecimal price = BigDecimal.valueOf(2.99);
Date bestBefore = new Date();
MessageFormat formatter = new MessageFormat(pattern, locale);
String result = MessageFormat.format(pattern, name, quantity, price, bestBefore);
  
```

Sometimes, a program may need to format a different message depending on actual supplied values. For example, a different message should be produced based on a product quantity. This could be achieved with the help of the `ChoiceFormat` class. In the following code example, we assume that we need to produce a "no products" message if quantity is 0 or less, "a product" message if quantity is between 0 and 2 and a quantity value followed by the word "products" if quantity is 2 or greater.

```
// describe ranges of values as array elements:  
double[] quantities = {0,1,2};  
  
// describe an array of text patterns in the same order ranges of values:  
String[] patterns = {"no products", "a product", "{0} products"};  
  
// describe a ChoiceFormat object associated with ranges and patterns:  
ChoiceFormat choiceFormat = new ChoiceFormat(quantities, patterns);  
  
// describe a MessageFormat with required message pattern:  
MessageFormat formatter = new MessageFormat("Ordered {0, choice}");  
  
// associate element at index 0 within the MessageFormat pattern with a ChoiceFormat object:  
formatter.setFormat(0,choiceFormat);  
  
// supply values that should be formatted, this could be a quantity of 0, 1, or more:  
Object[] values = {0};  
  
// perform formatting:  
String result = formatter.format(values);
```

The result is:

```
"Ordered no products" when value is 0 or less.  
"Ordered a product" when value is between 0 and 2.  
"Ordered 2 products" when values is 2 or more.
```

In addition to formatting, `MessageFormat` and `ChoiceFormat` classes also provides parsing 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: Summary

```
resources/messages_en_GB.properties
```

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

```
Locale locale = Locale.of("en", "GB");
ResourceBundle bundle = ResourceBundle.getBundle("resources.messages", locale);
String pattern = bundle.getString("product");

String name = "Cookie";
BigDecimal price = BigDecimal.valueOf(2.99);
int quantity = 4;
LocalDate bestBefore = LocalDate.of(2019, Month.APRIL, 1);

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 result = MessageFormat.format(pattern, name, fPrice, fQuantity, fBestBefore);
```

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

**✖ Note:** See notes for an alternative example.

29



Format objects can be placed into the array in the order in which they should be applied to the elements of the message pattern. For example, assume that a pattern contains the following text:

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

In this case, we expect to place a string representing product name at a position marked as {0}, a BigDecimal value for a price at the position marked {1}, an int value for a quantity at the position marked {2} and a LocalDate value at the position marked {3}. This means that we need to supply format objects in the following order: no formatter, a CurrencyFormat, a NumberFormat and a DateFormat. This is the order in which format objects must be placed into the array. Likewise, all values that should be substituted into the pattern can also be supplied as an array, with the exact same order of elements.

```
// define Locale
Locale locale = Locale.of("en", "GB");
// load resource bundle
ResourceBundle bundle = ResourceBundle.getBundle("resources.messages", locale);
// retrieve pattern from the bundle
String pattern = bundle.getString("product");
// prepare values
String name = "Cookie";
BigDecimal price = BigDecimal.valueOf(2.99);
int quantity = 4;
LocalDate bestBefore = LocalDate.of(2019, Month.APRIL, 1);
// place values into the array
Object[] values = {name, price, Integer.valueOf(quantity), bestBefore};
// example continues on the next page
```

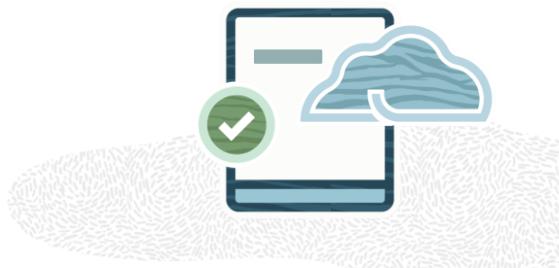
```
// prepare format objects
NumberFormat currencyFormat =
    NumberFormat.getCurrencyInstance(locale);
NumberFormat numberFormat =
    NumberFormat.getNumberInstance(locale);
DateTimeFormatter dateFormat =
    DateTimeFormatter.ofPattern("dd MMM yyyy", locale);
// place format objects into an array
Format[] formats = {null, currencyFormat, numberFormat, dateFormat.toFormat()};
// create a message format object associated with a pattern and locale
MessageFormat messageFormat = new MessageFormat(pattern, locale);
// apply format objects to the message format
messageFormat.setFormats(formats);
// prepare a StringBuffer object to store formatting result
StringBuffer result = new StringBuffer();
// format message
messageFormat.format(values, result,
    new FieldPosition(MessageFormat.Field.ARGUMENT));
// use the result
System.out.println(result);
```

MessageFormat expects all format object to be subtypes of the format class. However, DateTimeFormatter is designed differently and it is not a subtype of the format class. For this reason, DateTimeFormatter provides a method `toFormat()` that returns a DateFormat object that is compatible with the type that MessageFormat expects.

## Summary

In this lesson, you should have learned how to:

- Manipulate text values by using String, Text Blocks, and StringBuilder classes
- Describe primitive wrapper classes
- Perform string and primitive conversions
- Handle decimal numbers by using the BigDecimal class
- Handle date and time values
- Describe localization and formatting classes



## Practices for Lesson 3: Overview

In this practice, you will:

- Manipulate text values by using String and StringBuilder classes
- Manipulate text blocks
- Manipulate and format numeric, date, and time values
- Apply localization and format messages



# Classes and Objects

---

Java classes define objects. An object is an instance of a class. Every object has state and behavior.

# Objectives

After completing this lesson, you should be able to:

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

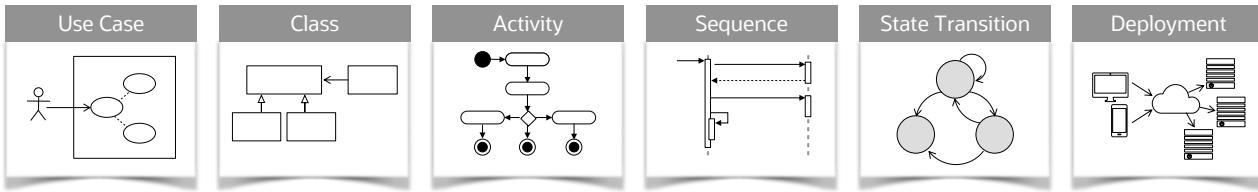


# UML: Introduction

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

O

Snippets of diagram types shown in the slide:

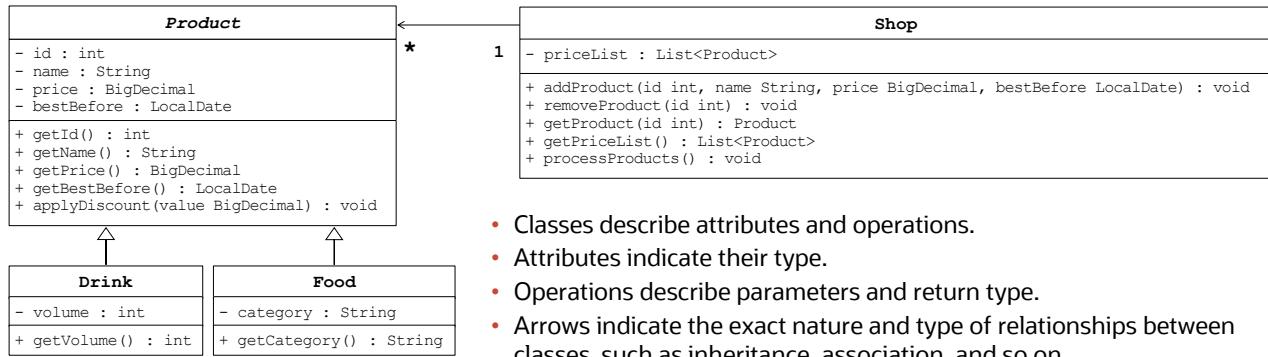
- Use Case diagrams capture high-level business requirements.
- Class diagrams represent classes and their relationships.
- Activity diagrams describe program logic.
- Sequence diagrams describe interactions between objects.
- State Transition diagrams describe the life cycle of an object.
- Deployment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

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

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 a product management system for a shop (see detailed description in notes).

❖ Also see notes for naming convention details.

O

The example in the slide represents a set of Java classes **Shop**, **Product**, **Food**, and **Drink**. The **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. The model can look ambiguous when a name can be misinterpreted as a verb. In this case, "Shop" is considered as a 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 the 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.

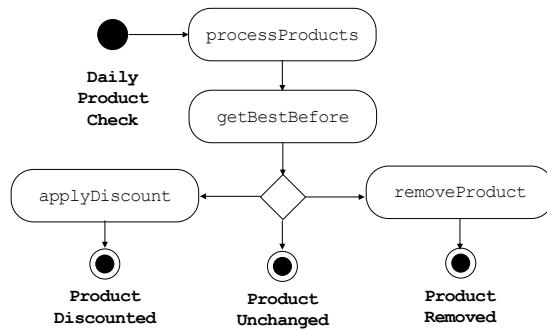
- It is typical for methods that provide access to information to follow the `getXXX`/`setXXX` naming pattern. If a method provides access to Boolean variable, it is usually named using `isXXX` pattern:

`getBestBefore`, `setTotalDiscount`, `isFresh`, and so on

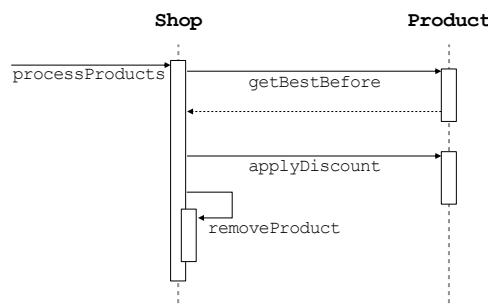
Because of the way in which initial letters of words are capitalized, these naming conventions are sometimes referred to as "camel-case."

# Modeling Interactions and Activities

An Activity diagram represents a flow of operations.



A Sequence diagram represents interactions between program objects.



- 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.

- 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.

O

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 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);
    }
}
```

O

# 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 a variable or methods of the object by using the `".` operator.

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

O

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 to never inform a programmer about the exact memory address where an object is physically stored. Thus, Java reference should be considered as a purely logical way of pointing to an object.

# Defining 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);
}
```

O

# Defining 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 modifier** determines 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>(<Type> <name>, <Type> <name>) {
        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.

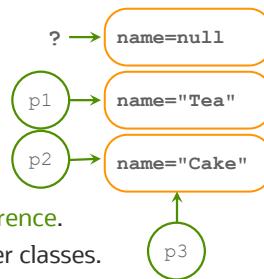
```
package demos.shop;
public class Product {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```

O

# 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.
- You can assign the same object reference to more than one variable.
- Invoke operations or access variables of the object by using the `.` operator, via object reference.
- Access modifiers may restrict the 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
```

O

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 the next lesson.

The type of a reference should match a class that has been instantiated, or any of its parents. You can find more details about the reference type option in a later lesson of this course that covers inheritance and polymorphism.

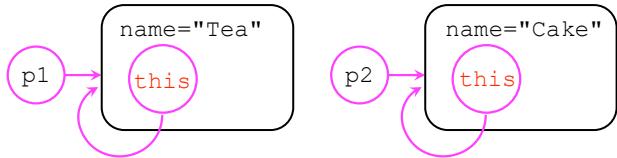
Multiple variables that reference the same object do not duplicate the object.

# Local Variables and Recursive Object Reference

**Local variables** are declared inside methods.

- They cannot have an access modifier. They are not visible outside of the method anyway.
- Method **parameters** are essentially local variables.
- A **local variable** can "shadow" an **instance variable** if their names coincide.
- Use the **this** keyword (recursive reference to current object) to refer to an instance, rather than a local variable.
- Variables defined in inner code blocks (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;
    }
}
```

O

It's best to avoid local and instance variable name clashes by applying consistent naming conventions.

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

In the example, 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, the 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 a 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
}
```

**Note:** Polymorphism and loops are covered later in this course.

O

Local variable type inference was introduced in Java 10. 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.

An implicit typed lambda expression must use `var` for all its formal parameters or for none of them.

`var` is permitted only for the formal parameters of implicitly typed lambda expressions.

The following examples are illegal and illustrate cases when `var` **cannot** be used:

Declarations without an initial value, or an initialization with a null value:

`var price;`

`var tax = null;`

Compound declarations:

`var price = 9.95, tax = 0.05;`

Array Initializers:

`var prices = {9.95, 5, 3.50};`

Fields:

`public var price;`

Parameters:

`public void setPrice(var price) {...}`

Method return type:

```
public var getPrice() {
    return price;
}
```

Cannot mix 'var' and 'no var' in implicitly typed lambda expression:

`(var x, y) -> x.process(y);`

Cannot mix 'var' and manifest types in explicitly typed lambda expression:

`(var x, int y) -> x.process(y);`

It is legal, but dangerous to use `var` with unqualified Generics (diamond operator), because the inferred type will be defaulted to the object:

```
var listA = new ArrayList<>();
// DANGEROUS: infers as ArrayList<Object>
var listB = List.of();
// DANGEROUS: infers as List<Object>
```

For more information about why this is considered dangerous refer to the Appendix D "Advanced Generics"

# Defining 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 constants** must be either initialized immediately or via all constructors.
- Local variables and parameters** can also be marked as final.
- An attempt to reassign a constant will result in a 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.

O

A constant 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 a 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.

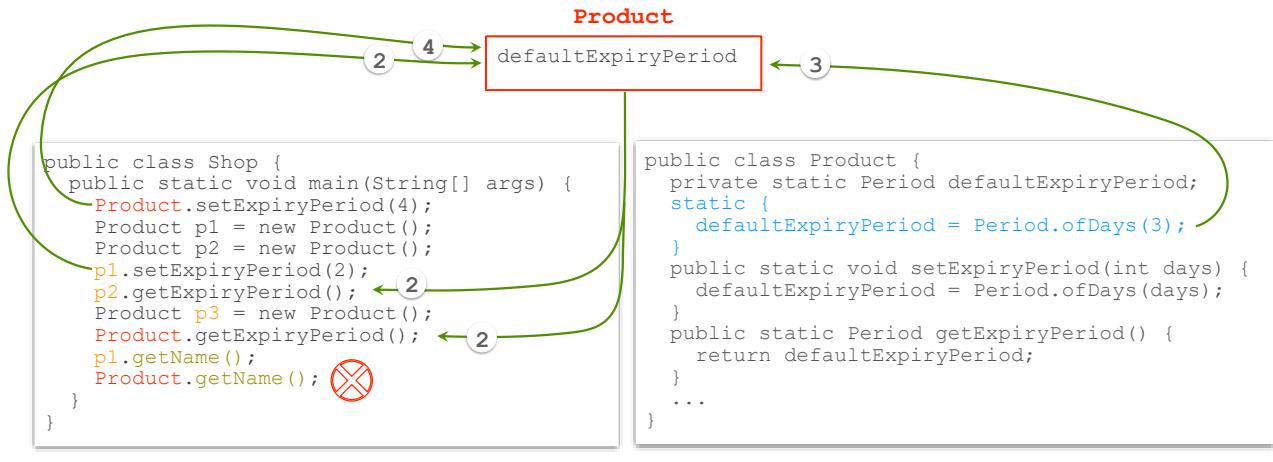
Although it is not possible to access the current instance context from static context, it is possible to access a specific instance:

```
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.



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.setExpiryPeriod(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 the 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);  
        ...  
    }  
}
```

O

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. 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

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

- The `main` method is static.
- All operations of the `Math` class are static.
- Factory methods are static methods that create and return a new instance.
- The `System.out` is a static variable, which refers to the `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.

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

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

Complete code example illustrating 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");
    }
}
```

O

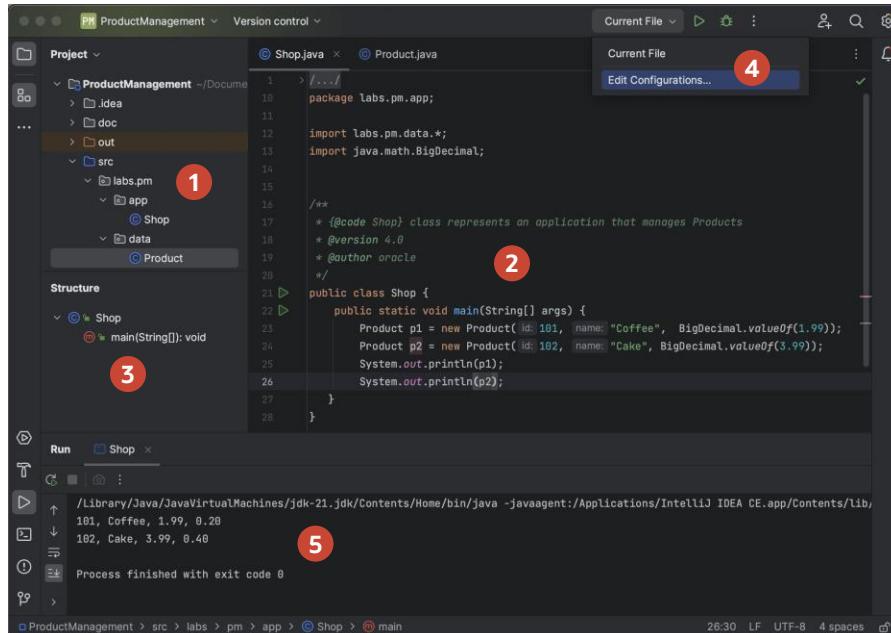
# IntelliJ IDE: Introduction

## IDE Windows:

1. Project
2. Editor
3. Structure
4. Edit Configurations
5. Output

## Toolbar Actions:

- Build Project
- Run
- Debug
- Stop



O

The Projects window is the main entry point to your project sources. It shows a logical view of important project content.

The Editor in IntelliJ is much more than a text editor. The IntelliJ 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 by using classes
- Define instance variables and methods
- Describe the `this` object reference
- Explain object instantiation
- Explain local variables and local variable type inference
- Define static variables and methods
- Invoke methods and access variables
- Describe IntelliJ IDE



## Practices for Lesson 4: Overview

In this practice, you will:

- Use IntelliJ to create a Product Management application project
- Create classes Product and Shop
- Document your code
- Compile and execute your application





## Improved Class Design

---

Improved class design is a key component of Oracle's commitment to quality and reliability. By following best practices and using modern design patterns, we can ensure that our code is maintainable, efficient, and easy to understand. This section will cover some of the key concepts and techniques used in improved class design.

# 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



# Overload Methods

Define more than one version of a method within a given class:

- Two or more methods, within the same class, that have **identical names**
- They must have a **different number**, or **different types**, or different order of parameters.
- Invoker does not need to learn different method names, just set different parameters.

```
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;
    }
}
```

Methods cannot be overloaded:

- Using different parameter names
- Using different return types

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

O

A method signature comprises the method's name and its parameter types.

The following example will not compile, because two versions of `aMethod` have identical signatures and only differ by return type, which is not allowed.

```
public class Some {
    public int aMethod(int x) {
        return 0;
    }
    public String aMethod(int y) {
        return "";
    }
}
```

The following example will compile, because two version of `aMethod` have different signatures. The fact that they have different return types is irrelevant from the method overloading perspective.

```
public class Some {
    public int aMethod(int x) {
        return 0;
    }
    public String aMethod(double x) {
        return "";
    }
}
```

Note that parameter names are irrelevant from the method overloading perspective.

# 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—the 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);
```

**Note:** Arrays are covered later in the course.

```
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]);
        }
    }
}
```

O

## 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);
```

Arrays are overed in the lesson titled "Arrays and Loops."

# Defining Constructors

A constructor is a special method that initializes the object:

- It is invoked using the `new` operator.
- It must be **named after the class**, must not have a return type, or be defined as void.
- Usually public, **it 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 the constructor, if you want to be able to create objects 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(); 
```

O

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, whereas the instance initializer is executed every time this class is instantiated.

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

# Reusing 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();` ✗

O

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 the 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 the `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
<b>public</b>	✓	✓	✓	✓
<b>protected</b>	✗	✓	✓	✓
<b>&lt;default&gt;</b>	✗	✗	✓	✓
<b>private</b>	✗	✗	✗	✓

## 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.

O

# Defining Encapsulation

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

- Instance variables are typically `private`, 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 modifiers: `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);
    }
}
```

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

```
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();
    }
}
```

O

Hiding information and behaviors within the class is done to be able to provide a more suitable interface in the form of methods that are visible to outside callers. This approach allows an operation to validate, calculate, and prepare values and prevents external observers from directly accessing 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.

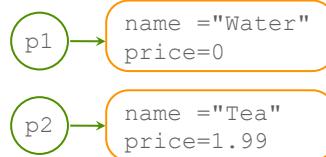
# Defining Immutability

Immutable objects present read-only data that is not modifiable 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 the 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);
    }
}
```



O

The following rules define a simple strategy for creating immutable objects:

1. Don't provide "setter" methods—i.e., methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify the mutable objects.
  - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies.

Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

# 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:** An instance initializer is a block of code that is triggered before the invocation of the constructor.

O

**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 switch constructs

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

```
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());
    }
}
```

```
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!");
        }
        return this;
    }
}
```

O

In the example on the screen, 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
```

You could use the switch -> syntax:

```
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!");
            case Condition.WARM -> this.addCaution("Just right");
            case Condition.HOT -> this.addCaution("Warning HOT!");
        }
        return this;
    }
}
```

# 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.
- ❖ After **caution** becomes a property of the **enum**, the switch from the previous example becomes redundant.

O

Java programming language **enum** types are much more powerful than their counterparts in other languages. The **enum** declaration defines a class (called an **enum** type). The **enum** class body can include methods and other fields.

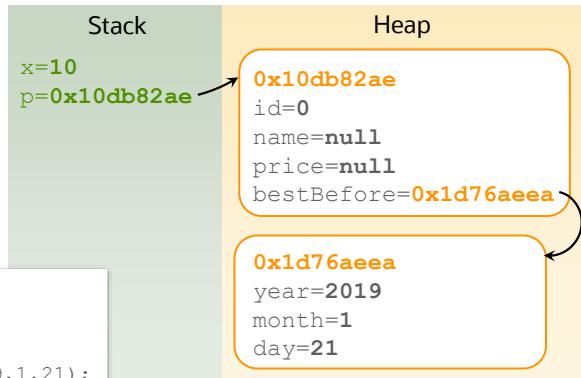
```
public enum Condition {
    HOT("Warning HOT!"),
    WARM("Just right"),
    COLD("Warning COLD!");
    private String caution;
    private Condition(String caution) {
        this.caution = caution;
    }
    public String getCaution() {
        return caution;
    }
}
public class Product {
  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

# 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.

O

You may initialize an 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 “Implement Inheritance and Use Records.”

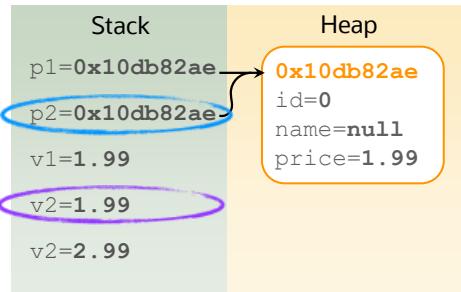
# Parameter Passing

Just like any other local method variables, parameters are stored in 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.

O

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.

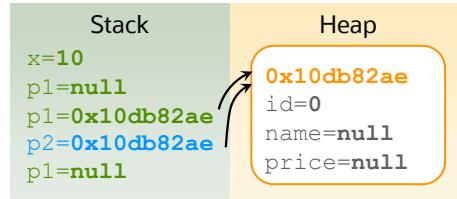
If you intend to inform the invoking method about value changes of locally scoped variables within your method, you can define such methods as returning a value, rather than void.

# Java Memory Cleanup

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

- An object reference is `null` until it is initialized.
- Assigning `null` does not destroy the object; it just indicates the absence of a 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 the Java run time.

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

O

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 (for example, passed as a 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 the Java run time to run garbage collection using `System.gc();` or an equivalent `Runtime.getRuntime().gc();` operation. However, there is no guarantee that the 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 tries 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



## Practices for Lesson 5: Overview

In this practice, you will:

- Create an enumeration to represent the product rating
- Add custom constructors to the product class
- Make product objects immutable



# Implement Inheritance and Use Records

In this section, you will learn how to implement inheritance and use records.

# Objectives

After completing this lesson, you should be able to:

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



# Extending Classes

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

- Object class is an ultimate parent of any other class in Java.
- Object class 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.

O

# Object Class

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

- `toString` method creates text value for the object.
- `equals` method compares a pair of objects.
- `hashCode` method generates int hash value for the object.
- `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.

O

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

# Reusing 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  
}
```

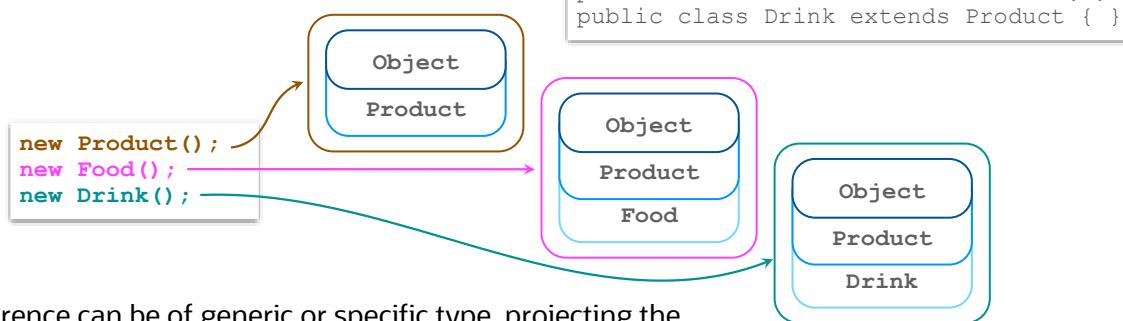
```
public class Food extends Product {  
    private LocalDate bestBefore;  
    public LocalDate getBestBefore() {  
        return bestBefore;  
    }  
    // other Food specific methods and variables  
}  
  
public class Drink extends Product {  
    // other Drink specific methods and variables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

O

# Instantiating Classes and Accessing Objects

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

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



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

- `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();
```

O

# Rules of Reference Type Casting

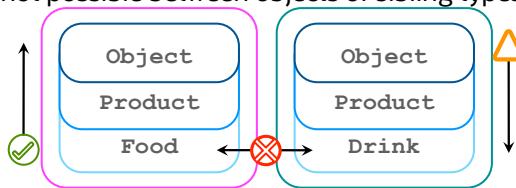
An object can be referenced by using either of the following:

- 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;
```

O

# Verifying Object Type Before Casting the Reference

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

- There is no need to verify or cast the reference type to **invoke generic operations**.
- **Verify type of the object** using the `instanceof` operator.
- **Cast 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)))
                    ? price.multiply(BigDecimal.valueOf(0.2))
                    : BigDecimal.ZERO;
    }
    price = price.subtract(p.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.

O

Using generic (superclass) types to define method parameters and return values helps 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 Food and Drink classes.
- getPrice method is a generic operation declared within the parent class Product.
- getBestBefore method is specific to the Food class.
- 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 a 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.

## Pattern Matching for instanceof

- Pattern matching combines an `instanceof` check with a `reference type-casting`.
- It enables simple and safe access to `subtype specific operations`.

