



Integrated Cloud Applications & Platform Services



Java SE: Programming I

Student Guide | Volume II

D102470GC20

Edition 2.0 | January 2019 | D105823

Learn more from Oracle University at education.oracle.com

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

Disclaimer

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

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

Restricted Rights Notice

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

U.S. GOVERNMENT RIGHTS

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

Trademark Notice

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

Contents

1 Introduction

- Audience 1-2
- Introductions 1-3
- Course Objectives 1-4
- Schedule 1-5
- Lesson Format 1-8
- Course Environment 1-9
- How Do You Learn More After the Course? 1-10
- Additional Resources 1-11
- Summary 1-13

2 What Is a Java Program?

- Objectives 2-2
- Topics 2-3
 - Purpose of a Computer Program 2-4
 - Translating High-Level Code to Machine Code 2-5
 - Linked to Platform-Specific Libraries 2-6
 - Platform-Dependent Programs 2-7
 - Topics 2-8
 - Key Features of the Java Language 2-9
 - Java Is Platform-Independent 2-10
 - Java Programs Run In a Java Virtual Machine 2-11
 - Procedural Programming Languages 2-12
 - Java Is an Object-Oriented Language 2-13
 - Topics 2-14
 - Verifying the Java Development Environment 2-15
 - Examining the Installed JDK: The Tools 2-16
 - Topics 2-17
 - Compiling and Running a Java Program 2-18
 - Compiling a Program 2-19
 - Executing (Testing) a Program 2-20
 - Output for a Java Program 2-21
 - Exercise 2-1 2-22
 - JDK 11: Launch Single-File Source-Code Programs 2-23
 - Exercise 2-2 2-24

Quiz 2-25

Summary 2-26

3 Creating a Java Main Class

Objectives 3-2

Topics 3-3

Java Classes 3-4

Program Structure 3-5

Java Packages 3-6

Java IDEs 3-7

The NetBeans IDE 3-8

Creating a Java Project 3-9

Creating a Java Class 3-10

Exercise 3-1:Creating a New Project and Java Class 3-11

Opening an Existing Java Project 3-12

Creating a New Java Package 3-13

Topics 3-14

The main Method 3-15

A main Class Example 3-16

Output to the Console 3-17

Avoiding Syntax Errors 3-18

Compiling and Running a Program by Using NetBeans 3-19

Exercise 3-2: Creating a main Method 3-20

Quiz 3-21

Summary 3-22

4 Data in a Cart

Objectives 4-2

Topics 4-3

Variables 4-4

Variable Types 4-5

Naming a Variable 4-6

Java SE 9: The Underscore Character Is Not a Legal Name 4-7

Uses of Variables 4-8

Topics 4-9

Examples: Variable Declaration and Initialization 4-10

String Concatenation 4-11

Exercise 4-1: Using String Variables 4-13

Quiz 4-14

Topics 4-15

int and double Values 4-16

Initializing and Assigning Numeric Values	4-17
Topics	4-18
Standard Mathematical Operators	4-19
Increment and Decrement Operators (++ and --)	4-20
Operator Precedence	4-21
Using Parentheses	4-23
Exercise 4-2: Using and Manipulating Numbers	4-24
Quiz	4-25
Summary	4-27

5 Managing Multiple Items

Objectives	5-2
Topics	5-3
Making Decisions	5-4
The if/else Statement	5-5
Boolean Expressions	5-6
Relational Operators	5-7
Examples	5-8
Exercise 5-1: Using if Statements	5-9
Quiz	5-10
Topics	5-11
What If There Are Multiple Items in the Shopping Cart?	5-12
Introduction to Arrays	5-13
Array Examples	5-14
Array Indices and Length	5-15
Declaring and Initializing an Array	5-16
Accessing Array Elements	5-18
Exercise 5-2: Using an Array	5-19
Quiz	5-20
Topics	5-22
Loops	5-23
Processing a String Array	5-24
Using break with Loops	5-25
Exercise 5-3: Using a Loop to Process an Array	5-26
Quiz	5-27
Summary	5-28

6 Describing Objects and Classes

Interactive Quizzes	6-2
Objectives	6-3
Topics	6-4

Object-Oriented Programming	6-5
Duke's Choice Order Process	6-6
Characteristics of Objects	6-7
Classes and Instances	6-8
Quiz	6-9
Topics	6-10
The Customer Properties and Behaviors	6-11
The Components of a Class	6-12
Modeling Properties and Behaviors	6-13
Exercise 6-1: Creating the Item Class	6-14
Topics	6-15
Customer Instances	6-16
Object Instances and Instantiation Syntax	6-17
The Dot (.) Operator	6-18
Objects with Another Object as a Property	6-19
Quiz	6-20
Topics	6-21
Accessing Objects by Using a Reference	6-22
Working with Object References	6-23
References to Different Objects	6-26
References and Objects in Memory	6-28
Assigning a Reference to Another Reference	6-29
Two References, One Object	6-30
Exercise 6-2: Modifying the ShoppingCart to Use Item Fields	6-31
Topics	6-32
Arrays Are Objects	6-33
Declaring, Instantiating, and Initializing Arrays	6-34
Storing Arrays in Memory	6-35
Storing Arrays of Object References in Memory	6-36
Quiz	6-37
Topics	6-39
Soccer Application	6-40
Creating the Soccer Application	6-41
Soccer Web Application	6-42
Summary	6-43
Practices Overview	6-44

7 Manipulating and Formatting the Data in Your Program

Objectives	7-2
Topics	7-3
String Class	7-4

Concatenating Strings	7-5
String Method Calls with Primitive Return Values	7-8
String Method Calls with Object Return Values	7-9
Topics	7-10
Java API Documentation	7-11
JDK 11 API Documentation	7-12
Java Platform SE and JDK Version 11 API Specification	7-13
Java Platform SE 11: Method Summary	7-14
Java Platform SE 11: Method Detail	7-15
indexOf Method Example	7-16
Exercise 7-1: Use indexOf and substring Methods	7-17
Topics	7-18
StringBuilder Class	7-19
StringBuilder Advantages over String for Concatenation (or Appending)	7-20
StringBuilder: Declare and Instantiate	7-21
StringBuilder Append	7-22
Quiz	7-23
Exercise 7-2: Instantiate the StringBuilder object	7-24
Topics	7-25
Primitive Data Types	7-26
Some New Integral Primitive Types	7-27
Floating Point Primitive Types	7-28
Textual Primitive Type	7-29
Java Language Trivia: Unicode	7-30
Constants	7-31
Quiz	7-32
Topics	7-33
Modulus Operator	7-34
Combining Operators to Make Assignments	7-35
More on Increment and Decrement Operators	7-36
Increment and Decrement Operators (++ and —)	7-37
Topics	7-38
Promotion	7-39
Caution with Promotion	7-40
Type Casting	7-42
Caution with Type Casting	7-43
Using Promotion and Casting	7-45
Compiler Assumptions for Integral and Floating Point Data Types	7-46
Automatic Promotion	7-47
Using a long	7-48
Using Floating Points	7-49

Floating Point Data Types and Assignment	7-50
Quiz	7-51
Exercise 7-3: Declare a long, float, and char	7-52
Summary	7-53
Practices Overview	7-54

8 Creating and Using Methods

Objectives	8-2
Topics	8-3
Basic Form of a Method	8-4
Calling a Method from a Different Class	8-5
Caller and Worker Methods	8-6
A Constructor Method	8-7
Writing and Calling a Constructor	8-8
Calling a Method in the Same Class	8-9
Topics	8-10
Method Arguments and Parameters	8-11
Method Parameter Examples	8-12
Method Return Types	8-13
Method Return Types Examples	8-14
Method Return Animation	8-15
Passing Arguments and Returning Values	8-16
More Examples	8-17
Code Without Methods	8-18
Better Code with Methods	8-19
Even Better Code with Methods	8-20
Variable Scope	8-21
Advantages of Using Methods	8-22
Exercise 8-1: Declare a setColor Method	8-23
Topics	8-24
Static Methods and Variables	8-25
Example: Setting the Size for a New Item	8-26
Creating and Accessing Static Members	8-27
When to Use Static Methods or Fields	8-28
Some Rules About Static Fields and Methods	8-29
Static Fields and Methods Versus Instance Fields and Methods	8-30
Static Methods and Variables in the Java API	8-31
Examining Static Variables in the JDK Libraries	8-32
Using Static Variables and Methods: System.out.println	8-33
More Static Fields and Methods in the Java API	8-34
Converting Data Values	8-35

Topics	8-36
Passing an Object Reference	8-37
What If There Is a New Object?	8-38
A Shopping Cart Code Example	8-39
Passing by Value	8-40
Reassigning the Reference	8-41
Passing by Value	8-42
Topics	8-43
Method Overloading	8-44
Using Method Overloading	8-45
Method Overloading and the Java API	8-47
Exercise 8-2: Overload a setItemFields Method, Part 1	8-48
Exercise 8-2: Overload a setItemFields Method, Part 2	8-49
Quiz	8-50
Summary	8-51
Practices Overview	8-52

9 Using Encapsulation

Interactive Quizzes	9-2
Objectives	9-3
Topics	9-4
What Is Access Control?	9-5
Access Modifiers	9-6
Access from Another Class	9-7
Another Example	9-8
Using Access Control on Methods	9-9
Topics	9-10
Encapsulation	9-11
Get and Set Methods	9-12
Why Use Setter and Getter Methods?	9-13
Setter Method with Checking	9-14
Using Setter and Getter Methods	9-15
Exercise 9-1: Encapsulate a Class	9-16
Topics	9-17
Initializing a Shirt Object	9-18
Constructors	9-19
Shirt Constructor with Arguments	9-20
Default Constructor and Constructor with Args	9-21
Overloading Constructors	9-22
Quiz	9-23
Exercise 9-2: Create an Overloaded Constructor	9-24

Summary 9-25
Practices Overview 9-26

10 More on Conditionals

Objectives 10-2
Topics 10-3
Review: Relational Operators 10-4
Testing Equality Between String variables 10-5
Common Conditional Operators 10-9
Ternary Conditional Operator 10-10
Using the Ternary Operator 10-11
Exercise 10-1: Using the Ternary Operator 10-12
Topics 10-13
Handling Complex Conditions with a Chained if Construct 10-14
Determining the Number of Days in a Month 10-15
Chaining if/else Constructs 10-16
Exercise 10-2: Chaining if Statements 10-17
Topics 10-18
Handling Complex Conditions with a switch Statement 10-19
Coding Complex Conditions: switch 10-20
switch Statement Syntax 10-21
When to Use switch Constructs 10-22
Exercise 10-3: Using switch Construct 10-23
Quiz 10-24
Topics 10-25
Working with an IDE Debugger 10-26
Debugger Basics 10-27
Setting Breakpoints 10-28
The Debug Toolbar 10-29
Viewing Variables 10-30
Summary 10-31
Practices Overview 10-32

11 Working with Arrays, Loops, and Dates

Objectives 11-2
Topics 11-3
Displaying a Date 11-4
Class Names and the Import Statement 11-5
Working with Dates 11-6
Working with Different Calendars 11-7
Some Methods of LocalDate 11-8

Formatting Dates	11-9
Exercise 11-1: Declare a LocalDateTime Object	11-10
Topics	11-11
Using the args Array in the main Method	11-12
Converting String Arguments to Other Types	11-13
Pass Arguments to the args Array in NetBeans	11-14
Exercise 11-2: Parsing the args Array	11-15
Topics	11-16
Describing Two-Dimensional Arrays	11-17
Declaring a Two-Dimensional Array	11-18
Instantiating a Two-Dimensional Array	11-19
Initializing a Two-Dimensional Array	11-20
Quiz	11-21
Topics	11-22
Some New Types of Loops	11-23
Repeating Behavior	11-24
Coding a while Loop	11-25
A while Loop Example	11-26
while Loop with Counter	11-27
Coding a Standard for Loop	11-28
Standard for Loop Compared to a while loop	11-29
Standard for Loop Compared to an Enhanced for Loop	11-30
do/while Loop to Find the Factorial Value of a Number	11-31
Coding a do/while Loop	11-32
Comparing Loop Constructs	11-33
The continue Keyword	11-34
Exercise 11-3: Processing an Array of Items	11-35
Topics	11-36
Nesting Loops	11-37
Nested for Loop	11-38
Nested while Loop	11-39
Processing a Two-Dimensional Array	11-40
Output from Previous Example	11-41
Quiz	11-42
Topics	11-44
ArrayList Class	11-45
Benefits of the ArrayList Class	11-46
Importing and Declaring an ArrayList	11-47
Working with an ArrayList	11-48
Exercise 11-4: Working with an ArrayList	11-49

Summary 11-50
Practices Overview 11-51

12 Using Inheritance

Interactive Quizzes 12-2
Objectives 12-3
Duke's Choice Classes: Common Behaviors 12-4
Code Duplication 12-5
Inheritance 12-6
Inheritance Terminology 12-7
Topics 12-8
Implementing Inheritance 12-9
More Inheritance Facts 12-10
Clothing Class: Part 1 12-11
Shirt Class: Part 1 12-12
Constructor Calls with Inheritance 12-13
Inheritance and Overloaded Constructors 12-14
Exercise 12-1: Creating a Subclass, Part 1 12-15
Exercise 12-1: Creating a Subclass, Part 2 12-16
Topics 12-17
More on Access Control 12-18
Overriding Methods 12-19
Review: Duke's Choice Class Hierarchy 12-20
Clothing Class: Part 2 12-21
Shirt Class: Part 2 12-22
Overriding a Method: What Happens at Run Time? 12-23
Exercise 12-2: Overriding a Method in the Superclass 12-24
Topics 12-25
Polymorphism 12-26
Superclass and Subclass Relationships 12-27
Using the Superclass as a Reference 12-28
Polymorphism Applied 12-29
Accessing Methods Using a Superclass Reference 12-30
Casting the Reference Type 12-31
instanceof Operator 12-32
Exercise 12-3: Using the instanceof Operator, Part 1 12-33
Exercise 12-3: Using the instanceof Operator, Part 2 12-34
Topics 12-35
Abstract Classes 12-36
Extending Abstract Classes 12-38

Summary	12-39
Practice Overview	12-40

13 Using Interfaces

Objectives	13-2
Topics	13-3
The Object Class	13-4
Calling the <code>toString</code> Method	13-5
Overriding <code>toString</code> in Your Classes	13-6
Topics	13-7
The Multiple Inheritance Dilemma	13-8
The Java Interface	13-9
No Multiple Inheritance of State	13-10
Multiple Hierarchies with Overlapping Requirements	13-11
Using Interfaces in Your Application	13-12
Implementing the Returnable Interface	13-13
Access to Object Methods from Interface	13-14
Casting an Interface Reference	13-15
Quiz	13-16
Topics	13-18
What is This Feature?	13-19
Benefits	13-20
Where Can it be Used?	13-21
Why is The Scope So Narrow?	13-22
Exercise 13-1: Local Variable Type Inference	13-23
Topics	13-24
The Collections Framework	13-25
ArrayList Example	13-26
List Interface	13-27
Example: <code>Arrays.asList</code>	13-28
Exercise 13-2: Converting an Array to an ArrayList, Part 1	13-30
Exercise 13-2: Converting an Array to an ArrayList, Part 2	13-31
Topics	13-32
Example: Modifying a List of Names	13-33
Using a Lambda Expression with <code>replaceAll</code>	13-34
Lambda Expressions	13-35
The Enhanced APIs That Use Lambda	13-36
Lambda Types	13-37
The UnaryOperator Lambda Type	13-38
The Predicate Lambda Type	13-39
Exercise 13-3: Using a Predicate Lambda Expression	13-40

Summary 13-41
Practice Overview 13-42

14 Handling Exceptions

Objectives 14-2
Topics 14-3
What Are Exceptions? 14-4
Examples of Exceptions 14-5
Code Example 14-6
Another Example 14-7
Types of Throwable classes 14-8
Error Example: OutOfMemoryError 14-9
Quiz 14-10
Topics 14-11
Normal Program Execution: The Call Stack 14-12
How Exceptions Are Thrown 14-13
Topics 14-14
Working with Exceptions in NetBeans 14-15
The try/catch Block 14-16
Program Flow When an Exception Is Caught 14-17
When an Exception Is Thrown 14-18
Throwing Throwable Objects 14-19
Uncaught Exception 14-20
Exception Printed to Console 14-21
Summary of Exception Types 14-22
Exercise 14-1: Catching an Exception 14-23
Quiz 14-24
Exceptions in the Java API Documentation 14-25
Calling a Method That Throws an Exception 14-26
Working with a Checked Exception 14-27
Best Practices 14-28
Bad Practices 14-29
Somewhat Better Practice 14-30
Topics 14-31
Multiple Exceptions 14-32
Catching IOException 14-33
Catching IllegalArgumentException 14-34
Catching Remaining Exceptions 14-35
Summary 14-36
Practices Overview 14-37

15 Deploying and Maintaining the Soccer Application

- Interactive Quizzes 15-2
- Objectives 15-3
- Topics 15-4
- Packages 15-5
 - Packages Directory Structure 15-6
 - Packages in NetBeans 15-7
 - Packages in Source Code 15-8
- Topics 15-9
- SoccerEnhanced.jar 15-10
- Set Main Class of Project 15-11
- Creating the JAR File with NetBeans 15-12
- Topics 15-14
 - Client/Server Two-Tier Architecture 15-15
 - Client/Server Three-Tier Architecture 15-16
- Topics 15-17
 - Client/Server Three-Tier Architecture 15-18
 - Different Outputs 15-20
 - The Soccer Application 15-21
 - IDisplayDataItem Interface 15-22
 - Running the JAR File from the Command Line 15-23
 - Text Presentation of the League 15-24
 - Web Presentation of the League 15-25
 - Topics 15-26
 - Enhancing the Application 15-27
 - Adding a New GameEvent Kickoff 15-28
 - Game Record Including Kickoff 15-29
 - Summary 15-30

16 Understanding Modules

- Objectives 16-2
- Topics 16-3
- Reusing Code in Java: Classes 16-4
- Reusing Code in Java: Packages 16-5
- Reusing Code in Java: Programming in the Large 16-6
- What Is a Module? 16-7
- Example: java.base Module 16-8
- Module System 16-9
- Modular Development in JDK 9 16-10
- Topics 16-11
- JARs 16-12

JAR Files and Distribution Issues	16-13
Class Path Problems	16-14
Example JAR Duplicate Class Problem 1	16-15
Example JAR Duplicate Class Problem 2	16-16
Module System: Advantages	16-17
Accessibility Between Classes	16-18
Access Across Non-Modular JARs	16-19
Access Across Modules	16-20
Topics	16-21
module-info.java	16-22
Example: module-info.java	16-23
Creating a Modular Project	16-24
exports Module Directive	16-25
exports...to Module Directive	16-26
requires Module Directive	16-27
requires transitive Module Directive	16-28
Implementing a Requires Transitively Relationship	16-29
Summary of Keywords	16-30
Compiling Modules	16-31
Running a Modular Application	16-32
Topics	16-33
The JDK	16-34
The Modular JDK	16-35
Listing the Modules in JDK 9	16-36
Java SE Modules	16-37
The Base Module	16-38
Summary	16-39
Practice Overview	16-40

17 JShell

Objectives	17-2
Topics	17-3
A Million Test Classes and Main Methods	17-4
JShell Provides a Solution	17-5
Topics	17-6
Comparing Normal Execution with REPL	17-7
Getting Started with JShell and REPL	17-8
Scratch Variables	17-9
Declaring Traditional Variables	17-10
Code Snippets	17-11
Completing a Code Snippet	17-12

Tab Completion and Tab Tips	17-13
Semicolons	17-14
JShell Commands	17-15
Importing Packages	17-16
Quiz 17-1	17-17
Topics	17-18
Why Incorporate JShell in an IDE?	17-19
Use Cases	17-20
Two Ways to Open JShell in NetBeans	17-21
Summary	17-22
Practices	17-23

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

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

More on Conditionals



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



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

Objectives

After completing this lesson, you should be able to:

- Use a `ternary` statement
- Test equality between strings
- Chain an `if/else` statement
- Use a `switch` statement
- Use the NetBeans debugger



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



Topics

- Relational and conditional operators
- More ways to use if/else statements
- Using a switch statement
- Using the NetBeans debugger



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



Review: Relational Operators

Condition	Operator	Example
Is equal to	<code>==</code>	<code>int i=1; (i == 1)</code>
Is not equal to	<code>!=</code>	<code>int i=2; (i != 1)</code>
Is less than	<code><</code>	<code>int i=0; (i < 1)</code>
Is less than or equal to	<code><=</code>	<code>int i=1; (i <= 1)</code>
Is greater than	<code>></code>	<code>int i=2; (i > 1)</code>
Is greater than or equal to	<code>>=</code>	<code>int i=1; (i >= 1)</code>



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

Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = "Fred Smith";  
    public String name2 = "Sam Smith";  
  
    public void areNamesEqual() {  
        if (name1.equals(name2)) {  
            System.out.println("Same name.");  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



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

If you use the `==` operator to compare object references, the operator tests to see whether both object references are the same (that is, do the `String` objects point to the same location in memory). For a `String` it is likely that instead you want to find out whether the characters within the two `String` objects are the same. The best way to do this is to use the `equals` method.

Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = "Fred Smith";  
    public String name2 = "fred smith";  
  
    public void areNamesEqual() {  
        if (name1.equalsIgnoreCase(name2)) {  
            System.out.println("Same name.");  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



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

There is also an `equalsIgnoreCase` method that ignores the case when it makes the comparison.

Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = "Fred Smith";  
    public String name2 = "Fred Smith";  
  
    public void areNamesEqual() {  
        if (name1 == name2) {  
            System.out.println("Same name."); ✓  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



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

- Depending on how the String variables are initialized, == might actually be effective in comparing the values of two String objects, but only because of the way Java deals with strings.
- In this example, only one object was created to contain "Fred Smith" and both references (`name1` and `name2`) point to it. Therefore, `name1 == name2` is true. This is done to save memory. However, because String objects are immutable, if you assign `name1` to a different value, `name2` is still pointing to the original object and the two references are no longer equal.

Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = new String("Fred Smith");  
    public String name2 = new String("Fred Smith");  
  
    public void areNamesEqual() {  
        if (name1 == name2) {  
            System.out.println("Same name.");  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



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



- When you initialize a String using the new keyword, you force Java to create a new object in a new location in memory even if a String object containing the same character values already exists. Therefore in the following example, name1 == name2 would return false.
- It makes sense then that the safest way to determine equality of two string values is to use the equals method.

Common Conditional Operators

Operation	Operator	Example
If one condition AND another condition	&&	<pre>int i = 2; int j = 8; (i < 1) && (j > 6)</pre>
If either one condition OR another condition		<pre>int i = 2; int j = 8; (i < 1) (j > 10)</pre>
NOT	!	<pre>int i = 2; (! (i < 3))</pre>



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

Ternary Conditional Operator

Operation	Operator	Example
If some condition is true, assign the value of value1 to the result. Otherwise, assign the value of value2 to the result.	? :	condition ? value1 : value2 Example: int x = 2, y = 5, z = 0; z = (y < x) ? x : y;

Equivalent statements

z = (y < x) ? x : y;

```
if(y<x){  
    z=x;  
}  
else{  
    z=y;  
}
```



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

Using the Ternary Operator

Advantage: Usable in a single line

```
int numberOfGoals = 1;  
String s = (numberOfGoals==1 ? "goal" : "goals");  
  
System.out.println("I scored " +numberOfGoals +" "  
+s );
```

Advantage: Place the operation directly within an expression

```
int numberOfGoals = 1;  
  
System.out.println("I scored " +numberOfGoals +" "  
+(numberOfGoals==1 ? "goal" : "goals") );
```

Disadvantage: Can have only two potential results

(numberOfGoals==1 ? "goal" : "goals" : "More goals");

 boolean

 true

 false

 ???



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

Based on the number of goals scored, these examples will print the appropriate singular or plural form of "goal."

The operation is compact because it can only yield two results, based on a boolean expression.

Exercise 10-1: Using the Ternary Operator

In this exercise, you use a ternary operator to duplicate the same logic shown in this `if/else` statement:

```
01 int x = 4, y = 9;  
02 if ((y / x) < 3) {  
03     x += y;  
04 }  
05 else x *= y;
```



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

Perform this exercise by opening the project **Exercise_10-1** or create your own project with a **Java Main Class** named `TestClass`.

Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- Using a switch statement
- Using the NetBeans debugger



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



Handling Complex Conditions with a Chained `if` Construct

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

The chained `if` statement:

- Connects multiple conditions together into a single construct
- Often contains nested `if` statements
- Tends to be confusing to read and hard to maintain



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

Determining the Number of Days in a Month

```
01  if (month == 1 || month == 3 || month == 5 || month == 7  
02      || month == 8 || month == 10 || month == 12) {  
03      System.out.println("31 days in the month.");  
04  }  
05  else if (month == 2) {  
06      if(!isLeapYear){  
07          System.out.println("28 days in the month.");  
08      }else System.out.println("29 days in the month.");  
09  }  
10  else if (month ==4 || month == 6 || month == 9  
11      || month == 11) {  
12      System.out.println("30 days in the month.");  
13  }  
14  else  
15      System.out.println("Invalid month.");
```



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

- The code example above shows how you would use a chained and nested `if` to determine the number of days in a month.
- Notice that, if the month is 2, a nested `if` is used to check whether it is a leap year.

Note: Debugging (covered later in this lesson) would reveal how every `if/else` statement is examined up until a statement is found to be true.

Chaining if/else Constructs

Syntax:

```
01  if <condition1> {  
02      //code_block1  
03  }  
04  else if <condition2> {  
05      // code_block2  
06  }  
07  else {  
08      // default_code  
09  }
```



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

You can chain `if` and `else` constructs together to state multiple outcomes for several different expressions. The syntax for a chained `if/else` construct is shown in the slide example, where:

- Each of the conditions is a boolean expression.
- `code_block1` represents the lines of code that are executed if `condition1` is true.
- `code_block2` represents the lines of code that are executed if `condition1` is false and `condition2` is true.
- `default_code` represents the lines of code that are executed if both conditions evaluate to false.

Exercise 10-2: Chaining if Statements

1. Open the project `Exercise_10-2` in NetBeans.

In the `Order` class:

2. Complete the `calcDiscount` method so it determines the discount for three different customer types:
 - Nonprofits get a discount of 10% if total > 900, else 5%.
 - Private customers get a discount of 7% if total > 900, else 0%.
 - Corporations get a discount of 8% if total < 500, else 5%.

In the `ShoppingCart` class:

3. Use the `main` method to test the `calcDiscount` method.



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

Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- **Using a switch statement**
- Using the NetBeans debugger



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



Handling Complex Conditions with a switch Statement

The switch statement:

- Is a streamlined version of chained if statements
- Is easier to read and maintain
- Offers better performance



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

Coding Complex Conditions: switch

```
01 switch (month) {  
02     case 1: case 3: case 5: case 7:  
03     case 8: case 10: case 12:  
04         System.out.println("31 days in the month.");  
05         break;  
06     case 2:  
07         if (!isLeapYear) {  
08             System.out.println("28 days in the month.");  
09         } else  
10             System.out.println("29 days in the month.");  
11         break;  
12     case 4: case 6: case 9: case 11:  
13         System.out.println("30 days in the month.");  
14         break;  
15     default:  
16         System.out.println("Invalid month.");  
17     }  
18 }
```



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

Here you see an example of the same conditional logic (from the previous chained `if` example) implemented as a `switch` statement. It is easier to read and understand what is happening here.

- The `month` variable is evaluated only once, and then matched to several possible values.
- Notice the `break` statement. This causes the `switch` statement to exit without evaluating the remaining cases.

Note: Debugging (covered later in this lesson) reveals why the `switch` statement offers better performance compared to an `if/else` construct. Only the line containing the true case is executed in a `switch` construct, whereas every `if/else` statement must be examined up until a statement is found to be true.

switch Statement Syntax

Syntax:

```
01 switch (<variable or expression>) {  
02     case <literal value>:  
03         //code_block1  
04         [break;]  
05     case <literal value>:  
06         // code_block2  
07         [break;]  
08     default:  
09         //default_code  
10 }
```



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

When to Use switch Constructs

Use when you are testing:

- Equality (not a range)
- A *single* value
- Against fixed known values at compile time
- The following data types:
 - Primitive data types: int, short, byte, char
 - String or enum (enumerated types)
 - Wrapper classes (special classes that wrap certain primitive types):
Integer, Short, Byte and Character

Only a single variable can be tested.

```
01 switch (month) {  
02     case 1: case 3: case 5: case 7:  
03     case 8: case 10: case 12:  
04         System.out.println("31 days in the month.");  
05         break;  
06     case 2:  
07         if (!isLeapYear) {  
08             System.out.println("28 days in the month.");  
09         } else  
10             System.out.println("29 days in the month.");
```

} Known values



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

If you are not able to find values for individual test cases, it would be better to use an if/else construct instead.

Exercise 10-3: Using switch Construct

1. Continue editing `Exercise_10-2` or open `Exercise_10-3`.

In the `Order` class:

2. Rewrite `calcDiscount` to use a `switch` statement:

- Use a ternary expression to replace the nested `if` logic.
- For better performance, use a `break` statement in each case block.
- Include a `default` block to handle invalid `custType` values.

In the `ShoppingCart` class:

3. Use the `main` method to test the `calcDiscount` method.



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

Quiz

Which of the following sentences describe a valid case to test in a `switch` construct?

- a. The `switch` construct tests whether values are greater than or less than a single value.
- b. Variable or expression where the expression returns a supported `switch` type.
- c. The `switch` construct can test the value of a `float`, `double`, `boolean`, or `String`.
- d. The `switch` construct tests the outcome of a `boolean` expression.



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

Answer: b

- Answer a is incorrect because you must test for a single value, not a range of values. Relational operators are not allowed.
- Answer b is correct.
- Answer c is incorrect. The `switch` construct tests the value of types `char`, `byte`, `short`, `int`, or `String`.
- Answer d is incorrect. The `switch` construct tests of value of expressions that return `char`, `byte`, `short`, `int`, or `String`—not `boolean`.

Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- Using a `switch` statement
- Using the NetBeans debugger



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



Working with an IDE Debugger

Most IDEs provide a debugger. They are helpful to solve:

- Logic problems
 - (Why am I not getting the result I expect?)
- Runtime errors
 - (Why is there a `NullPointerException`?)



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

Debugging can be a useful alternative to print statements.

Debugger Basics

- Breakpoints:
 - Are stopping points that you set on a line of code
 - Stop execution at that line so you can view the state of the application
- Stepping through code:
 - After stopping at a break point, you can “walk” through your code, line by line to see how things change.
- Variables:
 - You can view or change the value of a variable at run time.
- Output:
 - You can view the System output at any time.



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

Setting Breakpoints

- To set breakpoints, click in the margin of a line of code.
- You can set multiple breakpoints in multiple classes.

The screenshot shows a Java code editor with the following code:

```
3 public class DebugTestIfElse {
4     public static void main(String[] args) {
5         int month =11;
6         boolean isLeapYear = true;
7
8         if(month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 || month == 12){
9             System.out.println("31 days in the month.");
10        }
11        else if(month == 2){
12            if(!isLeapYear){
13                System.out.println("28 days in the month.");
14            }
15            else{
16                System.out.println("29 days in the month.");
17            }
18        }
19        else if(month == 4 || month ==6 || month == 9 || month ==11){
20            System.out.println("30 days in the month.");
21        }
22        else{
23            System.out.println("Invalid month");
24        }
25    }
26}
```

Two breakpoints are marked with red circles and squares in the margin of lines 9 and 20.



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

The Debug Toolbar

1. Start debugger
2. Stop debug session
3. Pause debug session
4. Continue running
5. Step over
6. Step over an expression
7. Step into
8. Step out of



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

Here you see the Debug toolbar in NetBeans. Each button is numbered and the corresponding description of the function of that button appears in the list on the left.

1. Start the debug session for the current project by clicking button 1. After a session has begun, the other buttons become enabled. The project runs, stopping at the first breakpoint.
2. You can exit the debug session by clicking button 2.
3. Button 3 allows you to pause the session.
4. Button 4 continues running until the next breakpoint or the end of the program.
5. Buttons 5 through 8 give you control over how far you want to drill down into the code. For example:
 - If execution has stopped just before a method invocation, you may want to skip to the next line after the method.
 - If execution has stopped just before an expression, you may want to skip over just the expression to see the final result.
 - 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.
 - If you have stepped into a method or another class, use the last button to step back out into the original code block.

Viewing Variables

Current line of execution

Breakpoint

Name	Type	Value
Static	String[]	#72[length=0]
args	String[]	
month	int	2
isLeapYear	boolean	true

Value of variables

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

Here you see a debug session in progress. The debugger stopped at the breakpoint line, but then the programmer began stepping through the code. The current line of execution is indicated by the green arrow in the margin.

Notice that the `isLeapYear` variable on the current line appears in the Variables tab at the bottom of the window. Here you can view the value or even change it to see how the program would react.

Note: Debugging reveals why the `switch` statement offers better performance compared to an `if/else` construct. Only the line containing the true case is executed in a switch construct, whereas every `if/else` statement must be examined up until a statement is found to be true.

Summary

In this lesson, you should have learned how to:

- Use a `ternary` statement
- Test equality between strings
- Chain an `if/else` statement
- Use a `switch` statement
- Use the NetBeans debugger



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



Practices Overview

- 10-1: Using Conditionals
- 10-2: Debugging



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

Working with Arrays, Loops, and Dates



ORACLE®



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

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

Objectives

After completing this lesson, you should be able to:

- Create a `java.time.LocalDateTime` object to show the current date and time
- Parse the `args` array of the `main` method
- Nest a `while` loop
- Develop and nest a `for` loop
- Code and nest a `do/while` loop
- Use an `ArrayList` to store and manipulate objects



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



Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- The `ArrayList` class



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

The first topic, “Working with dates,” focuses on the new Date Time API. This is a new feature of Java SE 8.



Displaying a Date

```
LocalDate myDate = LocalDate.now();  
System.out.println("Today's date: "+ myDate);
```

Output: 2018-12-20

- `LocalDate` belongs to the package `java.time`.
- The `now` method returns today's date.
- This example uses the default format for the default time zone.



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

The `now` static method returns an object of type `LocalDate`. Of course, `System.out.println` calls the `toString` method of the `LocalDate` object. Its `String` representation is 2018-12-20 in this example.

Class Names and the Import Statement

- Date classes are in the package `java.time`.
- To refer to one of these classes in your code, you can fully qualify `java.time.LocalDate`
or, add the import statement at the top of the class.

```
import java.time.LocalDate;
public class DateExample {
    public static void main (String[] args) {
        LocalDate myDate;
    }
}
```



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

Classes in the Java programming language are grouped into packages depending on their functionality. For example, all classes related to the core Java programming language are in the `java.lang` package, which contains classes that are fundamental to the Java programming language, such as `String`, `Math`, and `Integer`. Classes in the `java.lang` package can be referred to in code by just their class names. They do not require full qualification or the use of an import statement.

All classes in other packages (for example, `LocalDate`) require that you fully qualify them in the code or that you use an import statement so that they can be referred to directly in the code.

The `import` statement can be:

- For just the class in question
`java.time.LocalDate;`
- For all classes in the package
`java.time.*;`

Working with Dates

java.time

- Main package for date and time classes

java.time.format

- Contains classes with static methods that you can use to format dates and times

Some notable classes:

- java.time.LocalDate
- java.time.LocalDateTime
- java.time.LocalTime
- java.time.format.DateTimeFormatter

Formatting example:

```
myDate.format(DateTimeFormatter.ISO_LOCAL_DATE);
```



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

The Java API has a `java.time` package that offers many options for working with dates and times. A few notable classes are:

- `java.time.LocalDate` is an immutable date-time object that represents a date, often viewed as year-month-day. Other date fields, such as day-of-year, day-of-week, and week-of-year, can also be accessed. For example, the value “2nd October 2007” can be stored in a `LocalDate`.
- `java.time.LocalDateTime` is an immutable date-time object that represents a date-time, often viewed as year-month-day-hour-minute-second. Other date and time fields, such as day-of-year, day-of-week, and week-of-year, can also be accessed. Time is represented to nanosecond precision. For example, the value “2nd October 2007 at 13:45.30.123456789” can be stored in a `LocalDateTime`.
- `java.time.LocalTime` is an immutable date-time object that represents a time, often viewed as hour-minute-second. Time is represented to nanosecond precision. For example, the value “13:45.30.123456789” can be stored in a `LocalTime`. It does not store or represent a date or time-zone. Instead, it is a description of the local time as seen on a wall clock. It cannot represent an instant on the time-line without additional information such as an offset or time zone.

Working with Different Calendars

- The default calendar is based on the Gregorian calendar.
- If you need non-Gregorian type dates:
 - Use the `java.time.chrono` classes
 - They have conversion methods.
- Example: Convert a `LocalDate` to a Japanese date:

```
LocalDate myDate = LocalDate.now();
JapaneseDate jDate = JapaneseDate.from(mydate);
System.out.println("Japanese date: " + jDate);
```

- Output:
Japanese date: Japanese Heisei 26-01-16



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

In the above example, `JapaneseDate` is a class belonging to the `java.time.chrono` package. `myDate` is passed to the static `from` method, which returns a `JapaneseDate` object (`jDate`). The result of printing the `jDate` object is shown as output.

Some Methods of LocalDate

LocalDate overview: A few notable methods and fields

- Instance methods:
 - myDate.minusMonths (15); ~~long monthsToSubtract~~
 - myDate.plusDays (8); ~~long daysToAdd~~
- Static methods:
 - of(int year, Month month, int dayOfMonth)
 - parse(CharSequence text, DateTimeFormatter formatter)
 - now()



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

LocalDate has many methods and fields. Here are just a few of the instance and static methods that you might use:

- minusMonths returns a copy of this LocalDate with the specified period in months subtracted.
- plusDays returns a copy of this LocalDate with the specified number of days added.
- of(int year, Month month, int dayOfMonth) obtains an instance of LocalDate from a year, month, and day.
- parse(CharSequence text, DateTimeFormatter formatter) obtains an instance of LocalDate from a text string using a specific formatter.

Read the LocalDate API reference for more details.