```
public void order(Product p) {
    if (p instanceof Food f && f.getBestBefore().isBefore(LocalDate.now())) {
        LocalDate bestBefore = f.getBestBefore();
        /* Food specific logic using variable f */
    } else {
        /* variable f is out of scope */
    }
}
```



- The example below fails to compile, and variable `f` is considered to be out of scope:

```
public void order(Product p) {
    if (p instanceof Food f || f.getBestBefore().isBefore(LocalDate.now())) {
    }
}
```



O

Pattern matching for the `instanceof` operator allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely.

A pattern is a combination of a predicate, or test, that can be applied to a target, and a set of local variables, known as pattern variables, that are extracted from the target only if the predicate successfully applies to it.

Note that in the first example the conditional-AND operator (`&&`) is short-circuiting. The program can reach the `f.getBestBefore()` expression only if the `instanceof` operator returns `true` and thus guarantees that the object is indeed a type of `Food`, making safe type-casting possible.

However, in the second example, the compiler is able to detect that the program can potentially reach the `f.getBestBefore()` expression even if the `instanceof` check returns `false`, because the logic is reversed to use conditional-OR operator (`||`) thus, you cannot use the pattern variable `f` here, because it is not guaranteed to be of the appropriate type.

# Reference Code Within the Current or Parent Object

Reference the current object and the parent object.

- `this` keyword to reference variables and methods of the current object
- `super` keyword to reference variables and methods of the parent object
- `this` or `super` keyword 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 a subclass from accessing parent class variables and methods.
- ❖ Well-encapsulated code should only expose methods and should hide variables.

O

Variables that are not private are potentially visible within subclasses. This could create 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 by 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 ever be put in this ambiguous situation.

# Defining Subclass Constructors

- The subclass constructor **must** invoke the superclass constructor.
- Subclass can implicitly invoke the superclass constructor only when superclass contains the no-arg constructor (default or explicitly defined).
- 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.



## Reminders :

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

O

**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() { }
}
```

O

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 initializer

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 an 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.

# Overriding Methods and Using Polymorphism

The subclass can override parent class methods.

- The subclass defines the method whose signature matches the parent class method.
- The subclass can widen, but cannot narrow, the 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 8).
- ❖ 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
    }
}
```

O

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 by using a public access modifier, but not by using default to private.

Overriding parent class methods allows Java classes to utilize the 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.

## Reusing Parent Class Logic in Overwritten Method

The subclass may invoke a parent class method by 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);  
    }  
}
```

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

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

```
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);  
    }  
}
```

O

# Defining Abstract Classes and Methods

Abstract classes are used to encourage class extensibility:

- Marked by the **abstract** keyword
- Cannot be directly instantiated
- Meant to be extended by one or more concrete subclasses
- May contain normal variables and methods, which are inherited by subclasses as usual
- 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
    }
}
```

O

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 available only 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, a 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 a 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 in this example.)

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

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

## Defining 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:** Attempting to override a final method or extend a final class would result in a compilation error.  
❖ **Reminder:** A variable becomes a constant (cannot be reassigned) if it is marked with the `final` keyword.

O

# Sealed Classes and Interfaces

Restrict which other types can extend or implement them:

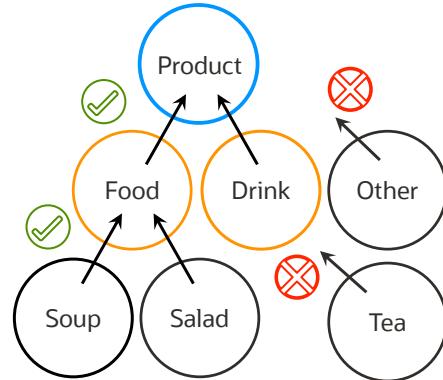
- Permitted subclasses must belong to the same module and, if declared in an unnamed module, the same package as the sealed parent class.
- Every permitted subclass must directly extend the sealed class.
- Every permitted subclass must be defined as either `final`, `sealed`, or `non-sealed`.

```
package shop;
public abstract sealed class Product
    permits Food, Drink { }
```

```
package shop;
public non-sealed class Food extends Product { }
```

```
package shop;
public final class Drink extends Product { }
```

```
package shop;
public class Other extends Product { }
```



O

Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.

Sealed classes allow the author of a class or interface to control which code is responsible for implementing it.

Sealed classes provide a more declarative way than access modifiers to restrict the use of a superclass.

Sealed classes support future directions in pattern matching by providing a foundation for the exhaustive analysis of patterns.

New keywords allow control of inheritance.

- sealed**: Class or interface can only be extended, implemented by permitted classes or interfaces.
- permits**: Used to specify which classes or interfaces are allowed to extend or implement
- non-sealed**: No restriction on any classes or interfaces for extension
- final**: This class cannot be extended.

Any class that is `sealed` must define `permits`. Any permitted class or interface must either be `sealed` and `permits`, `final` or `non-sealed`.

Any class not using any of the above will behave as usual.

A sealed class imposes three constraints on its permitted subclasses:

1. The sealed class and its permitted subclasses must belong to the same module and, if declared in an unnamed module, to the same package.
2. Every permitted subclass must directly extend the sealed class.
3. Every permitted subclass must use a modifier to describe how it propagates the sealing initiated by its superclass:
  - A permitted subclass may be declared final to prevent its part of the class hierarchy from being extended further. (Record classes are implicitly declared final.)
  - A permitted subclass may be declared sealed to allow its part of the hierarchy to be extended further than envisaged by its sealed superclass, but in a restricted fashion.
  - A permitted subclass may be declared non-sealed so that its part of the hierarchy reverts to being open for extension by unknown subclasses. A sealed class cannot prevent its permitted subclasses from doing this. (The modifier non-sealed is the first hyphenated keyword proposed for Java.)

# Overriding 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 the 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
```

O

# Overriding 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  
    }  
}
```

O

The example on the screen 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.

Note on comparing the use of `instanceof` vs `Class.getName()`:

`instanceof` is used for object references of a class or a subclass of the class being compared. If you are using interfaces, it returns true if the class or a super class of the object reference implements that interface.

`getClass().getName().equals("somename")` shows if the class of the object reference is precisely from another class; therefore a subclass will not pass this test.

# Override Object Class Operations: hashCode

It is recommended that all Java classes override certain 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);
    }
}
```

## Notes

- 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).

O

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();
    }
}
```

The `Objects` class provides a convenient operation `hash` to generate such hash codes for any number of objects.

`HashMap` is developed as an array of linked lists. Each element is a bucket that contains a pointer to the beginning of a linked list of map entries.

The buckets number is a power of 2 to accelerate the computations.

The `hashcode` of an object is a 32-bit signed integer value, evaluated by invoking the `hashCode()` method.

If the key is null, then the stored `hashcode` is zero.

To optimize collections in memory based on hashing algorithms, some collections use the notion of buckets that will be used to store the entries for even distribution. Access to entries is faster because they are evenly distributed.

# Comparing String Objects

The String class 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 = "heILlo";
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
```

## Reminders

- ❖ 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 quotation marks: "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.

O

## Java Records

- Immutable and implicitly final data classes.
- Require only the types and names of fields.
- Implicitly extend `java.lang.Record` class.
- A record declaration specifies a description of its contents in a header.
- Accessors, constructor, `equals`, `hashCode`, and `toString` methods are created automatically.

```
public record Product(String name, double price) { }
```

```
Product p1 = new Product("Tea", 1.99);
Product p2 = new Product("Tea", 1.99);
boolean same = p1.equals(p2);
int hashCode = p1.hashCode();
String name = p1.name();
String text = p1.toString();
```

❖ **Note:** Records behave similar to other final classes, may contain methods, implement interfaces, and so on.

O

Rules for record classes:

- A record class declaration does not have an `extends` clause. The superclass of a record class is always `java.lang.Record`.
- A record class is implicitly `final`, and cannot be `abstract`. These restrictions emphasize that the API of a record class is defined solely by its state description, and cannot be enhanced later by another class.
- The fields derived from the record components are `final`.
- A record class cannot explicitly declare instance fields, and cannot contain instance initializers.
- Any explicit declarations of a member that would otherwise be automatically derived must match the type of the automatically derived member exactly, disregarding any annotations on the explicit declaration.
- A record class cannot declare native methods.
- Instances of record classes are created by using a `new` expression.
- A record class can be declared top level or nested, and can be generic.
- A record class can declare static methods, fields, and initializers.
- A record class can declare instance methods.
- A record class can implement interfaces. A record class cannot specify a superclass because that would mean inherited state, beyond the state described in the header.
- A record class can declare nested types, including nested record classes.
- A record class, and the components in its header, may be decorated with annotations.
- Instances of record classes can be serialized and deserialized. However, the process cannot be customized by providing `writeObject`, `readObject`, `readObjectNoData`, `writeExternal`, or `readExternal` methods.

## Custom Record Constructors

- Constructor is generated automatically based on a record header.
- Optionally, custom constructors may also be created in two styles:

```
public record Product(String name) {  
    /* Conventional Constructor - works as usual */  
    public Product(String name) {  
        name = name.toUpperCase();  
        this.name = name;  
    }  
}
```

```
public record Product(String name) {  
    /* Compact Constructor  
     * Implies the argument list from the header  
     * Automatically perform instance variable initializations */  
    public Product {  
        name = name.toUpperCase();  
        // this.name = name; is auto-generated  
    }  
}
```

O

## Record Patterns

- Pattern matching can be used to unravel (deconstruct) a record structure.
- Deconstruction can be nested for records within other records.
- Inference of type arguments of generic record patterns is supported.

```
public record Product(String name, double price) { }
public record Delivery(Product product, LocalDateTime time) { }
```

```
if (obj instanceof Delivery(Product(name, price), time)) {
    System.out.println(name + " " + price + " " + time);
}
```

O

The Records Patterns specification became production in Java SE 21. It is intended to simplify the deconstruction of the record into its components. It is based on the idea that records have well defined declarative structure which allows an automation of the extraction of the individual constituent components from the record. Basically, you can get required parts of the record immediately without having to extract intermediate components and create excessive amount of unnecessary variables.

This is what the example on this page would look like without using record patterns:

```
if (obj instanceof Delivery d) {
    Product p = d.product();
    String name = p.name();
    double price = p.price();
    System.out.println(name + " " + price + " " + time);
}
```

## Pattern Matching for switch

- Pattern matching cases must be exhaustive - generally guaranteed with the default case.
- Cases for enumerations and sealed classes are considered exhaustive without the default case.
- Specific cases (with when conditions) must be placed before generic cases.

```
String result = switch(obj) {  
    case null -> "No data";  
    case Delivery d when d.time().isBefore(LocalDateTime.now()) -> "Delivered";  
    case Delivery d -> "Due "+d.time();  
    default -> obj.toString();  
};
```

- ❖ To maintain backward compatibility, switch with just a default case does not match the null.
- ❖ The null case can be used in combination with a default case explicitly.

O

Pattern matching requires cases to be exhaustive, covering all possible type test outcomes for a given switch expression. This of course could be achieved using the default case. However, to maintain backward compatibility, switch with just a default case does not match the null, but of course null case can be combined with a default explicitly.

Case listings for enumerations and sealed classes that enlist all possible variatns are considered exhaustive without the default case, which is otherwise required.

More specific cases (that use when conditions) must be placed before generic cases in the order of cases, to avoid compilation error.

# Factory Methods

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(...) { }
}
```

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

O

The example on this page illustrates the idea rather than provides a complete code. The actual decision on which sub-type a factory method should produce and return can be based on method parameters or other dynamically determined factors. Factory methods can be used to return differently constructed objects of the same type, or even subtypes of a given type.

Example of factory methods that return different variants of a NumberFormat:

```
NumberFormat currency = NumberFormat.getCurrencyInstance(Locale.FRANCE);
NumberFormat number = NumberFormat.getNumberInstance(Locale.FRANCE);
NumberFormat percent = NumberFormat.getPercentInstance() (Locale.FRANCE);
```

## 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
- Define record classes

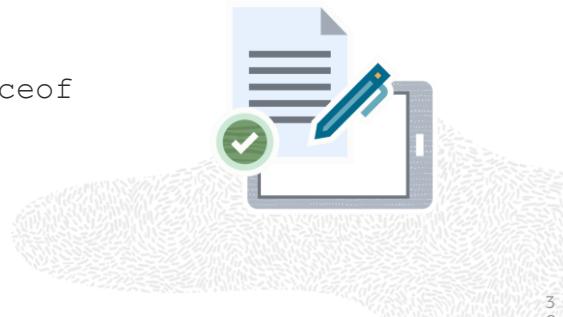


29

## Practices for Lesson 6: Overview

In this practice, you will:

- Create Food and Drink classes that extend the Product class
- Make Product an abstract class
- Add subclass-specific attributes and methods
- Override parent class methods
- Create a ProductManager class to provide factory methods that create instances of Food and Drink
- Work with records and pattern matching for `instanceof`
- Work with sealed classes



30

# Interfaces and Generics

---

# 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



# Java Interfaces

Java interfaces are used to define abstract types.

- Are similar to abstract classes except that:
  - Methods are implicitly abstract (except default, static, and private methods)
  - A class does not *extend* it, but *implements* it
  - A class may implement more than one interface
- All abstract methods from the interface must be implemented by the class.
- Can only contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns

```
public interface Consumable {  
    public void consume();  
}
```

```
public class Food implements Consumable {  
    public void consume() {  
        // eat  
    }  
}  
  
public class Drink implements Consumable {  
    public void consume() {  
        // drink  
    }  
}
```

O

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

# Java 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.



Early version of Java only allowed abstract methods and constants to be placed into interfaces. The reason behind this restriction was that if interfaces would be allowed to have regular methods, then this could potentially cause an ambiguity when two or more interfaces define methods with exact same signature. The problem would be that any class that may implement these interfaces would be inheriting two different versions of operation that are indistinguishable just by looking at the method signatures. This is not a problem for an abstract method, because an abstract method does not have an implementation, and all abstract methods must be implemented by a class that inherits such methods.

Starting from Java SE 8, static and default methods are also allowed in interfaces.

- Interfaces allow static methods to be defined in the exact same way as classes. Invoker can specify exact type (specific interface) where the static method is defined and, therefore, there is no ambiguity on which method is being invoked.
- The default method is a special type of operation that allows provision of concrete implementation code within the interface. If a given class implements two or more interfaces that define default methods with exact same signature, Java compiler would force programmer to override this default method in this class, essentially treating this method as if it was abstract. However, if no such conflict exists between different interfaces, than default methods work just like any other regular methods.
- Default methods, are implicitly **public**, so you can omit the **public** modifier.

Starting from Java SE 9, private methods are also allowed in interfaces.

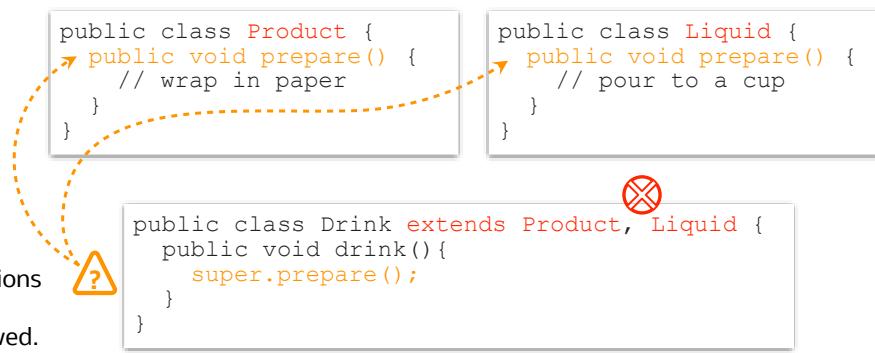
- Private interface methods don't lead to inheritance complications, because they cannot be invoked from anywhere, but a given interface itself, thus do not cause ambiguity for classes that implement interfaces.
- Typically, private interface methods are used to place common logic in one place that is shared between different default methods of this interface.

# 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.



O

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.
- A **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.

O

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.

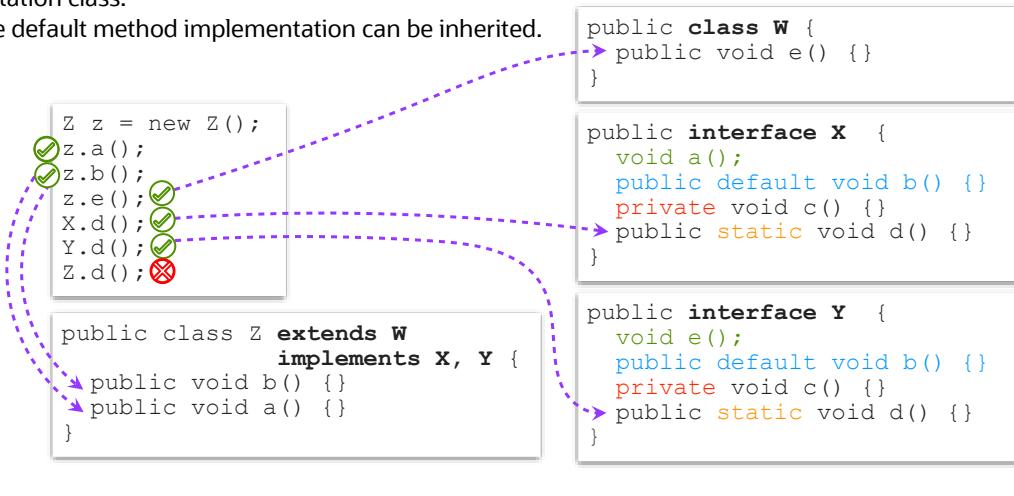
A class must provide only one implementation of a method, including the case when two interfaces define methods with identical signatures.

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.



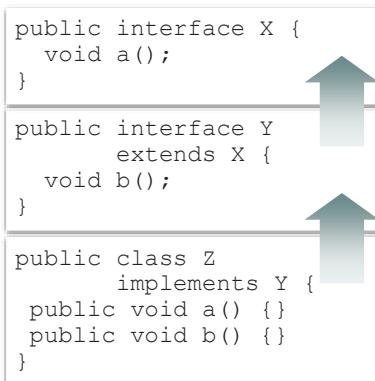
A class can inherit a concrete method that has a signature identical to the signature of a method defined by an interface that this class implements. In this case, the class does not have to override this method, because concrete implementation of it is already available for any given instance of this 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.

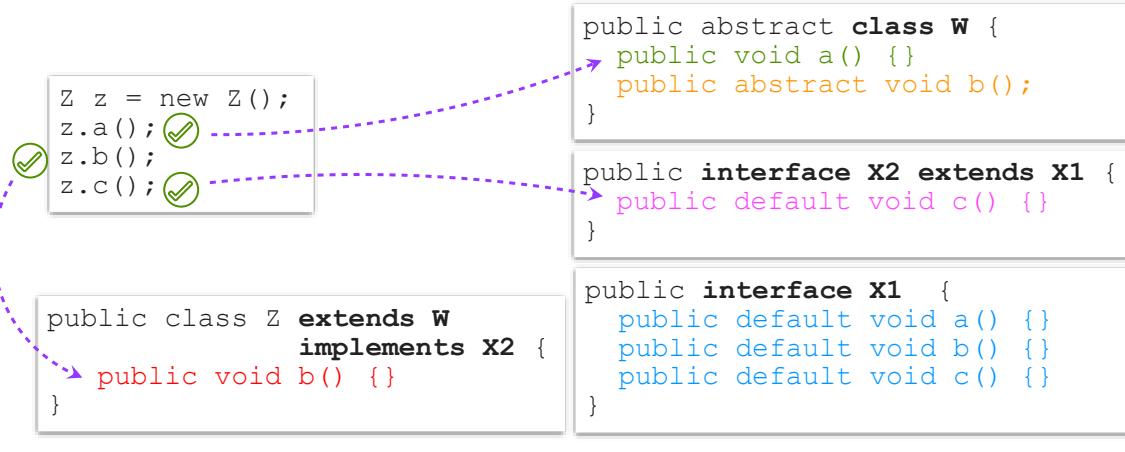
```
public interface X {  
    void a();  
}  
  
public interface Y  
    extends X {  
    void b();  
}  
  
public class Z  
    implements Y {  
    public void a() {}  
    public void b() {}  
}
```



O

## Default Methods Inheritance

- A superclass method takes priority over an **interface default method**.
- An **abstract superclass method must be overridden in a subclass** regardless of the default method availability.
- A **subtype interface's default method takes priority over a super-type interface's default method**.



O

# Interface Is a Type

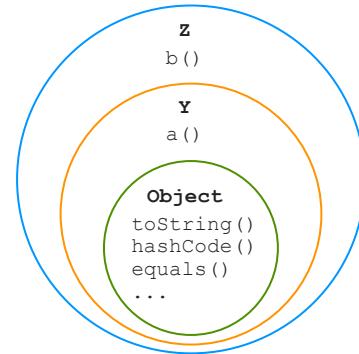
An interface is a type just like a class.

- Interface is a valid reference type that 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();  
}
```



O

# 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" have 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() {}
}
```

O

An interface is considered to be functional if it defines just one abstract method.

Optionally, the `@FunctionalInterface` annotation can be applied to such an interface. This annotation is not required to make interface functional, but instead acts as a safeguard that enables Java compiler to verify that this interface indeed defines only one abstract method and raises a compilation error if a programmer attempts to define more than one, or no abstract methods in this interface.

See the appendix titled “Annotations” for more details.

There is an edge-case regarding methods that interface inherits. As you already know, Object class is an ultimate parent type of all other classes and interfaces in Java. Thus, all methods defined by the Object class are inherited by all other types, which of course includes interfaces. Normally, this is not a problem and most of the time programmers do not even notice this because a class that implements an interface inherits exact same operations from the Object class anyway. However, consider the following unusual case:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2)
    boolean equals(Object obj);
}
```

Notice that this interface redefines the `equals` method inherited from the Object class as abstract. The confusing part, is that from the Java compiler point of view this method is inherited from the Object class and thus it is not considered as defined by this interface. So, the `@FunctionalInterface` annotation applied to this interface does not cause compiler to produce an error indicating that this interface defines more than one abstract method. Also, notice that if you remove the `compare` method definition from this interface, compiler would produce an error indicating that no abstract method is found in this interface.

Essentially this means that methods defined by the Object class are ignored from the consideration that verifies that a given interface conforms to the function interface specification.

In addition to a single abstract method, a Functional interface may also define static, private, and default methods just like any other interface.

# Generics

Generics is a feature of Java language available since Java SE 5.

- It allows operations on objects of various types while providing compile-time type safety.
- Prior to Java SE 5 (**no generics style**), values were wrapped within the class using the type Object.
- Post Java SE 5 (**generics style**), values are wrapped within the class with deferred 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&lt;T&gt; {     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.

O

Generics cannot be applied to:

- Descendants of class Throwable
- Anonymous classes
- Enumerations

Generics allow the usage of parameterized types in methods, classes, and interfaces.

Generics provide code reuse where we can write a method/class/interface once and use it with any type. They provide flexible type safety to your code. They help move many common errors from run time to compile time. They reduce the need for casting with collections. They are used heavily in the Java Collections API.

# 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	With Generics
<pre>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; }</pre> 	<pre>Some&lt;Product&gt; some = new Some&lt;&gt;(); some.setValue(new Product("Tea", 1.99)); <b>✓</b>some.setValue("something"); <b>✗</b>Product product = some.getValue();</pre>

- ❖ **Note:** Generics can be used with both classes and interfaces. Many existing Java interfaces utilize generics.

O

Ways of declaring initializing objects that utilize generics:

- The following example instantiates “Some” object using matching generic expression:

```
Some<Product> some1 = 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. In the following example, the 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<>();
```

- The following 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> some3 = new Some();
```

- The following 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>();
```

- The following example does not use generics and is sometimes described as using the "raw type" and essentially reverts to using Object as type:

```
Some some5 = new Some();
```

- The following example does uses Object as the generic type, which is really the same as not using generics at all:

```
Some<Object> some6 = new Some<Object>();
```

All examples of the generics 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.

- It defines a single abstract method `int compareTo(T o)`.
- It 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);
}
```

O

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 the Comparable interface, giving you a convenient natural way of sorting objects of different types.

## Examples of Java Interfaces: `java.util.Comparator`

The 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
}
```

O

The example in the slide uses arrays.

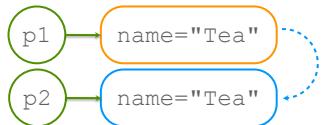
The Comparable interface is implemented as a part of the class, comparing the current object to the parameter. The main advantage of using the Comparator interface is that it is implemented as a separate class. You can have as many implementations of it as required, thereby providing different types 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.



```
package java.lang;  
public interface Cloneable { }
```

```
public class Product  
    implements Cloneable {  
protected Object clone()  
throws CloneNotSupportedException {  
    return super.clone();  
}  
}
```

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

```
package java.lang;  
public class Object {  
protected Object clone()  
throws CloneNotSupportedException {  
    if (!(this instanceof Cloneable)) {  
        throw new CloneNotSupportedException();  
    }  
    // clone object  
}
```

O

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() { }
}
```

O

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, because 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
- Utilize some of the commonly used Java Interfaces
- Implement the Composition design pattern



## Practices for Lesson 7: Overview

In this practice, you will:

- Design Rateable interface to enable the rating of various objects
- Add logic to the `ProductManager` class to handle product reviews, format, and print reports



# Arrays and Loops

---

# 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 by using a loop
- Use multidimensional arrays
- Use embedded loops
- Use break and continue operators



O

# 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" an array element to operate on it.

Accessing 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	<code>null</code>
2	

O

There are two alternative styles of declaring an 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 `{...}`

O

# 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

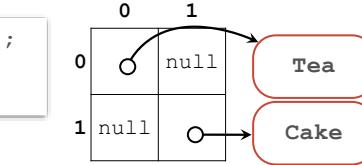
```
int[][] matrix = new int[2][3];
matrix[0][1] = 5;
matrix[1][2] = 7;
```

	0	1	2
0	0	5	0
1	0	0	7

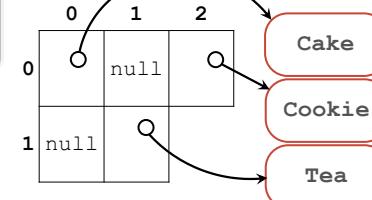
```
int[][] matrix = {{4,1},{2,0,5}};
```

	0	1	2
0	4	1	
1	2	0	5

```
Product[][] products = new Product[2][2];
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")}};
```



O

# 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(b1,5);
```

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

O

The `Arrays.copy` and `Arrays.copyOfRange` methods are overloaded for each primitive data type 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 convenient methods for handling arrays, such as:

- Filling an 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());
```

0	Mary
1	Ann
2	Jane
3	Tom

```
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	0	Ann	0	Ann
1	Ann	1	John	1	Tom
2	John	2	Mary	2	John
3	Tom	3	Tom	3	Mary

O

**Reminder:** `Comparable` and `Comparator` interfaces were covered in the lesson titled “Interfaces and Generics.”

In the example on the screen, 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 the 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).

```
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
}
```

```
while (someMethod()) {
    // iterative logic
}
```

```
int i = 0;
for ( ; i < 10 ; ) {
    // iterative logic
    i++;
}
```

❖ Note: while or do/while loops are best used with a method call that returns Boolean to control iterations.

❖ Note: The example demonstrates the positional nature of the for loop construct (not a recommended way of writing a for loop.)

O

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. Alternatively, 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 a 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** by 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.

O

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

O

Syntactically, a `for` loop construct represents a very unusual case in Java language. It has positional structure, and it uses 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
```

	0	1	2	x
0	O	X	X	
1	X	X	O	
2	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');  
}
```

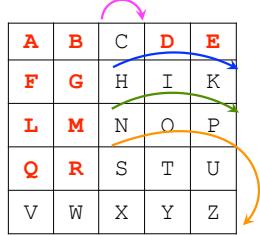
0

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.



Result:

```
ABDE
FGLM
QR
```

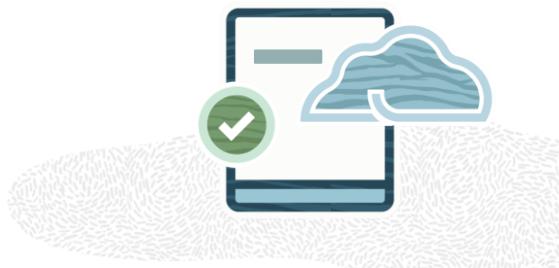
```
char[][] matrix = {{"A", "B", "C", "D", "E"},  
                  {"F", "G", "H", "I", "K"},  
                  {"L", "M", "N", "O", "P"},  
                  {"Q", "R", "S", "T", "U"},  
                  {"V", "W", "X", "Y", "Z"}};  
  
StringBuilder txt = new StringBuilder();  
outerLoopLabel:  
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');  
}
```

O

## 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 by using a loop
- Use multidimensional arrays
- Use embedded loops
- Use break and continue operators



## Practices for Lesson 8: Overview

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



# Collections

# Objectives

After completing this lesson, you should be able to:

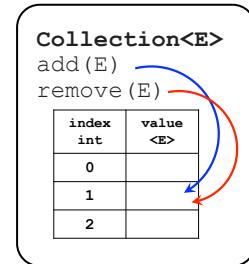
- Introduce Java Collection API interfaces and implementation classes
- Use List, Set, Deque, and Map collections
- Iterate through collection content
- Use Collections class
- Access collections concurrently

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.

O

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

Java Collection API located in the `java.util` package provides automations and convenience wrappers for various types of collections.

## Collection API interfaces:

- `Iterable<T>` iterate through collection.
- `Collection<E>` add and remove elements.
- `SequencedCollection<E>` \* describes a collection with a well-defined encounter order of elements.
- `List<E>` describes a list of elements indexed by `int`.
- `Deque<E>` describes a double-ended queue, providing (FIFO) and (LIFO) behaviors.
- `Set<E>` describes a set of unique elements.
- `SequencedSet<E>` \* is a set-specific variant of `SequencedCollection`.
- `SortedSet<E>` adds ordering ability.
- `Map<K, V>` interface contains a `Set` of unique keys and a `Collection` of values.
- `SequencedMap<E>` \* describes a map with a well-defined encounter order of elements.

❖ Note: `SequencedCollection`, `SequencedSet`, and `SequencedMap` are Java SE 21 new features.



Each interface in the Collection API defines a particular feature such as iterating through all elements in the collection, adding or removing elements, enforcement of element uniqueness, ordering elements, access mechanism, or type of element indexing. Each Collection API class provides implementation of these interfaces, resulting in a collection that has corresponding capabilities.

This page describes frequently used collection types - more variants of collections are available.

- `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.
- `SequencedCollection<E>` interface extends `Collection` and describes a well-defined encounter order of elements, supports operations at both ends, and is reversible. This interface is a Java SE 21 new feature – interfaces that extend this interface in SE 21 were directly extending the `Collection` interface in earlier versions.
- `List<E>` interface extends `SequencedCollection` (in SE21) or `Collection` (in earlier versions) and describes a list of elements indexed by `int`.
- `Deque<E>` interface extends `SequencedCollection` (in SE21) or `Collection` (in earlier versions) and describes a double-ended queue, providing First-In-First-Out (FIFO) and Last-In-First-Out (LIFO) behaviors.
- `Set<E>` interface extends `Collection` and describes a set of unique elements.
- `SortedSet<E>` interface inherits from both the `Set` and (in Java SE 21 the `SequencedCollection`) interfaces and defines ordering abilities.
- `Map<K, V>` interface contains a `Set` of unique keys and a `Collection` of values.

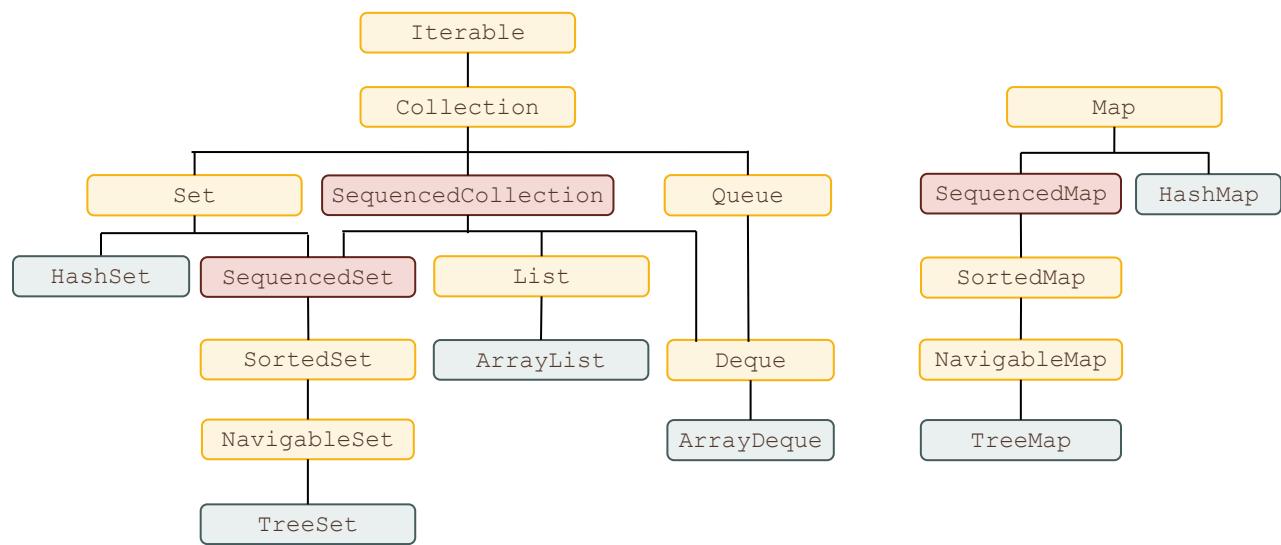
# Java Collection API Implementation Classes

## Examples of Collection API classes:

- `ArrayList<E>` implements `List` interface.
- `ArrayDeque<E>` implements the `Deque` interface.
- `HashSet<E>` implements the `Set` interface.
- `TreeSet<E>` implements the `NavigableSet` interface.
- `HashMap<K, V>` implements the `Map` interface.
- `TreeMap<K, V>` implements the `NavigableMap` interface.

O

# Java Collection API Interfaces and Implementation Classes



O

Note: SequencedCollection, SequencedSet, and SequencedMap are Java SE 21 new features.

Before sequenced collections were formally introduced in Java as these three interfaces, some of the collections were anyway effectively sequenced. For example, the encounter order of elements was well defined in the ArrayList or in the ArrayDeque. However, absence of a common interface specification meant that operations related to the encounter order of elements did not have same names in these different classes. Sequenced collections interfaces are intended to provide common definitions for such operations.

# 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(p1,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.

O

The `LinkedList` class is similar to the `ArrayList`. It has the same methods as the `ArrayList` class because they both implement the `List` interface. You can add elements, modify elements, remove elements, and empty the list.

Elements inside the `ArrayList` are placed into an array. If the array run out of available positions, a new bigger array is instantiated to replace the old one which will be garbage collected.

The `LinkedList` stores its elements as nodes, where each node contains an element and points to previous and next nodes in this list.

**Example:**

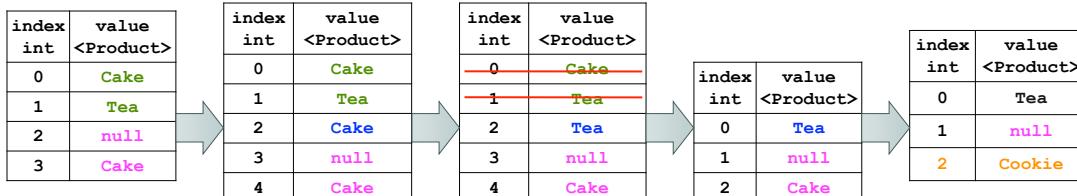
```
public class MainClass {
    public static void main(String[] args) {
        LinkedList<String> pets = new LinkedList<>();
        pets.add("Dog");
        pets.add("Cat");
        pets.add("Fish");
        System.out.println("LinkedList of pets: " + pets);
    }
}
```

# Manage List Contents

List represents collection of elements indexed 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 ✗
```



O

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.

O

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.

The `HashSet` and `TreeSet` classes are two examples of `Set` implementations. They both present collections of unique elements. However, there are some important distinctions.

`HashSet` is used for operations such as search, insert and delete. `HashSet` is optimized in a way that its operations should take relatively constant execution constant time, regardless of the size of the set. Elements in `HashSet` are not ordered.

A `TreeSet` is similar to the `HashSet`, but presents its elements as sorted. The order of elements in the `TreeSet` can be determined by elements themselves if they implement `Comparable` interface, known as natural order, or elements can be sorted in any other way by supplying an implementation of a `Comparator` interface. `TreeSet` performance would degrade as the number of elements placed into the set grows.

`TreeSet` offers several operations that benefit from its ordered nature, such as `first()`, `last()`, `headSet()`, `tailSet()`, `higher()`, `floor()`, and `ceiling()`.

`HashSet` allows null object. `TreeSet` doesn't allow null elements, because the `compareTo()` method that is used to establish the order of elements may throw `java.lang.NullPointerException`.

`HashSet` uses `equals()` method to compare two objects in `Set` and for detecting duplicates. `TreeSet` uses the `compareTo()` method for the same purpose.

**Example:**

```
public class MainClass {  
    public static void main(String[] args) {  
        Set<String> set = new TreeSet<>();  
        set.add("one");  
        set.add("two");  
        set.add("three");  
        set.add("three"); // not added, because such element is already in the set  
        for (String item:set){  
            System.out.println("Item: " + item);  
        }  
    }  
}
```

# 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 the `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);
```

O

In this slide example, values in the Set are references to the 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);

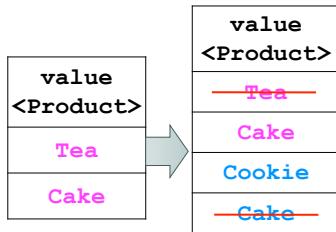
Deque<Product> productDeque1 = new ArrayDeque<>();
Deque<Product> productDeque2 = new ArrayDeque<>(20);
Deque<Product> productDeque3 = new ArrayDeque<>(list);
```

O

## 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); ✘
```

O

First In First Out (FIFO) and Last In First Out (LIFO) are typical deque handling mechanisms.

- `boolean offerFirst(E e)` inserts the specified element at the front of this deque.
- `boolean offerLast(E e)` inserts the specified element at the end of this deque.
- `void addFirst(T)` is the equivalent of `offerFirst` but throws an exception instead of returning Boolean false value.
- `void addLast(T)` is the equivalent of `offerLast` but throws an exception instead of returning Boolean false value.

**Note:** It is preferable to use `offer` instead of `add` methods when working with the capacity restricted 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.

A Deque is a collection that can be used as a stack or a queue.

- It means a “double-ended queue” (and is pronounced “deck”).
- A queue provides FIFO (first in, first out) operations:
  - `add(e)` and `remove()` methods
- A stack provides LIFO (last in, first out) operations:
  - `push(e)` and `pop()` methods

Example of a stack with Deque:

```
public class TestStack {  
    public static void main(String[] args) {  
        Deque<String> stack = new ArrayDeque<>();  
        stack.push("one");  
        stack.push("two");  
        stack.push("three");  
  
        int size = stack.size() - 1;  
        while (size >= 0) {  
            System.out.println(stack.pop());  
            size--;  
        }  
    }  
}
```

## 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 Collection 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 10 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));
```

O

`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 ➔ 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);
```

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 the `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 the `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();
}
```

O

# Sequenced Collections

Sequenced Collections are variants of the Collection interface:

- Describe a well-defined encounter order of elements.
- Support operations for elements at the start and the end of the collection.
- Add the reverse capability.
- **Are not the same as sorted collections!**

```
interface SequencedMap<K, V>
    extends Map<K, V> {
    // New methods:
    SequencedMap<K, V> reversed();
    SequencedSet<K> sequencedKeySet();
    SequencedCollection<V> sequencedValues();
    SequencedSet<Entry<K, V>> sequencedEntrySet();
    V putFirst(K, V);
    V putLast(K, V);
    // Methods promoted from NavigableMap interface:
    Entry<K, V> firstEntry();
    Entry<K, V> lastEntry();
    Entry<K, V> pollFirstEntry();
    Entry<K, V> pollLastEntry();
}
```

```
interface SequencedCollection<E>
    extends Collection<E> {
    // New method:
    SequencedCollection<E> reversed();
    // Methods promoted from Deque interface:
    void addFirst(E);
    void addLast(E);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
}

interface SequencedSet<E>
    extends SequencedCollection<E>,
    Set<E> {
    // New method:
    SequencedSet<E> reversed();
}
```

O

Java SE 21 introduced Sequenced Collections feature with three new interfaces SequencedCollection, SequencedSet and SequencedMap. These interfaces are designed to define behaviours of collections with well-defined order of elements. In fact, many of these behaviours are not new and existed in other Java Collection API interfaces before version 21. So this "new feature" is basically a refactoring on existing operations that are now grouped together in these three new interfaces.

It is very important not to confuse Sequenced and Sorted principles. You may take a collection and sort its elements so that they will be rearranged in a particular order, such as ascending or descending order of numbers, strings, dates etc. However, sequenced collections do not try to rearrange their elements in this manner. Instead, sequenced collections guarantee the encounter order of elements that is not related to the concept of sorting. For example, elements added to an ArrayList (which is an implementation of a SequencedCollection) appear in the order in which they were added:

```
ArrayList<String> list = new ArrayList<>();
list.add("c");
list.add("a");
list.add("b");
list.addFirst("f");
list.addLast("d");
```

The order of elements here is: f, c, a, b, d

However, that would not be the case, if collection is sorted. For example, elements added to a TreeSet (which is an implementation of a SequencedSet) appear in the natural sorting order:

```
TreeSet<String> set = new TreeSet<>();
set.add("c");
set.add("a");
set.add("b");
```

The order of elements here is: a, b, c

Because of this, operations such as addFirst() or addLast() don't make sense in the context of a sorted collection such as a TreeMap. In fact, they actually throw an UnsupportedOperationException in the sorted collection implementations.

This is not a problem for the unsorted SequencedSet collections. For example, the LinkedHashSet collection has no problem providing working implementations of the addFirst() and the addLast() methods, because LinkedHashSet is not an ordered Set.

Java language adheres to the all or nothing inheritance principle, i.e. a class may not partially inherit from a class that it extends, or an interface that it implements. This implies that when you make the decision to extend a class or implement an interface you want to make sure that all of this class or interface methods make sense for the class that extends or implements it. This fundamental design principle appears to be broken in the implementation of sequenced collections, because of the sorted collections implementing sequenced collection interfaces have to effectively disable the addFirst() and the addLast() operations, because these methods cannot logically work in a scenario where collection reorders its content.

# Other Collection Behaviors

Some other examples of generic Collection behaviors include:

- Convert collection to an array using the `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());
```

## Note for the example:

- ❖ Product array is re-created to accommodate extra elements.
- ❖ Method `removeIf` removes all products except tea.

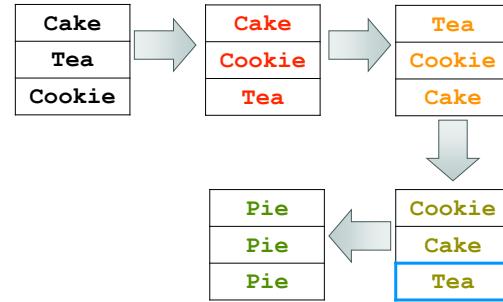
O

## Use `java.util.Collections` Class

Class `java.util.Collections` provides convenience methods for handling collections.

- Filling collection with values
- Searching through the collection
- 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"));
```



O

The example assumes that `Product` class implements `Comparable` interface and overrides the `compareTo` method. Alternatively, an implementation of `Comparator` can be provided:

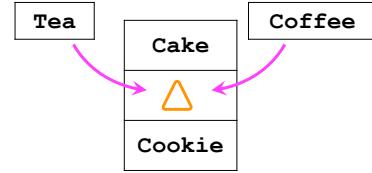
```
public class LengthCompare implements Comparator<Product> {
    public int compare(Product p1, Product p2) {
        if (p1.getName().length() > p2.getName().length()) { return 1; }
        if (p1.getName().length() < p2.getName().length()) { return -1; }
        return 0;
    }
}
Collections.sort(menu, 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 lesson 14 "Java Concurrency and Multithreading".

O

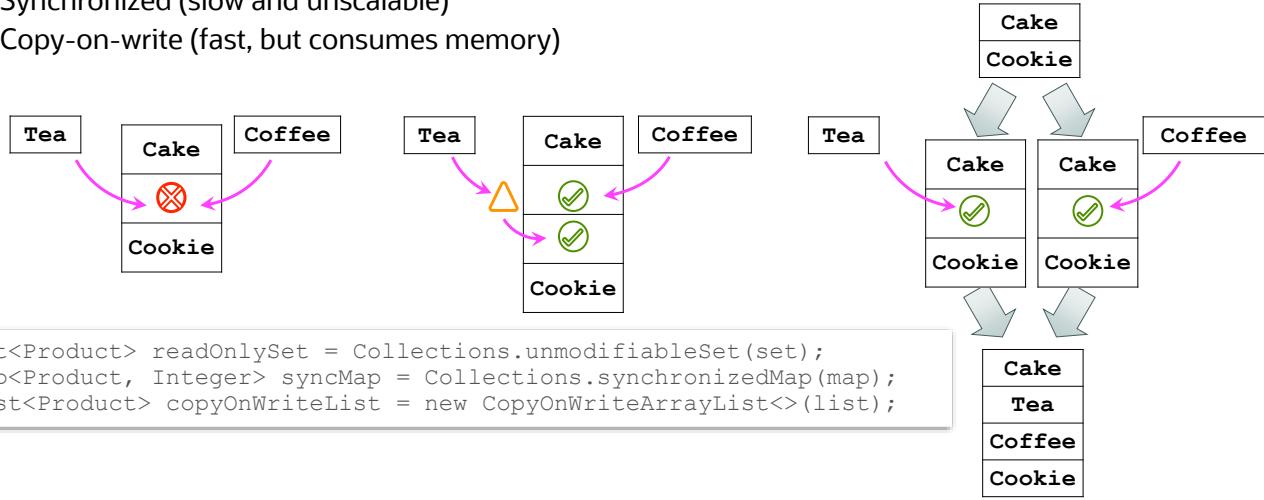
Making collection thread-safe is not the same as making objects inside this collection thread-safe as well:

- 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)



O

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 the 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 the `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`.

O

## 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



## Practices for Lesson 9: Overview

In this practice, you will:

- Use HashMap to store products and reviews
- Provide sorting mechanism for reviews and searching mechanism for products





## Nested Classes and Lambda Expressions

---

Java 8 introduced two new features that make Java code more concise and readable: Nested Classes and Lambda Expressions.

Nested classes allow you to define a class inside another class, which provides a way to encapsulate related data and behavior.

Lambda expressions provide a concise way to represent anonymous functions, which can be passed as arguments to other functions or methods.

# Objectives

- After completing this lesson, you should be able to use:
  - Nested classes
  - Static
  - Member
  - Local
  - Anonymous
  - Lambda expressions



# 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 a 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();
```

O

## Nested Classes Implementation Notes

- The `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, because a probability of the serialized object across different implementations of Java run times 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  
        }  
    }  
}
```

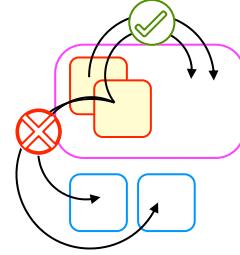
## Static Nested Classes

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();
```



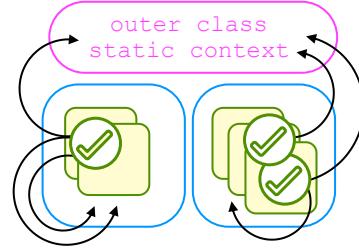
O

# 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);
```

O

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.

## Composite Relationship

There are three types of relationships: Association, Aggregation, and Composition

### Association

An association is a relationship in which one object holds a reference to another object as an instance variable.

Example: Customer -----> Survey

### Aggregation

A special form of association that specifies a relation between the aggregate and a component

It is a weak relationship with independent objects. The associated objects are an essential part of the containing object.

Example: Room -----> Reservation

### Composition

A form of aggregation that requires that a part instance be included in one composite. It is a strong relationship. The composite object is responsible for the creation and destruction of the part instances.

Example: Reservation -----> Payment

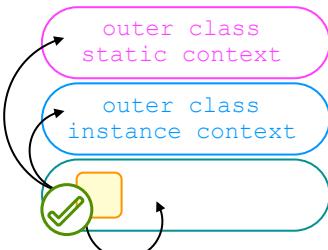
Code example of the member inner class `Item` and the outer containing class `Order` demonstrating the outer class managing the life cycle of the inner class:

```
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 a specific method.

- Instances of the local inner class can only be created within the outer method context.
- It 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();  
    }  
}
```

O

# Anonymous Inner Classes

Anonymous inner class is an implementation of an interface or extension of a class.

- It extends a parent class or implement an interface to override operations.
- It is 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);  
    }  
};
```

O

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)

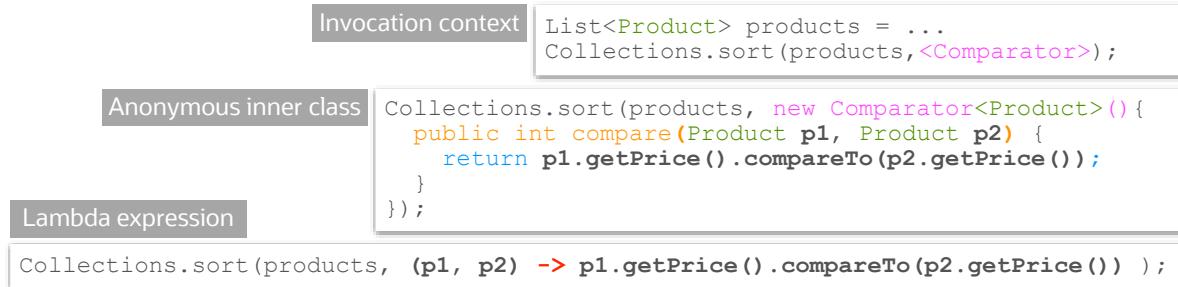
```
List<Product> products = ...  
Collections.sort(products, new Comparator<Product>() {  
    public int compare(Product p1, Product p2) {  
        return p1.getName().compareTo(p2.getName());  
    }  
});  
  
Collections.sort(products, new Comparator<Product>() {  
    public int compare(Product p1, Product p2) {  
        return p1.getPrice().compareTo(p2.getPrice());  
    }  
});
```

O

# 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.



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.

O

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()); // same as the line above
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
```

❖ Note: Class `String` has an instance method `compareToIgnoreCase` that implements non-case-sensitive text sorting.

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);
// Fill collection the menu List object with new instances of Food
Collections.fill(menu, new Food());
// Alternative syntax that does the same as the line above
Collections.fill(menu, Food::new);
```

# Default and Static Methods in Functional Interfaces

Interfaces may provide additional nonabstract 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() { }  
}
```

0

# 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.

O

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 the `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))));
```

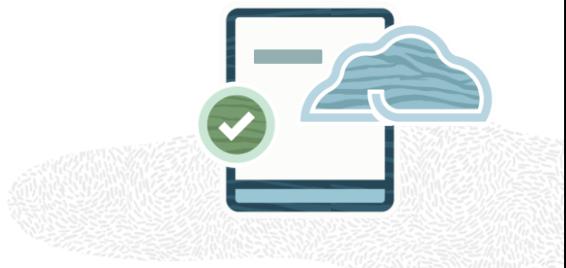
O

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



## Practices for Lesson 10: Overview

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



## Java Streams API

---

# Objectives

After completing this lesson, you should be able to:

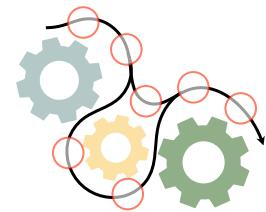
- Describe Java Streams API
- Process stream pipelines
- Implement functional interfaces using lambda expressions
- Describe parallel stream processing
- Use Spliterator



# 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.

O

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).

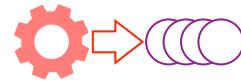
```
BaseStream
└ Stream
└ DoubleStream
└ IntStream
└ LongStream
```

```
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));
```



O

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 the `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
filter	foreach
map	foreachOrdered
flatMap	count
peek	min
distinct	max
sorted	sum
dropWhile	average
skip	collect
limit	reduce
takeWhile	allMatch
	anyMatch
	noneMatch
	findAny
	findFirst

O

Operations `sum` and `average` are available only for primitive stream variants `DoubleStream`, `IntStream`, and `LongStream`.

# Using Functional Interfaces

Stream operations use functional interfaces:

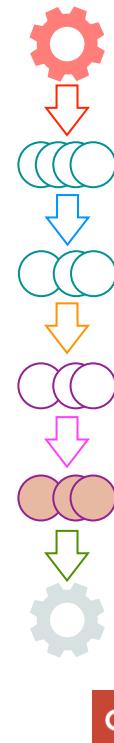
- Located in `java.util.function` package
- Can be implemented using lambda expressions
- Basic function shapes are:
  - `Predicate<T>` defines method `boolean test(T t)` to apply conditions to filter elements.
  - `Function<T, R>` defines method `R apply(T t)` to convert types of elements.
  - `UnaryOperator<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&lt;T&gt;</code>	<code>Function&lt;T, R&gt;</code>	<code>UnaryOperator&lt;T&gt;</code>	<code>Consumer&lt;T&gt;</code>	<code>Supplier&lt;T&gt;</code>

```
List<Product> list = new ArrayList<>();
list.stream()
    .filter(p -> p.getDiscount() == 0)
    .peek(p -> p.applyDiscount(0.1))
    .map(p -> p.getBestBefore())
    .forEach(d -> d.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



O

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

	<b>Predicate</b>	<b>Function</b>	<b>UnaryOperator</b>	<b>Consumer</b>	<b>Supplier</b>
<b>primitive output</b>		<code>ToIntFunction&lt;T&gt;</code> <code>ToLongFunction&lt;T&gt;</code> <code>ToDoubleFunction&lt;T&gt;</code>			<code>IntSupplier</code> <code>LongSupplier</code> <code>DoubleSupplier</code> <code>BooleanSupplier</code>
<b>primitive input</b>	<code>IntPredicate</code> <code>LongPredicate</code> <code>DoublePredicate</code>	<code>IntFunction&lt;R&gt;</code> <code>LongFunction&lt;R&gt;</code> <code>DoubleFunction&lt;R&gt;</code>		<code>IntConsumer</code> <code>LongConsumer</code> <code>DoubleConsumer</code>	
<b>primitive input-output</b>		<code>IntToLongFunction</code> <code>IntToDoubleFunction</code> <code>LongToIntFunction</code> <code>LongToDoubleFunction</code> <code>DoubleToIntFunction</code> <code>DoubleToLongFunction</code>	<code>IntUnaryOperator</code> <code>LongUnaryOperator</code> <code>DoubleUnaryOperator</code>		

✖ 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();
```



The example demonstrates conversions from primitive to object and from object to primitive streams.

- 1) Create a stream of double primitive numbers.
- 2) Remove hole numbers from the stream using a DoublePredicate interface.
- 3) Convert a primitive stream to an object stream.
- 4) Convert Double objects into BigDecimal objects using Function interface.
- 5) Round BigDecimal values using UnaryOperator interface.
- 6) Convert an object stream to an int primitive stream using theToIntFunction interface.
- 7) Compute sum of all int values in the stream.

```
int x = DoubleStream.of(1.234, 1.0, 3.987, 0.321, 4.0) // (1)
    .filter(n -> n != (int)n) // (2)
    .boxed() // (3)
    .map(n->BigDecimal.valueOf(n)) // (4)
    .map(n->n.round(new MathContext(0, RoundingMode.HALF_UP))) // (5)
    .mapToInt(n->n.intValue()) // (6)
    .sum(); // (7)
```

The next example demonstrates an int variant of the Supplier:

- 1) Generate an infinite stream of random int values between 0 and 10.
- 2) Terminate stream pipeline processing when value of 3 is generated.
- 3) Compute the sum of int values generated within this stream.