Formatting Dates

```
1 LocalDateTime today = LocalDateTime.now();
2 System.out.println("Today's date time (no formatting): "
3                     + today);
4
5 String sdate =
6     today.format(DateTimeFormatter.ISO_DATE_TIME);Format the date in
standard ISO format.
7 System.out.println("Date in ISO_DATE_TIME format: "
8                     + sdate);
9
10 String fdate =
11    today.format(DateTimeFormatter.ofLocalizedDateTime
12                  (FormatStyle.MEDIUM));Localized date time in
Medium format
13
14 System.out.println("Formatted with MEDIUM FormatStyle: "
15                     + fdate);
```

Output:

```
Today's date time (no formatting): 2013-12-23T16:51:49.458
Date in ISO_DATE_TIME format: 2013-12-23T16:51:49.458
Formatted with MEDIUM FormatStyle: Dec 23, 2013 4:51:49 PM
```



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

The code example in the slide shows you some options for formatting the output of your dates.

- **Line 1:** Get a `LocalDateTime` object that reflects today's date.
- **Lines 6 - 7:** Get a `String` that shows the date object formatted in standard `ISO_DATE_TIME` format. As you see in the output, the default format when you just print the `LocalDateTime` object uses the same format.
- **Lines 11 - 12:** Call the `ofLocalizedDateTime` method of the `DateTimeFormatter` to get a `String` representing the date in a medium localized date-time format. The third line of the output shows this shorter version of the date.

Exercise 11-1: Declare a `LocalDateTime` Object

1. Open the project `Exercise_11-1` or create your own project with a Java Main Class named `TestClass`.
2. Declare a `LocalDateTime` object to hold the order date.
3. Initialize the object to the current date and time by using the `now()` static method of the class.
4. Print the `orderDate` object with a suitable label.
5. Format `orderDate` by using the `ISO_LOCAL_DATE` static constant field of the `DateTimeFormatter` class.
6. Add the necessary package imports.
7. Print the formatted `orderDate` with a suitable label.



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

Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- The `ArrayList` class



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



Using the args Array in the main Method

- Parameters can be typed on the command line:

```
> java ArgsTest Hello World!
args[0] is Hello
args[1] is World!
```

Diagram annotations: 'Hello' is highlighted with a red box and connected by a blue bracket to 'args[0] is Hello'. 'World!' is highlighted with a red box and connected by a blue bracket to 'args[1] is World!'. Handwritten notes: 'Goes into args[1]' is written next to the bracket under 'args[1] is World!', and 'Goes into args[0]' is written next to the bracket under 'args[0] is Hello'.

- Code for retrieving the parameters:

```
public class ArgsTest {
    public static void main (String[] args) {
        System.out.println("args[0] is " + args[0]);
        System.out.println("args[1] is " + args[1]);
    }
}
```



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

When you pass strings to your program on the command line, the strings are put in the `args` array. To use these strings, you must extract them from the `args` array and, optionally, convert them to their proper type (because the `args` array is of type `String`).

The `ArgsTest` class shown in the slide extracts two `String` arguments passed on the command line and displays them.

To add parameters on the command line, you must leave one or more spaces after the class name (in this case, `ArgsTest`) and one or more spaces between each parameter added.

NetBeans does not allow you a way to run a Java class from the command line, but you can set command-line arguments as a property of the NetBeans project.

Converting String Arguments to Other Types

- Numbers can be typed as parameters:

```
> java ArgsTest 2 3  
Total is: 23  
Total is: 5
```

Concatenation, not addition!

- Conversion of String to int:

```
public class ArgsTest {  
    public static void main (String[] args) {  
        System.out.println("Total is:"+ (args[0]+args[1]));  
        int arg1 = Integer.parseInt(args[0]);  
        int arg2 = Integer.parseInt(args[1]);  
        System.out.println("Total is: " + (arg1+arg2));  
    }  
}
```

Strings

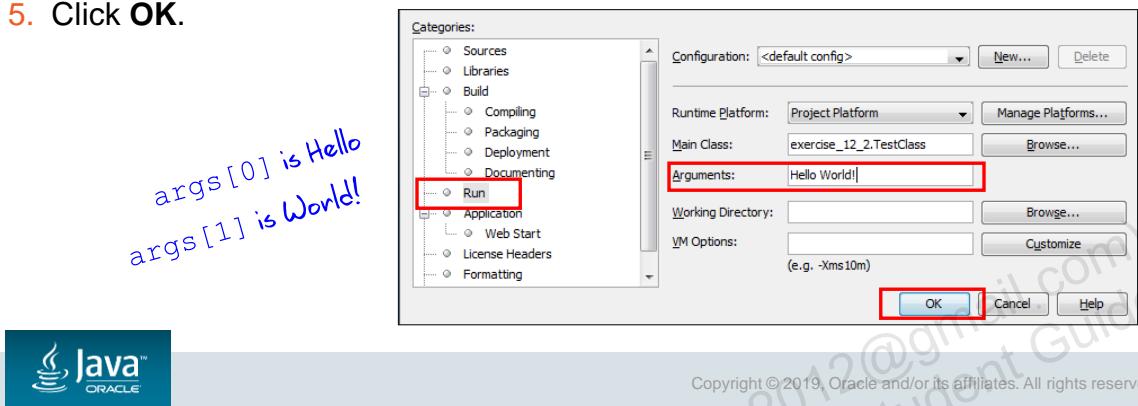
Note the parentheses.



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

Pass Arguments to the `args` Array in NetBeans

1. Right-click on your project.
2. Select **Properties**.
3. Select **Run**.
4. Type your arguments into the **Arguments** field.
 - Separate each argument with a space, not a comma.
5. Click **OK**.



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

Exercise 11-2: Parsing the args Array

1. Open the project **Exercise_11-2** or create your own project with a **Java Main Class** named TestClass.
2. Parse the args array to populate name and age.
 - If args contains fewer than two elements, print a message telling the user that two arguments are required.
 - Remember that the age argument will have to be converted to an `int`.
 - Hint: Use a static method of the Integer class to convert it.
3. Print the name and age values with a suitable label.



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

Topics

- Working with dates
- Parsing the `args` array
- **Two-dimensional arrays**
- Alternate looping constructs
- Nesting loops
- The `ArrayList` class



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



Describing Two-Dimensional Arrays



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

You can store matrices of data by using multidimensional arrays (arrays of arrays, of arrays, and so on). A two-dimensional array (an array of arrays) is similar to a spreadsheet with multiple columns (each column represents one array or list of items) and multiple rows.

The diagram in the slide shows a two-dimensional array. Note that the descriptive names Week 1, Week 2, Monday, Tuesday, and so on would not be used to access the elements of the array. Instead, Week 1 would be index 0 and Week 4 would be index 3 along that dimension, whereas Sunday would be index 0 and Saturday would be index 6 along the other dimension.

Declaring a Two-Dimensional Array

Syntax:

```
type [][] array_identifier;
```

Example:

```
int [][] yearlySales;
```



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

Two-dimensional arrays require an additional set of brackets. The process of creating and using two-dimensional arrays is otherwise the same as with one-dimensional arrays. The syntax for declaring a two-dimensional array is:

```
type [][] array_identifier;
```

where:

- `type` represents the primitive data type or object type for the values stored in the array
- `[][]` informs the compiler that you are declaring a two-dimensional array
- `array_identifier` is the name you have assigned the array during declaration

The example shown declares a two-dimensional array (an array of arrays) called `yearlySales`.

Instantiating a Two-Dimensional Array

Syntax:

```
array_identifier = new type [number_of_arrays] [length];
```

Example:

```
// Instantiates a 2D array: 5 arrays of 4 elements each  
yearlySales = new int[5][4];
```

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Year 1				
Year 2				
Year 3				
Year 4				
Year 5				



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

The syntax for instantiating a two-dimensional array is:

```
array_identifier = new type [number_of_arrays] [length];
```

where:

- `array_identifier` is the name that you have assigned the array during declaration
- `number_of_arrays` is the number of arrays within the array
- `length` is the length of each array within the array

The example shown in the slide instantiates an array of arrays for quarterly sales amounts over five years. The `yearlySales` array contains five elements of the type `int` array (five subarrays). Each subarray is four elements in size and tracks the sales for one year over four quarters.

Initializing a Two-Dimensional Array

Example:

```
int[][] yearlySales = new int[5][4];  
yearlySales[0][0] = 1000;  
yearlySales[0][1] = 1500;  
yearlySales[0][2] = 1800;  
yearlySales[1][0] = 1000;  
yearlySales[3][3] = 2000;
```

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Year 1	1000	1500	1800	
Year 2	1000			
Year 3				
Year 4				2000
Year 5				



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

When setting (or getting) values in a two-dimensional array, indicate the index number in the array by using a number to represent the row, followed by a number to represent the column. The example in the slide shows five assignments of values to elements of the `yearlySales` array.

Note: When you choose to draw a chart based on a 2D array, they way you orient the chart is arbitrary. That is, you have the option to decide if you would like to draw a chart corresponding to `array2DName [x] [y]` or `array2DName [y] [x]`.

Quiz



A two-dimensional array is similar to a _____.

- a. Shopping list
- b. List of chores
- c. Matrix
- d. Bar chart containing the dimensions for several boxes



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

Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- **Alternate looping constructs**
- Nesting loops
- The `ArrayList` class



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



Some New Types of Loops

Loops are frequently used in programs to repeat blocks of code while some condition is true. There are three main types of loops:

- A `while` loop repeats *while* an expression is true.
- A `for` loop simply repeats a *set number* of times.
 - * A variation of this is the **enhanced** `for` loop. This loops through the elements of an array.
- A `do/while` loop executes once and then continues to repeat *while* an expression is true.

**You have already learned this one!*



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

Up to this point, you have been using the enhanced for loop, which repeats a block of code for each element of an array.

Now you can learn about the other types of loops as described above.

Repeating Behavior



```
while (!areWeThereYet) {  
  
    read book;  
    argue with sibling;  
    ask, "Are we there yet?";  
  
}  
  
Woohoo!;  
Get out of car;
```



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

A common requirement in a program is to repeat a number of statements. Typically, the code continues to repeat the statements until something changes. Then the code breaks out of the loop and continues with the next statement.

The pseudocode example above, shows a `while` loop that loops until the `areWeThereYet` boolean is true.

Coding a while Loop

Syntax:

```
while (boolean_expression) {  
    code_block;  
}
```

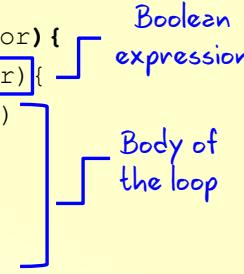


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

The `while` loop first evaluates a boolean expression and, while that value is true, it repeats the code block. To avoid an infinite loop, you need to be sure that something will cause the boolean expression to return false eventually. This is frequently handled by some logic in the code block itself.

A while Loop Example

```
01 public class Elevator {  
02     public int currentFloor = 1;  
03  
04     public void changeFloor(int targetFloor){  
05         while (currentFloor != targetFloor){  
06             if(currentFloor < targetFloor)  
07                 goUp();  
08             else  
09                 goDown();  
10         }  
11     }
```



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

The code in the slide shows a very simple while loop in an `Elevator` class. The elevator accepts commands for going up or down only one floor at a time. So to move a number of floors, the `goUp` or `goDown` method needs to be called a number of times.

- The `goUp` and `goDown` methods increment or decrement the `currentFloor` variable.
- The boolean expression returns `true` if `currentFloor` is not equal to `targetFloor`. When these two variables are equal, this expression returns `false` (because the elevator is now at the desired floor), and the body of the while loop is not executed.

while Loop with Counter

```
01 System.out.println("/*");
02 int counter = 0;
03 while (counter < 3){
04     System.out.println(" *");
05     counter++;
06 }
07 System.out.println("*/");
```

Output:

```
/*
 *
 *
 *
 */

```



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

Loops are often used to repeat a set of commands a specific number of times. You can easily do this by declaring and initializing a counter (usually of type `int`), incrementing that variable inside the loop, and checking whether the counter has reached a specific value in the `while` boolean expression.

Although this works, the standard `for` loop is ideal for this purpose.

Coding a Standard `for` Loop

The standard `for` loop repeats its code block for a set number of times using a counter.

- Syntax:

```
01 for(<type> counter = n; <boolean_expression>; <counter_increment>){  
02     code_block;  
03 }
```

- Example:

```
01 for(int i = 1; i < 5; i++){  
02     System.out.print("i = " + i +"; ");  
03 }
```

Output: `i = 1; i = 2; i = 3; i = 4;`



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

The three essential elements of a standard `for` loop are the counter, the boolean expression, and the increment. All of these are expressed within parentheses following the keyword `for`.

1. A counter is declared and initialized as the first parameter of the `for` loop. (`int i = 1`)
2. A boolean expression is the second parameter. It determines the number of loop iterations. (`i < 5`)
3. The counter increment is defined as the third parameter. (`i++`)

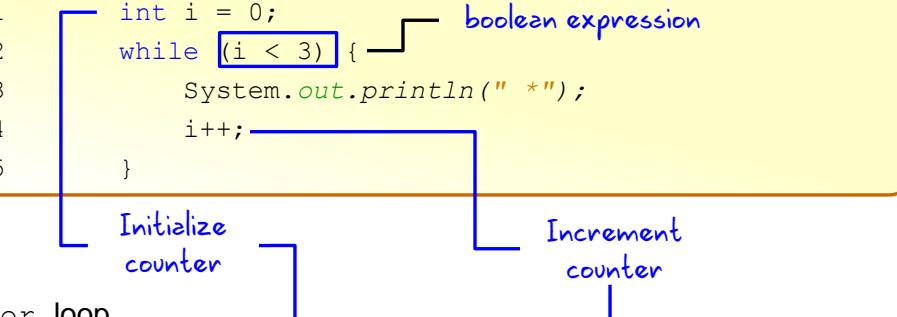
The code block (shown on line 2) is executed in each iteration of the loop. At the end of the code block, the counter is incremented by the amount indicated in the third parameter.

As you can see, in the output shown above, the loop iterates four times.

Standard for Loop Compared to a while loop

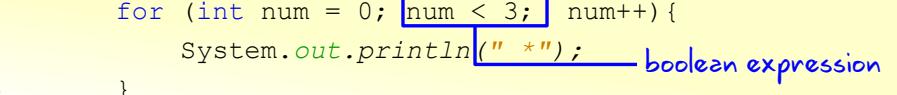
while loop

```
01 int i = 0;  
02 while (i < 3) {  
03     System.out.println(" *");  
04     i++;  
05 }
```



for loop

```
01 for (int num = 0; num < 3; num++) {  
02     System.out.println(" *");  
03 }
```



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

Standard `for` Loop Compared to an Enhanced `for` Loop

Enhanced `for` loop

```
01 for(String name: names){  
02     System.out.println(name);  
03 }
```

Standard `for` loop

```
01 for (int idx = 0; idx < names.length; idx++){  
02     System.out.println(names[idx]);  
03 }
```

boolean expression

Counter used as the
index of the array



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

This slide compares the standard `for` loop to the enhanced `for` loop that you learned about in the lesson titled “Managing Multiple Items.” The examples here show each type of `for` loop used to iterate through an array. Enhanced `for` loops are used only to process arrays, but standard `for` loops can be used in many ways.

- **The enhanced `for` loop example:** A `String` variable, `name`, is declared to hold each element of the array. Following the colon, the `names` variable is a reference to the array to be processed. The code block is executed as many times as there are elements in the array.
- **The standard `for` loop example:** A counter, `idx`, is declared and initialized to 0. A boolean expression compares `idx` with the length of the `names` array. If `idx < names.length`, the code block is executed. `idx` is incremented by one at the end of each code block.
- Within the code block, `idx` is used as the array index.

The output for each statement is the same.

do/while Loop to Find the Factorial Value of a Number

```
1 // Finds the product of a number and all integers below it
2 static void factorial(int target) {
3     int save = target;
4     int fact = 1;
5     do {
6         fact *= target--;
7     }while(target > 0);
8     System.out.println("Factorial for "+save+": "+ fact);
9 }
```

Executed once before evaluating the condition

Outputs for two different targets:

Factorial value for 5: 120

Factorial value for 6: 720



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

The do/while loop is a slight variation on the while loop.

The example above shows a do/while loop that determines the factorial value of a number, called target. The factorial value is the product of an integer, multiplied by each positive integer smaller than itself. For example if the target parameter is 5, this method multiples $1 * 5 * 4 * 3 * 2 * 1$ resulting in a factorial value of 120.

do/while loops are not used as often as while loops. The code above could be rewritten as a while loop like this:

```
while (target > 0) {
    fact *= target--;
}
```

The decision to use a do/while loop instead of a while loop usually relates to code readability.

Coding a do/while Loop

Syntax:

```
do {  
    code_block; }  
    while (boolean_expression); // Semicolon is mandatory.
```

This block executes at least once.



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

In a do/while loop, the condition (shown at the bottom of the loop) is evaluated *after* the code block has already been executed once. If the condition evaluates to true, the code block is repeated continually until the condition returns false.

The *body of the loop* is, therefore, processed at least once. If you want the statement or statements in the body to be processed at least once, use a do/while loop instead of a while or for loop.

Comparing Loop Constructs

- Use the `while` loop to iterate indefinitely through statements and to perform the statements zero or more times.
- Use the standard `for` loop to step through statements a predefined number of times.
- Use the enhanced `for` loop to iterate through the elements of an `Array` or `ArrayList` (discussed later).
- Use the `do/while` loop to iterate indefinitely through statements and to perform the statements *one* or more times.



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

The continue Keyword

There are two keywords that enable you to interrupt the iterations in a loop of any type:

- `break` causes the loop to exit. *
- `continue` causes the loop to skip the current iteration and go to the next.

```
01 for (int idx = 0; idx < names.length; idx++){  
02     if (names[idx].equalsIgnoreCase("Unavailable"))  
03         continue;  
04     System.out.println(names[idx]);  
05 }
```

* Or any block of code to exit



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

- `break` allows you to terminate an execution of a loop or switch and skip to the first line of code following the end of the relevant loop or switch block.
- `continue` is used only within a loop. It causes the loop to skip the current iteration and move on to the next. This is shown in the code example above. The `for` loop iterates through the elements of the `names` array. If it encounters an element value of “Unavailable”, it does not print out that value, but skips to the next array element.

Exercise 11-3: Processing an Array of Items

1. Open the project **Exercise_11-3** in NetBeans:

In the `ShoppingCart` class:

2. Code the `displayTotal` method. Use a standard `for` loop to iterate through the `items` array.
3. If the current item is out of stock (call the `isOutOfStock` method of the item), skip to the next loop iteration.
4. If it is not out of stock, add the item price to a total variable that you declare and initialize before the `for` loop.
5. Print the Shopping Cart total with a suitable label.



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

Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- The `ArrayList` class



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



Nesting Loops

All types of loops can be nested within the body of another loop. This is a powerful construct used to:

- Process multidimensional arrays
- Sort or manipulate large amounts of data

How it works:

1st iteration of outer loop triggers:

 Inner loop

2nd iteration of outer loop triggers:

 Inner loop

3rd iteration of outer loop triggers:

 Inner loop

and so on...



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



Nested for Loop

Example: Print a table with 4 rows and 10 columns:

```
01 int height = 4, width = 10;
02
03 for(int row = 0; row < height; row++) {
04     for (int col = 0; col < width; col++) {
05         System.out.print("@");
06     }
07     System.out.println();
08 }
```

Output:

```
run:
@@@@@@@
@@@@@@@
@@@@@@@
@@@@@@@
BUILD SUCCESSFUL (total time: 0 seconds)
```



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

The code in the slide shows a simple nested loop to output a block of @ symbols with height and width given in the initial local variables.