```
int x = IntStream.generate(() -> (int)(Math.random()*10)) // (1)
    .takeWhile(n -> n !=3)                                // (2)
    .sum();                                                 // (3)
```

Class `java.util.Random` provides various operations that generate a stream of random numbers. In the example, class `Random` acts as a `DoubleSupplier`. Example also uses a `DoubleToIntFunction` interface to convert double values into int values in the mapping operation.

- 1) Generate a stream of 10 random double values between 0 and 1.
- 2) Produce stream of int values by multiplying each number by 100 and rounding.
- 3) Compute the sum of all int values in the stream:

```
Random random = new Random();
int x = random.doubles(10)                      // (1)
    .mapToInt(n->(int)(Math.round(n*100))) // (2)
    .sum();                                 // (3)
```

The next example shows a stream of Boolean primitive values generated using a `java.util.function.BooleanSupplier`, which is a boolean primitive variant of the `Supplier` functional interface. The functional method of `BooleanSupplier` is `getAsBoolean()` which returns a boolean result.

```
@FunctionalInterface
public interface BooleanSupplier {
    boolean getAsBoolean();
}
```

**Example:**

```
BooleanSupplier boolSup = () -> "Cake".length() > 3;
System.out.println(boolSup.getAsBoolean());
```

# 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, U>` defines method `void accept(T t, U u)` to process a pair of elements.

	<b>Predicate</b>	<b>Function</b>	<b>UnaryOperator</b>	<b>Consumer</b>
<b>primitive variants</b>	<code>BiPredicate&lt;T, U&gt;</code>	<code>BiFunction&lt;T, U, R&gt;</code>	<code>BinaryOperator&lt;T&gt;</code>	<code>BiConsumer&lt;T, U&gt;</code>
		<code>ToIntBiFunction&lt;T, U&gt;</code>	<code>IntBinaryOperator</code>	<code>ObjIntConsumer&lt;T&gt;</code>
		<code>ToLongBiFunction&lt;T, U&gt;</code>	<code>LongBinaryOperator</code>	<code>ObjLongConsumer&lt;T&gt;</code>
		<code>ToDoubleBiFunction&lt;T, U&gt;</code>		<code>ObjDoubleConsumer&lt;T&gt;</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.

O

# 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 the abstract `void accept(T t)` method.
- The default `andThen` method provided by the `Consumer` interface combines consumers together.
- Unlike the `forEachOrdered`, 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).

O

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 have 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`:
  - `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)));
```



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` interface:
  - `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`.

O

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.

Object streams values can be mapped to primitive values, and primitive stream values can be mapped to objects. The `mapToObj()` operation of the `IntStream` returns an object-valued Stream consisting of the results of applying the given function. It is a lazy intermediate operation.

Syntax:

```
<U> Stream<U> mapToObj(IntFunction<? extends U> mapper)
```

Example:

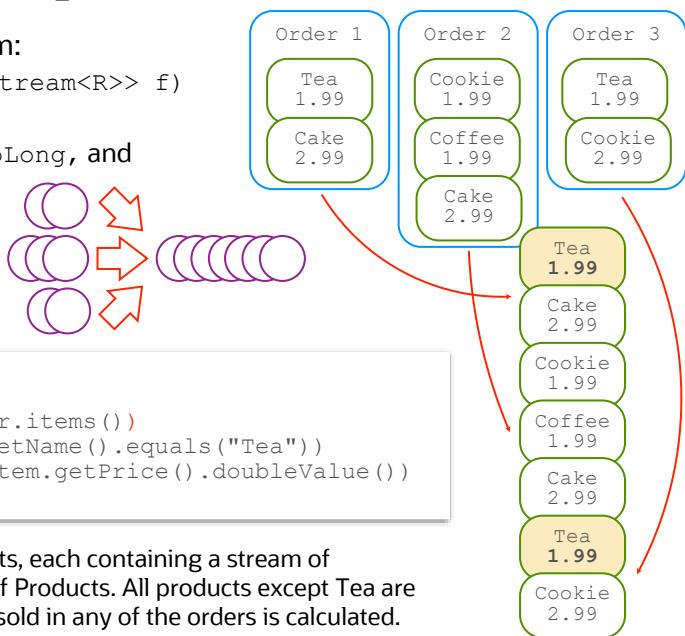
```
IntStream is = IntStream.range(1, 5);
Stream<String> str = is.mapToObj(value -> Integer.toBinaryString(value));
str.forEach(System.out::println);
```

# 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.

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();
```

The `Stream.concat()` operation concatenates the elements of the second stream to the elements of the first stream. The stream created is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel.

Syntax:

```
static <T> Stream<T>
    concat(Stream<? extends T> stream1, Stream<? extends T> stream2)
```

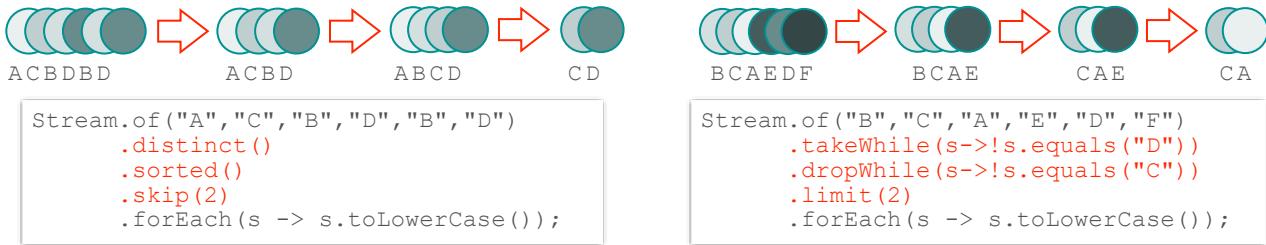
Example:

```
Stream<String> str1 = Stream.of("car", "truck");
Stream<String> str2 = Stream.of("motorcycle", "bicycle");
Stream.concat(str1, str2).forEach(element -> System.out.println(name));
```

# 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 a given size.



- Operations `takeWhile` and `dropWhile` may produce different results if the stream is ordered.
- Their cost can be significantly increased for ordered and parallel streams.

O

Example related to the intermediate operations shown in the slide:

```

public class Test {
    public static void main(String[] args) {
        Arrays.asList(3,4,2,1,4,5,1).stream()
            .distinct().sorted().skip(2)
            .forEach(n -> {
                System.out.println(n);
            });
        Arrays.asList("B", "C", "A", "E", "D", "F").stream()
            .takeWhile(s->!s.equals("D"))
            .dropWhile(s->!s.equals("C"))
            .limit(2).forEach(s -> {
                System.out.println(s);
            });
    }
}

```

Output:

```

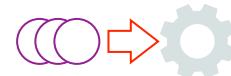
3
4
5
C
A

```

# 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 the `Optional` object.
- Operation `findFirst()` returns the first element from the stream wrapped in the `Optional` object.



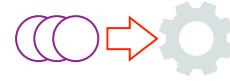
```
String[] values = {"RED", "GREEN", "BLUE"};
boolean allGreen = Arrays.stream(values).allMatch(s->s.equals("GREEN"));
boolean anyGreen = Arrays.stream(values).anyMatch(s->s.equals("GREEN"));
boolean noneGreen = Arrays.stream(values).noneMatch(s->s.equals("GREEN"));
Optional<String> anyColour = Arrays.stream(values).findAny();
Optional<String> firstColour = Arrays.stream(values).findFirst();
```

0

## 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();        // 4
double avgValue = v3.isPresent() ? v3.getAsDouble() : 0;
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.



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

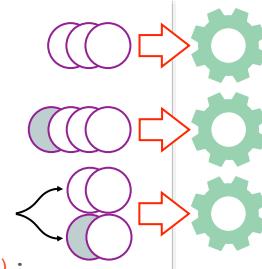
Produce a single result from the stream of values using the `reduce` operation:

- `Optional<T> reduce(BinaryOperator<T> accumulator)` performs accumulation of elements.
- `T reduce(T identity, BinaryOperator<T> accumulator)` `identity` acts as the 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.

O

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. After the maximum value is located, the `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);
```

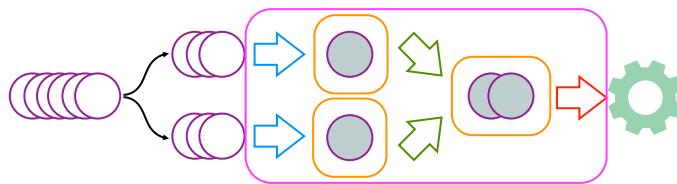
# General Logic of the collect Operation

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.



O

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());
```

0

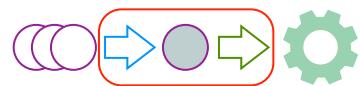
Calculating and using summary statistics object:

```
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) ));
```

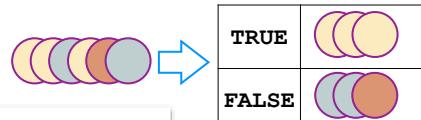
0

# 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()));
```

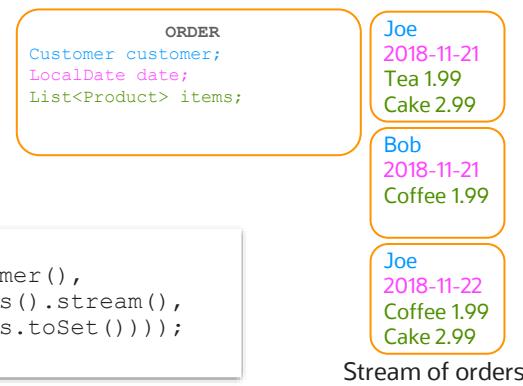


0

# 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.



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, whereas the flatMapping function used at collect stage created a single set of products per customer.

This example demonstrates the effect of using the filter function on a stream compared to a filtering function:

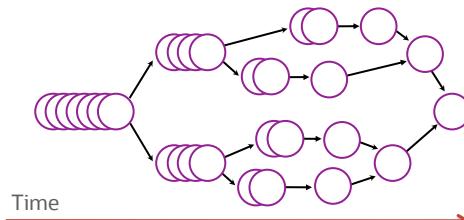
```
Map<Customer, Set<Order>> customerOrderOnDate =  
orders.stream().filter(o -> o.getDate().equals(LocalDate.of(2018,11,22)))  
.collect(Collectors.groupingBy(o -> o.getCustomer(),  
Collectors.toSet()));  
{Joe=[Order[date=2018-11-22, customer Joe, products=[Coffee, Cake]]]}
```

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();
```

O

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, because that was the last call:

```
list.stream().sequential().mapToDouble(p-
>p.getPrice().doubleValue()).parallel().sum();
```

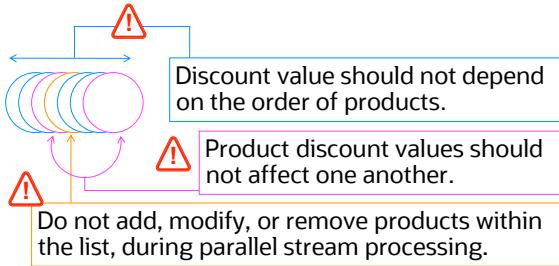
The term "Stochastic" describes the processing order of execution of which is randomly determined and cannot be predicted.

# Parallel Stream Processing Guidelines

Parallel stream processing should observe the following guidelines:

- **Stateless** (state of one element must not affect another element)
- **Noninterfering** (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 the 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.

O

**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()));
```



O

Factors that can degrade parallel stream processing performance:

- Triggering parallel stream processing on a system that does not have multicore 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:

$\text{OverallProcessingCost} = \text{NumberOfStreamElements} * \text{IndividualElementProcessingCost}$

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.

# Spliterator

`Spliterator` is analogue of `Iterator`, with parallel processing capability.

`Spliterator` is used to process content of streams and collections.

- Process next element if it exists
- Process all remaining elements
- Once element is processed it is no longer available
- Attempt to split `Spliterator` object in half

```
Spliterator<Integer> s1 = new Random().ints(10,0,10).spliterator();
s1.tryAdvance(v->System.out.print(v));
Spliterator s2 = s1.trySplit();
if (s2 == null) {
    System.out.println("Did not split");
} else {
    s1.forEachRemaining(v->System.out.print(v));
    s2.forEachRemaining(v->System.out.print(v));
}
```

## Notes:

- ❖ Method `tryAdvance()` is an alternative to the combination of `hasNext()` and `next()` methods of the `Iterator`.
- ❖ Method `forEachRemaining()` is an alternative to the entire `Iterator` loop



Method `trySplit()` may return null if for example there are elements available, or inability to split because traversal has already commenced, or constraints imposed by the source data structure, or efficiency considerations.

In this example processing of both spliterators is sequential, which means that they did not need to be split. However, it is possible to process each part in a different thread, which would justify the split action.

It may be useful to convert the `Iterable` or `Iterator` object into a `Stream`. This enables the use of lambda expressions and a further processing of data using the `Spliterator`.

## Convert Iterable to Stream:

```
Iterable<String> iterable = Arrays.asList("Tea", "Cake");
Stream<String> stream = StreamSupport.stream(iterable.spliterator(), false);
stream.forEach(System.out::println);
```

## Convert Iterator to Stream:

```
Iterator<String> iterator = Arrays.asList ("Tea", "Cake").listIterator();
Spliterator<String> spl =
    Spliterators.spliteratorUnknownSize(iterator, Spliterator.ORDERED);
Stream<String> stream = StreamSupport.stream(spl, false);
stream.forEach(System.out::println);
```

# Spliterator Characteristics

Analyze Spliterator characteristics

- A Boolean test to determine if spliterator possesses a given characteristic
- Attempt to determine a number of available elements
- Characteristics change upon a split

```
Spliterator<Integer> s1 = new Random().ints(10,0,10).spliterator();
String characteristics =
    "Concurrent "+s.hasCharacteristics(Spliterator.CONCURRENT)+"\n"+
    "Distinct "+s.hasCharacteristics(Spliterator.DISTINCT)+"\n"+
    "Immutable "+s.hasCharacteristics(Spliterator.IMMUTABLE)+"\n"+
    "NonNull "+s.hasCharacteristics(Spliterator.NONNULL)+"\n"+
    "Ordered "+s.hasCharacteristics(Spliterator.ORDERED)+"\n"+
    "Sized "+s.hasCharacteristics(Spliterator.SIZED)+"\n"+
    "Sorted "+s.hasCharacteristics(Spliterator.SORTED)+"\n"+
    "Subsized "+s.hasCharacteristics(Spliterator.SUBSIZED);
System.out.println(characteristics);
System.out.println("Size "+s.getExactSizeIfKnown());
System.out.println("Estimate Size "+s.estimateSize());
```

O

## Summary

- In this lesson, you should have learned how to:
  - Describe Java Streams API
  - Process stream pipelines
  - Implement functional interfaces using lambda expressions
  - Describe parallel stream processing
  - Use Spliterator



## Practices for Lesson 11: Overview

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





# Exception Handling, Logging, and Debugging

---

# 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

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;
}
```

✿ Note: Use of the `module-info` class is covered in the lesson titled “Modules and Deployment.”

O

Java Logging API is part of the `java.logging` module. Development and deployment of modularized Java applications are covered in the lesson titled “Modules and Deployment.”

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.

4

O

Example of using different logger levels and methods:

```
public class TestLogger {
    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.

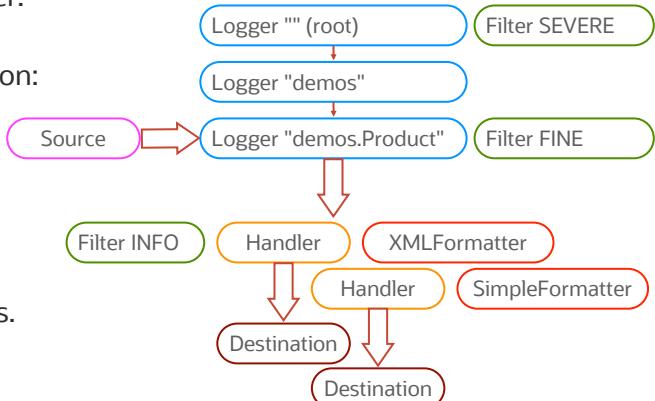
1. Logging level can be set programmatically or via the configuration.
2. Message is concatenated, but is not recorded because it is below the logging-level threshold.
3. Message is not processed if it is below the logging-level threshold.
4. Object parameters can be used to avoid concatenating messages 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 of 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



6

O

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
```

7

O

By default, `logging.properties` is used to configure Java Logging. It is located in the `JAVA_HOME/conf` folder. (For Java version 8 or earlier that was `JAVA_HOME/jre/lib` folder). 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

## Notes:

- ❖ 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
```

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);  
    }  
}
```

❖ **Note:** 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.

9

O

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 the exception propagation process

When an 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 */  
}
```

❖ **Note:** 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.

10

O

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

```
try {  
    doThings();  
}catch(NullPointerException |  
      ArithmeticException e){  
    /* exception handler actions */  
}catch(NoSuchElementException 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 */  
}
```

## Notes:

- 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.

# 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.ArithmaticException: / by zero
    at Test.divide(Test.java:9)
    at Test.main(Test.java:5)
```

No explicit exception handler is required in this example, because `ArithmaticException` is a subclass of `RuntimeException` and thus represents an unchecked exception.

## Helpful NullPointerExceptions

- Since version 14, Java NullPointerException produces extra error message information that helps to troubleshoot the issue.
- Option is on by default, but can be turned off for security reasons.

```
1 public class Test {  
2     private static String value;  
3     public static void main(String[] args) {  
4         System.out.println(value.indexOf("a"));  
5     }  
6 }
```



```
java Test  
Exception in thread "main" java.lang.NullPointerException: Cannot  
invoke "String.indexOf(String)" because "Test.value" is null  
    at Test.main(Test.java:4)
```

NEW

```
java Test  
Exception in thread "main" java.lang.NullPointerException  
    at Test.main(Test.java:4)
```

OLD



13

Since version 14 Java NullPointerException produces extra error message information that helps to troubleshoot the issue. This feature was present but turned off by default in Java 14. The -XX:-ShowCodeDetailsInExceptionMessages flag was used to turn it on. However, this feature was changed to be on by default in Java version 15 or later.

# 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 the 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;
    }
}
```

```
java Test
Exception in thread "main" java.lang.ArithmetricException: Error: 5/0
        at Test.divide(Test.java:10)
        at Test.main(Test.java:5)
```

## Notes:

- ❖ 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.

# 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 the `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");
    }
}
```

**Notes:**

- ❖ You may catch unchecked exceptions.
- ❖ You must catch checked exceptions  
(unless you listed them in a `throws` clause to be propagated further).

15

O

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, because it makes it less clear as to 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);
    }
}
```

# Resource Auto-Closure

Try-with-parameters syntax provides auto-closure of resource.

- Classes that implement the `AutoCloseable` interface can be instantiated using the try-with-parameters syntax.
- Multiple resources may be initialized inside the 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"));
      PrintWriter out = new PrintWriter(new FileWriter("other.txt"))) {
    String text = in.readLine();
    out.println(text);
} catch (FileNotFoundException ex) {
    logger.log(Level.SEVERE, "Opening file error", ex);
} catch (IOException ex) {
    logger.log(Level.SEVERE, "Read-write error", ex);
}

```

17

O

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"));
PrintWriter out = new PrintWriter(new FileWriter("other.txt"));
try (in ; out) {
    String text = null;
    while ((text = in.readLine()) != null) {
        out.println(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);
    }
}
```

## 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();
}
```

# Java Debugger

- Debugger is used to find and fix bugs in Java platform programs.
- Command-line debugger tool `jdb` is provided with JDK.
- Java IDEs provide visual debug capabilities:
  - Toggle breakpoints in your source code



```

30 > public class Shop {
31
32 >     public static void main(String[] args) {
33         ProductManager pm = new ProductManager(Locale.UK);
34         ● pm.createProduct( id: 101, name: "Tea", BigDecimal.valueOf(1.99),
35             Rating.NOT_RATED);
36         pm.printProductReport( id: 101);
37         pm.reviewProduct( id: 101, Rating.FOUR_STAR, comments: "Nice hot cup of tea");
38         ● pm.reviewProduct( id: 101, Rating.TWO_STAR, comments: "Rather weak tea");
39         pm.reviewProduct( id: 101, Rating.FOUR_STAR, comments: "Fine tea");
40         pm.reviewProduct( id: 101, Rating.FOUR_STAR, comments: "Good tea");

```

- Launch program in debug mode (from the main toolbar, or right click on a Java class)



**Notes:**

- ❖ Debugger actions are covered next.
- ❖ JDB tool documentation is available at  
<https://docs.oracle.com/en/java/javase/21/docs/specs/man/jdb.html>

20

O

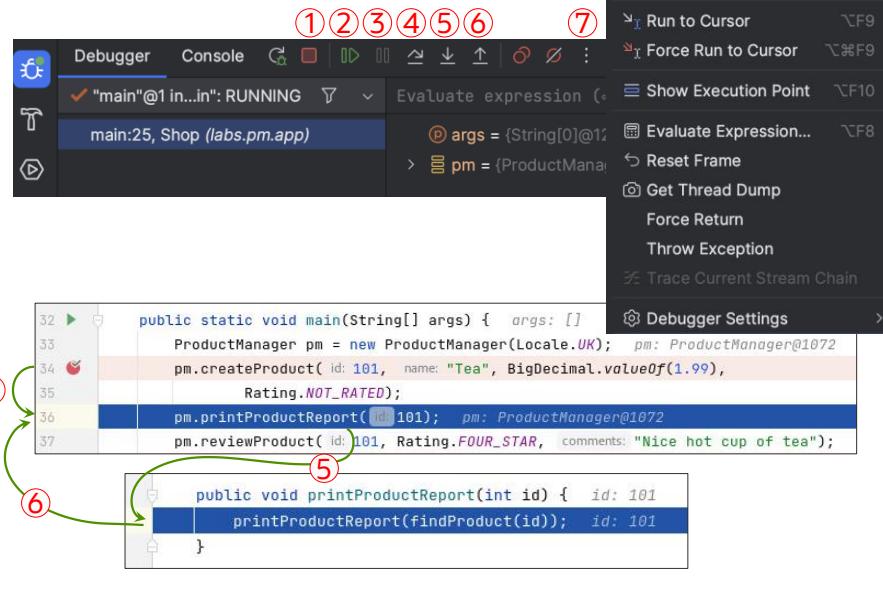
To set breakpoints, click in the margin of a line of code.

You can set multiple breakpoints in multiple classes.

# Debugger Actions

Once debugger is running, you can execute the following debug actions:

1. Stop debugger session
2. Resume
3. Pause
4. Step over
5. Step into
6. Step out
7. More actions



21

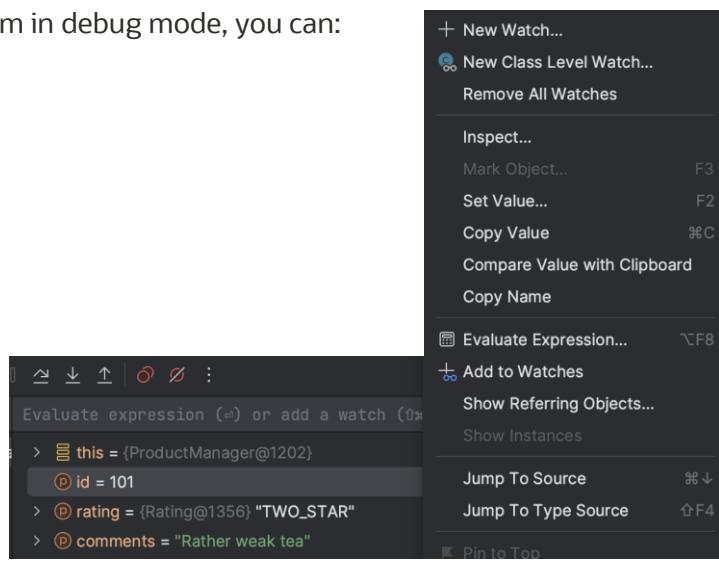
Action available in debug mode allows you to do the following:

1. Exit the debug session by clicking the button.
2. Pause the debug session.
3. Continue running until the next breakpoint or the end of the program.
4. Shows the current execution point in the corresponding class file.
5. If execution has stopped just before a method invocation, you may want to fast-forward to the next line after the method.
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. You may want to step into by ignoring stepping filters for libraries, constructors, and so on.
8. If you have stepped into a method or another class, use the last button to step back out into the original code block.
9. 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.
10. Evaluate arbitrary expressions.

# Manipulate Program Data in Debug Mode

When your IDE is executing program in debug mode, you can:

- Study all the available variables
- Evaluate expressions
- Modify the values
- Watch the specific variables



22

O

To modify a variable value, right-click it and invoke the "Set Value..." menu. This allows you to set your own values and evaluate your logic code.

To focus on a specific variable, right-click it and invoke the "Add to Watches" 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.

```
assert <boolean expression>;
assert <boolean expression>: <error text expression>;
```

- When assertion expression is false, application terminates and assertion error information is printed.