- The outer `for` loop produces the rows. It loops four times.
- The inner `for` loop prints the columns for a given row. It loops ten times.
- Notice how the outer loop prints a new line to start a new row, whereas the inner loop uses the `print` method of `System.out` to print an @ symbol for every column. (Remember that, unlike `println`, `print` does not generate a new line.)
- The output is shown at the bottom: a table containing four rows of ten columns.

Nested while Loop

Example:

```
01 String name = "Lenny";
02 String guess = "";
03 int attempts = 0;
04 while (!guess.equalsIgnoreCase(name)) {
05     guess = "";
06     while (guess.length() < name.length()) {
07         char asciiChar = (char) (Math.random() * 26 + 97);
08         guess += asciiChar;
09     }
10     attempts++;
11 }
12 System.out.println(name+" found after "+attempts+" tries!");
```

Output:

Lenny found after 20852023 tries!



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

The nested `while` loop in the above example is a little more complex than the previous `for` example. The nested loop tries to guess a name by building a String of the same length completely at random.

- Looking at the inner loop first, the code initializes `char asciiChar` to a lowercase letter randomly. These `chars` are then added to `String guess`, until that `String` is as long as the `String` that it is being matched against. Notice the convenience of the concatenation operator here, allowing concatenation of a `String` and a `char`.
- The outer loop tests to see whether `guess` is the same as a lowercase version of the original name. If it is not, `guess` is reset to an empty `String` and the inner loop runs again, usually millions of times for a five-letter name. (Note that names longer than five letters will take a very long time.)

Processing a Two-Dimensional Array

Example: Quarterly Sales per Year

```
01 int sales[][] = new int[3][4];
02 initArray(sales); //initialize the array
03 System.out.println
04 ("Yearly sales by quarter beginning 2010:");
05 for(int i=0; i < sales.length; i++){
06     for(int j=0; j < sales[i].length; j++){
07         System.out.println("\tQ"+(j+1)+" "+sales[i][j]);
08     }
09     System.out.println();
10 }
```



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

Output from Previous Example

```
Yearly sales by quarter beginning 2010:  
Q1 36631  
Q2 62699  
Q3 60795  
Q4 11975  
  
Q1 72535  
Q2 37363  
Q3 20527  
Q4 36670  
  
Q1 3195  
Q2 98608  
Q3 21433  
Q4 98519
```



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

Quiz



enable you to check and recheck a decision to execute and re-execute a block of code.

- a. Classes
- b. Objects
- c. Loops
- d. Methods



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

Quiz



Which of the following loops always executes at least once?

- a. The `while` loop
- b. The nested `while` loop
- c. The `do/while` loop
- d. The `for` loop



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

Answer: c

Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- **The `ArrayList` class**



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



ArrayList Class

Arrays are not the only way to store lists of related data.

- ArrayList is one of several list management classes.
- It has a set of useful methods for managing its elements:
 - add, get, remove, indexOf, and many others
- It can store *only objects*, not primitives.
 - Example: an ArrayList of Shirt objects:
 - shirts.add(shirt04);
 - Example: an ArrayList of String objects:
 - names.remove ("James");
 - Example: an ArrayList of ages:
 - ages.add(5) //NOT ALLOWED!
 - ages.add(new Integer(5)) // OK



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

The ArrayList is one of several list management classes included in the `java.util` package. The other classes of this package (often referred to as the “collections framework”) are covered in greater depth in the *Java SE Programming II* course.

- ArrayList is based on the Array object and has many useful methods for managing the elements. Some of these are listed above, and you will see examples of how to use them in an upcoming slide.
- An important thing to remember about ArrayList variables is that you cannot store primitive types (`int`, `double`, `boolean`, `char`, and so on) in them—only object types. If you need to store a list of primitives, use an Array, or store the primitive values in the corresponding object type as shown in the final example above.

Benefits of the ArrayList Class

- Dynamically resizes:
 - An ArrayList grows as you add elements.
 - An ArrayList shrinks as you remove elements.
 - You can specify an initial capacity, but it is not mandatory.
- Option to designate the object type it contains:

```
ArrayList<String> states = new ArrayList();
```

Contains only String objects

- Call methods on an ArrayList or its elements:

```
states.size(); //Size of list
```

```
states.get(49).length(); //Length of 49th element
```



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

For lists that are very dynamic, the ArrayList offers significant benefits such as:

- ArrayList objects dynamically allocate space as needed. This can free you from having to write code to:
 - Keep track of the index of the last piece of data added
 - Keep track of how full the array is and determine whether it needs to be resized
 - Increase the size of the array by creating a new one and copying the elements from the current one into it
- When you declare an ArrayList, you have the option of specifying the object type that will be stored in it using the diamond operator (<>). This technique is called “generics.” This means that when accessing an element, the compiler already knows what type it is. Many of the classes included in the collections framework support the use of generics.
- You may call methods on either the ArrayList or its individual elements.
 - Assume that all 50 US states have already been added to the list.
 - The examples show how to get the size of the list, or call a method on an individual element (such as the length of a String object).

Importing and Declaring an ArrayList

- You must `import java.util.ArrayList` to use an `ArrayList`.
- An `ArrayList` may contain any object type, including a type that you have created by writing a class.

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main (String[] args) {
        ArrayList<Shirt> myList;
    }
}
```

You may specify any object type.



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

- If you forget to import `java.util.ArrayList`, NetBeans will complain but also correctly suggest that you add the `import` statement.
- In the example above, the `myList` `ArrayList` will contain `Shirt` objects. You may declare that an array list contains any type of object.

Working with an ArrayList

```

01 ArrayList<String> names;
02 names = new ArrayList();
03
04 names.add("Jamie");
05 names.add("Gustav");
06 names.add("Alisa");
07 names.add("Jose");
08 names.add(2,"Prashant");
09
10 names.remove(0);
11 names.remove(names.size() - 1);
12 names.remove("Gustav");
13
14 System.out.println(names);

```

Declare an ArrayList of Strings.
 Instantiate the ArrayList.
 Initialize it.
 Modify it.



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

Declaring an ArrayList, you use the diamond operator (`<>`) to indicate the object type. In the example above:

- Declaration of the `names` ArrayList occurs on line 1.
- Instantiation occurs on line 2.

There are a number of methods to add data to the ArrayList. This example uses the `add` method, to add several String objects to the list. In line 8, it uses an overloaded `add` method that inserts an element at a specific location:

`add(int index, E element).`

There are also many methods available for manipulating the data. The example here shows just one method, but it is very powerful.

- `remove(0)` removes the first element (in this case, "Jamie").
- `remove(names.size() - 1)` removes the last element, which would be "Jose".
- `remove("Gustav")` removes an element that matches a specific value.
- You can pass the ArrayList to `System.out.println`. The resulting output is: [Prashant, Alisa]

Exercise 11-4: Working with an ArrayList

1. Open the project **Exercise_11-4**.
2. Create a String ArrayList with at least three elements.
 - Be sure to add the correct import statement.
 - Print the ArrayList and test your code.
3. Add a new element to the middle of the list.
 - **Hint:** Use the overloaded `add` method that takes an index number as one of the arguments.
 - Print the list again to see the effect.
4. Test for a particular value in the ArrayList and remove it.
 - **Hint:** Use the `contains` method. It returns a boolean and takes a single argument as the search criterion.
 - Print the list again.



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

Summary

In this lesson, you should have learned how to:

- Create a `java.time.LocalDateTime` object to show the current date and time
- Parse the `args` array of the `main` method
- Nest a `while` loop
- Develop and nest a `for` loop
- Code and nest a `do/while` loop
- Use an `ArrayList` to store and manipulate objects



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



Practices Overview

- 11-1: Iterating Through Data
- 11-2: Working with `LocalDateTime`



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



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

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

Using Inheritance



ORACLE®

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



Interactive Quizzes



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

Before you start today's lessons, test your knowledge by answering some quiz questions that relate to yesterday's lessons. Open the Quiz files by clicking the quizzes.html shortcut from the desktop of your VM. In the welcome page, JavaSEProgrammingI.html, click the links for Lessons 9, 10, and 11.

Objectives

After completing this lesson, you should be able to:

- Define inheritance in the context of a Java class hierarchy
- Create a subclass
- Override a method in the superclass
- Use the `super` keyword to reference the superclass
- Define polymorphism
- Use the `instanceof` operator to test an object's type
- Cast a superclass reference to the subclass type
- Explain the difference between abstract and non-abstract classes
- Create a class hierarchy by extending an abstract class



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



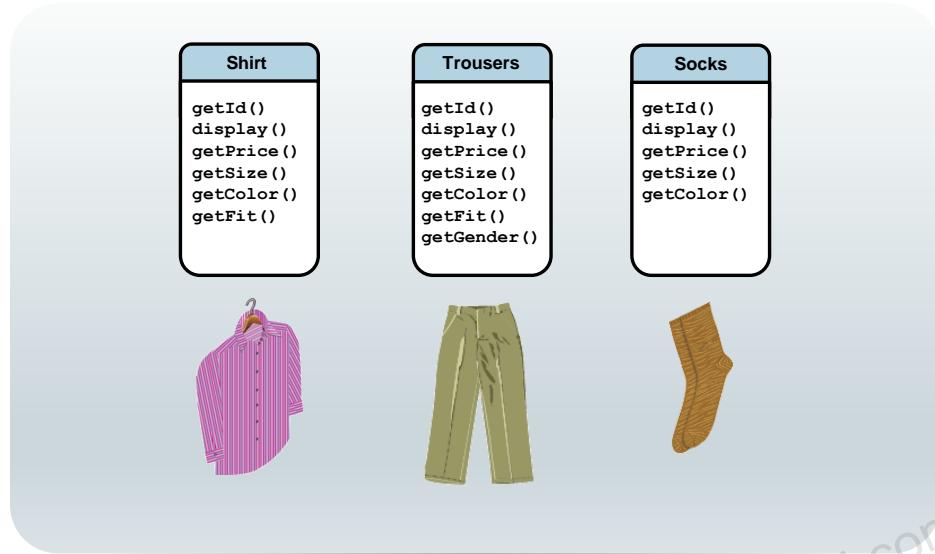
Duke's Choice Classes: Common Behaviors

Shirt	Trousers
<code>getId()</code> <code>getPrice()</code> <code>getSize()</code> <code>getColor()</code> <code>getFit()</code>	<code>getId()</code> <code>getPrice()</code> <code>getSize()</code> <code>getColor()</code> <code>getFit()</code> <code>getGender()</code>
<code>setId()</code> <code>setPrice()</code> <code>setSize()</code> <code>setColor()</code> <code>setFit()</code>	<code>setId()</code> <code>setPrice()</code> <code>setSize()</code> <code>setColor()</code> <code>setFit()</code> <code>setGender()</code>
<code>display()</code>	<code>display()</code>



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

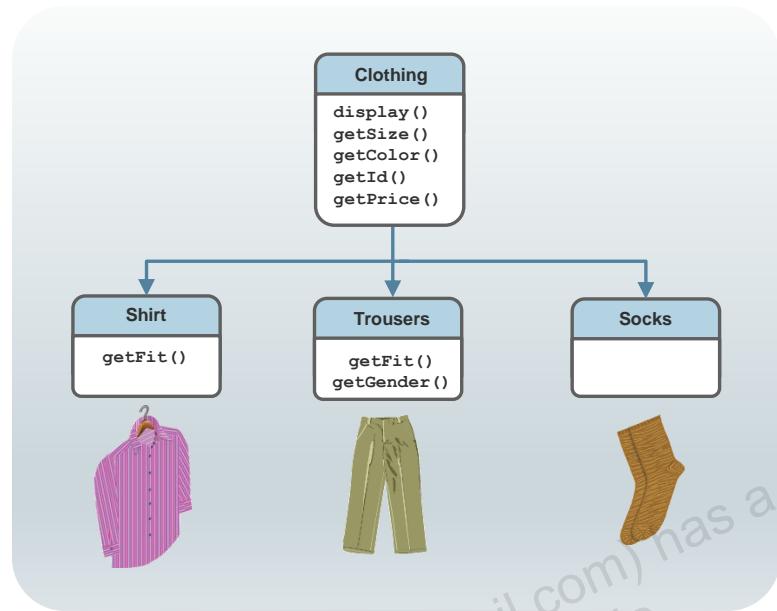
Code Duplication



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

Inheritance

- **Inheritance** allows one class to be derived from another.
- Fields and methods are written in one class, and then inherited by other classes.
 - There is less code duplication.
 - Edits are done in one location



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

You can eliminate code duplication in the classes by implementing inheritance. Inheritance enables programmers to put common members (fields and methods) in one class (the superclass) and have other classes (the subclasses) inherit these common members from this new class.

An object instantiated from a subclass behaves as if the fields and methods of the subclass were in the object. For example,

- The `Clothing` class can be instantiated and have the `getId` method called, even though the `Clothing` class does not contain `getId`. It is inherited from the `Item` class.
- The `Trousers` class can be instantiated and have the `display` method called even though the `Trousers` class does not contain a `display` method; it is inherited from the `Clothing` class.
- The `Shirt` class can be instantiated and have the `getPrice` method called, even though the `Shirt` class does not contain a `getPrice` method; it is inherited from the `Clothing` class.

Inheritance Terminology

- The term inheritance is inspired by biology
 - A child inherits properties and behaviors of the parent.
 - A child *class* inherits the fields and method of a parent *class*.
- The parent class is known as the **superclass**.
 - A superclass is the common location for fields and methods.
- The child class is known as the **subclass**. A subclass extends its superclass.
 - Subclasses share the same methods as the superclass.
 - Subclasses may have additional methods that aren't found in their superclass.
 - Subclasses may override the methods they inherit from their superclass.



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

Inheritance is a mechanism by which a class can be derived from another class, just as a child derives certain characteristics from the parent.

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes



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

Implementing Inheritance

```
public class Clothing {  
    public void display() {...}  
    public void setSize(char size) {...}  
}
```

```
public class Shirt extends Clothing {...}
```



Use the **extends** keyword.

```
Shirt myShirt = new Shirt();  
myShirt.setSize ('M');
```

This code works!



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

More Inheritance Facts

- A subclass has access to all of the public fields and methods of its superclass.
- A subclass may have unique fields and methods not found in the superclass.

```
subclass      superclass
public class Shirt extends Clothing {
    private int neckSize;
    public int getNeckSize() {
        return neckSize;
    }
    public void setNeckSize(int nSize) {
        this.neckSize = nSize;
    }
}
```



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

The subclass not only has access to all of the public fields and methods of the superclass, but it can also declare additional fields and methods that are specific to its own requirements.

Clothing Class: Part 1

```
01 public class Clothing {  
02     // fields given default values  
03     private int itemID = 0;  
04     private String desc = "-description required-";  
05     private char colorCode = 'U';  
06     private double price = 0.0;  
07  
08     // Constructor  
09     public Clothing(int itemID, String desc, char color,  
10                     double price) {  
11         this.itemID = itemID;  
12         this.desc = desc;  
13         this.colorCode = color;  
14         this.price = price;  
15     }  
16 }
```



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

The code in the slide shows the fields and the constructor for the Clothing superclass.

Shirt Class: Part 1

```
01 public class Shirt extends Clothing {  
03     private char fit = 'U';  
04  
05     public Shirt(int itemID, String description, char  
06                 colorCode, double price, char fit) {  
07         super(itemID, description, colorCode, price);  
08         this.fit = fit;  
09     }  
12     public char getFit() {  
13         return fit;  
14     }  
15     public void setFit(char fit) {  
16         this.fit = fit;  
17     }  
}
```

Reference to the superclass constructor
Reference to this object



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



Constructor Calls with Inheritance

```
public static void main(String[] args) {
    Shirt shirt01 = new Shirt(20.00, 'M');
}

public class Shirt extends Clothing {
    private char fit = 'U';

    public Shirt(double price, char fit) {
        super(price);           //MUST call superclass constructor
        this.fit = fit;
    }
}

public class Clothing{
    private double price;

    public Clothing(double price) {
        this.price = price;
    }
}
```



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

Inheritance and Overloaded Constructors



```
public class Shirt extends Clothing {  
    private char fit = 'U';  
  
    public Shirt(char fit){  
        this(15.00, fit); //Call constructor in same class  
    } //Constructor is overloaded  
  
    public Shirt(double price, char fit) {  
        super(price); //MUST call superclass constructor  
        this.fit = fit; //}  
  
public class Clothing{  
    private double price;  
  
    public Clothing(double price){  
        this.price = price;  
    } }
```



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

Use the `this` keyword to call another constructor within the same class. This is how you call an overloaded constructor.

Use the `super` keyword to call a constructor in the superclass. When you have overloaded subclass constructors, all of your constructors must eventually lead to the superclass constructor. If you call a superclass constructor, the call must be the first line of your constructor.

If your superclass constructors are overloaded, Java will know which superclass constructor you are calling based on the number, type, and order of arguments that you supply.

Exercise 12-1: Creating a Subclass, Part 1

1. Open the project `Exercise_12-1`.
2. Examine the `Item` class. Pay close attention to the overloaded constructor and also the `display` method.
3. In the `exercise_12_1` package, create a new class called `Shirt` that inherits from `Item`.
4. In the `Shirt` class, declare two private `char` fields: `size` and `colorCode`.
5. Create a constructor method that takes 3 args (`price`, `size`, `colorCode`). The constructor should:
 - Call the 2-arg constructor in the superclass
 - Pass a String literal for the `desc` arg ("Shirt").
 - Pass the `price` argument from this constructor.
 - Assign the `size` and `colorCode` fields.



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

In this exercise, you create the `Shirt` class, which extends the `Item` class.

Exercise 12-1: Creating a Subclass, Part 2

In the `ShoppingCart` class:

6. Declare and instantiate a `Shirt` object, using the 3-arg constructor.
7. Call the `display` method on the object reference.
 - Notice where the `display` method is actually coded.



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

In this exercise, you create the `Shirt` class, which extends the `Item` class.

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- **Overriding superclass methods**
- Introducing polymorphism
- Creating and extending abstract classes



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



More on Access Control

Access level modifiers determine whether other classes can use a particular field or invoke a particular method

- At the top level—public, or *package-private* (no explicit modifier).
- At the member level—public, private, protected, or *package-private* (no explicit modifier).

Stronger
access privileges



Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>No modifier</i>	Y	Y	N	N
private	Y	N	N	N



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

In the diagram, the first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member

Note: packages will be covered in greater detail in lesson titled “Deploying and Maintaining the Soccer Application”.

Overriding Methods

Overriding: A subclass implements a method that already has an implementation in the superclass.

Access Modifiers:

- The method can only be overridden if it is accessible from the subclass
- The method signature in the subclass cannot have a more restrictive (stronger) access modifier than the one in the superclass



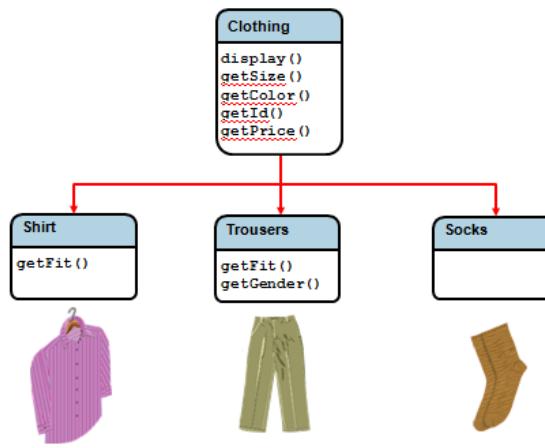
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Subclasses may implement methods that already have implementations in the superclass. In this case, the methods in the subclass are said to override the methods from the superclass.

- For example, although the `colorCode` field is in the superclass, the color choices may be different in each subclass. Therefore, it may be necessary to override the accessor methods (getter and setter methods) for this field in the individual subclasses.
- Although less common, it is also possible to override a field that is in the superclass. This is done by simply declaring the same field name and type in the subclass.

Review: Duke's Choice Class Hierarchy

Now consider these classes in more detail.



Clothing Class: Part 2

```
29 public void display() {  
30     System.out.println("Item ID: " + getItemID());  
31     System.out.println("Item description: " + getDesc());  
32     System.out.println("Item price: " + getPrice());  
33     System.out.println("Color code: " + getColorCode());  
34 }  
35 public String getDesc (){  
36     return desc;  
37 }  
38 public double getPrice() {  
39     return price;  
40 }  
41 public int getItemID () {  
42     return itemID;  
43 }  
44 protected void setColorCode(char color){  
45     this.colorCode = color; }
```

Assume that the remaining
get/set methods are included
in the class.



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

The code in the slide shows the display method for the Clothing superclass and also some of the getter methods and one of the setter methods. The remaining getter and setter methods are not shown here. Of course, this display method prints out only the fields that exist in Clothing. You would need to override the display method in Shirt in order to display all of the Shirt fields.

Shirt Class: Part 2

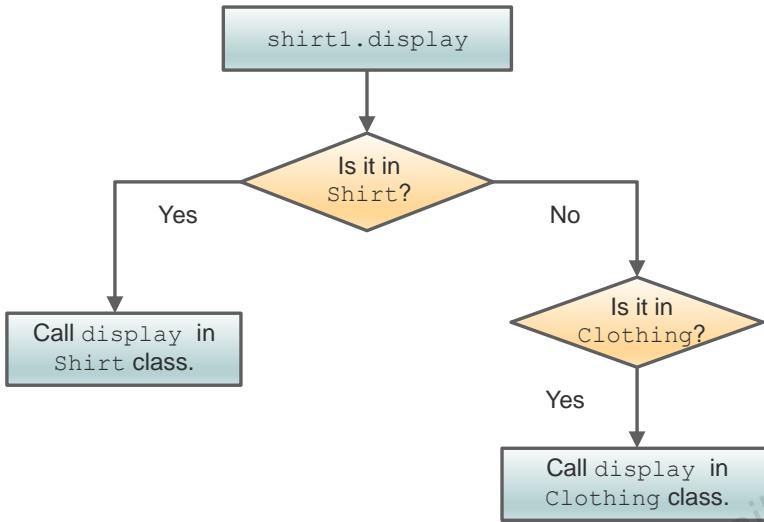
```
17 // These methods override the methods in Clothing
18 public void display() {
19     System.out.println("Shirt ID: " + getItemID());
20     System.out.println("Shirt description: " + getDesc());
21     System.out.println("Shirt price: " + getPrice());
22     System.out.println("Color code: " + getColorCode());
23     System.out.println("Fit: " + getFit());
24 }
25
26 protected void setColorCode(char colorCode) {
27     //Code here to check that correct codes used
28     super.setColorCode(colorCode);
29 }
30}
```

Call the superclass's version of `setColorCode`.



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

Overriding a Method: What Happens at Run Time?



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

The `shirt01.display` code is called. The Java VM:

- Looks for `display` in the `Shirt` class
 - If it is implemented in `Shirt`, it calls the `display` in `Shirt`.
 - If it is not implemented in `Shirt`, it looks for a parent class for `Shirt`.
- If there is a parent class (`Clothing` in this case), it looks for `display` in that class.
 - If it is implemented in `Clothing`, it calls `display` in `Clothing`
 - If it is not implemented in `Clothing`, it looks for a parent class for `Clothing`... and so on.

This description is not intended to exactly portray the mechanism used by the Java VM, but you may find it helpful in thinking about which method implementation gets called in various situations.

Exercise 12-2: Overriding a Method in the Superclass

1. Open `Exercise_12-2` or continue editing `Exercise_12-1`.

In the `Shirt` class:

2. Override the `display` method to do the following:
 - Call the superclass's `display` method.
 - Print the `size` field and the `colorCode` field.
3. Run the code. Do you see a different display than you did in the previous exercise?



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

In this exercise, you override a method the `display` method to show the additional fields from the `Shirt` class.

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- **Introducing polymorphism**
- Creating and extending abstract classes



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



Polymorphism

- Polymorphism means that the same message to two different objects can have different results.
 - “Good night” to a child means “Start getting ready for bed.”
 - “Good night” to a parent means “Read a bedtime story.”
- In Java, it means the same method is implemented differently by different classes.
 - This is especially powerful in the context of inheritance.
 - It relies upon the “**is a**” relationship.



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

You already used polymorphism when you overrode a method in the superclass, thereby allowing two classes to have the same method name but with a different outcome. In this lesson, we will examine this relationship in more detail and also introduce some other ways of implementing polymorphism.

Superclass and Subclass Relationships



Use inheritance only when it is completely valid or unavoidable.

- Use the “*is a*” test to decide whether an inheritance relationship makes sense.
- Which of the phrases below expresses a valid inheritance relationship within the Duke’s Choice hierarchy?



- A Shirt *is a* piece of Clothing.
- A Hat *is a* Sock.
- Equipment *is a* piece of Clothing.
- Clothing and Equipment *are* Items.



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

In this lesson, you have explored inheritance through an example:

- In the Duke’s Choice shopping cart, shirts, trousers, hats, and socks are all types of clothing. So `Clothing` is a good candidate for the superclass to these subclasses (types) of clothing.
- Duke’s Choice also sells equipment, but a piece of equipment is *not* a piece of clothing. However, clothing and equipment are both items, so `Item` would be a good candidate for a superclass for these classes.

Using the Superclass as a Reference

So far, you have referenced objects only with a reference variable of the same class:

- To use the `Shirt` class as the reference type for the `Shirt` object:

```
Shirt myShirt = new Shirt();
```

- But you can also use the superclass as the reference:

```
Clothing garment1 = new Shirt();
Clothing garment2 = new Trousers();
```

Shirt is a (type of) Clothing.
Trousers is a (type of) Clothing.



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

A very important feature of Java is this ability to use not only the class itself but any superclass of the class as its reference type. In the example shown in the slide, notice that you can refer to both a `Shirt` object and a `Trousers` object with a `Clothing` reference. This means that a reference to a `Shirt` or `Trousers` object can be passed into a method that requires a `Clothing` reference. Or a `Clothing` array can contain references to `Shirt`, `Trousers`, or `Socks` objects as shown below.

- `Clothing[] clothes = {new Shirt(), new Shirt(), new Trousers(), new Socks()};`

Polymorphism Applied

```
Clothing c1 = new ??();  
c1.display();  
c1.setColorCode('P');
```

c1 could be a Shirt, Trousers, or Socks object.

The method will be implemented differently on different types of objects. For example:

- Trousers objects show more fields in the display method.
- Different subclasses accept a different subset of valid color codes.



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

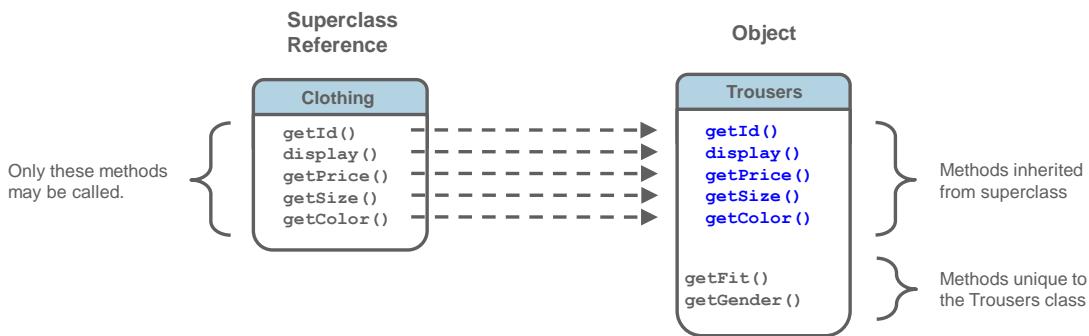
Polymorphism is achieved by invoking one of the methods of the superclass—in this example, the `Clothing` class.

This is a polymorphic method call because the runtime engine does not know, or *need* to know, the type of the object (sometimes called the *runtime* type). The correct method—that is, the method of the actual object—will be invoked.

In the example in the slide, the object could be any subclass of `Clothing`. Recall that some of the subclasses of `Clothing` implemented the `display` and `setColorCode` methods, thereby overriding those methods in the `Clothing` class.

Here you begin to see the benefits of polymorphism. It reduces the amount of duplicate code, and it allows you to use a common reference type for different (but related) subclasses.

Accessing Methods Using a Superclass Reference

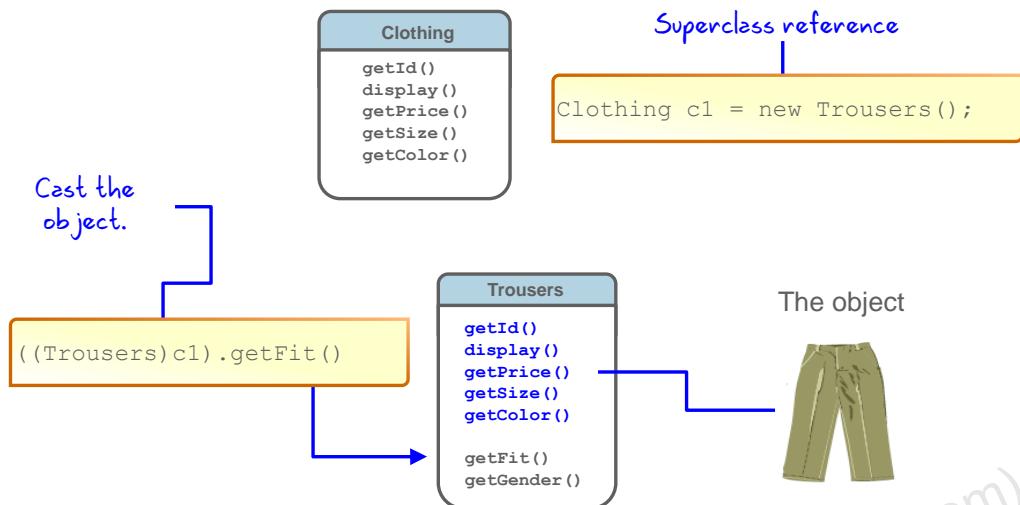


```
Clothing c1 = new Trousers();
c1.getId();   OK
c1.display(); OK
c1.getFit();  NO!
```

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



Casting the Reference Type



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

instanceof Operator

Possible casting error:

```
public static void displayDetails(Clothing cl) {  
    cl.display();  
    char fitCode = ((Trousers)cl).getFit();  
    System.out.println("Fit: " + fitCode);  
}
```

What if `cl` is not a Trousers object?

`instanceof` operator used to ensure there is no casting error:

```
public static void displayDetails(Clothing cl) {  
    cl.display();  
    if (cl instanceof Trousers) {  
        char fitCode = ((Trousers)cl).getFit();  
        System.out.println("Fit: " + fitCode);  
    }  
    else { // Take some other action }  
}
```

instanceof returns true if `cl` is a Trousers object.



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

The first code example in the slide shows a method that is designed to receive an argument of type `Clothing`, and then cast it to `Trousers` to invoke a method that exists only on a `Trousers` object. But it is not possible to know what object type the reference, `cl`, points to. And if it is, for example, a `Shirt`, the attempt to cast it will cause a problem. (It will throw a `ClassCastException`. Throwing exceptions is covered in the lesson titled “Handling Exceptions.”)

You can code around this potential problem with the code shown in the second example in the slide. Here the `instanceof` operator is used to ensure that `cl` is referencing an object of type `Trousers` before the cast is attempted.

If you think your code requires casting, be aware that there are often ways to design code so that casting is not necessary, and this is usually preferable. But if you do need to cast, you should use `instanceof` to ensure that the cast does not throw a `ClassCastException`.

Exercise 12-3: Using the instanceof Operator, Part 1

1. Open `Exercise_12-3` or continue editing `Exercise_12-2`.

In the `Shirt` class:

2. Add a public `getColor` method that converts the `colorCode` field into the corresponding color name:
 - Example: 'R' = "Red"
 - Include at least 3 `colorCode/color` combinations.
3. Use a `switch` statement in the method and return the color String.



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

Exercise 12-3: Using the instanceof Operator, Part 2

In the ShoppingCart class:

4. Modify the Shirt object's declaration so that it uses an Item reference type instead.
5. Call the display method of the object.
6. Use instanceof to confirm that the object is a Shirt.
 - If it is a Shirt:
 - Cast the object to a Shirt and call the getColor method, assigning the return value to a String variable.
 - Print out the color name using a suitable label.
 - If it is not a Shirt, print a message to that effect.
7. Test your code. You can test the non-Shirt object condition by instantiating an Item object instead of a Shirt object.



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

In this exercise, you use the instanceof operator to test the type of an object before casting it to that type.

Topics

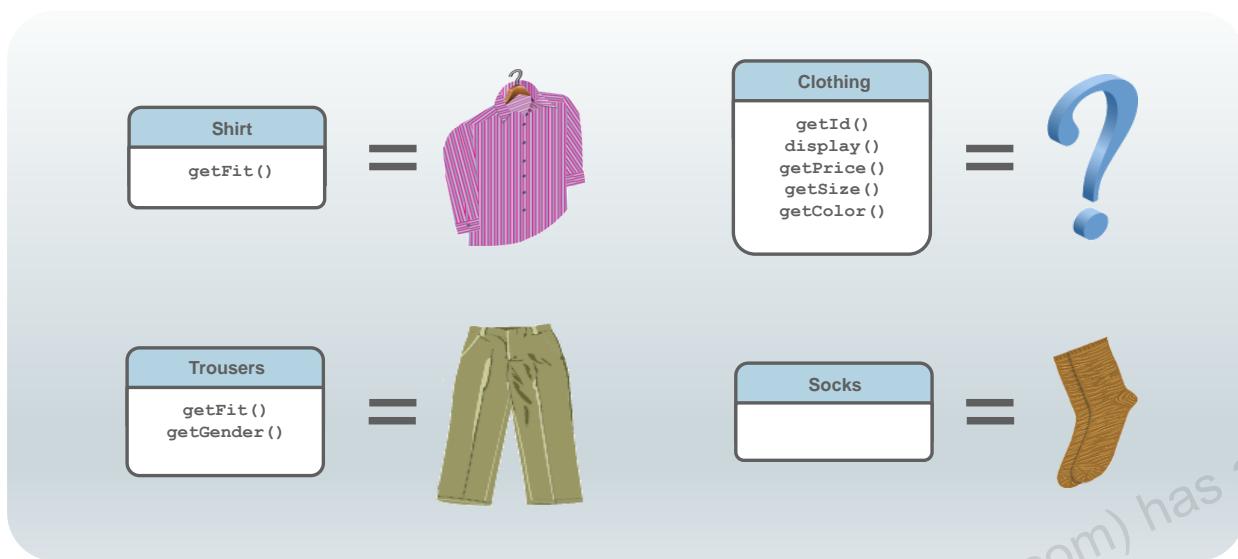
- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes



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



Abstract Classes



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

Abstract Classes

Use the `abstract` keyword to create a special class that:

- Cannot be instantiated
- May contain concrete methods
- May contain abstract methods that **must** be implemented later by any non-abstract subclasses



`Clothing cloth01 = new Clothing()`

```
public abstract class Clothing{
    private int id;

    public int getId() {
        return id;
    }

    public abstract double getPrice();
    public abstract void display();
}
```

Concrete method

Abstract methods



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

An abstract class cannot be instantiated. In fact, in many cases it would not make sense to instantiate them (Would you ever want to instantiate a `Clothing`?). However these classes can add a helpful layer of abstraction to a class hierarchy. The abstract class imposes a requirement on any subclasses to implement all of its abstract methods. Think of this as a contract between the abstract class and its subclasses.

- The example above has a concrete method, `getId`. This method can be called from the subclass or can be overridden by the subclass.
- It also contains two abstract methods: `getPrice` and `display`. Any subclasses of `Clothing` must implement these two methods.

Extending Abstract Classes

```
public abstract class Clothing{  
    private int id;  
  
    public int getId(){  
        return id;  
    }  
    protected abstract double getPrice(); //MUST be implemented  
    public abstract void display(); } //MUST be implemented
```

```
public class Socks extends Clothing{  
    private double price;  
  
    protected double getPrice(){  
        return price;  
    }  
    public void display(){  
        System.out.println("ID: " +getId());  
        System.out.println("Price: $" +getPrice());  
    } }
```



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

The `Socks` class extends the `Clothing` class. The `Socks` class implements the abstract `getPrice` and `display` methods from the `Clothing` class. A subclass is free to call any of the concrete methods or newly implemented abstract methods from an abstract superclass, including within the implementation of an inherited abstract method, as shown by the call to `getID` and `getPrice` within the `display` method.

Summary

In this lesson, you should have learned the following:

- Define inheritance in the context of a Java class hierarchy
- Create a subclass
- Override a method in the superclass
- Use the `super` keyword to reference the superclass
- Define polymorphism
- Use the `instanceof` operator to test an object's type
- Cast a superclass reference to the subclass type
- Explain the difference between abstract and non-abstract classes
- Create a class hierarchy by extending an abstract class



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

Inheritance enables programmers to put common members (variables and methods) in one class and have other classes *inherit these common members* from this new class.

The class containing members common to several other classes is called the *superclass* or the *parent class*. The classes that inherit from, or extend, the superclass are called *subclasses* or *child classes*.

Inheritance also allows object methods and fields to be referred to by a reference, that is, the type of the object, the type of any of its superclasses, or an interface that it implements.

Abstract classes can also be used as a superclass. They cannot be instantiated, but by including abstract methods that must be implemented by the subclasses, they impose a specific public interface on the subclasses.

Practice Overview

- 12-1: Creating a Class Hierarchy
- 12-2: Creating a GameEvent Hierarchy



Using Interfaces



ORACLE®

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



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

Objectives

After completing this lesson, you should be able to:

- Override the `toString` method of the `Object` class
- Implement an interface in a class
- Cast to an interface reference to allow access to an object method
- Use the local variable type inference feature to declare local variables using `var`
- Write a simple lambda expression that consumes a `Predicate`



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



Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using local variable type inference
- Using the `List` interface
- Introducing lambda expressions



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



In this section, you will look at a few examples of interfaces found in the foundation classes.

The Object Class

The screenshot shows two JavaDoc pages side-by-side. The left page is for the `ArrayList` class, which is part of the `java.util` package and extends the `Object` class. A callout box points from the `Object` link in the `ArrayList` documentation to the right page. The right page is for the `Object` class, which is part of the `java.lang` package and implements the `Serializable` interface. It also extends the `Object` class. The `Object` class is described as the root of the class hierarchy, with every class having `Object` as a superclass. It includes a note about arrays implementing the methods of `Object`, a `toString` method example, and a history section mentioning it was introduced in version 1.0.

Module java.base
Package java.util
Class ArrayList

Type Parameters:
E - the type of elements contained in this collection

All Implemented Interfaces:
Serializable

Direct Known Subclasses:
java.util.ArrayList

Attributes:

Java lang.Object

java.util.AbstractCollection<E>
java.util.AbstractList<E>
java.util.ArrayList<E>

Module java.base
Package java.lang
Class Object

Java lang.Object

public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:
1.0

The Object class is the base class.



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

All classes have at the very top of their hierarchy the `Object` class. It is so central to how Java works that all classes that do not explicitly extend another class automatically extend `Object`.

So all classes have `Object` at the root of their hierarchy. This means that all classes have access to the methods of `Object`. Being the root of the object hierarchy, `Object` does not have many methods—only very basic ones that all objects must have.

An interesting method is the `toString` method. The `Object` `toString` method gives very basic information about the object; generally classes will override the `toString` method to provide more useful output. `System.out.println` uses the `toString` method on an object passed to it to output a string representation.



Calling the `toString` Method