```
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>
```

23

O

Assertions are designed for code testing purposes. If you need to 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 IntelliJ:

1. In IntelliJ, choose Run → Edit Configurations.
2. Select the run configuration of interest.
3. Click the Modify options link and choose Add VM options.
4. Add `-ea` to the VM options box.

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.

## JUnit Testing

JUnit is an open-source Java unit testing framework. It is one of the best testing methods for regression testing. It is used to write and run repeatable automated tests.

When using JUnit, the scope contains the least amount of code that performs a separate task that can be a single method or class. There is a good reason that we limit scope when unit testing.

Regression testing guarantees that latest errors fix or improvement does not break the functionality of your method or class when testing the changes you make to your code.

There is a tutorial to set up JUnit on IntelliJ for your projects, create tests, and run them to test your code and make sure it works correctly.

## 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();  
    }  
}
```

# 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();  
    }  
}  
public void a() throws IOException {  
    if (false) {  
        throw new IOException();  
    }  
}
```

# Program Flow Producing a Runtime Exception

Which operations are going to be invoked in this scenario?



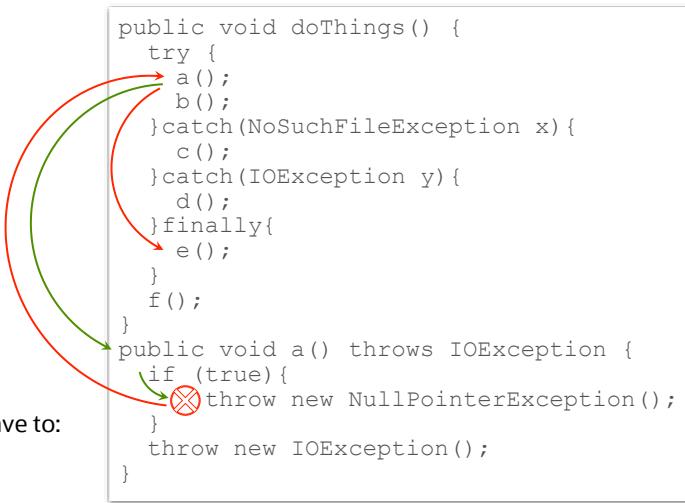
```
public void doThings() {  
    try {  
        a();  
        b();  
    } catch (NoSuchFileException x) {  
        c();  
    } catch (IOException y) {  
        d();  
    } finally {  
        e();  
    }  
    f();  
}  
public void a() throws IOException {  
    if (true) {  
        throw new NullPointerException();  
    }  
    throw new IOException();  
}
```

# 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 associated with it



You could also add a `catch` clause to catch unchecked exceptions, in which case the handling would be no different from any other checked exception scenario.

# Program Flow Catching Specific Checked Exception

Which operations are going to be invoked in this scenario?



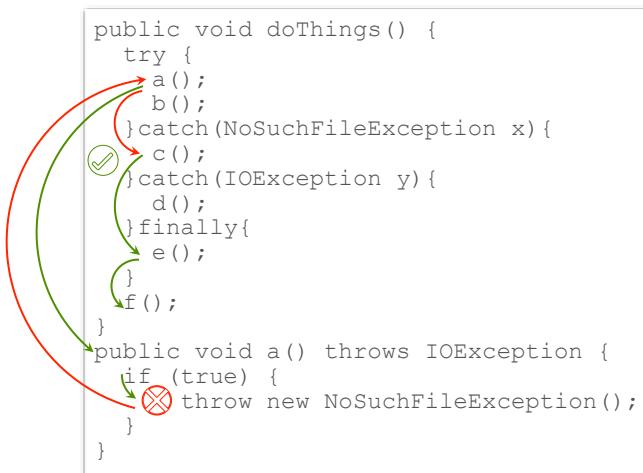
```
public void doThings() {  
    try {  
        a();  
        b();  
    } catch(NoSuchFileException x){  
        c();  
    } catch(IOException y){  
        d();  
    } finally{  
        e();  
    }  
    f();  
}  
public void a() throws IOException {  
    if (true) {  
        throw new NoSuchFileException();  
    }  
}
```

# Program Flow Catching Specific Checked Exception

When `NoSuchFileException` is produced:

- Execution path invokes methods `a`, `c`, `e`, and `f`
- `NoSuchFileException` catch block is triggered
- Finally block is executed
- The rest of the program continues normally because exception was intercepted

- `NoSuchFileException` is a subclass of `IOException`, so the `throws` clause may (but does not have to) explicitly list it.
- `IOException` catch block could intercept the `NoSuchFileException`, unless a more specific catch is provided.



29

O

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?



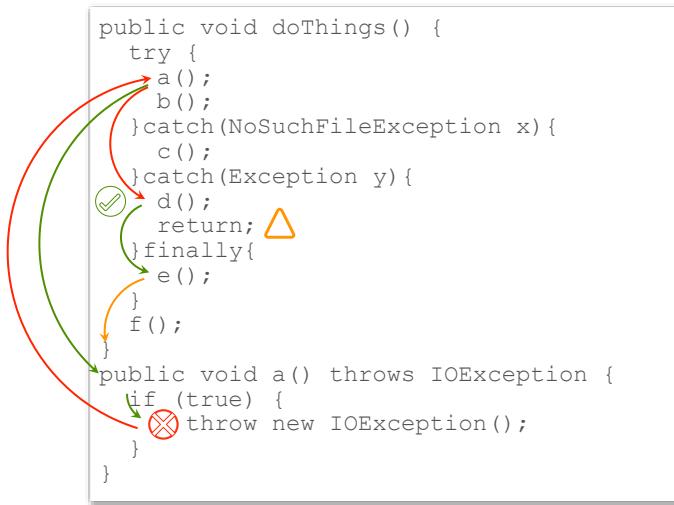
```
public void doThings() {  
    try {  
        a();  
        b();  
    } catch(NoSuchFileException x){  
        c();  
    } catch(Exception y){  
        d();  
        return;  
    } finally{  
        e();  
    }  
    f();  
}  
public void a() throws IOException {  
    if (true) {  
        throw new IOException();  
    }  
}
```

# 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 run time using `System.exit(0)`.



## 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

32



## Practices for Lesson 12: Overview

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





## Java IO API

---

# Objectives

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

- IO Fundamentals
- Serialization/Deserialization
- Working with filesystems



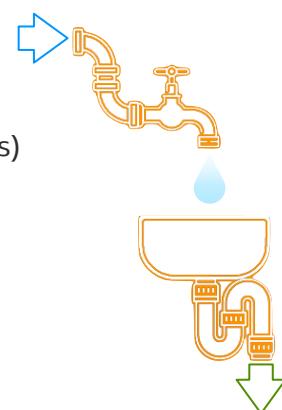
# Java Input-Output Principles

Characteristics of Java Input-Output (IO):

- Read information from various **sources** - input direction
- Write information to various **destinations** - output direction
- Information transferred through a series of **interconnected streams** (pipes)

Streams are categorized based on:

- The type of data that a 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



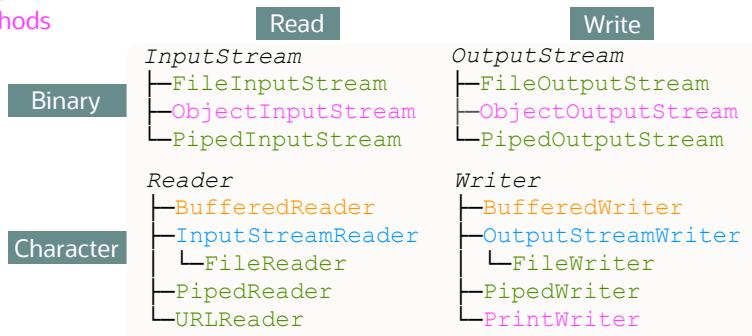
❖ **Note:** Sources and destinations could be Files, Network Sockets, Console, Memory, and so on.

O

# 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 because many IO streams provide combinations of abilities.

O

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	
int read(byte[] buffer, int offset, int length)	read binary data from the stream
void mark(int position)	mark position in the stream
boolean markSupported()	
long transferTo(OutputStream out)	transfer all data from input to output
int available()	check the amount of available data
void skip(long length)	skip data
void reset()	reset the stream
void close()	close the stream
OutputStream	
void write(byte[] buffer, int offset, int length)	write binary data to the stream
void flush()	flush the stream
void close()	close the stream

❖ See notes for overloaded versions of read and write methods.

O

## Operation read is overloaded:

- int `read()` read one byte at a time.
- int `read(byte[] buffer)` read a complete buffer.
- int `read(byte[] buffer, int offset, int length)` read a portion of the buffer.

## Operation write is overloaded:

- void `write(int value)` write a single byte.
- void `write(byte[] buffer)` write a complete buffer.
- void `write(byte[] buffer, int offset, int length)` write a portion of the 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 */ }
```

O

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.

O

## Operation read is overloaded:

- int `read()` read one character at a time.
- int `read(char[] buffer)` read a 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 a single character.
- void `write(char[] buffer)` write a complete buffer.
- void `write(char[] buffer, int offset, int length)` write a portion of the 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 */ }
```

O

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 */
}
```

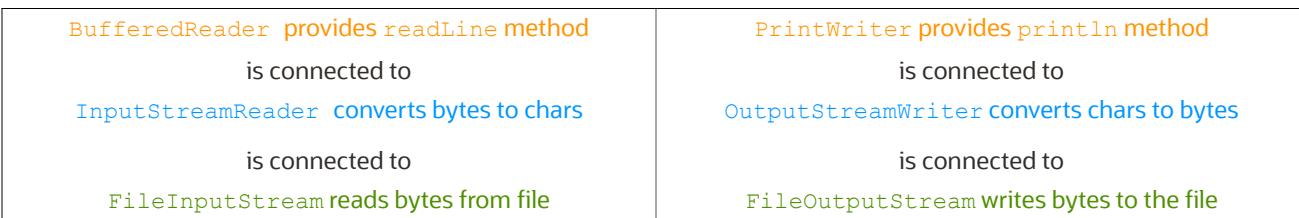
Generally, for any source and destination, data could be transferred in a more automated way:

```
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 */ }
```

O

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 your 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);  
        }  
    }  
}
```

O

It is not advisable to overuse standard input and output, because they represent sequential access to resources and may significantly degrade performance and scalability of the program. For example, only one thread at a time can print to the console blocking all other threads that are also trying to print at the same time.

# 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 the `Reader` object associated with the console
- `writer()` - retrieves the `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

O

Compare characteristics of `PrintStream` and `PrintWriter`:

- Generally, they 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, Charset.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");  
        }  
    }  
}
```

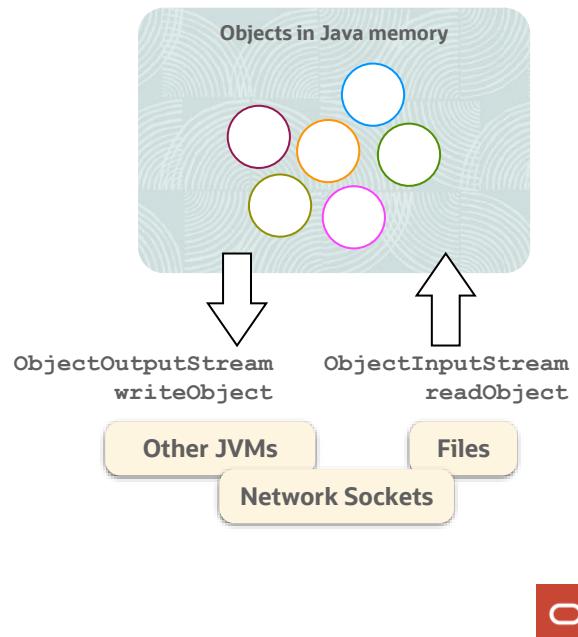
# Understand Serialization

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.



Serialization allows direct read/write operations on memory to extract values and place them into streams or restore values directly to memory from streams.

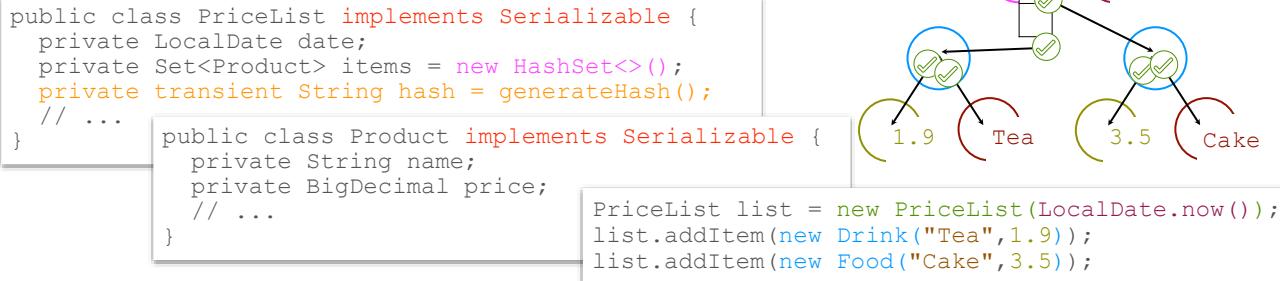
It provides a good mechanism to replicate values across network or perform memory swap.

The reason serialization is not suitable for storing program data for a long term is because 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, because 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 nontransient variable that is of nonserializable type.



- ❖ Keyword `transient` indicates that a field should exist only in memory and is not going to be written into an `ObjectOutputStream`.

O

# 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

O

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) throws 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 Appendix C "Java Security".

O

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

O

You can decide which fields to serialize using the `serialPersistentFields` static array of fields instead of using a `transient` keyword. Only fields included into this array would be considered for the serialization.

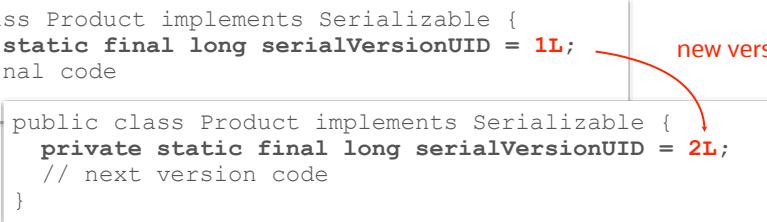
```
public class Product implements Serializable {
    private String name;
    private BigDecimal price;
    // private transient BigDecimal discount;
    private BigDecimal discount;
    private static final ObjectStreamField[] serialPersistentFields = {
        new ObjectStreamField("name", String.class),
        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 run time 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  
}
```



- ✿ 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.

O

Note that persistence API is not technically part of the Java Standard (SE) edition but originates from Java Enterprise Edition (EE). In later versions, Java EE has been rebranded as Jakarta and relevant packages were refactored from `javax.persistence` to be `jakarta.persistence` instead.

# 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:" and so on.
  - 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.

O

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);
  }});
```

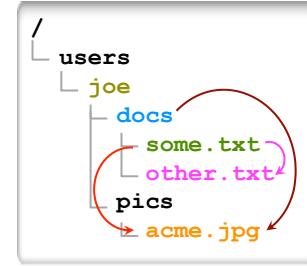
# Constructing Filesystem Paths

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 nonexistent 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
```

O

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, that is, an attempt to guess the directory structure on a computer by using ".../somepath" relative paths.

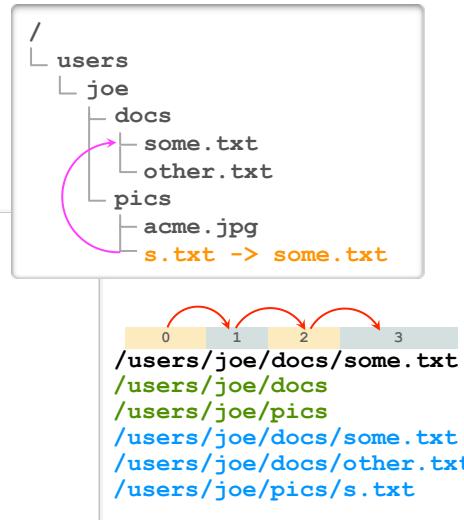
Method `relativize` constructs a relative path between two other path objects.

# 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);
```



O

Both `list` and `walk` operations return `Stream` objects, to which `map`, `peek`, `filter`, `forEach`, and so on 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 the method `walk`:

```
Path startPath = Path.of(".");
int depth = 10;
Files.walk(startPath, depth);
```

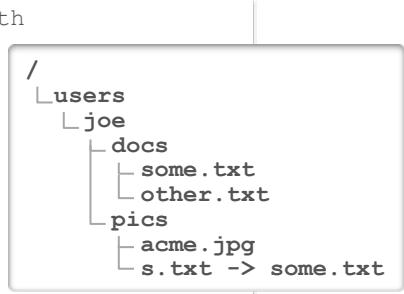
Many operations of the `Files` class can be augmented to follow or not to follow symbolic links:

```
Files.walk(startPath, FileVisitOption.FOLLOW_LINKS);
or
Files.exists(Path.of("acme"), LinkOption.NOFOLLOW_LINKS);
```

# Analyze 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.

O

Depending on a type, filesystems may present different sets of path properties:

```
Path path = Path.of("some.txt");
```

Generic attributes:

```
BasicFileAttributes a1 = Files.readAttributes(path, BasicFileAttributes.class);
```

POSIX attributes:

```
PosixFileAttributes a2 = Files.readAttributes(path, PosixFileAttributes.class);
```

DOS attributes:

```
DosFileAttributes a3 = Files.readAttributes(path, DosFileAttributes.class);
```

Example of setting the 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.

O

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 write
--wx-wx-w 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:

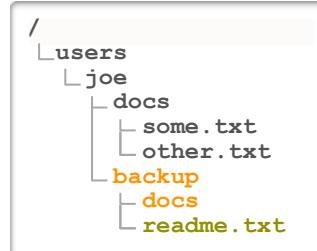
```
Set<PosixFilePermission> perms = Set.of(PosixFilePermission.OWNER_READ,
                                         PosixFilePermission.OWNER_WRITE,
                                         PosixFilePermission.GROUP_READ,
                                         PosixFilePermission.GROUP_WRITE,
                                         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.exists(backup)) {
    Files.createDirectories(backup);
}
Path readme = backup.resolve("../readme.txt").normalize();
Files.createFile(readme);
Files.writeString(readme, "Backup time: " + Instant.now());
Files.lines(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.

O

`createDirectory` operation throws an exception when trying to create a subfolder within a nonexistent 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);
```

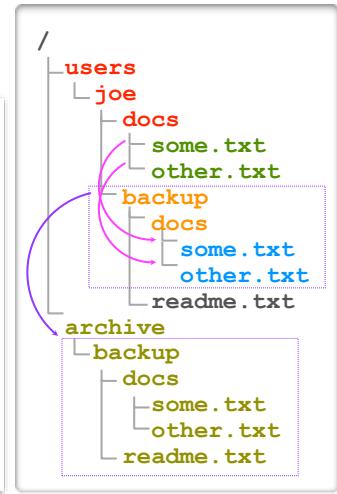
O

# 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.

O

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.

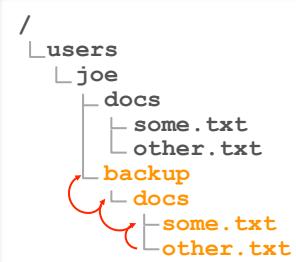
`StandardCopyOption.ATOMIC_MOVE` treats the entire move operations as an atomic action, which will not be allowed to partially succeed or fail.

## Delete Paths

Class `Files` provides operations to delete folders and files:

- `delete` and `deleteIfExists` delete files and folders.
- Attempting to delete a nonempty 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);
        }
    });
});
```



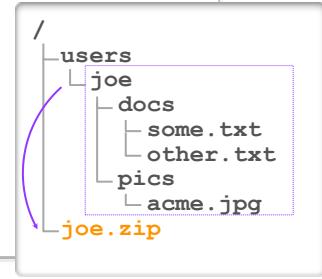
O

# 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.



The example shows extracting content from zip archive:

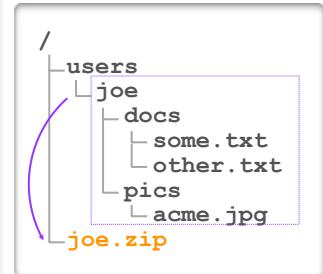
```
Path target = Path.of("extract_to");
Path zip = Path.of("/joe.zip");
try (ZipInputStream in = new ZipInputStream(Files.newInputStream(zip))) {
    ZipEntry e = null;
    while((e = in.getNextEntry()) != null) {
        Path p = Paths.get(target.toString(), e.getName());
        if (e.isDirectory()) {
            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);
}
```



## Notes:

- In the example, all files and folders are copied from `joe` to the `joe.zip` file.
- Zip File System provider is supplied as a separate module `jdk.zipfs`.  
Modules are covered in the Lesson 15 "Modules and Deployment".

O

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 the `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 = 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));
```

## Notes:

- ❖ The example shows an html document downloaded into a file.
- ❖ Use of the `module-info` class is covered in the Lesson 15 “Modules and Deployment.”

O

Java HTTP access API is part of the `java.net.http` module. Development and deployment of modularized Java applications are covered in the lesson titled “Modules and Deployment.”

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:

- IO Fundamentals
- Serialization/Deserialization
- Working with filesystems



## Practices for Lesson 13: Overview

In this practice, you will:

- Write reports about products and reviews to text files
- Bulk-load data from comma-delimited files
- Serialize and deserialize Java objects using temporary files







# Java Concurrency and Multithreading

---

# 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
- Use Virtual Threads
- Ensure thread safety using volatile variables, atomic actions, and locks



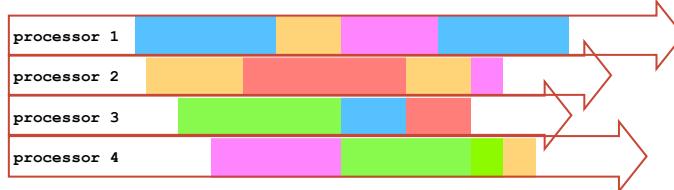
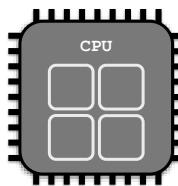
# Java Concurrency Concepts

Java thread is an execution path.

- Thread actions are implemented by using the `run` method of a `Runnable` interface.
- Thread is scheduled to run using the `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 the `main` or `run` method 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
    }
}
```



O

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 a Java program starts, at least one thread is created. This thread executes the `main` method. 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 numOfHardwareThreads = 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 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

```
// 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();
```

❖ 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();
```

O

# Thread Life Cycle



- ❖ 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();
t1.start();
```

O

An attempt to start the same thread object twice will throw a `java.lang.IllegalThreadStateException`.

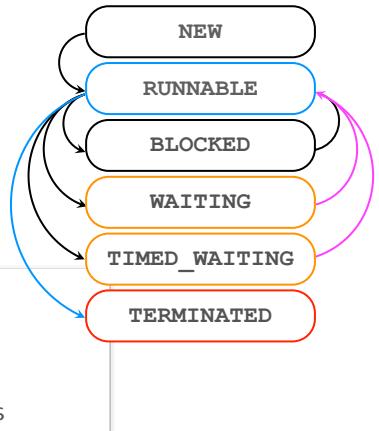
The `getState` method is designed for monitoring the system state, not for synchronization control.

# Interrupt Thread

Logic of a `run` method is in charge of making lifecycle 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, a thread may choose to terminate (get to the end of run method) or it may decide to continue.

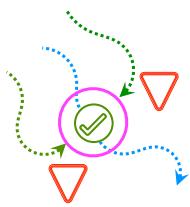
O

If a 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, the thread goes back to the running state and again can make a decision whether to terminate or not.

After the timed wait expires, the thread goes back to the running state. However, it would not happen at that very moment, but rather when the thread scheduler allocates the next variable CPU time slot for this thread.

## Block Thread

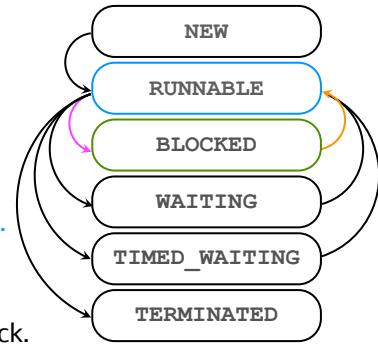
- **Monitor object** helps to coordinate the order of execution of threads.
- Any object or a class can be used as a monitor.
- It allows threads to enter blocked or waiting states.
- It enables mutual exclusion of threads and signaling mechanisms.
- The keyword `synchronized` enforces exclusive access to the block of code.
- A 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();
```

O

The `synchronized` keyword:

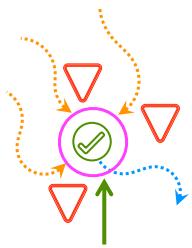
- Enforces exclusive access to an object or a class
- Establishes the order of threads accessing a locked object (monitor object) known as intrinsic lock
- Ensures that a thread completes the sequence of actions within the synchronized block (even if it was interrupted and then resumed) before another thread is allowed to enter this block of code

# Make Thread Wait Until Notified

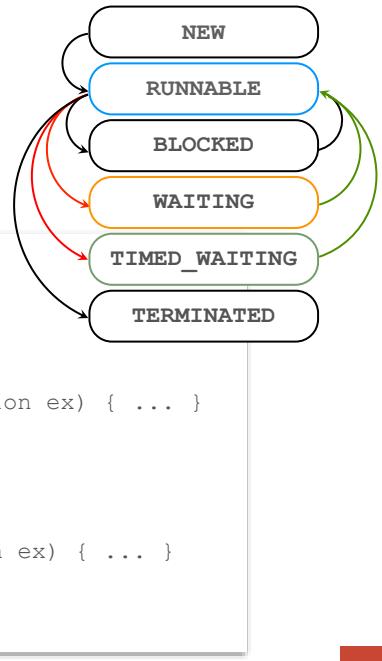
Suspend a thread waiting indefinitely.

- The `wait` method puts a thread into **waiting state** against a **specific monitor**.
- Any number of threads can be waiting against the same monitor.
- The `notify` method **wakes up** one of the waiting threads (stochastic).
- The `notifyAll` method **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();
}
```



O

The `wait` method of the `Object` class is overloaded to allow you to put a thread into a **WAITING** or a **TIMED\_WAITING** state.

The current thread waits until it is awakened by being notified or interrupted:

```
void wait()
```

The current thread waits until it is awakened by being notified or interrupted, or until a certain amount of real time has elapsed:

- `void wait(long timeoutMillis)`
- `void wait(long timeoutMillis, int nanos)`

There is a nuanced difference in the way threads transit from **WAITING** and **TIMED\_WAITING** states back to **RUNNABLE** state, depending on how it was placed into these states.

Earlier in this lesson you have observed a scenario that used the `Thread.sleep(timeout)` method. When timeout is over, or the thread is interrupted, it transitions from **TIMED\_WAITING** to a **RUNNABLE** state, which can be illustrated like this:

`RUNNABLE -> TIMED_WAITING      Thread.sleep(timeout)`

`RUNNABLE <- TIMED_WAITING      sleep method returns or is interrupted`

However, this state transition works slightly differently when thread was placed into a **WAITING** or **TIMED\_WAITING** state using the `wait()` or `wait(timeout)` method.

Apparently, in this scenario, there is a period when the thread goes into a **BLOCKED** state just before it goes back into a **RUNNABLE** state. Usually, this a very brief period and may even go undetected even if you constantly poll for the thread state using the `getState()` method.

This is indeed a very nuanced behavior, which is often ignored, because logically you are actually trying to put your thread into a RUNNABLE state and its brief transition through the BLOCKED state is down to an implementation of the wait/notify methods mechanics, which can be illustrated like this:

RUNNABLE -> WAITING	wait()
RUNNABLE <- BLOCKED <- WAITING	notify(), notifyAll(), interrupt()
RUNNABLE -> TIMED_WAITING	wait(timeout)
RUNNABLE <- BLOCKED <- TIMED_WAITING	timeout expires, notify(),
notifyAll(), interrupt()	

## Common Thread Properties

- Thread could be given a **custom name** using constructor and `set/get name` methods.
- It has a **unique ID**.
- It can be marked as a **daemon** or a **user** (default) thread.
- It may **wait for another thread to terminate**.
- It could be assigned a **priority**.

<code>Thread.MAX_PRIORITY</code>	10	
<code>Thread.MIN_PRIORITY</code>	1	
<code>Thread.NORM_PRIORITY</code>	5 (default)	

- ❖ The JVM exits when the only remaining threads still running are all daemon threads.
- ❖ The `setDaemon` method must be invoked before the thread is started.

❖ 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) { }
```

O

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 can be reused.

# Create Executor Service Objects

The `java.util.concurrent.Executors` class 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 every 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.

O

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. (However, note 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 60 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. (However, note 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 */ } ;  
// Create pool of 2 threads:  
ScheduledExecutorService es = Executors.newScheduledThreadPool(2) ;  
// Schedule a runnable task to be executed in 10 seconds:  
ScheduledFuture result = es.schedule(task, 10, TimeUnit.SECONDS) ;  
// Get delay amount in milliseconds:  
long delay = result.getDelay(TimeUnit.MILLISECONDS) ;  
// Wait for 3 times as long as the delay amount for the task to complete its actions:  
Thread.sleep(delay*3) ;  
// Check if the task has not yet been completed or canceled:  
if (! (result.isDone() || result.isCancelled())) {  
    result.cancel(true); /* request the task to be interrupted */  
}
```

UnconfigurableExecutorService 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>) ;
```

Since Java SE 21, an executor service for Virtual threads that starts a new virtual Thread for each task became available:

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

Virtual threads are covered later in this lesson.

# 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 */
}
```

O

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
        try {
            Thread.sleep(10); // pause
        } catch (InterruptedException ex) {
            System.out.println("Thread "+name+" was interrupted while sleeping");
            return; // terminate thread if it was interrupted while in timed wait
state
        }
    };
}

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 the `public void run()`; method of the `Runnable` interface
  - Launched using `execute` or `submit` method of `ExecutorService`
- Callable objects
  - Implementing the `public <T> call() throws Exception;` method of the `Callable` interface
  - Launched using the `submit` method of `ExecutorService`
  - Returned value is wrapped into the `Future` object, which is returned immediately.
  - The `get` method 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 */ }
```

O

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.

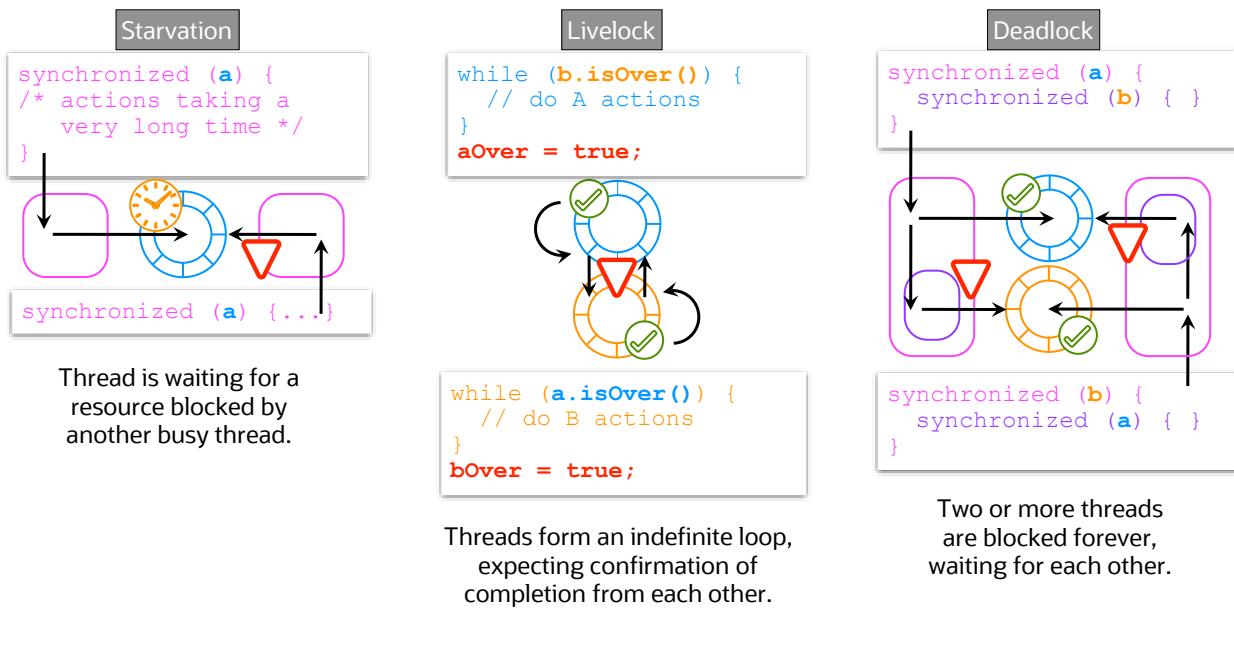
When getting a value from the `Future` result object, the following exception can occur:

```
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 returns the value immediately if it was already produced by the call method; otherwise, it
blocks the invoker until the value is produced. */
}
```

```
String value = result.get(); //try to retrieve the value immediately, and wait if not available
//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

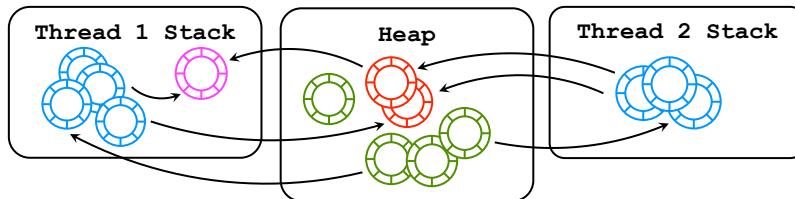


Well-designed software should embrace the stochastic nature of the concurrently executed code. This means that it's best to write code that is not dependent upon the order in which concurrent actions are executed, because such an order is inherently unpredictable. The order of executions should not really 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.