```

1  public class Main {
2      public static void main(String[] args) {
3          // Output an Object to the console
4          System.out.println(new Object());
5
6          // Output this StringBuilder object to the console
7          System.out.println(new StringBuilder("Some text for StringBuilder"));
8
9          //Output a class that does not override the toString() method
10         System.out.println(new First());
11
12         //Output a class that *does* override the toString() method
13         System.out.println(new Second());
14
15     }
16 }
```

Object's `toString` method is used.

`StringBuilder` overrides Object's `toString` method.

First inherits Object's `toString` method.

Second overrides Object's `toString` method.

The output for the calls to the `toString` method of each object

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



All objects have a `toString` method because it exists in the `Object` class. But the `toString` method may return different results depending on whether or not that method has been overridden. In the example in the slide, `toString` is called (via the `println` method of `System.out`) on four objects:

- **An `Object` object:** This calls the `toString` method of the base class. It returns the name of the class (`java.lang.Object`), an @ symbol, and a hash value of the object (a unique number associated with the object).
- **A `StringBuilder` object:** This calls the `toString` method on the `StringBuilder` object. `StringBuilder` overrides the `toString` method that it inherits from `Object` to return a `String` object of the set of characters it is representing.
- **An object of type `First`, a test class:** `First` does not override the `toString` method, so the `toString` method called is the one that is inherited from the `Object` class.
- **An object of type `Second`, a test class:** `Second` is a class with one method named `toString`, so this overridden method will be the one that is called.

There is a case for re-implementing the `getDescription` method used by the `Clothing` classes to instead use an overridden `toString` method.

Overriding `toString` in Your Classes

Shirt class example

```
1 public String toString() {
2     return "This shirt is a " + desc + ";"
3     + " price: " + getPrice() + ","
4     + " color: " + getColor(getColorCode());
5 }
```

Output of `System.out.println(shirt)`:

- Without overriding `toString`
`examples.Shirt@73d16e93`
- After overriding `toString` as shown above
`This shirt is a T Shirt; price: 29.99, color: Green`



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

The code example here shows the `toString` method overridden in the `Shirt` class.

When you override the `toString` method, you can provide useful information when the object reference is printed.

Topics

- Polymorphism in the JDK foundation classes
- **Using Interfaces**
- Using local variable type inference
- Using the `List` interface
- Introducing lambda expressions



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



The Multiple Inheritance Dilemma

Can I inherit from *two* different classes? I want to use methods from both classes.

```
public class Red{  
    public void print(){  
        System.out.print("I am Red");  
    }  
}
```

```
public class Blue{  
    public void print(){  
        System.out.print("I am Blue");  
    }  
}
```

```
public class Purple extends Red, Blue{  
    public void printStuff() {  
        print();  
    }  
}
```

Which implementation
of print() will occur?



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

The Java Interface

- An interface is similar to an abstract class, 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.

```
1 public interface Printable {  
2     public void print();  
3 }
```

Implicitly
abstract

```
1 public class Shirt implements Printable {  
2     ...  
3     public void print(){  
4         System.out.println("Shirt description");  
5     }  
6 }
```

Implements the
print()
method.



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

When a class implements an interface, it enters into a contract with the interface to implement all of its abstract methods. Therefore, using an interface lets you enforce a particular public interface (set of public methods).

- In first example above, you see the declaration of the `Printable` interface. It contains only one method, the `print` method. Notice that there is no method block. The method declaration is just followed by a semicolon.
- In the second example, the `Shirt` class implements the `Printable` interface. The compiler immediately shows an error until you implement the `print` method.

Note: A method within an interface is assumed to be abstract unless it uses the `default`, `static`, or `private` keywords. Default methods are new as of Java 8. They're covered in more detail in the course *Java SE Programming II*.

No Multiple Inheritance of State

- Multiple Inheritance of methods is not a problem
- Multiple Inheritance of state is a big problem
 - Abstract classes may have instance and static fields.
 - Interface fields must be static final.

Key difference
between abstract
classes and interfaces

```
public abstract class Red{  
    public String color = "Red";  
}
```

```
public abstract class Blue{  
    public String color = "Blue";  
}
```

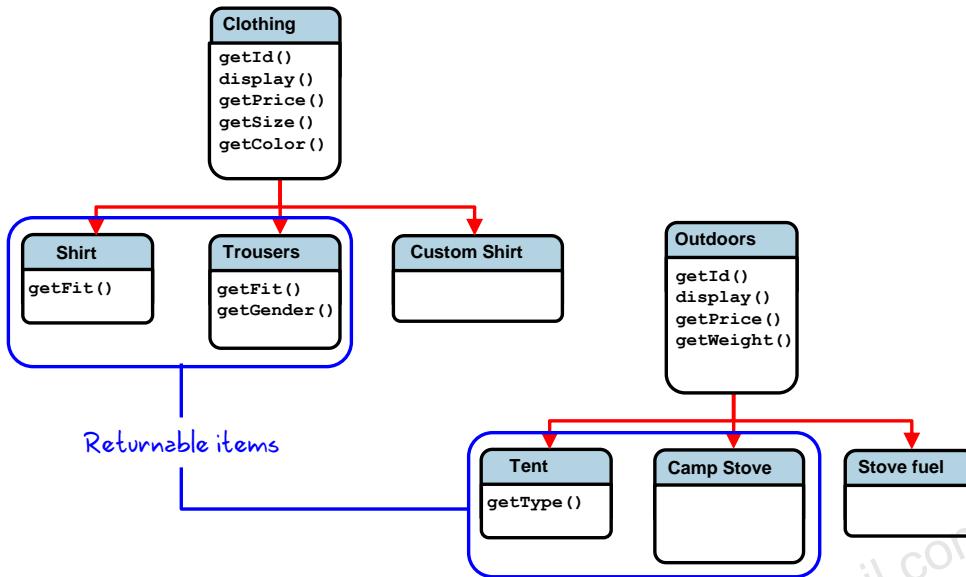
```
public class Purple extends Red, Blue{  
    public void printStuff() {  
        System.out.println(color);  
    }  
}
```

Which value of color
will print?



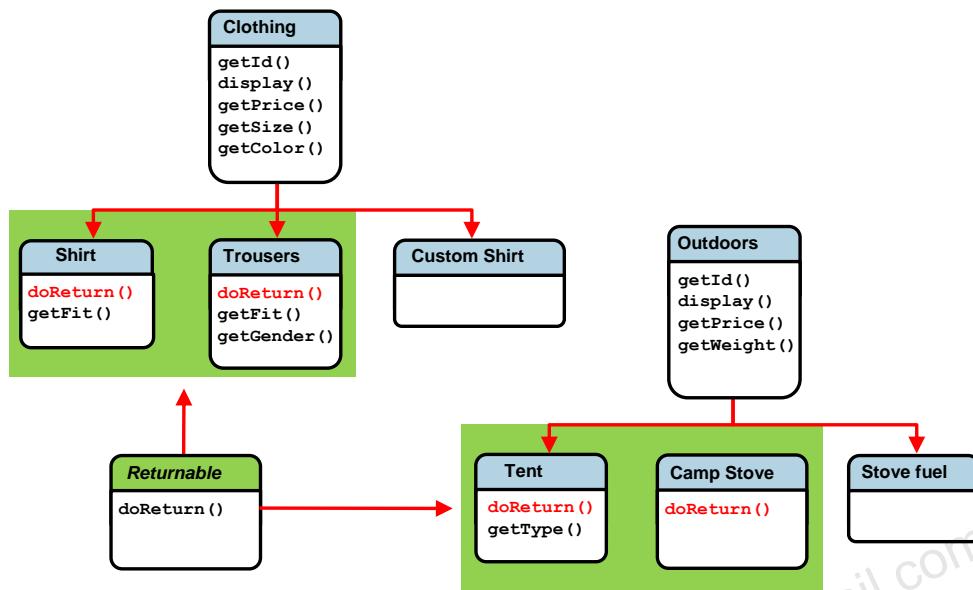
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Multiple Hierarchies with Overlapping Requirements



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

Using Interfaces in Your Application



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

Implementing the Returnable Interface

Returnable interface

```
01 public interface Returnable {  
02     public String doReturn(); Implicitly abstract method  
03 }
```

Shirt class

Now, Shirt 'is a' Returnable.

```
01 public class Shirt extends Clothing implements Returnable {  
02     public Shirt(int itemID, String description, char colorCode,  
03                  double price, char fit) {  
04         super(itemID, description, colorCode, price);  
05         this.fit = fit;  
06     }  
07     public String doReturn() { Shirt implements the method declared in Returnable.  
08         // See notes below  
09         return "Suit returns must be within 3 days";  
10     }  
11     ...< other methods not shown > ... } // end of class
```



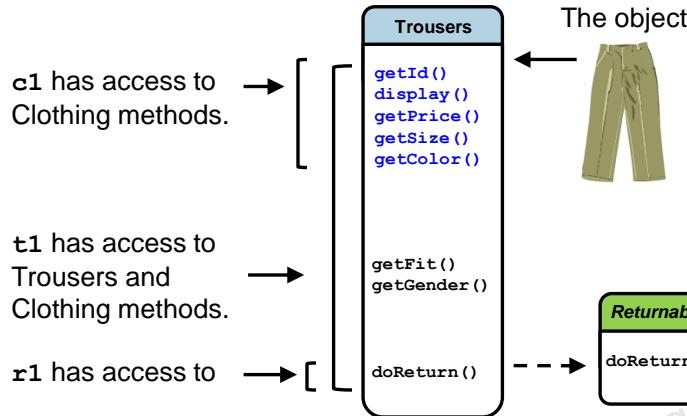
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

The code in this example shows the Returnable interface and the Shirt class. Notice that the abstract methods in the Returnable class are stub methods (that is, they contain only the method signature).

- In the Shirt class, only the constructor and the doReturn method are shown.
- The use of the phrase “implements Returnable” in the Shirt class declaration imposes a requirement on the Shirt class to implement the doReturn method. A compiler error occurs if doReturn is not implemented. The doReturn method returns a String describing the conditions for returning the item.
- Note that the Shirt class now has an “is a” relationship with Returnable. Another way of saying this is that Shirt *is a* Returnable.

Access to Object Methods from Interface

```
Clothing c1 = new Trousers();
Trousers t1 = new Trousers();
Returnable r1 = new Trousers();
```



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



Casting an Interface Reference

```
Clothing c1 = new Trousers();  
Trousers t1 = new Trousers();  
Returnable r1 = new Trousers();
```

- The Returnable interface does not know about Trousers methods:

```
r1.getFit() //Not allowed
```

- Use **casting** to access methods defined outside the interface.

```
((Trousers)r1).getFit();
```

- Use **instanceof** to avoid inappropriate casts.

```
if(r1 instanceof Trousers) {  
    ((Trousers)r1).getFit();  
}
```



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

If a method receives a Returnable reference and needs access to methods that are in the Clothing or Trousers class, the reference can be cast to the appropriate reference type.

Quiz

Which methods of an object can be accessed via an interface that it implements?

- a. All the methods implemented in the object's class
- b. All the methods implemented in the object's superclass
- c. The methods declared in the interface



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

Quiz

How can you change the reference type of an object?

- a. By calling `getReference`
- b. By casting
- c. By declaring a new reference and assigning the object



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



Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- **Using local variable type inference**
- Using the `List` interface
- Introducing lambda expressions



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



What is This Feature?

- Local variable type inference is a new language feature in Java 10.
- Use `var` to declare local variables.
- The compiler infers the datatype from the variable initializer.

Before Java 10

```
ArrayList list = new ArrayList<String>();
```

Datatype declared twice

Now

```
var list = new ArrayList<String>();
```

Datatype declared once



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

Benefits

- There's less boilerplate typing.
- Code is easier to read with variable names aligned.

```
String desc = "shirt";
ArrayList<String> list = new
ArrayList<String>();
int price = 20;
double tax = 0.05;
```

```
var desc = "shirt";
var list = new ArrayList<String>();
var price = 20;
var tax = 0.05;
```

- It won't break old code.
 - Keywords cannot be variables names.
 - `var` is not a keyword.
 - `var` is a reserved type name.
 - It's only used when the compiler expects a variable type.
 - Otherwise, you can use `var` as a variable name.

But it's a bad name...



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

You're starting to notice variable type declarations growing more complex. In larger scale applications seeing a declaration as "ArrayList<String>" is only the beginning. With longer declarations, it becomes harder to read code and perceive functionality. Not only does the var keyword simplify the declarations, if variables are declared near each other, their names align for easier readability.

Where Can it be Used?

Yes

- Local variables
`var x = shirt1.toString();`
- for loop
`for(var i=0; i<10; i++)`
- for-each loop
`for(var x : shirtArray)`

No

- Declaration without an initial value
`var price;`
- Declaration and initialization with a null value
`var price = null;`
- Fields
`public var price;`
- Parameters
`public void setPrice(var price){}`
- Method return types
`public var getPrice(){
 return price;
}`



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

Additionally, `var` cannot be used in these scenarios:

- Compound declarations
`var price=19.95, tax=0.08;`
- Array initializer
`var price = {9.99, 19.95, 15.00};`

Why is The Scope So Narrow?

- Larger scopes increase the potential for issues or uncertainty in inferences.
- To prevent issues, Java restricts the usage of `var`.

```
public var getSomething(var something) {  
    return something;  
}
```

How should this compile?
something could be anything!



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

Exercise 13-1: Local Variable Type Inference

1. Open the project **Exercise_13-1** in NetBeans.
2. Edit `TestClass.java`.
3. Replace the variable declarations with the `var` variable type inference feature in the following cases. Note which cases produce an error.
 - As a local variables
 - As a reference to Collection
 - In the enhanced for loop
 - As the index counter in the traditional for loop
 - Saving the returned value from a method
 - As a method return type



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

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using local variable type inference
- **Using the List interface**
- Introducing lambda expressions



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



The Collections Framework

The collections framework is located in the `java.util` package. The framework is helpful when working with lists or collections of objects. It contains:

- Interfaces
- Abstract classes
- Concrete classes (Example: `ArrayList`)



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

You were introduced to the `java.util` package when you learned to use the `ArrayList` class. Most of the classes and interfaces found in `java.util` provide support for working with collections or lists of objects. You will consider the `List` interface in this section.

The collections framework is covered in much more depth in the *Java SE Programming II* course.

ArrayList Example

Class `ArrayList<E>`

```
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>
```

Type Parameters:
E - the type of elements in this list

All Implemented Interfaces:

```
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess
```

Direct Known Subclasses:

```
AttributeList, RoleList, RoleUnresolvedList
```

```
public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList **extends** AbstractList, which in turn extends AbstractCollection.

ArrayList **implements** a number of interfaces.

The List interface is principally what is used when working with ArrayList.



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

Some of the best examples of inheritance and the utility of Interface and Abstract types can be found in the Java API.

List Interface

```

Module: java.base
Package: java.util
Interface List<E>

Type Parameters:
E - the type of elements in this list

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
ObservableList<E>, ObservableListValue<E>, WritableListValue<E>

All Known Implementing Classes:
AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, FilteredList,
LinkedList, ListBinding, ListExpression, ListProperty, ListPropertyBase,
ModifiableObservableListBase, ObservableListBase, ReadOnlyListProperty, ReadOnlyListPropertyBase,
ReadOnlyListWrapper, RoleList, RoleUnresolvedList, SimpleListProperty, SortedList, Stack,
TransformationList, Vector

```

Many classes implement the List interface.

All of these object types can be assigned to a List variable:

```

1   ArrayList words = new ArrayList<String>();
2   List mylist = words;

```

```

1   var words = new ArrayList();
2   var mylist = words;

```

Using local variable type inference



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

The List interface is implemented by many classes. This means that any method that requires a List may actually be passed a List reference to any objects of these types (but not the abstract classes, because they cannot be instantiated). For example, you might pass an ArrayList object, using a List reference. Likewise, you can assign an ArrayList object to a List reference variable as shown in the code example above.

- In line 1, an ArrayList of String objects is declared and instantiated using the reference variable words.
- In line 2, the words reference is assigned to a variable of type List<String>.

Example: Arrays.asList

The `java.util.Arrays` class has many static utility methods that are helpful in working with arrays.

- Converting an array to a List:

```
1  String[] nums = {"one", "two", "three"};  
2  List<String> myList = Arrays.asList(nums);
```

List objects can be of many different types. What if you need to invoke a method belonging to `ArrayList`?

```
mylist.replaceAll()  This works! replaceAll comes  
from List.  
mylist.removeIf()  Error! removeIf comes from  
Collection (superclass of ArrayList).
```



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

As you saw on the previous slide, you can store an `ArrayList` object reference in a variable of type `List` because `ArrayList` implements the `List` interface (therefore, `ArrayList` is a `List`).

Occasionally you need to convert an array to an `ArrayList`. How do you do that? The `Arrays` class is another very useful class from `java.util`. It has many static utility methods that can be helpful in working with arrays. One of these is the `asList` method. It takes an array argument and converts it to a `List` of the same element type. The example above shows how to convert an array to a `List`.

- In line 1, a `String` array, `nums`, is declared and initialized.
- In line 2, the `Arrays.asList` method converts the `nums` array to a `List`. The resulting `List` object is assigned to a variable of type `List<String>` called `myList`.

Recall that any object that implements the `List` interface can be assigned to a `List` reference variable. You can use the `myList` variable to invoke any methods that belong to the `List` interface (example: `replaceAll`). But what if you wanted to invoke a method belonging to `ArrayList` or one of its superclasses that is not part of the `List` interface (example: `removeIf`)? You would need a reference variable of type `ArrayList`.

Example: Arrays.asList

Converting an array to an ArrayList:

```
1 String[] nums = {"one", "two", "three"};  
2 List<String> myList = Arrays.asList(nums);  
3 ArrayList<String> myArrayList = new ArrayList(myList);  
  
or  
var myArrayList = new ArrayList(myList);
```

Shortcut:

```
1 String[] nums = {"one", "two", "three"};  
2 ArrayList<String> myArrayList = new ArrayList(Arrays.asList(nums));  
  
or  
var myArrayList = new ArrayList(Arrays.asList(nums));
```



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

Building upon the previous example, this slide example shows how to convert an array to an ArrayList.