O

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();
}
```

Because of 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.

The `volatile` keyword 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 the `run` method of the `Runnable` interface.

O

The `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();
```

# Nonblocking 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 `+` `-` `*` `%` `++` `--` and so on are not atomic.
- The `java.util.concurrent.atomic` package 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();
```

O

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 nonatomic actions that can be interrupted, resulting in memory inconsistencies in a multithreaded environment.

```
public class Shared {
    public int x;
}

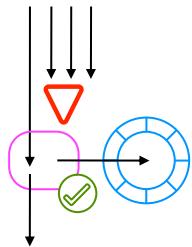
Shared s = new Shared();

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();  
}
```

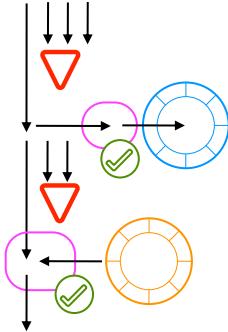
❖ The example ensures that only one thread at a time is allowed to add elements to the list.

O

# Intrinsic Lock Automation

Some Java APIs provide synchronized versions of objects, for example:

- `Collections` class 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.

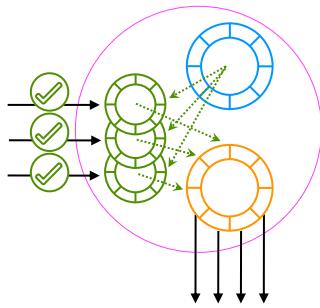
O

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());
}
```

# Nonblocking Concurrency Automation

- The `java.util.concurrent` package 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.

O

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.

- It allows actions to be performed on an object, without interference from other threads.
- It is 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(); }  
    }  
}
```

O

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();  
    }  
}
```

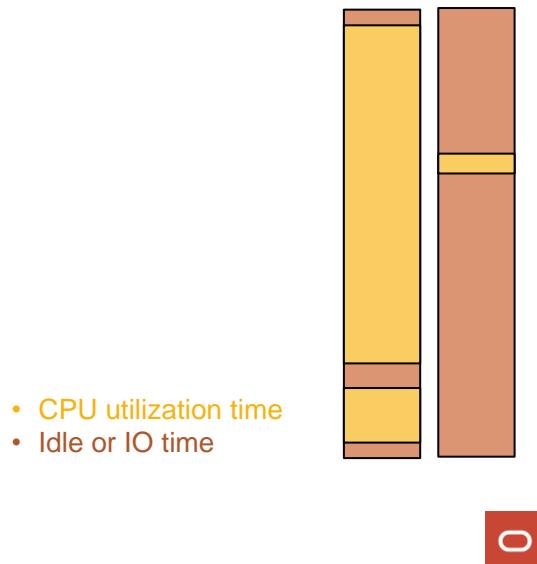
# CPU Versus IO Bound Concurrent Tasks

## CPU Bound Task:

- Computationally intensive
- Uses all available CPU resources
- Example: Processing large volumes of calculations

## IO Bound Task:

- Input/Output intensive
- Uses few CPU resources
- Example: Handling HTTP request and responses



Historically, JVM Thread scheduler mapped each Java Thread to a specific "carrier" OS Thread. JVM had to reserve at least 2 kilobytes of memory for each thread metadata and also at least 1 megabyte of heap memory per such thread. The model is limited by the number of threads host platform can create, and its memory utilization is not efficient because of a minimum amount of memory that must be allocated per thread, regardless if a given thread actually needs this memory or not.

This model does make sense for CPU bound type of concurrency tasks, but not for the IO bound type of task. The reason is that IO bound tasks tend to underutilize CPU resources, so they do not actually need a dedicated "carrier" OS thread for each task. Consider a typical web server application which needs to allocate threads to serve concurrent callers. It could require many threads to serve large numbers of concurrent callers, but it does not actually use a lot of CPU time to perform calculations, because it is mostly waiting for relatively slow input/output operations to complete.

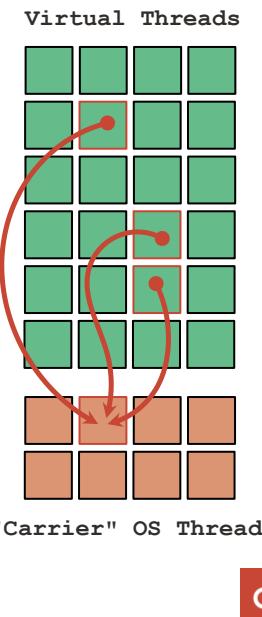
A new Java Virtual Threads API has been proposed to allow Java to create Virtual threads that can transparently and dynamically be mapped to "carrier" OS threads, to allow multiple IO bound concurrent tasks to be performed by a given "carrier" OS thread.

## Virtual Threads API

Virtual threads are lightweight threads:

- Efficiently serve IO bound concurrency tasks
- Dynamically mapped to "carrier" OS threads
- Transparently managed by JVM Scheduler
- Stacks stored as continuations in the heap
- Use less heap memory for metadata

```
try(var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    /* submit tasks for executor to run */
} catch(ExecutionException | InterruptedException e) {
    /* handle any errors */
} /* executor is automatically closed */
```



Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.

The purpose of the Virtual threads API is to allow Java programmers to write code that looks exactly like the one thread per task code, but these Java virtual threads would in fact be mapped (mounted) on to the "carrier" OS threads, which would allow near perfect optimization of the CPU utilization for IO bound type tasks.

The utilization of virtual threads has important implications. They are abundant and cost-effective, making pooling unnecessary. Instead, it is recommended to create a new virtual thread for each task within an application, as they tend to have shallow call stacks and short lifetimes. In contrast, platform threads are heavy and costly, and therefore often need to be pooled. These threads have deep call stacks and are typically shared among multiple tasks.

Overall, virtual threads maintain the traditional thread-per-request approach that is consistent with the Java Platform's design, while making optimal use of hardware resources. Although the use of virtual threads does not require new concepts. By adopting virtual threads, application developers can benefit from improved scalability, and framework designers can offer easy-to-use APIs that are compatible with the platform's design without compromising on scalability.

## Virtual Thread Operations

- `Thread.Builder` is used to create `Thread` and `ThreadFactory` objects.
- `ThreadFactory` is used to create multiple threads with identical properties.
- `Thread.ofVirtual()` creates a virtual thread.
- `Thread.ofPlatform()` creates a platform thread.
- `Thread.startVirtualThread(Runnable)` creates and starts a virtual thread.
- `Thread.isVirtual()` tests whether a thread is a virtual thread.
- `Thread.getAllStackTraces()` returns a map of all platform threads rather than all threads.

```
Runnable task = () -> {/* task instructions */};  
Thread t1 = Thread.ofVirtual().name("acme").unstarted(task);  
t1.start();  
Thread t2 = Thread.startVirtualThread(task);
```

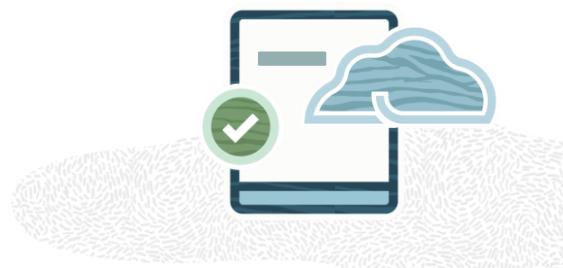
- ❖ All virtual threads are always daemon threads, with normal priority, and use a "VirtualThreads" placeholder as a thread group name name.

O

## Summary

In this lesson, you should have learned how to:

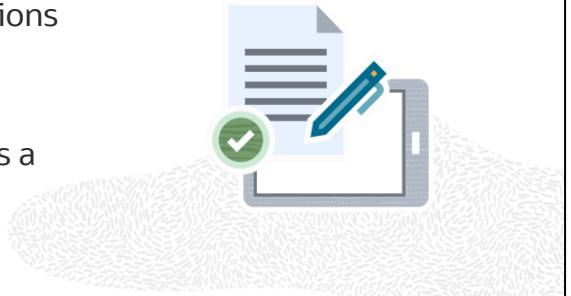
- Describe multithreading
- Manage thread life cycle and execution order
- Automate management and execution of concurrent tasks
- Ensure thread-safety using volatile variables, atomic actions, and locks



## Practices for Lesson 14: Overview

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





## Modules and Deployment

---

Oracle Database 12c includes a new feature called "Modular Database". This allows you to create a database with multiple schemas, each containing its own set of tables, procedures, functions, and other objects. These schemas can be deployed independently, allowing for easier management and maintenance of the database.

# Objectives

After completing this lesson, you should be able to:

- Compare modular and nonmodular Java applications
- Describe Java Modules
- Create module dependencies
- Define and use module services
- Create runtime images
- Deploy and execute modular and nonmodular Java applications



Java Modules are also known as Java Platform Module System (JPMS)

# Compile, Package, and Execute Nonmodular Java Applications

Legacy (nonmodular) Java application deployment and execution:

- Compile Java classes using the `javac` utility:

```
javac -cp /project/classes:other.jar:some.jar  
-d /project/classes  
-sourcepath /project/sources
```

- Package Java class into Java Archive (JAR) using the `jar` utility.
- Optionally, provide JAR descriptor `MANIFEST.MF` file:
  - May contain name of the main application class
  - May set up classpath 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 the Java application using `java` runtime:

```
java -jar shop.jar
```

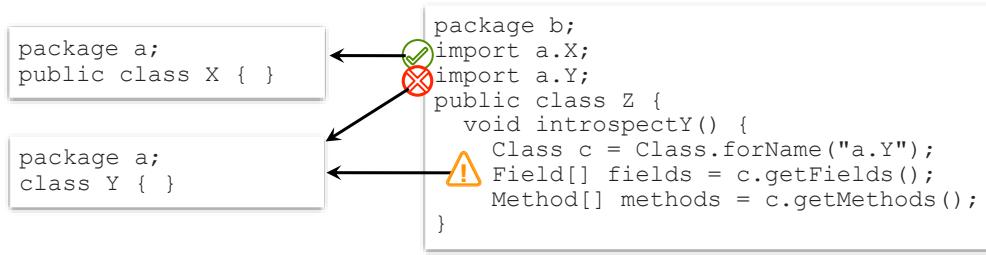
O

The `MANIFEST.MF` file is placed in the `META-INF` folder inside the `JAR` root directory.

## Nonmodular Java Characteristics

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 access modifiers.
- ⚠ Encapsulation can always be bypassed using reflection.
- ⚠ It's impossible to restrict in which exact other packages your code can be used.



O

Before the introduction of module, it was not possible to make some classes 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:** Nonmodular 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.
- The `module-info.class` module descriptor stored in the module's root folder contains:
  - A unique module name (recommended reverse-dns convention `com.yourcompany.whatever`)
  - Required module dependencies (other module that this module depends on)
  - Packages that this module exports, making them available to the other module (All other packages contained within the module are unavailable to the other module.)
  - Permissions to open content of this module to other module using reflection
  - Services this module offers to other module
  - Services this module consumes
- Module does 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 module that require them>;  
    opens <packages of this module to other module via reflection>;  
    uses <services provided by other module>;  
    provides <services to other module> with <service implementations>;  
    version <value>;  
}
```

O

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

The Module system improved security as it can explicitly expose some packages and strongly encapsulate other packages.

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 module. Also, a module can explicitly define what other module it wants to use.

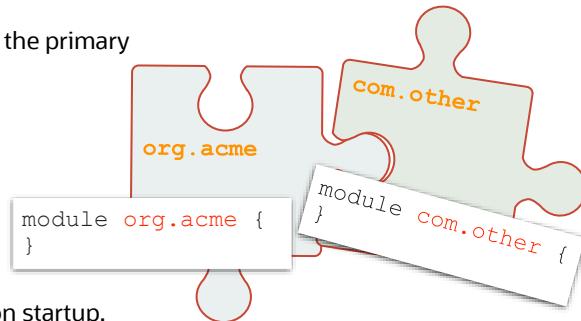
A module helps you to achieve:

- **Reliable configuration:** Modularity provides mechanisms for explicitly declaring dependencies between module 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 module required to support your app.
- **Strong encapsulation:** The packages in a module are accessible to other module 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 module. (This number might change as Java evolves.) You can create custom run times consisting of only module 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 module, 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 a specific module.

# Java Modules

Module is a group of packages.

- Starting from version 9, all JDK APIs were repackaged as module.
- You can describe your own module in the same way by using the `module-info.java` descriptor file, placed into the **module root directory**, and named after the **module name**.
- The 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 a specific module or to all other modules.
- Module declares which other module it uses.
  - It 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 module, are detected at startup.



O

Java Modules are also known as the Java Platform Module System and 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, that is, the `org.acme` folder contains the `module-info.java` file that describes the `org.acme` module.

Alternatively, the `module-info.java` file can be placed in the root of the archive that contains the module.

# Java Module Categories

- Java SE module:
  - Core Java platform for general-purpose APIs
  - Module names start with "java"
  - Examples: `java.base`, `java.se`, `java.logging`
- JDK module:
  - Additional implementation-specific module
  - Module names start with "jdk"
  - Examples: `jdk.httpserver`, `jdk.jconsole`, `jdk.jshell`
- Other module:
  - Create your own and use third-party modules.

```
java --list-module
```

- ❖ Example: List the existing set of modules supplied by the Java SE environment.

O

When listing the existing module, notice that they have a version indicator, for example, "@9" for Java 9.

# Define Module Dependencies

A module defines which other module it needs:

- **requires <module>** directive specifies a normal module dependency.  
Example: The "com.some" module needs access to content provided by the "java.logging" module.
- **requires transitive <module>** directive makes dependent modules available to other module.  
Example: Any module that uses "com.some" will also use "org.acme" without having to declare explicit dependency.
- **requires static <module>** directive indicates module dependency only at compile time.  
Example: The "com.some" module optionally uses the "com.foo" module; this usage is not required at run time.

```
module com.some {  
    requires java.logging;  
    requires transitive org.acme;  
    requires static com.foo;  
}
```

## Notes

- ❖ These instructions accept comma-separated lists of module names.
- ❖ The **requires java.base** directive is implied for all modules; but any other module has to be referenced explicitly.

O

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 module reading your module also read that dependency. This is known as implied readability.

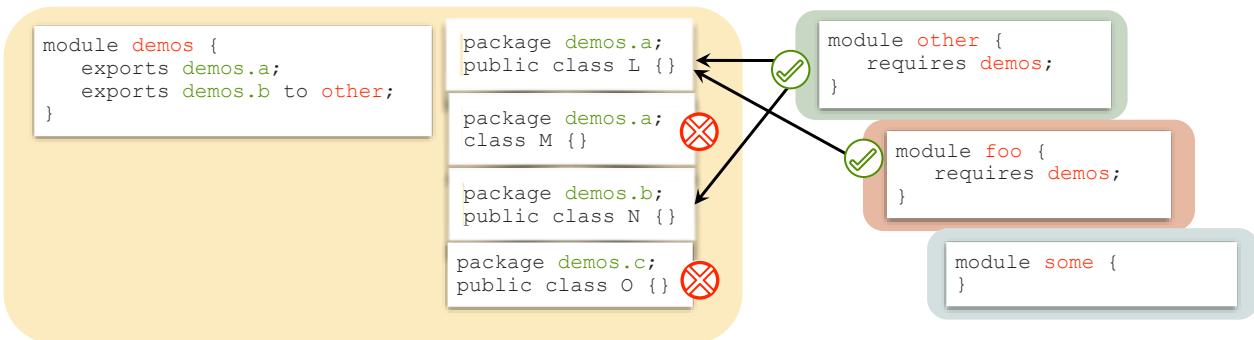
A **requires static** directive indicates that a module is required at compile time, but is optional at run time. This is known as an optional dependency.

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

A module defines dependencies by exporting packages and requiring other module.

- Exporting a package means making all of its public types (and their nested public and protected types) available to other module.
- `exports <packages>` directive specifies packages whose public types should be accessible to all other modules.
- `exports <packages> to <other module>` restricts exported packages to a list of specific modules.



❖ **Note:** These instructions accept comma-separated lists of package and module names.

❖ **Reminder:** A module registers its interest in other module by using the `requires <module>` directive.

O

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 module.

An exports...to directive enables you to specify in a comma-separated list precisely which module or module's code can access the exported package—this is known as a qualified export.

In the example in the slide, the 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 the "other" module.

Module expresses an interest in accessing content from another module with the help of the requires directive.

Class "L" is visible to the "other" and "foo" modules because it is a public class in the package exported to all other module.

Class "M" is not visible to the "other" module because it is not public.

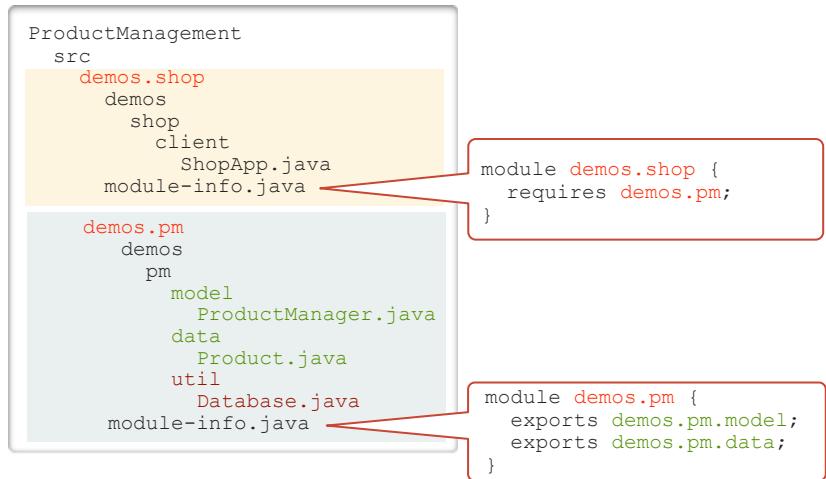
Class "N" is not visible to the "foo" module because its package has only been exposed to the "other" module.

Class "O" is not visible to anyone outside of the module "demos" because this module never exported the 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.

## Module 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 module.
- The `demos.pm.util` package remains private to this module.
- In this example, both modules share a common application folder. However, a module can be created by different developers and is likely to originate from different folders.



✿ **Note:** In this example, the module provides an ability to create different client applications using business logic exposed by the `demos.pm.model` module, yet encapsulating its implementation.

O

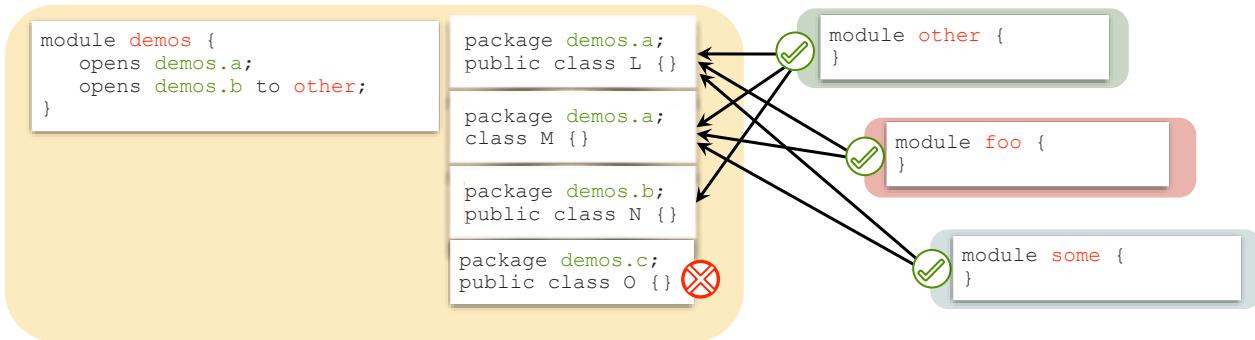
Module names are derived from primary package names of respected module. They do not have to coincide, but they often do to ensure that module names are unique.

The `demos.pm.util` package in this example is a concealed package.

# Open Module Content

A 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 module at run time.
- `opens <packages> to <module>` restricts opened package to a list of specific module.
- Opening a package works similar to export, but also makes all of its nonpublic types available via reflection.
- Module 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.

O

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 module 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.

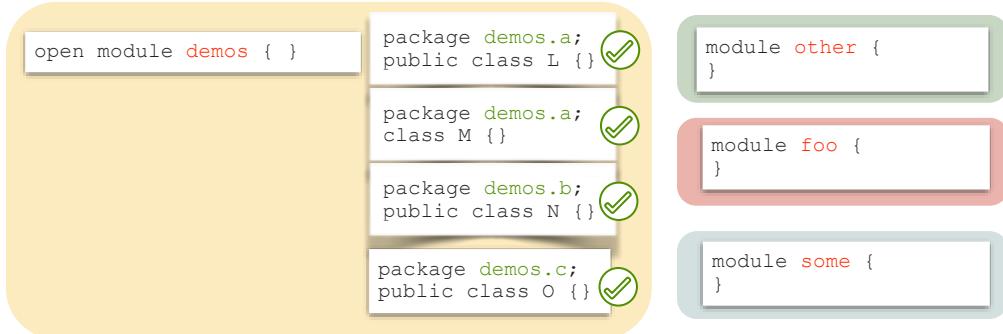
An `opens <packages>` directive allows you to specify a package whose content is accessible to code in other module at run time only; it makes all the types in the specified package (and all of the types' members) accessible via reflection.

An `opens <packages> to <module>` directive indicates that a specific package's public types (and their nested public and protected types) are accessible to code in the listed module 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 module.

# Open an Entire Module

A 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.



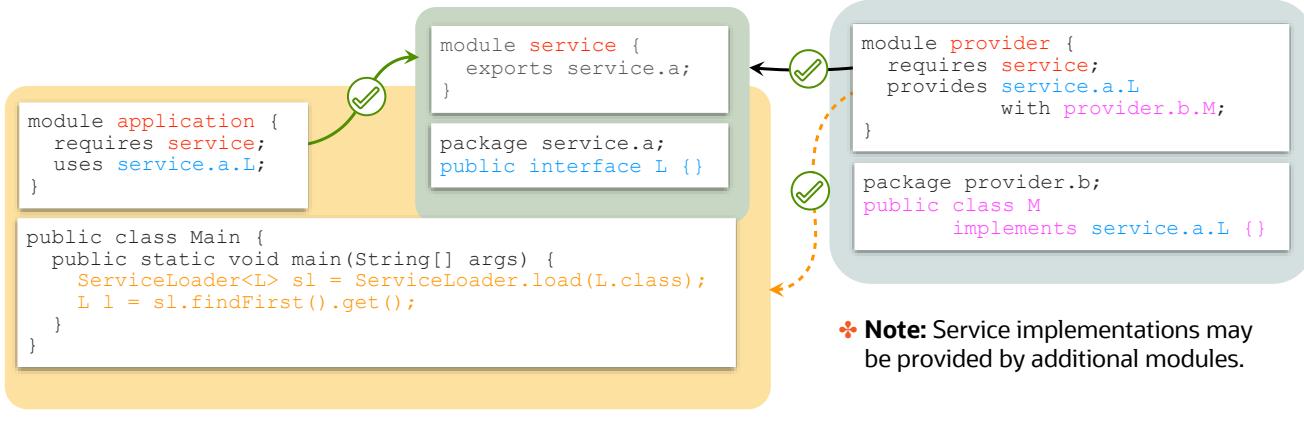
O

All classes in all packages of the module `demos` are accessible at run time to any module via reflection.

# Produce and Consume Services

A module 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.
- The **provides <service interface> with <classes>** directive specifies that the module provides one or more service implementations that can be **dynamically discovered by service consumer**.
- The **uses <service interface>** directive specifies an interface or an abstract class that defines a service that this module would like to consume.



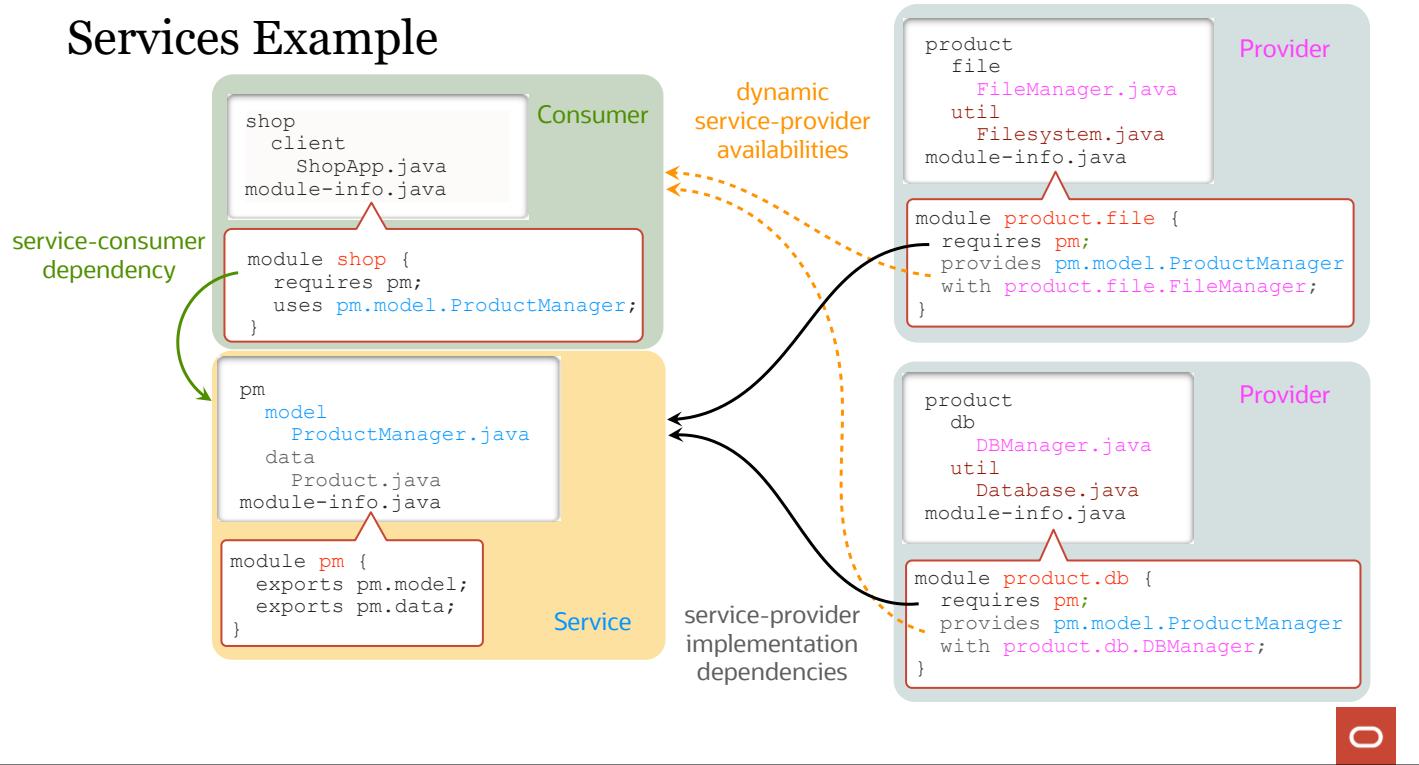
A **uses <service>** directive specifies a service used by this module, making the module a service consumer. A service is an object of a class that implements the interface or extends the abstract class specified in the **uses** directive.

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 the `java.util.ServiceLoader` class:

```
ServiceLoader<L> sl = ServiceLoader.load(L.class);
sl.stream().forEach(s -> s.get().getClass().getName());
```

# Services Example



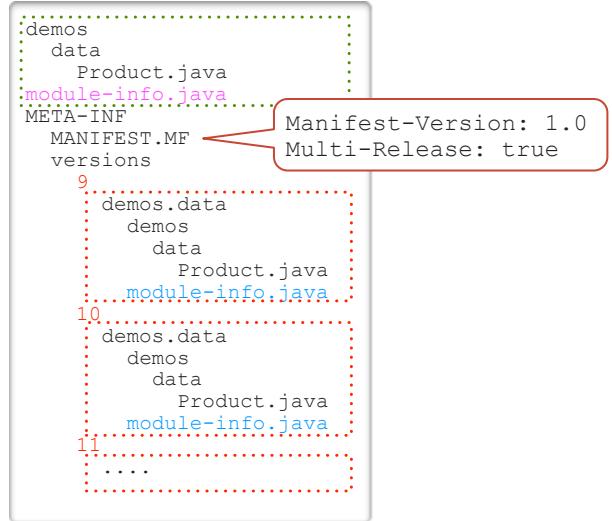
In this example, the `pm` module presents a service with two alternative implementations, provided by `product.db` and `product.file` modules. This service is used by the `shop` module. The content of the `pm` module contains classes and interfaces required by both service consumers and service providers.

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.
- The module root directory may contain either a **default version of the module** or a **nonmodularized 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 nontransitive requires clauses of `java.*` and `jdk.*` module
  - Can have different use clauses



O

The root folder of this archive may contain a nonmodular 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 module, 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 module, you may include explicit version identity to avoid such errors.

# Compile and Package a Module

Compile module:

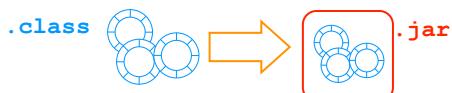
- Specify all of your Java sources from various packages that you want this module to contain.
- Include the packages that are exported by this module to other module and a module-info.
- Reference the other module required for this module to compile.



```
javac --module-path <paths to other module>
      -d <compiled output folder>
      -sourcepath <path to source code>
```

Package module into a JAR file:

- 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> .
```

```
java --module-path <path to compiled module>
      --describe-module <module name>
```

O

The `--create` option instructs the `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 on an implicit inclusion, then the `--describe-module` command would display this as requires `java.base` mandated.

An existing module can be updated using the `--update` option.

# Execute a Modularized Application

## Execute Modular Application Using Module Path

- Classes are located by using the `-p` or `--module-path` option.
- Reference the main class by using the `-m` or `--module` option.
- Nonmodular JARs are treated as Automatic module (described in the next slide).

```
java -p <path to module>
      -m <module name>/<package name>.<main class name> <arguments>
```

## Execute Nonmodular Application (reminder)

- Classes are located by using the `-cp` or `--class-path` option.
- Modular JARs located via classpath are treated as nonmodular for backward compatibility.

```
java -cp <path to jars including modularised jars>
      <package name>.<main class name> <arguments>
```

- ❖ **Note:** In Java 9 and later versions, any classes accessed via `classpath` are assigned to the "unnamed module" as a temporary solution for the period of migration to the modular application structure.

O

### Unnamed module:

- Is created for each class loader for all classes that are loaded via class-path
- Can read all other module, and all module can read content of the unnamed module

## Migrating Legacy Java Applications Using Automatic module

Legacy JAR files, which are placed into the module-path, are treated as Automatic module.

- Such JARs can be referenced as a module by other module by using the `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` by using the `MANIFEST.MF` file placed into the legacy JAR as a temporary measure.

`whatever.jar`

`META-INF/MANIFEST.MF`

```
Manifest-Version: 1.0
Created-By: 11.0.2+9-LTS (Oracle Corporation)
Automatic-Module-Name: demos.data
```

✿ **Note:** To complete migration, create the `module-info` descriptor and repackage legacy JAR as a module.

O

# Create Custom Runtime Image

Use the `jlink` utility to create a custom runtime image:

```
jlink --module-path <paths to compiled module and $JAVA_HOME/jmods folder>
      --add-module <list of module names>
      --bind-services
      --launcher <command name>=<module name>
      --output <name of the runtime image>
```

Structure of the runtime image:

- Is 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
  module
```

Only modules used by the application are included.

## Notes

- ✖ `--module-path` may optionally reference the `$JAVA_HOME/jmods` folder when creating a module for the JDK of another platform.
- ✖ Launcher option creates an alias for running Java with a specific main executable class.

Examine runtime image:

```
<image>/bin/java -version
<image>/bin/java --list-module
```

O

For more information about the `jlink` option, refer to:

<https://docs.oracle.com/en/java/javase/21/docs/specs/man/jlink.html>

When adding module 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 module contains provider implementations for this service:

```
jlink --module-path <paths to compiled module and $JAVA_HOME/jmods folder> --
suggest-providers=<name of the service interface or class>
```

A custom runtime image is fully self-contained. It bundles the application modules with the JVM and everything else it needs to execute your application.

A custom image is platform-specific and is not portable to other platforms.

JLink utility can also be used to create a custom runtime image for a non-modular Java application. You would not get benefits of a modular design, but you'll still be able to create a platform specific application distribution with JDK packaged with it.

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. It 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

The `--bind services` option is used to bind any available service providers. You don't have to use this option if the module that contain required service providers are already included in the `--add-module` list.

## Execute Runtime Image

The runtime image contains all required modules and actual Java Runtime to execute an 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.

O

# Optimize a Custom Runtime Image

Runtime image optimization options:

```
jlink <options that were discussed earlier>
    --limit-module <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-opt <file name>
```

O

- **--limit-module mod [,mod...]**: Limits the universe of observable module to those in the transitive closure of the named module, mod, plus the main module, if any, plus any further module specified in the **--add-module** 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

## Check Dependencies

The `jdeps` command allows:

- Study dependencies by looking at the JAR files.
- Identify which modules required by an application.
- Dependency analysis for both modular and non-modularized applications.

```
jdeps -s labs.client.jar labs.file.jar labs.pm.jar
labs.client -> java.base
labs.client -> java.logging
labs.client -> labs.pm
labs.file -> java.base
labs.file -> java.logging
labs.file -> labs.pm
labs.pm -> java.base
```

O

A brief description of some of the parameters of the `jdeps` utility:

-h -? --help	Print this help message
-s -summary	Print dependency summary only.
-v -verbose	Print all class level dependences
-cp <path>	Specify where to find classes
--module-path <module path>	Specify module path
-m <module-name>	Specify the root module for analysis
--list-deps	Lists the module dependences.

For more option refer to `jdeps` documentation available as part of the JDK documentation.

## Summary

In this lesson, you should have learned how to:

- Compare modular and nonmodular Java applications
- Describe modularization
- Create module dependences
- Define and use module services
- Create runtime images
- Deploy and execute modular and nonmodular Java applications



## Practices for Lesson 15: Overview

In this practice, you will:

- Compare modular and nonmodular deployment formats
- Migrate application to modular Java format
- Restructure application module to achieve more flexible design
- Create and execute application using runtime image



## Annotations

---

# Objectives

After completing this lesson, you should be able to:

- Describe annotations
- Create custom annotations
- Dynamically discover annotations
- Identify frequently used Java annotations



# Annotations: Introduction



- 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 types 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.

O

# Design Annotations

- Annotations can have **elements**.
- Elements can be of the following types:
  - Primitive
  - String
  - Class
  - Enum
  - Annotation
  - An array of any of the above
- An element can have a **default value**, which could be a constant expression.
- An element 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();
}
```

O

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.

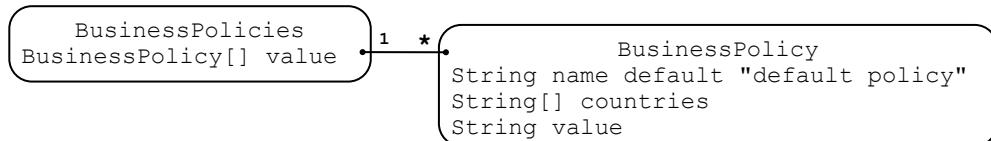
- First, mark your annotation with the meta annotation `@Repeatable`. The container annotation type that the Java compiler generates to store repeating annotations is in parentheses. In this example, the container is `BusinessPolicies`.
- Second, declare the container annotation type. It must have an 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.

- Elements are set as a list of name-value pairs `@Annotation(name=value, arrayName={e1,e2})`.
- When no elements are used, `()` can be omitted.
- The `value` element does not have to be specified by name, when it's the only element that needs to be set.
- If array has only one value, `{}` can be omitted.
- Elements 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 { }
```



## Notes

- Annotation applied to a class is not inherited by its subclasses, unless it is marked as `@Inherited`.
- Annotation `BusinessPolicy` is Repeatable and thus can be applied without explicitly using the `BusinessPolicy` annotation.

O

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 can omit the braces if the array contains only one value.

Compiler rejects code that tries to set annotation element to a `null` value.

See the verbose example of applying annotations that explicitly use all element names, including element "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 elements.

```
// 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 elements:  
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.



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")
```

Values of BusinessPolicy annotation elements:

```
Returns Policy 4 weeks GB  
default policy Ship via Dover-Calais GB FR
```

# Document the Use of Annotations

An 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(){}
}
```

0

# Annotations that Validate Design

Annotation can 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;
    }
}
```

O

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.

`FunctionalInterface` is actually marked as retained at run time. However, practically it is used only to perform a compile-time check.

The `@Override` annotation makes the 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.
- The `since` element indicates a version after which this code should no longer be used.
- The `forRemoval` element is a Boolean indicator:
  - True indicates intent to remove the annotated program element in a future version.
  - False is the default which 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
  - Section of the 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() { ... }
}
```



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 the use of the deprecated element is discouraged, there's no intent to remove it yet. Because "forRemoval=false" is the default value, it is normally omitted in this scenario.

# 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.
- ❖ Nonparameterized (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 pregenerics or deprecated code.

O

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 (for example, `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.

## Varargs and Heap Pollution

- Incorrect use of varargs with generics can lead to **heap pollution**.
- Vararg 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.
- The `@SafeVarargs` annotation suppresses heap-pollution warning when using varargs.
- Ensure that your code does not produce actual heap pollution.
- An 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"));  
}
```

O

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:

- Describe annotations
- Create custom annotations
- Dynamically discover annotations
- Identify frequently used Java annotations



# Java Database Connectivity

---

# Objectives

After completing this lesson, you should be able to:

- Explain the Java Database Connectivity (JDBC) API structure
- Manage database connections
- Execute SQL statements
- Process query results
- Manage transactions
- Discover metadata
- Customize query results processing



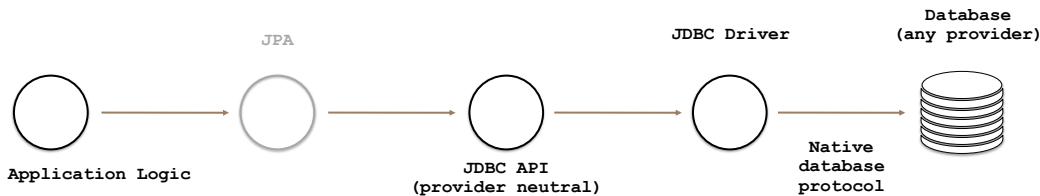
# Java Database Connectivity (JDBC)

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 classpath or module path).
  - Driver is selected using JDBC connection URL:

```
jdbc:<provider>:<driver type>:<connection details>
jdbc:oracle:thin:@<host>:<port>:<database name>
jdbc:derby:<host>:<port>:<database name>
```

*General JDBC url pattern  
Oracle thin driver example  
Java DB (Derby) driver example*



❖ **Note:** Java Persistence API (JPA) is covered in the *Developing Application for the Java EE 7 Platform* course.



Provider-specific JDBC driver implementation libraries must be made available to the application class loader using either classpath 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 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();
}
```

O

If you provide explicit `finally` block, you must close objects in specific order:

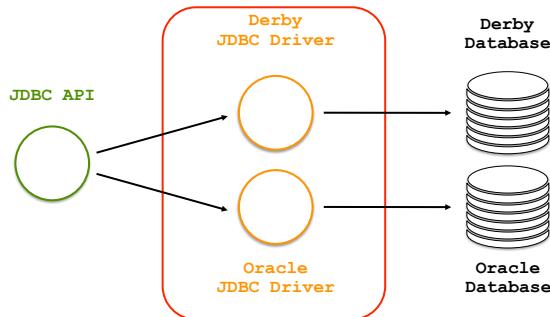
```
Connection c = null;
Statement s = null;
ResultSet rs = null;
try {
    c = DriverManager.getConnection(...);
    s = c.createStatement(...);
    rs = s.executeQuery();
    /* process statement execution results */
} catch (SQLException e) {
    /* handle exceptions */
} finally{
    rs.close(); // first close ResultSet
    s.close(); // then close Statement, PreparedStatement or CallableStatement
    objects
    c.close(); // last close Connection
}
```

# Manage Database Connections

The `java.sql.Connection` interface represents JDBC connection to any database.

- From JDBC 4.0, driver is auto-selected based on a JDBC URL.
- The `getConnection` method 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 pre-JDBC 4.0 drivers.

O

Provider-specific JDBC driver implementation libraries must be made available to the application class loader using either classpath or module path.

Oracle provides different implementations of JDBC driver, such as `thin` (generally recommended driver), `OCI` (platform-specific driver), and `default` or `kprb` (server-side internal database driver).

The `getConnection` method 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 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

The `java.sql.Statement` interface represents all types of **SQL statements**.

All types of statements (`Statement`, `PreparedStatement`, `CallableStatement`) can:

- Execute a query (`select`) operation to get a `ResultSet` object
- Execute 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();
}
```

O

Examples of executing `select`, `insert`, `update`, and `delete` operations using the 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

The `java.sql.PreparedStatement` interface 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.

O

Examples of 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

The `java.sql.CallableStatement` interface 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, String.class); // some.getString(1);
```

❖ **Note:** In/out parameter **position index** starts from 1.

O

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, String.class);
```

# Process Query Results

The `java.sql.ResultSet` interface is used to traverse a set of records returned by the query.

- The `next()` method:
  - 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 by 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.

O

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 can improve the 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 completes successfully.
- The `setAutoCommit` method changes commit mode behavior.
- The `getAutoCommit` method returns current commit mode behavior.

Transaction must end in commit or rollback.

- Use the `commit` method to make all pending transaction changes permanent.
- Use the `rollback` method to discard pending transaction changes.
- If required, use the `setSavepoint` method 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  
}
```

O

Connection is closed implicitly if it was initialized using try-with-resources construct, so it is important to remember to roll back when runtime exceptions occur, but it does not on checked exceptions, because checked exceptions are supposed to be explicitly captured and the corresponding catch block is the place where rollback decisions can be made explicitly.

## Discover Metadata

Metadata objects are available for the 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.

O

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 of 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:** The next two pages explain these options in more detail.

O

CallableStatement can also present a ResultSet object; for example, when a 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 the `next()` method 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 `DatabaseMetaData.supportsResultSetType` method returns true if the specified `ResultSet` type is supported and false otherwise.

O

`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.

The `absolute(int row)` method 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()`.

The `relative(int rows)` method 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 `relative(1)` method is identical to calling the method `next()` and calling the `relative(-1)` method is identical to calling the `previous()` method.

# 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/nonholdable cursors.

Methods of `DatabaseMetaData` object indicate specific database support levels:

- `supportsResultSetConcurrency` returns true if the specified concurrency level is supported.
- `getResultSetHoldability` returns the 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.

O

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:

- Explain the Java Database Connectivity (JDBC) API structure
- Manage database connections
- Execute SQL statements
- Process query results
- Manage transactions
- Discover metadata
- Customize query results processing







# Java Security

---

# Objectives

After completing this lesson, you should be able to:

- Understand security requirements
- Protect code
- Validate values
- Protect sensitive data
- Prevent injections
- Discover and document security issues



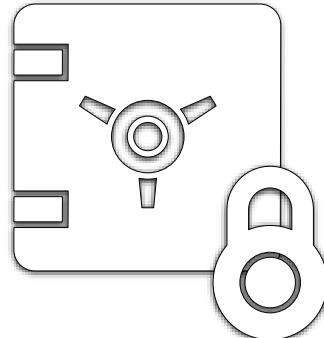
# Security as Nonfunctional Requirement

- The key to successful software development is that all stakeholders develop a clear and uniform understanding of application requirements.
- Software requirements can be broadly classified into two groups:
  - Functional requirements
  - Nonfunctional requirements
- In a problem domain, the focus is on the functional or business requirements. It is recommended that you create a domain model of your functional requirements before you start thinking of the solution domain.
  - In a solution domain, we focus on how to deliver the solution for functional or business requirements.
- Some of the important nonfunctional requirements of an application are:
  - Quality attributes such as security, high availability, scalability, performance, and reliability.
- Designing a secure enterprise application system requires consideration in three inter-related areas:
  - Network services
  - Hosting environment
  - Applications
- The security level of the system is determined by the weakest link of these three components.

O

# 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



O

## 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.

O

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" signed "Jane, John" {
    permission java.net.SocketPermission "localhost:7777", "listen";
    permission java.io.FilePermission "/someFile", "read, write";
};
```

❖ See <https://docs.oracle.com/en/java/javase/21/security/java-se-platform-security-architecture.html> for more information.



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 ...
};
```

To get a certificate from a CA, you must perform the following steps:

- Create a keystore.
  - This will create a public and private key.
  - Use keytool, which comes with Java.
- Generate a certificate signing request.
  - Create a request form using your public key.
- Go to the vendor's website to purchase and submit the request.
- The vendor sends you a certificate.
- Store the certificate in your keystore.
  - Now you can use the certificate to sign your applications.

After you have done all the steps outlined in the slide, you are ready to go. When you sign a .jar, your private key is used. The certificate is included with the .jar so that a user can verify that the file comes from you and has not been tampered with.

### Generating Public and Private Keys with keytool

- keytool
  - It is a key and certificate management utility.
  - Users can create public/private key pairs.
  - Users can create certificates for use in self-authentication.
  - It allows users to cache the public keys (in the form of certificates).

- Sample keytool command:

```
- keytool -genkey -alias signFiles -keystore examplestore
```

```
keytool -genkey -alias signFiles -keystore examplestore
```

What does each of the keytool subparts mean?

- genkey: Generates the keys
- -alias signFiles: The alias used to refer to the keystore entry containing the keys that are generated
- keystore examplestore: The name (and optionally path) of the keystore that you are creating

## Changes in the Security API

Java SE 17 release has marked Security API classes `AccessController` and `SecurityManager` deprecated for removal.

```
SecurityManager sm = System.getSecurityManager();  
AccessController.checkPermission(permission);  
AccessController.doPrivileged(privilegedAction);
```



O

Refer to JEP-411 <https://openjdk.org/jeps/411> to read about the reasons explaining why this API has been deprecated.

# Secure File System and I/O 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 I/O 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.



# Best Practices for Protecting Your Code

Enforce tight encapsulation:

- Use Java Modules 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:

- Nonfinal and nonprivate 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 argument -Xverify:none or -noverify, which disables byte-code verification. This is almost always a very bad idea.



O

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 that all these components in your entire stack are completely safe and bug-free.

# Erroneous Value Guards

Guard against overflowed number space.

- `xxxExact` methods of the `Math` class provide mathematical operations.
- Throw `java.lang.ArithmException` in case of value overflow.

Guard against bad floating point values.

- The `isInfinite` method guards against positive or negative floating point number infinity.
- The `isNaN` method 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.isNaN(untrusted_double);
Optional<Product> o = someDatabase.findProduct(101);
if (o.isPresent()) {
    Product p = o.get();
}
```

O

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 `ArithmException` 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");
}
```

O

## 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/21/docs/specs/security/standard-names.html> for a list of supported standard security algorithm names.

O

# 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 the `PreparedStatement` object with substitution parameters.
  - Or use the basic statement method `enquoteLiteral(param)` to sanitize parameter value.

```
"Tea'; drop table products;"
```



```
s.executeQuery("select id, price from products where name like '"+param+"'");
```

❖ **Note:** Consider using JPA and BeanValidations APIs to automate and safely build database applications.

❖ JPA and BeanValidations APIs are covered in the [Developing Applications for the Java EE 7 Platform](#) course.

O

# 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.

❖ **Note:** This is easier said than done; consider using open-source libraries such as OWASP to automate this process.

Java Script Code



```
<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](#)

[Develop Web Applications with JavaScript, HTML5 and CSS](#)

O

# 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

```
<?xml version="1.0"?>
<!DOCTYPE some [
    <!ENTITY x "&y;">
    <!ENTITY y "&x;">
]>
<some>&x;</some>
```

- 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.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

**Notes:**

- ❖ Consider using JAXB API to map XML documents to Java Classes.
- ❖ JAXB API is covered in the [Developing Applications for the Java EE 7 Platform](#) course.

O

# 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	☒

O

## Summary

In this lesson, you should have learned how to:

- Understand security requirements
- Protect code
- Validate values
- Protect sensitive data
- Prevent injections
- Discover and document security issues



# Advanced Generics

---

# Objectives

After completing this lesson, you should be able to:

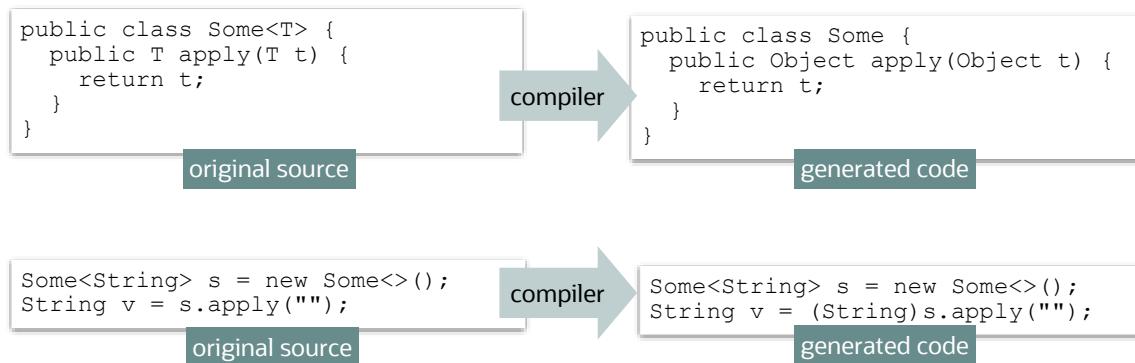
- Describe Generics erasure
- Use Generics with wildcards



# Compiler Erases Information About Generics

Generics information can be erased from the compiled code. Compiler:

- Verifies type-safety of your code before erasing generics
- Adds relevant type-casting operations

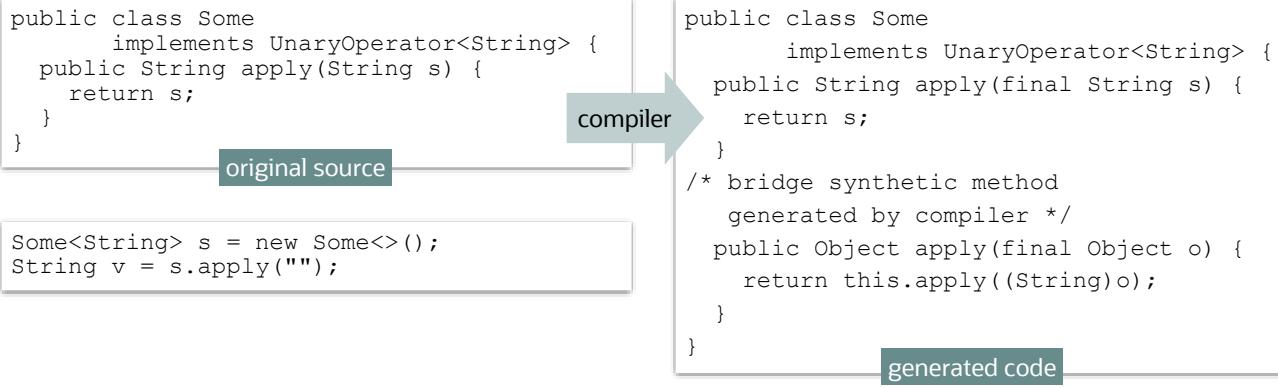


O

# Generic and Raw Type Compatibility

When applying generics to override methods:

- Compiler verifies type-safety of your code before erasing generics
- Compiler adds a synthetic (compiler generated) bridge method
- Bridge method compiles with the nongenerics signature of the method that your code is overriding
- Bridge method invokes your nonsynthetic method, applying type-casting to the generic type
- No type-casting needs to be applied to the code that invokes such an operation



You can use reflection API to dynamically discover the actual code that the compiler would generate:

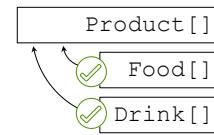
```
Stream.of(Some.class.getDeclaredMethods())
    .forEach(x-> {
        boolean isBridge = x.isBridge();           // true for bridge methods or
        false otherwise
        boolean isSynthetic = x.isSynthetic(); // true for synthetic (compiler-
        generated) methods false otherwise
        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.

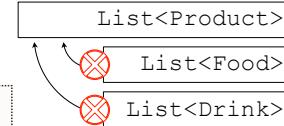
```
Product[] products = new Food[10];  
products[0] = new Drink("Tea"); ✖ java.lang.ArrayStoreException
```



- 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



O

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 operations defined by the Object class can be safely used.
  - Type-check and type-casting must be applied to access any subtype 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 subtype 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.