- In the first example, the conversion is accomplished in three steps:
 - Line 1 declares the `nums` String array.
 - Line 2 converts the `nums` array to a `List` object, just as you saw on the previous slide.
 - Line 3 uses the `List` object to initialize a new `ArrayList`, called `myArrayList`. It does this using an overloaded constructor of the `ArrayList` class that takes a `List` object as a parameter.
- The second example reduces this code to two lines by using the `Arrays.asList(nums)` expression as the `List` argument to the `ArrayList` constructor.
- The `myArrayList` reference could be used to invoke the `removeIf` method you saw on the previous slide.

Exercise 13-2: Converting an Array to an ArrayList, Part 1

1. Open the project **Exercise_13-2** in NetBeans or create your own Java Main Class named TestClass
2. Convert the days array to an ArrayList.
 - Use Arrays.asList to return a List.
 - Use that List to initialize a new ArrayList.
 - Preferably do this all on one line.
3. Iterate through the ArrayList, testing to see if an element is "sunday".
 - If it is a "sunday" element, print it out, converting it to upper case.

Use String class methods:

 - public boolean equals (Object o);
 - public void toUpperCase();
 - Else, print the day anyway, but not in upper case.



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

In this exercise, you convert a String array to an ArrayList and manipulate list values.

Exercise 13-2: Converting an Array to an ArrayList, Part 2

4. After the `for` loop print out the `ArrayList`.

- While within the loop, was "sunday" printed in upper case?
- Was the "sunday" array element converted to upper case?
- Your instructor will explain what's going on in the next topic.



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

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using local variable type inference
- Using the `List` interface
- Introducing lambda expressions



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



Example: Modifying a List of Names

Suppose you want to modify a List of names, changing them all to uppercase. Does this code change the elements of the List?

```
1 String[] names = {"Ned", "Fred", "Jessie", "Alice", "Rick"};
2 List<String> mylist = new ArrayList(Arrays.asList(names));
3
4 // Display all names in upper case
5 for( var s: mylist){
6     System.out.print(s.toUpperCase() +", ");
7 }
8 System.out.println("After for loop: " + mylist);
```

Returns a new
String to print

Output:

```
NED, FRED, JESSIE, ALICE, RICK,
After for loop: [Ned, Fred, Jessie, Alice, Rick]
```

The list
elements are
unchanged.



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

You have already seen, in the previous exercise, that the technique shown here is not effective. The above code succeeds in printing out the list of names in uppercase, but it does not actually change the list element values themselves. The `toUpperCase` method used in the `for` loop simply changes the *local String* variable (`s` in the example above) to uppercase.

Remember that `String` objects are immutable. You cannot change them in place. All you can do is create a new `String` with the desired changes and then reassign the reference to point to the new `String`. You could do that here, but it would not be trivial.

A lambda expression makes this much easier!

Using a Lambda Expression with `replaceAll`

`replaceAll` is a default method of the `List` interface. It takes a lambda expression as an argument.

```
mylist.replaceAll( s -> s.toUpperCase() );  
System.out.println("List.replaceAll lambda: "+ mylist);
```

Lambda expression

Output:

```
List.replaceAll lambda: [NED, FRED, JESSIE, ALICE, RICK]
```



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

The `replaceAll` method belongs to the `List` interface. It is a default method, which means that it is a concrete method (not abstract) intended for use with a lambda expression. It takes a *particular type* of lambda expression as its argument. It iterates through the elements of the list, applying the result of the lambda expression to each element of the list.

The output of this code shows that the actual elements of the list were modified.

Lambda Expressions

Lambda expressions are like methods used as the argument for another method. They have:

- Input parameters
- A method body
- A return value

Long version:

```
mylist.replaceAll((String s) -> {return s.toUpperCase();});
```

Declare input parameter Arrow token Method body

Short version:

```
mylist.replaceAll( s -> s.toUpperCase() );
```



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

A lambda expression is a concrete method for an Interface expressed in a new way. A lambda expression looks very similar to a method definition. You can recognize a lambda expression by the use of an arrow token (->). A lambda expression:

- Has input parameters: These are seen to the left of the arrow token.
 - In the long version, the type of the parameter is explicitly declared.
 - In the short version, the type is inferred. The compiler derives the type from the type of the List in this example. (`List<String> mylist = ...`)
- Has a method body (statements): These are seen to the right of the arrow token. Notice that the long version even encloses the method body in braces, just as you would when defining a method. It explicitly uses the `return` keyword.
- Returns a value:
 - In the long version, the `return` statement is explicit.
 - In the short version it is inferred. Because the List was defined as a list of Strings, the `replaceAll` method is expecting a String to apply to each of its elements, so a return of String makes sense.

Note that you would probably never use the long version (although it does compile and run). You are introduced to this to make it easier for you to recognize the different method components that are present in a lambda expression.

The Enhanced APIs That Use Lambda

There are three enhanced APIs that take advantage of lambda expressions:

- `java.util.functions`
 - Provides target types for lambda expressions
- `java.util.stream`
 - Provides classes that support operations on streams of values
- `java.util`
 - Interfaces and classes that make up the collections framework
 - Enhanced to use lambda expressions
 - Includes List and ArrayList



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

A complete explanation of lambda expressions is beyond the scope of this course. You will, however, consider just a few of the target types for lambda expressions available in `java.util.functions`.

For a much more comprehensive treatment of lambda expressions, take the *Java SE 8 New Features* course, or the *Java SE Programming II* course.

Lambda Types

A lambda *type* specifies the type of expression a method is expecting.

- replaceAll takes a UnaryOperator type expression.

Method Summary	
All Methods	Instance Methods
Modifier and Type	Method and Description
default void replaceAll(UnaryOperator<E> operator)	Replaces each element of this list with the result of applying the operator to that element.

- All of the types do similar things, but have different inputs, statements, and outputs.



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

The lambda types can be viewed by looking at the `java.util.functions` package in the JDK API documentation. There are a great many of these, and they are actually interfaces. They specify the interface of the expression. Much like a method signature, they indicate the inputs, statements, and outputs for the expression.

The UnaryOperator Lambda Type

A `UnaryOperator` has a single input and returns a value of the same type as the input.

- Example: `String in – String out`
- The method body acts upon the input in some way, returning a value of the same type as the input value.
- `replaceAll` example:

The diagram shows the Java code `mylist.replaceAll(s -> s.toUpperCase());`. A yellow box highlights the lambda expression `s -> s.toUpperCase()`. Above the box, the text "UnaryOperator" is written in blue. Below the box, two vertical lines point to the "s" in "s.toUpperCase()". The left line is labeled "String input" and the right line is labeled "Method acts upon the string input, returning a string.".



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

A `UnaryOperator` has a single input and returns a value of the same type as the input. For example, it might take a single `String` value and return a `String` value, or it might take an `int` value and return an `int` value.

The method body acts upon the input in some way (possibly by calling a method), but must return the same type as the input value.

The code example here shows the `replaceAll` method that you saw earlier, which takes a `UnaryOperator` argument.

- A `String` is passed into the `UnaryOperator` (the expression). Remember that this method iterates through its list, invoking this `UnaryOperator` for each element in the list. The argument passed into the `UnaryOperator` is a single `String` element.
- The operation of the `UnaryOperator` calls `toUpperCase` on the string input.
- It returns a `String` value (the original `String` converted to uppercase).

The Predicate Lambda Type

A **Predicate type** takes a single input argument and returns a boolean.

- **Example:** String *in* – boolean *out*
- **removeIf** takes a **Predicate type expression**.
 - Removes all elements of the `ArrayList` that satisfy the **Predicate expression**

removeIf

```
public boolean removeIf(Predicate<? super E> filter)
```

- **Examples:**

```
mylist.removeIf (s -> s.equals("Rick"));  
mylist.removeIf (s -> s.length() < 5);
```



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

The **Predicate lambda expression type** takes a single input argument. The method body acts upon that argument in some way, returning a boolean.

In the examples shown here, `removeIf` is called on the `mylist` reference (an `ArrayList`). Iterating through the list and passing each element as a `String` argument into the `Predicate` expressions, it removes any elements resulting in a return value of `true`.

- In the first example, the `Predicate` uses the `equals` method of the `String` argument to compare its value with the string “Rick”. If it is equal, the `Predicate` returns `true`. The long version of the `Predicate` expression would look like this:
`mylist.removeIf ((String s) -> {return s.equals("Rick"); })`
- In the second example, the `Predicate` uses the `length()` method of the `String` argument, returning `true` if the string has less than 5 characters. The long version of this `Predicate` expression would look like this:
`mylist.removeIf ((String s) -> {return (s.length() < 5); })`

Exercise 13-3: Using a Predicate Lambda Expression

1. Open the project `Exercise_13-3`.

In the `ShoppingCart` class:

2. Examine the code. As you can see, the `items` list has been initialized with 2 shirts and 2 pairs of trousers.
3. In the `removeItemFromCart` method, use the `removeIf` method (which takes a `Predicate` lambda type) to remove all items whose description matches the `desc` argument.
4. Print the items list. Hint: the `toString` method in the `Item` class has been overloaded to return the item description.
5. Call the `removeItemFromCart` method from the main method. Try different description values, including ones that return `false`.
6. Test your code.



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

In this exercise, you use the `removeIf()` method to remove all items of the shopping cart whose description matches some value.

Summary

In this lesson, you should have learned the following:

- Override the `toString` method of the `Object` class
- Implement an interface in a class
- Cast to an interface reference to allow access to an object method
- Use local variable type inference feature to declare local variables using `var`
- Write a simple lambda expression that consumes a `Predicate`



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

Polymorphism means the same method name in different classes is implemented differently. The advantage of this is that the code that calls these methods does not need to know how the method is implemented. It knows that it will work in the way that is appropriate for that object.

Interfaces support polymorphism and are a very powerful feature of the Java language. A class that implements an interface has an “is a” relationship with the interface.

Practice Overview

- 13-1: Overriding the `toString` Method
- 13-2: Implementing an Interface
- 13-3: Using a Lambda Expression for Sorting



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

Handling Exceptions



ORACLE®

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



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

Objectives

After completing this lesson, you should be able to:

- Describe how Java handles unexpected events in a program
- List the three types of `Throwable` classes
- Determine what exceptions are thrown for any foundation class
- Describe what happens in the call stack when an exception is thrown and not caught
- Write code to handle an exception thrown by the method of a foundation class



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

Topics

- Handling exceptions: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors



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

What Are Exceptions?

Java handles unexpected situations using exceptions.

- Something unexpected happens in the program.
- Java doesn't know what to do, so it:
 - Creates an exception object containing useful information and
 - Throws the exception to the code that invoked the problematic method
- There are several different types of exceptions.



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

What if something goes wrong in an application? When an unforeseen event occurs in an application, you say “an exception was thrown.” There are many types of exceptions and, in this lesson, you will learn what they are and how to handle them.

Examples of Exceptions

- `java.lang.ArrayIndexOutOfBoundsException`
 - Attempt to access a nonexistent array index
- `java.lang.ClassCastException`
 - Attempt to cast an object to an illegal type
- `java.lang.NullPointerException`
 - Attempt to use an object reference that has not been instantiated
- You can create exceptions, too!
 - An exception is just a class.

```
public class MyException extends Exception { }
```



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

Here are just a few of the exceptions that Java can throw. You have probably seen one or more of the exceptions listed above while doing the practices or exercises in this class. Did you find the error message helpful when you had to correct the code?

Exceptions are classes. There are many of them included in the Java API. You can also create your own exceptions by simply extending the `java.lang.Exception` class. This is very useful for handling exceptional circumstances that can arise in the normal flow of an application. (Example: `BadCreditException`) This is not covered in this course, but you can learn more about it and other exception handling topics in the *Java SE Programming II* course.

Code Example

Coding mistake:

```
01 int[] intArray = new int[5];
02 intArray[5] = 27;
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at TestErrors.main(TestErrors.java:17)
```



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

This code shows a common mistake made when accessing an array. Remember that arrays are zero based (the first element is accessed by a zero index), so in an array like the one in the slide that has five elements, the last element is actually `intArray[4]`.

`intArray[5]` tries to access an element that does not exist, and Java responds to this programming mistake by throwing an `ArrayIndexOutOfBoundsException` exception. The information stored within the exception is printed to the console.

Another Example

Calling code in `main`:

```
19  TestArray myTestArray = new TestArray(5);
20  myTestArray.addElement(5, 23);
```

`TestArray` class:

```
13 public class TestArray {
14     int[] intArray;
15     public TestArray (int size) {
16         intArray = new int[size];
17     }
18     public void addElement(int index, int value) {
19         intArray[index] = value;
20     }
```

Stack trace:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
    at TestArray.addElement(TestArray.java:19)
    at TestException.main(TestException.java:20)
Java Result: 1
```



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

Here is a very similar example, except that this time the code that creates the array and tries to assign a value to a nonexistent element has been moved to a different class (`TestArray`). Notice how the error message, shown below, is almost identical to the previous example, but this time the methods `main` in `TestException`, and `addElement` in `TestArray` are explicitly mentioned in the error message. (In NetBeans the message is in red as it is sent to `System.err`).

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at TestArray.addElement(TestArray.java:19)
    at TestException.main(TestException.java:20)
Java Result: 1
```

This is called “the stack trace.” It is an unwinding of the sequence of method calls, beginning with where the exception occurred and going backwards.

In this lesson, you learn why that message is printed to the console. You also learn how you can catch or trap the message so that it is not printed to the console, and what other kinds of errors are reported by Java.

Types of Throwable classes

Exceptions are subclasses of `Throwable`. There are three main types of `Throwable`:

- `Error`
 - Typically an unrecoverable external error
 - `Unchecked`
- `RuntimeException`
 - Typically caused by a programming mistake
 - `Unchecked`
- `Exception`
 - Recoverable error
 - `Checked` (*Must be caught or thrown*)



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

As mentioned in the previous slide, when an exception is thrown, that exception is an object that can be passed to a `catch` block. There are three main types of objects that can be thrown in this way, and all are derived from the class, `Throwable`.

- Only one type, `Exception`, requires that you include a `catch` block to handle the exception. We say that `Exception` is a *checked exception*. You *may* use a `catch` block with the other types, but it is not always possible to recover from these errors anyway.

You learn more about `try/catch` blocks and how to handle exceptions in upcoming slides.

Error Example: OutOfMemoryError

Programming error:

```
01 ArrayList theList = new ArrayList();
02 while (true) {
03     String theString = "A test String";
04     theList.add(theString);
05     long size = theList.size();
06     if (size % 1000000 == 0) {
07         System.out.println("List has "+size/1000000
08             +" million elements!");
09     }
10 }
```

Output in console:

```
List now has 156 million elements!
List now has 157 million elements!
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```



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

OutOfMemoryError is an Error. Throwable classes of type Error are typically used for exceptional conditions that are external to the application and that the application usually cannot anticipate or recover from. In this case, although it is an external error, it was caused by poor programming.

The example shown here has an infinite loop that continually adds an element to an ArrayList, guaranteeing that the JVM will run out of memory. The error is thrown up the call stack, and because it is not caught anywhere, it is displayed in the console as follows:

```
List now has 156 million elements!
List now has 157 million elements!
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf((Arrays.java:2760)
    at java.util.Arrays.copyOf((Arrays.java:2734)
    at java.util.ArrayList.ensureCapacity(ArrayList.java:167)
    at java.util.ArrayList.add(ArrayList.java:351)
    at TestErrors.main(TestErrors.java:22)
```

Quiz



Which of the following objects are checked exceptions?

- a. All objects of type `Throwable`
- b. All objects of type `Exception`
- c. All objects of type `Exception` that are not of type `RuntimeException`
- d. All objects of type `Error`
- e. All objects of type `RuntimeException`



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

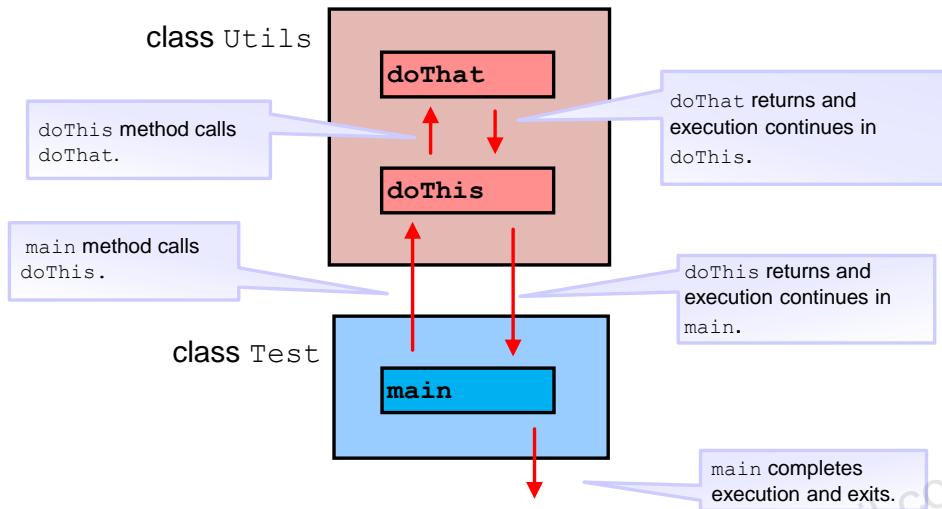
Topics

- Handling errors: an overview
- **Propagation of exceptions**
- Catching and throwing exceptions
- Multiple exceptions and errors



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

Normal Program Execution: The Call Stack



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

To understand exceptions, you need to think about how methods call other methods and how this can be nested deeply. The normal mode of operation is that a caller method calls a worker method, which in turn becomes a caller method and calls another worker method, and so on. This sequence of methods is called the *call stack*.

The example shown in the slide illustrates three methods in this relationship.

- The `main` method in the class `Test`, shown at the bottom of the slide, instantiates an object of type `Utils` and calls the method `doThis` on that object.
- The `doThis` method in turn calls a private method `doThat` on the same object.
- When a method either completes or encounters a return statement, it returns execution to the method that called it. So, `doThat` returns execution to `doThis`, `doThis` returns execution to `main`, and `main` completes and exits.

How Exceptions Are Thrown

Normal program execution:

1. Caller method calls worker method.
2. Worker method does work.
3. Worker method completes work and then execution returns to caller method.

When an exception occurs, this sequence changes. An exception object is thrown and either:

- Passed to a `catch` block in the current method
- or
- Thrown back to the caller method



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

An `Exception` is one of the subclasses of `Throwable`. `Throwable` objects are thrown either by the runtime engine or explicitly by the developer within the code. A typical thread of execution is described above: A method is invoked, the method is executed, the method completes, and control goes back to the calling method.

When an exception occurs, however, an `Exception` object containing information about what just happened is thrown. One of two things can happen at this point:

- The `Exception` object is caught by the method that caused it in a special block of code called a `catch` block. In this case, program execution can continue.
- The `Exception` is not caught, causing the runtime engine to throw it back to the calling method, and look for the exception handler there. Java runtime will keep propagating the exception up the method call stack until it finds a handler. If it is not caught in any method in the call stack, program execution will end and the exception will be printed to the `System.err` (possibly the console) as you saw previously.

Topics

- Handling errors: an overview
- Propagation of exceptions
- **Catching and throwing exceptions**
- Multiple exceptions and errors



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

Working with Exceptions in NetBeans

```

10  public class Utils {
11
12      public void doThis() {
13
14          System.out.println("Arrived in doThis()");
15          doThat();
16          System.out.println("Back in doThis()");
17
18      }
19
20      public void doThat() {
21          System.out.println("In doThat()");
22      }
23  }
24

```

No exceptions thrown;
nothing needs be done to
deal with them.

When you throw an
exception, NetBeans
gives you two options.