O

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, and 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 the 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<Product> products = List.of(p1, p2, p3);
List<Food> foods = List.of((Food)p1, (Food)p3);
setProducts(products);
setProductAndSubtypes(products);
setProductAndSubtypes(foods);
```

O

## 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 specific type is eventually a 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 cannot be assigned to its ancestor generic type.

# Lower Bound Wildcard

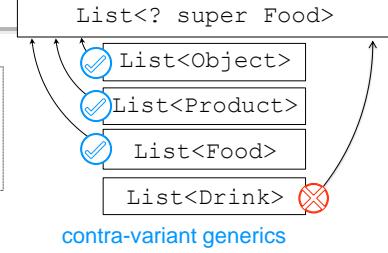
Lower-bounded wildcard `<? super Type>` allows the use of this type and its parents.

- A list of specific type `List<Food>`
  - Is **invariant**: You can assign only 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 the above methods only allow you to add food to such a list.



O

# Collections and Generics Best Practices

Collections and generics wildcard best practices:

- When class hierarchy (super/subtypes) 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(i);
    o.stream().forEach(p->p.prepare());
}
```

O

```
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);
}

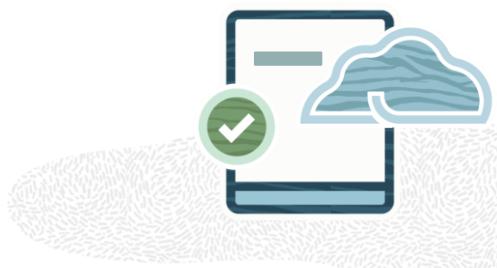
public void processOrder(List<? extends Product> order) {
    // use any Product behaviours
    order.stream().forEach(p->p.prepare());
}

public void addProductAndProcessOrder(List<Product> order, Product product) {
    // use any Product behaviours
    order.add(product);
    order.stream().forEach(p->p.prepare());
}
```

## Summary

In this lesson, you should have learned how to:

- Describe Generics erasure
- Use Generics with wildcards





# Java Applications on Oracle Cloud

---

# Objectives

After completing this lesson, you should be able to:

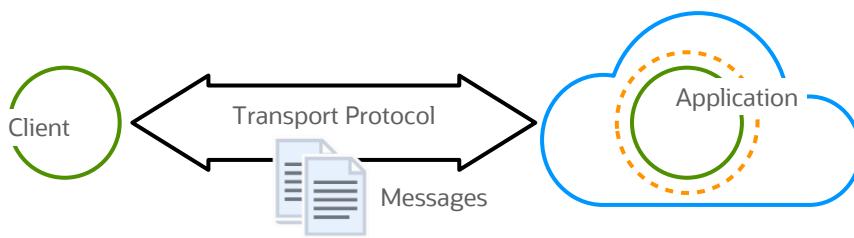
- Describe Cloud Application Design principles
- Deploy Java SE Application to Oracle Cloud



# Cloud Application Requirements

- Cloud Applications are managed and accessed remotely.
- You need to make a lot of design decisions and write implementation code to achieve that.
  - Select transport protocol.
  - Decide on message formats.
  - Enable application to accept remote requests and produce responses.
  - Consider mechanisms to handle concurrent invocations.
  - Manage application life cycle and availability.
  - and so on

Consider using **server-side containers** that automate many of these tasks.



O

So far, throughout this course, your application logic has been accessed locally. You simply executed Java application using command-line Java run time.

Your application accepted your input and output responses using console.

To communicate with a Cloud-based application, you will have to send input and receive output from it via the network. So the application would have to provide network interaction-handling functionalities.

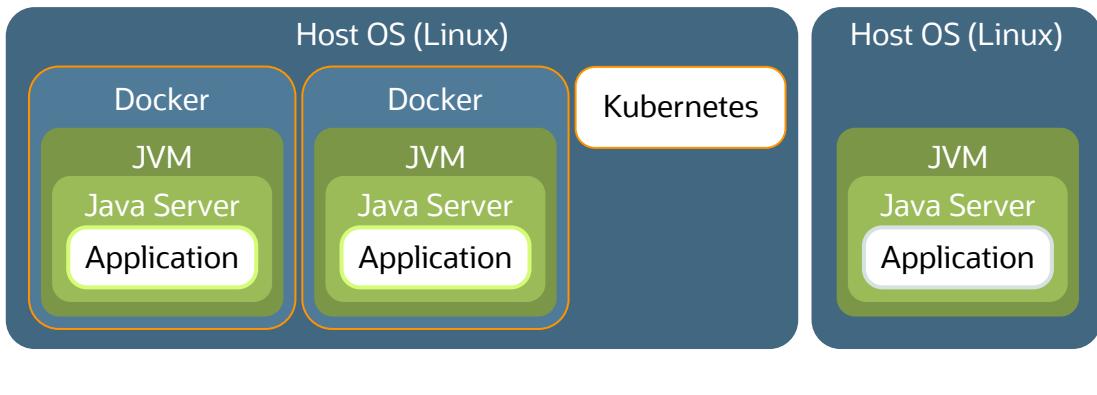
Likewise, when you have invoked application locally, its life cycle was essentially driven by you running the application from the console. Alternatively, cloud applications should be up and running and ready to accept remote requests, before any such request reaches the application and you also have to consider when to stop such cloud application.

A number of server-side containers and frameworks can be used to automate these tasks. The next page discusses available choice of such environments.

It is also worth considering that a separate user interface may have to be created to represent application functionalities within the client tier.

# Cloud Application Runtime Infrastructure

- **Server-side host OS:** Runs as a real or virtual machine
- **Docker:** Provides containers (virtualized environments) within the host OS
- **Kubernetes:** Provides management of deployment, scaling, and controls for Docker containers
- **JVM:** Provides environment to execute Java applications
- **Java Server:** Provides server-side automations for Java applications
- **Application code:** Actual code of your program that carries out business logic



O

Cloud provides a hosting environment for your applications. This environment can simply be a real or a virtual machine that provides the host operating system (OS) within which you can execute your application. However, cloud can provide much more sophisticated infrastructure services. For example, instead of deploying your application into a host OS directly, an application can be packaged as a Docker image, allowing multiple docker images to be hosted by the same host OS. Each Docker image acts as if it is a host of its own, but in fact is really just a virtualization of the host OS where it resides.

Technically, each docker image is a process within the host OS, but it behaves like an independent machine from the perspective of the hosted application. This approach enables better scalability, because you can quickly start and stop additional Docker instances as required by dynamically changing application workloads.

Kubernetes manages life cycle, deployment, and scalability of Docker containers.

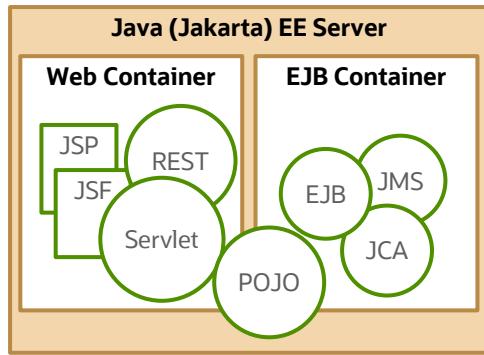
The cloud environment can provide different host operation systems and Linux is probably the most popular choice among them.

Cloud can also provide many other infrastructural services, such as database and file storage, identity and security services, monitoring services, network virtualization and traffic management, service orchestration, notification, workflow and many other features.

# Cloud Java Application Servers

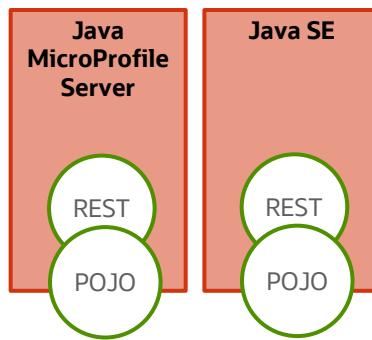
## Java (Jakarta) EE Server (*WebLogic*)

- Hosts Java (Jakarta) EE Applications
- Is a full-functionality application server
- Provides EJB and Web Containers
- Provides Security, Concurrency, Transaction Management, and so on



## Java SE (*Helidon*)

- Provides limited server capabilities
  - Hosts Microservice Applications
- Java MP Server (*Helidon*)**
- Is a lightweight application container
  - Provides a subset of Java EE features
  - Hosts Microservice Applications



O

There are a number of alternative solutions that help automate the development of server-side Java applications. What is common amongst all these solutions is that they provide environments where application can be hosted. These environments are called containers. Each container is designed to host components of specific types.

Jakarta EE is a new name for the Java EE (Enterprise Edition).

For example:

Java Enterprise Edition (EE) containers can host Java Server Pages (JSP), Java Server Faces (JSF), Servlets, REST Services, Enterprise Java Beans (EJB), Java Message Services (JMS), Java Connector Architecture (JCA) Adapters.

Java MicroProfile (MP) container can host REST Services.

Some products such as Helidon can provide REST Service capabilities for essentially Java SE applications without actually provisioning any container-used automation.

Java classes that you have been developing so far throughout this course are classified as Plain Old Java Objects (POJOs), which means that they are simple classes which are not associated with any special type of life cycle and does not have to be hosted in any particular type of container. However, they can be used in containers such as Java EE, MP and so on, but they do not have any special relation to such environments.

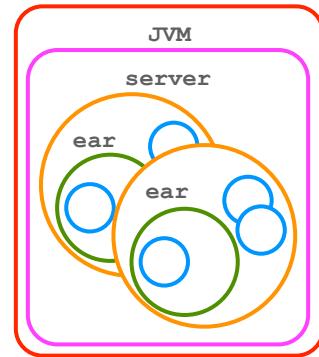
This lesson talks about the use of Helidon SE to provide a cloud-based deployment for a Java SE application.

This course does not cover EE or MP containers. You can learn more about container-based Java application development from "Developing Application for the Java EE 7 Platform" and "Develop Web Services with Java" courses.

# Package and Deploy Cloud Application

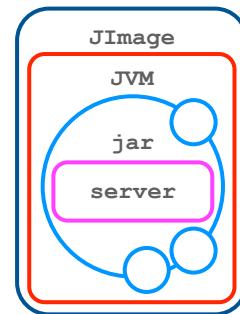
## Java (Jakarta) EE Applications

- Packaged as **Enterprise Archives**, **Web Archives**, and **Java Archives**
- Deployed to Java EE **Server**, which runs in its own **JVM**
- Can deploy multiple applications to the same Java EE Server



## Java SE/MP Applications

- Packaged as **Java Archives**
- Packaged as Java Images using Jlink
- Server** launched from the application code
- Each application runs in its own **JVM**.



❖ **Note:** You can automate production and management of application deployments using an open source tool called Maven.

O

An Enterprise Archive (EAR) is a zip archive that acts as a wrapper for all modules of the Java EE Application. Such modules include Web Archive (WAR) files and Java Archive (JAR) files. These are also zip files. The purpose of Web Modules presented by the WAR file is to contain all web components of the application, such as Servlets, Java Server Pages, Java Server Faces, Web Services, and so on. JAR files can contain libraries, classes that are responsible for the business logic implementation, such as Enterprise Java Beans, persistence management and so on. JAR modules can be included inside other modules, such as EAR and WAR, or even in other JARs.

You can learn more about Java EE Application development and design from the *Developing Applications for the Java EE 7 Platform* course.

Java SE or MP solution to the server-side deployment is to embed actual server directly inside the application. Thus it becomes part of your application deployment and any libraries that this server would require should be included into the deployment class-path or module-path.

If you are deploying Java SE 9 or later application, you can also benefit from packaging your application JARs into one deployment unit together with the actual Java Virtual Machine (JVM) as a JImage.

Apache Maven is an open source tool that is used to manage projects and produce application deployments. It describes application structure and dependencies using Project Object Model (POM) files and uses these files to produce actual deployment artefacts such as JAR, WAR, and EAR files. You can find more information about Maven in the *Develop Web Services and Microservices with Java* course.

# Optimise deployment with GraalVM

GraalVM is an alternative implementation of the JVM with extra compiler optimizations:

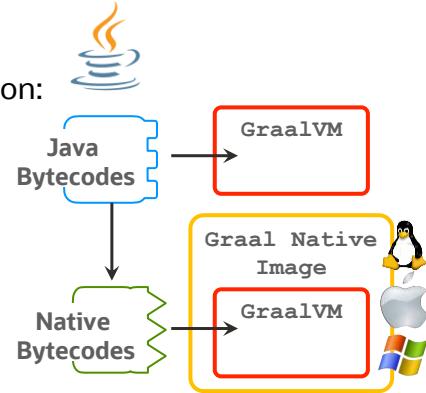
- Increase the size of the stack in proportion to heap.
- Try to spend less CPU time on Garbage Collection.
- Try to improve overall performance.

Graal can produce a runtime image with native code compilation:

- Compile Java bytecode to native code.
- Perform static code analysis and remove unused code.
- Pre-initialize application memory with a heap snapshot.

GraalVM provides a multi-lingual engine:

- Execute non-Java code in the same VM as Java.



**Note:** GraalVM best works with frameworks that implement native image support and Java version 17 or later.



Graal is an open source product that can be used to optimize and improve application performance. There are two main use-cases for the Graal.

- Use GraalVM as an alternative to the Java VM.

It is possible to use GraalVM as an alternative implementation of the Java Virtual Machine. The advantages of using GraalVM is that it provides sophisticated memory optimisations achieved through partial escape analysis that track the scope in which object references are visible and allocate data in the stack rather than heap if that is appropriate based on the object visibility. This increased use of stack can save CPU time when accessing objects and also reduce time needed for garbage collections.

- Produce native runtime image.

The main difference between Java runtime image (created via Jlink) and a Graal runtime image, is that Java runtime image contains a platform specific Java Virtual Machine, which executes normal Java bytecodes. However, Graal runtime image actually executes platform specific native code that is pre-created from Java bytecodes. This of course means that extra efforts are required to compile and assemble this code, but it may yield performance improvements, since application would not required Java bytecode to native code compilation at runtime. That said, it is still perhaps a good idea to first assemble a Java runtime image and ensure that works as intended before trying to optimise it using Graal.

To reduce application deployment footprint Graal removes code that it thinks is not used from the native runtime image. The decision which code to remove is the best guess that is taken as a result of static code analysis of the application code. Note, this is not a perfect process as it may not always detect that an application is actually using some libraries if they are used in a way that leaves them undetectable via the static code analysis, for example when application is using reflection. In which case such libraries should be explicitly listed as required as part of the configuration used in the Graal native image creation process.

Graal runtime image can pre-initialize application memory and create a heap memory snapshot, which includes pre-loading classes, executing static initializers and even pre-loading objects. This feature allows quicker application startup and initialization.

Many popular development frameworks such as Micronaut, Spring Boot, Helidon and Quarks provide out of the box support for the GraalVM. It is also possible to tailor GraalVM configuration to support other frameworks and libraries.

More information about Graal can be found here:

Documentation:

<https://www.graalvm.org/latest/docs/>

Get Started with Oracle GraalVM on Oracle Linux in OCI:

<https://luna.oracle.com/lab/3bodcf97-22d0-489b-a049-5d269199fa00>

# HTTP Protocol Basics

- HTTP Request is dispatched using **methods**, such as, GET (default), POST, PUT, DELETE, and so on.
- HTTP Response contains the **status code**, which indicates the nature of the response, such as 200 OK, 404 Not Found, and so on.
- Requests and Responses contain **headers** that describe metadata, such as Content Type, Length, Character Encoding, Date and Time, and so on.
- Requests and Responses may contain **body**, which transmits actual data.



**Note:** HTTP-based REST is a popular protocol choice for many Cloud-based applications.

O

Standard HTTP methods and their intended uses:

- OPTIONS: Describes the communication options for the target resource
- GET: Intended to retrieve data
- HEAD: Intended to return same headers as GET, but does not contain response body
- POST: Intended to accept request body to submit to the server, typically as a "create" action
- PUT: Intended to accept request body to submit to the server, typically as an "update" action
- DELETE: Intended to perform "delete" actions on server resources
- TRACE: Echoes the request body back
- CONNECT: Establishes a tunnel to the server
- PATCH: Applies partial modification to the server

HTTP Response code groups:

- 1xx codes: Informational
- 2xx codes: Successful Responses
- 3xx codes: Redirection
- 4xx codes: Client Errors
- 5xx codes: Server Errors

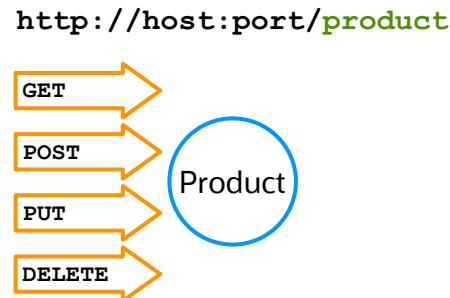
The body of an HTTP Request or Response may not necessarily be present; it may depend upon which HTTP method was used, or a status of the response.

HTTP is one of the most commonly used Internet protocols, utilized by many cloud-based applications and is a foundation for the REST protocol.

REST is basically a way of accessing application logic using HTTP URLs and interpreting HTTP methods as application actions. In fact, it is not really a standard, but rather a convention of how such HTTP methods can be interpreted as application functions. No requirements are imposed on REST communication data format, which enables more flexible design. However, absence of data format restrictions can also be a design challenge, making it difficult to consistently implement an application that consumes different services. Most common use-case for REST Services is to use them to implement back-end application functionalities to be consumed by the front-end user interface part of the application. Such front-end side is often implemented using JavaScript and thus the most common choice for the REST Service data format is JSON (JavaScript Object Notation) REST Communication Model.

# REST Service Conventions and Resources

- A REST service application represents one or more **resources**.
- Each resource represents a business entity and is **mapped to its own URL**.
- Each resource defines a number of operations mapped to HTTP methods.
- Typical conventional use of **HTTP methods**:
  - GET receives (queries) a collection of elements or a specific element identified by client.
  - POST creates a new element.
  - PUT updates an existing element.
  - DELETE removes an element.
- Other operations such as HEAD and OPTIONS can be used to retrieve metadata.
- REST resources can be represented using different message formats, such as JSON, Plain Text, and XML.



O

REST is centered around an abstraction known as a "resource." Any named piece of information can be a resource. A resource is identified by a uniform resource identifier (URI).

Clients request and submit representations of resources. REST Clients are often implemented using JavaScript in Browser or Mobile Applications.

There can be many different representations available to represent the same resource.

A representation consists of data and metadata. Metadata often takes the form of HTTP headers. Resources are interconnected by hyperlinks.

A RESTful web service is designed by identifying the resources. This is similar to Object Oriented Analysis and Design where you identify nouns in use-cases.

Unlike SOAP Services, there is no standard for REST Services. REST is considered to be a style of coding rather than an actual protocol.

Message format is not prescribed by REST Services, it could be from the resource representation. However, it is typical for REST Services to use Java Script Object Notation (JSON), because Java Script programs are often used as REST Service clients.

## Configure and Launch REST Service Application Using Helidon SE

- Define **handlers** to respond to HTTP calls and manage access to a resource.
- Describe **routing rules** that associate handlers with HTTP **URLs** and **methods**.
- Describe WebServer **configuration**.
- Start Helidon WebServer.

```
Handler productLookup = (ServerRequest req, ServerResponse res) -> {
    int id = Integer.parseInt(req.path().param("id"));
    String result = pm.findProduct(id).toString();
    res.status(Http.Status.OK_200);
    res.headers().put("Content-Type", "text/plain; charset=UTF-8");
    res.send(result);
};

Routing routing = Routing.builder()
    .get("/product/{id}", productLookup)
    .post(...)

ServerConfiguration config = ServerConfiguration.builder()
    .bindAddress(InetAddress.getLocalHost())
    .port(8080).build();

WebServer ws = WebServer.create(config, routing);
ws.start();
```

O

Helidon server greatly automates application development by taking care of aspects of networking I/O, concurrency, and many other "plumbing" tasks, which allows a developer to focus more on the application's business logic.

Helidon WebServer uses routing rules that you have described to trigger appropriate handling operation for each incoming HTTP request.

Server configuration typically describes IP address, host name, and listener port information.

Helidon SE provides some degree of automation, whereas Helidon MP, or a Java EE server, would provide even higher levels of automation and application management capabilities. This course's aim is to concentrate on Java SE. To learn more about automations that exist beyond Java SE, refer to *Develop Applications for the Java EE 7 Platform* and *Develop WebServices with Java* courses.

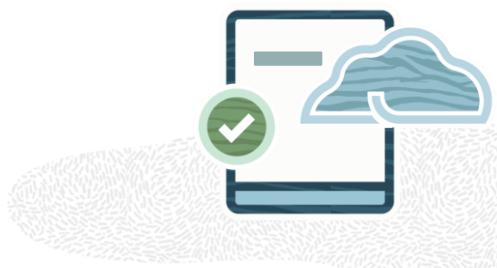
The `io.helidon.webserver.Handler` interface is a variant of the `java.util.function.BiConsumer` interface designed to handle server requests and responses.

Handlers are associated with different HTTP methods and URLs using the `io.helidon.webserver.Routing` object.

## Summary

In this lesson, you should have learned how to:

- Describe Cloud Application Design principles
- Deploy Java SE Application to Oracle Cloud





## Miscellaneous Java Topics

---

## Objectives

- Describe Builder and Singleton Design Patterns
- Describe Regular Expressions
- Explain Java IO, File Watch Service



# Builder Design Pattern

```
public class Product {  
    private String name;  
    private double price;  
    private Product(String name, double price) {  
        this.name = name;  
        this.price = price;  
    }  
    public String getName() { return name; }  
    public String getPrice() { return price; }  
    public static class Builder() {  
        private String name;  
        private double price;  
        public Builder() {}  
        public Product.Builder name(String value) {  
            this.name = value;  
            return this;  
        }  
        public Product.Builder price(double value) {  
            this.price = value;  
            return this;  
        }  
        public Product build() { return new Product(this.name, this.price); }  
    }  
}
```

```
Product p = new Product.Builder()  
    .name("Tea")  
    .price(1.99)  
    .build();
```

- Builder object holds intermediate state.
  - Uses method chaining that return "this" builder object
  - Allows objects to be created in phases
- Eventual object can be immutable.

O

## Singleton Design Pattern

- Uses static method to return an object instance, much like a factory pattern
- Unlike factory, always returns the same object instance

```
public class Application {  
    private static final Application theApp = new Application();  
    private Application() { }  
    public static Application get() {  
        return theApp;  
    }  
}
```

```
Application app1 = Application.get();  
Application app2 = Application.get();  
boolean alwaysTrue = app1 == app2;
```

O

# Java Regular Expression API

- Java `java.util.regex` package provides regular expression handling API.
- `Pattern` describes a regular expression pattern.
- `Matcher` is an engine that interprets a `Pattern` by applying it to input text.
  - The `results` method returns a stream of matched results.
  - The `find` method indicates the presence of a match and fetches it.
  - The `group` method returns last match value.
  - The `replaceAll` method replaces matched elements.

```
String text = "Find delimited elements: -tea-, -coffee- and -cookie-";
Pattern pattern = Pattern.compile("-(.*?)-");
Matcher matcher = pattern.matcher(text);
if (matcher.find()) {
    System.out.println(matcher.group());
}
matcher.results().map(v->v.group()).forEach(v->System.out.println(v));
String result = matcher.replaceAll(v-><" +v.group(1)+">");
System.out.println(result);
```

O

In this example, a pattern expression is looking for any text located between the dash - symbols.

An invocation of the `group()` or `group(0)` method will return matched values: "-tea-" "-coffee-" and "-cookie-".

An invocation of the `group(1)` method will return matched values: "tea" "coffee" and "cookie".

The `replaceAll` operation produces the text: "Find delimited text elements: <tea>, <coffee> and <cookie>"

# Regular Expressions: Character Classes

Character classes match:

- **abc** text exactly matching abc character sequence
- **[abc]** text containing 'a' or 'b' or 'c'
- **[^abc]** text not containing 'a' or 'b' or 'c'
- **[a-c]** a range of characters 'a' or 'b' or 'c'
- **.** (dot) any character
- **\d** any digit [0-9]
- **\D** any non-digit [^0-9]
- **\w** a word character [a-zA-Z0-9\_]
- **\W** not a word character [^a-zA-Z0-9\_]
- **\s** spaces [ \r\t\n\f\0XB]
- **\S** (non-spaces) [^ \r\t\n\f\0XB]
- **\t** a tab character
- **\n** a new-line
- **\r** a carriage return

and others

```
String text = "Find this fine number 2";  
Pattern.compile("th.s");  
Pattern.compile("th[ia]s");  
Pattern.compile("th[^eo]s");  
Pattern.compile("\s\d");  
Pattern.compile("f", Pattern.CASE_INSENSITIVE);
```

O

The concept of a change from one line of text to the next can be expressed in different ways:

**\r** is known as CR (Carriage Return)

**\f** is known as LF (Line Feed)

**\n** is known as a New Line

## Regular Expressions: Quantifiers

Quantifiers describe the number of times a character or a group of characters should appear.

- \* repeats zero or more times.
- + repeats one or more times.
- ? appears once or not at all.
- {n} appears exactly n times.
- {m,n} appears from m to n times.
- {m,} appears m or more times.
- (abc) {n} group of characters repeats n times.

```
String text = "Find this or that, but not these no-no";  
Pattern.compile("th[^e].{0,1}");  
Pattern.compile("th[ias|t].");  
Pattern.compile("(no)*");
```

O

## Regular Expressions: Boundaries

Boundary characters can be used to match different parts of a line.

- `^` matches the start of a line
- `$` matches the end of a line
- `\b` matches the start or the end of a word
- `\B` does not match the start or end of a word

```
String text = "look but, look look";
Pattern.compile("look$");
Pattern.compile("^look");
Pattern.compile("\\bo");
```

O

# File IO Watch Service

Watch Service allows detection of created, modified, and deleted paths within a given folder.

- Obtain `WatchService` object.
- Register `WatchService` with a specific folder opt observe `create, update` and `delete` events.
- `WatchKey` object identifies a path that was affected by the change.
- Acquire event using:
  - The `take()` method that blocks and waits for the next `WatchKey` event
  - The `poll()` method that returns null if no `WatchKey` is present
- Acquire events and identify a `kind` and a `context` (affected path) from each `WatchKey` event
- `WatchKey`, must be `reset`; otherwise, all subsequent events are ignored.

```
Path p = Path.of(someFolder);
try (WatchService ws = FileSystems.getDefault().newWatchService()) {
    WatchKey wk = p.register(ws, ENTRY_CREATE, ENTRY_MODIFY, ENTRY_DELETE);
    while ((wk = ws.take()) != null) {
        wk.pollEvents().stream().forEach(e -> e.kind()+" "+e.context());
        wk.reset();
    }
} catch (Exception e) { e.printStackTrace(); }
```

O

The registration of a `WatchService` returns a `WatchKey` object, which corresponds to a path that this `WatchService` will be tracking. This could be a file or a folder. For example, you may want to set up `WatchService` to check only when a certain file is updated, or to detect any changes of files and folders under a given path.

The `take()` method blocks until an event such as `create, update, or delete` is detected for the `Path` that was registered with the `WatchService`. Once such an event is detected, the `take` method returns a `WatchKey` for the corresponding path, that is, specific file or folder that has been created, updated or deleted.

The `poll()` method is overloaded to provide a block till timeout capability.

The new `WatchKey` that is detected using either `take()` or `poll()` method indicates that there is at least one event that has occurred for a given `WatchService`. You can use the `pollEvents()` method to find out any details of this and possible other events that may have occurred at the same time.

`Event kind()` method indicates a type of change (create, update or delete) that has occurred. `Event context()` method indicates for which specific path this event has been fired.

```
import java.io.*;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;

public class FileWatchTest {
    static class Watcher implements Runnable {
        public void run() {
            Path p = Path.of(".");
            System.out.println(p.toAbsolutePath());
            try (WatchService ws = FileSystems.getDefault().newWatchService()) {
                WatchKey wk = p.register(ws, ENTRY_CREATE, ENTRY_MODIFY,
                ENTRY_DELETE);
                while ((wk = ws.take()) != null &&
                !Thread.currentThread().isInterrupted()) {
                    wk.pollEvents().stream().forEach(e ->
System.out.println(e.kind()+" "+e.context()));
                    wk.reset();
                }
            } catch (IOException | InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        Watcher w = new Watcher();
        Thread t = new Thread(w);
        t.start();
        Console console = System.console();
        while(true) {
            if (console.readLine().equals("exit")) {
                t.interrupt();
                return;
            }
        }
    }
}
```

## Summary

- Describe Builder and Singleton Design Patterns
- Describe Regular Expressions
- Explain Java IO, File Watch Service