```

12  public void doThis() {
13
14      System.out.println("Arrived in doThis()");
15      doThat();
16      System.out.println("Back in doThis()");
17
18  }
19
20  public void doThat() {
21      System.out.println("In doThat()");
22
23      throw new Exception(); -- (Alt-Enter shows hints)
24  }
25

```

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



Here you can see the code for the `Utils` class shown in NetBeans.

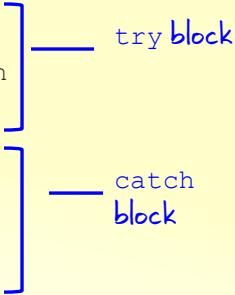
- In the first screenshot, no exceptions are thrown, so NetBeans shows no syntax or compilation errors.
- In the second screenshot, `doThat` explicitly throws an exception, and NetBeans flags this as something that needs to be dealt with by the programmer. As you can see from the tooltip, it gives the two options for handling the checked exception: Either catch it, using a `try/catch` block, or allow the method to be thrown to the calling method. If you choose the latter option, you must declare in the method signature that it throws an exception.

In these early examples, the `Exception` superclass is used for simplicity. However, as you will see later, you should not throw so general an exception. Where possible, when you catch an exception, you should try to catch a specific exception.

The try/catch Block

Option 1: Catch the exception.

```
try {  
    // code that might throw an exception  
    doRiskyCode();  
}  
catch (Exception e) {  
    String errMsg = e.getMessage();  
    // handle the exception in some way  
}
```



Option 2: Throw the exception.

```
public void doThat() throws Exception{  
    // code that might throw an exception  
    doRiskyCode();  
}
```



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

Here is a simple example illustrating both of the options mentioned in the previous slide.

- **Option 1: Catch the exception.**
 - The `try` block contains code that might throw an exception. For example, you might be casting an object reference and there is a chance that the object reference is not of the type you think it is.
 - The `catch` block catches the exception. It can be defined to catch a specific exception type (such as `ClassCastException`) or it can be the superclass `Exception`, in which case it would catch any subclass of `Exception`. The exception object will be populated by the runtime engine, so in the catch block, you have access to all the information bundled in it. By catching the exception, the program can continue although it could be in an unstable condition if the error is significant.
 - You may be able to correct the error condition within the `catch` block. For example, you could determine the type of the object and recast the reference to correct type.
- **Option 2: Declare the method to throw the exception:** In this case, the method declaration includes “`throws Exception`” (or it could be a specific exception, such as `ClassCastException`).

Program Flow When an Exception Is Caught

main method:

```
01 Utils theUtils = new Utils();
02 theUtils.doThis();
03 System.out.println("Back to main method");
```

3

Output

Utils class methods:

```
04 public void doThis() {
05     try{
06         doThat();
07     }catch(Exception e){
08         System.out.println("doThis - "
09             +" Exception caught: "+e.getMessage());
10    }
11 }
12 public void doThat() throws Exception{
13     System.out.println("doThat: Throwing exception");
14     throw new Exception("Ouch!");
15 }
```

2

1

```
run:
doThat: throwing Exception
doThis - Exception caught: Ouch!
Back to main method
BUILD SUCCESSFUL (total time: 0 seconds)
```



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

In this example, a `try/catch` block has been added to the `doThis` method. The slide also illustrates the program flow when the exception is thrown and caught by the calling method. The Output insert shows the output from the `doThat` method, followed by the output from the `catch` block of `doThis` and, finally, the last line of the main method.

main method code:

- In line 1, a `Utils` object is instantiated.
- In line 2, the `doThis` method of the `Utils` object is invoked.

Execution now goes to the `Utils` class:

- In line 6 of `doThis`, `doThat` is invoked from within a `try` block. Notice that in line 7, the `catch` block is declared to catch the exception.

Execution now goes to the `doThat` method:

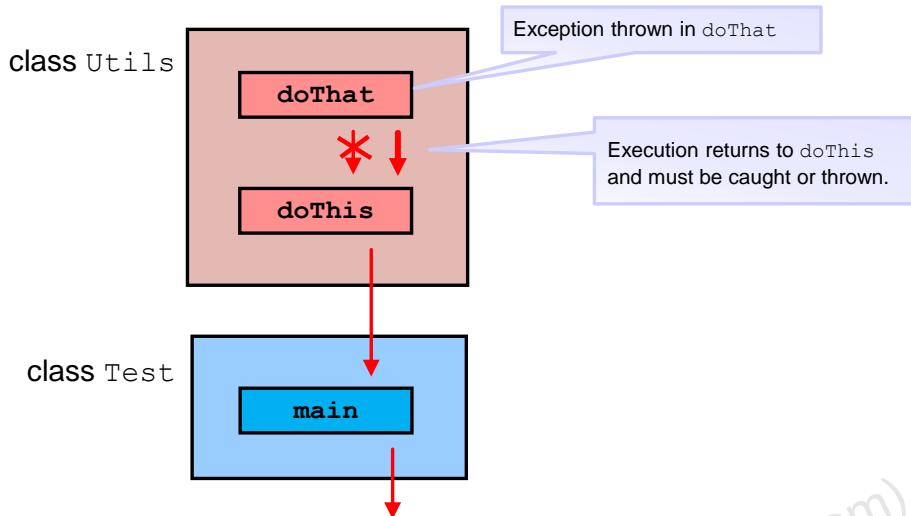
- In line 14, `doThat` explicitly throws a new `Exception` object.

Execution now returns to `doThis`:

- In line 8 of `doThis`, the exception is caught and the `message` property from the `Exception` object is printed. The `doThat` method completes at the end of the `catch` block.

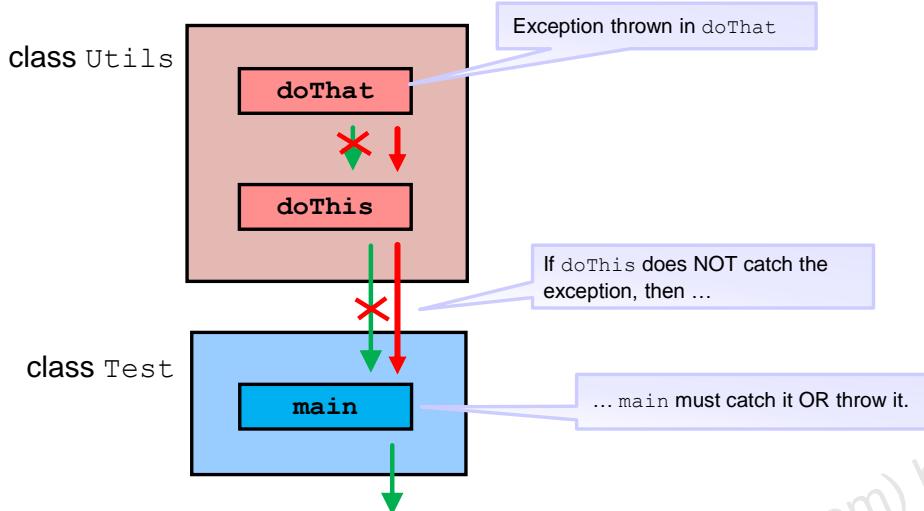
Execution now returns to the `main` method where line 3 is executed.

When an Exception Is Thrown



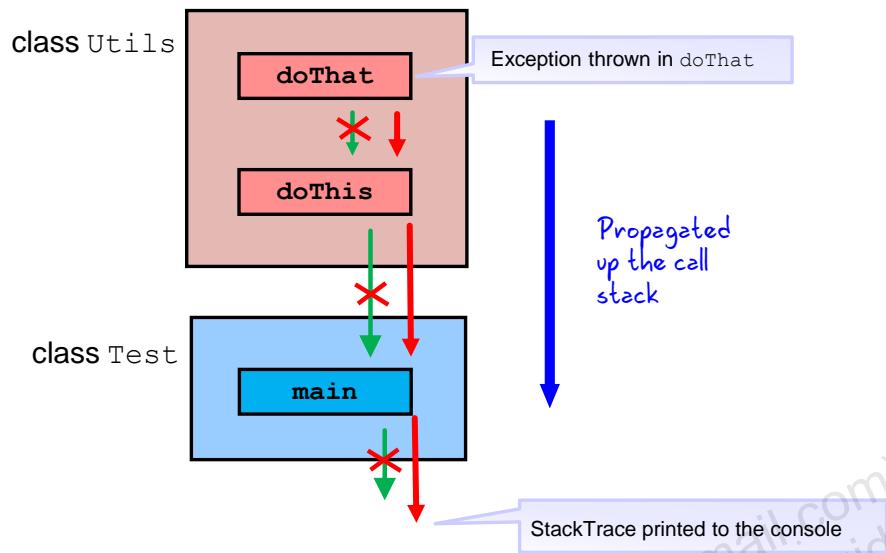
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Throwing Throwable Objects



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

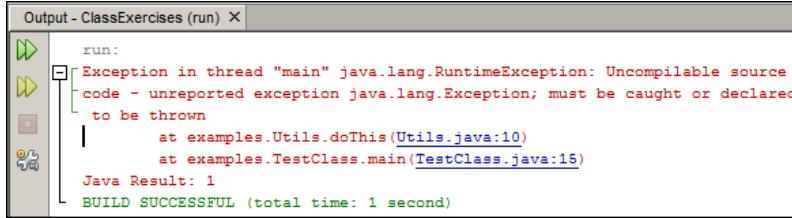
Uncaught Exception



But what happens if none of the methods in the call stack have `try/catch` blocks? That situation is illustrated by the diagram shown in this slide. Because there are no `try/catch` blocks, the exception is propagated all the way up the call stack. But what happens when it gets to the `main` method and is not handled there? This causes the program to exit, and the exception, plus a stack trace for the exception, is printed to the console.

Exception Printed to Console

When the exception is thrown up the call stack without being caught, it will eventually reach the JVM. The JVM will print the exception's output to the console and exit.



The screenshot shows the Java Output window titled "Output - ClassExercises (run)". It displays the following text:

```
run:
Exception in thread "main" java.lang.RuntimeException: Uncompilable source
code - unreported exception java.lang.Exception; must be caught or declared
to be thrown
|   at examples.Utils.doThis(Utils.java:10)
|   at examples.TestClass.main(TestClass.java:15)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```



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

In the example, you can see what happens when the exception is propagated up the call stack all the way to the `main` method. Did you notice how similar this looks to the first example you saw of an `ArrayIndexOutOfBoundsException`? In both cases, the exception is displayed as a stack trace to the console.

There was something different about the `ArrayIndexOutOfBoundsException`: None of the methods threw that exception! So how did it get passed up the call stack?

The answer is that `ArrayIndexOutOfBoundsException` is a `RuntimeException`. The `RuntimeException` class is a subclass of the `Exception` class, but it is not a checked exception so its exceptions are automatically propagated up the call stack without `throws` being explicitly declared in the method signature.

Summary of Exception Types

A `Throwable` is a special type of Java object.

- It is the only object type that:
 - Is used as the argument in a catch clause
 - Can be “thrown” to the calling method
- It has two direct subclasses:
 - `Error`
 - Automatically propagated up the call stack to the calling method
 - `Exception`
 - Must be explicitly handled and requires either:
 - A `try/catch` block to handle the error
 - A `throws` in the method signature to propagate up the call stack
 - Has a subclass `RuntimeException`
 - Automatically propagated up the call stack to the calling method



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

An `Exception` that is not a `RuntimeException` must be explicitly handled.

- An `Error` is usually so critical that it is unlikely that you could recover from it, even if you anticipated it. You are not required to check these exceptions in your code.
- An `Exception` represents an event that could happen and which may be recoverable. You are required to either catch an `Exception` within the method that generates it or throw it to the calling method.
- A `RuntimeException` is usually the result of a system error (out of memory, for instance). They are inherited from `Exception`. You are not required to check these exceptions in your code, but sometimes it makes sense to do so. They can also be the result of a programming error (for instance, `ArrayIndexOutOfBoundsException` is one of these exceptions).

The examples later in this lesson show you how to work with an `IOException`.

Exercise 14-1: Catching an Exception

1. Open the project **Exercise_14-1** in NetBeans.

In the `Calculator` class:

2. Change the `divide` method signature so that it throws an `ArithmeticException`.

In the `TestClass` class:

3. Surround the code that calls the `divide` method with a `try/catch` block. Handle the exception object by printing it to the console.

4. Run the `TestClass` to view the outcome.



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

Quiz



Which one of the following statements is true?

- a. A RuntimeException must be caught.
- b. A RuntimeException must be thrown.
- c. A RuntimeException must be caught or thrown.
- d. A RuntimeException is thrown automatically.



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

Answer: d

Exceptions in the Java API Documentation

Method Summary

Modifier and Type	Method and Description
boolean	<code>canExecute()</code> Tests whether the application can execute the file denoted by this abstract pathname.
boolean	<code>canRead()</code> Tests whether the application can read the file denoted by this abstract pathname.
boolean	<code>canWrite()</code> Tests whether the application can modify the file denoted by this abstract pathname.
int	<code>compareTo(File pathname)</code> Compares two abstract pathnames lexicographically.
boolean	<code>createNewFile()</code> Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.

These are methods of the `File` Class.

createNewFile

```
public boolean createNewFile()
                           throws IOException
```

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The `FileLock` facility should be used instead.

Returns:

`true` if the named file does not exist and was successfully created; `false` if the named file already exists

Throws:

`IOException` - If an I/O error occurred
`SecurityException` - If a security manager exists and its `SecurityManager.checkWrite(java.lang.String)` method denies write access to the file

Since:

1.2

Click to get the detail of `createNewFile`.

Note the exceptions that can be thrown.



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

When working with any API, it is necessary to determine what exceptions are thrown by the object's constructors or methods. The example in the slide is for the `File` class. `File` has a `createNewFile` method that can throw an `IOException` or a `SecurityException`. `SecurityException` is a `RuntimeException`, so `SecurityException` is unchecked but `IOException` is a checked exception.

Calling a Method That Throws an Exception

The figure consists of two vertically stacked screenshots of a Java code editor in NetBeans. Both screenshots show a method named `testCheckedException` with the following code:53 public void testCheckedException(){
54 File testFile = new File("//testFile.txt");
55
56 System.out.println("testFile exists: "+ testFile.exists());
57 testFile.delete();
58 System.out.println("testFile exists: "+ testFile.exists());
59 }
60
61 }

In the top screenshot, the line `File testFile = new File("//testFile.txt");` is highlighted with a red box. A callout bubble points to it with the text "Constructor causes no compilation problems.".

In the bottom screenshot, the line `testFile.createNewFile();` is highlighted with a red box. A callout bubble points to it with the text "createNewFile can throw a checked exception, so the method must throw or catch.".



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

The two screenshots in the slide show a simple `testCheckedException` method. In the first example, the `File` object is created using the constructor. Note that even though the constructor can throw a `NullPointerException` (if the constructor argument is null), you are not forced to catch this exception. However, in the second example, `createNewFile` can throw an `IOException`, and NetBeans shows that you must deal with this.

Note that `File` is introduced here only to illustrate an `IOException`. In the next course (*Java SE Programming II*), you learn about the `File` class and a new set of classes in the package `java.nio`, which provides more elegant ways to work with files.

Working with a Checked Exception

Catching IOException:

```
01 public static void main(String[] args) {
02     TestClass testClass = new TestClass();
03
04     try {
05         testClass.testCheckedException();
06     } catch (IOException e) {
07         System.out.println(e);
08     }
09 }
10
11 public void testCheckedException() throws IOException {
12     File testFile = new File("//testFile.txt");
13     testFile.createNewFile();
14     System.out.println("testFile exists:" +
15                         + testFile.exists());
16 }
```



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

The example in the slide is handling the possible raised exception by:

- Throwing the exception from the `testCheckedException` method
- Catching the exception in the caller method

In this example, the `catch` method catches the exception because the path to the text file is not correctly formatted. `System.out.println(e)` calls the `toString` method of the exception, and the result is as follows:

```
java.io.IOException: The filename, directory name, or volume label syntax is incorrect
```

Best Practices

- Catch the actual exception thrown, not the superclass type.
- Examine the exception to find out the exact problem so you can recover cleanly.
- You do not need to catch every exception.
 - A programming mistake should not be handled. It must be fixed.
 - Ask yourself, “Does this exception represent behavior I want the program to recover from?”



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

Bad Practices

```
01 public static void main(String[] args) {  
02     try {  
03         writeFile("c:/testFile.txt");  
04     } catch (Exception e) {           CATCHING SUPERCLASS?  
05         System.out.println("Error creating file.");    NO PROCESSING OF  
06     }                                EXCEPTION CLASS?  
07 }  
08 public static void writeFile(String name)  
09     throws IOException{  
10     File f = new File(name);  
11     f.createNewFile();  
12  
13     int[] intArray = new int[5];  
14     intArray[5] = 27;  
15 }
```



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

The code in the slide illustrates two poor programming practices.

1. The `catch` clause catches an `Exception` type rather than an `IOException` type (the expected exception from calling the `createFile` method).
2. The `catch` clause does not analyze the `Exception` object and instead simply assumes that the expected exception has been thrown from the `File` object.

A major drawback of this careless programming style is shown by the fact that the code prints the following message to the console:

There is a problem creating the file!

This suggests that the file has not been created, and indeed any further code in the `catch` block will run. But what is actually happening in the code?

Somewhat Better Practice

```

01 public static void main(String[] args) {
02     try {
03         writeFile("c:/testFile.txt");
04     } catch (Exception e) {
05         System.out.println(e);
06     //<other actions>
07 }
08 }

09 public static void writeFile(String fname)
10    throws IOException{
11    File f = new File(name);
12    System.out.println(name+ " exists? "+f.exists());
13    f.createNewFile();
14    System.out.println(name+ " exists? "+f.exists());
15    int[] intArray = new int[5];
16    intArray[5] = 27;
17 }

```

What is the object type?

toString() is called on this object.



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

Putting in a few `System.out.println` calls in the `writeFile` method may help clarify what is happening. The output now is:

```

C:/testFile.txt exists? false (from line 12)
C:/testFile.txt exists? true (from line 14)
java.lang.ArrayIndexOutOfBoundsException: 5

```

So the file is being created! And you can see that the exception is actually an `ArrayIndexOutOfBoundsException` that is being thrown by the final line of code in `writeFile`.

In this example, it is obvious that the array assignment can throw an exception, but it may not be so obvious. In this case, the `createNewFile` method of `File` actually throws another exception—a `SecurityException`. Because it is an unchecked exception, it is thrown automatically.

If you check for the specific exception in the `catch` clause, you remove the danger of assuming what the problem is.

Topics

- Handling errors: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors



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

Multiple Exceptions

```
01 public static void createFile() throws IOException {
02     File testF = new File("c:/notWriteableDir");
03
04     File tempF = testF.createTempFile("te", null, testF);
05
06     System.out.println
07         ("Temp filename: "+tempF.getPath()); ArgumentException
08     int myInt[] = new int[5];
09     myInt[5] = 25;
10 }
```

Directory must be writeable:
IOException

Arg must be greater than 3
characters:
ArgumentException

Array index must be valid:
ArrayIndexOutOfBoundsException



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

The example in the slide shows a method that could potentially throw three different exceptions. It uses the `createTempFile` method, which creates a temporary file. (It ensures that each call creates a new and different file and also can be set up so that the temporary files created are deleted on exit.)

The three exceptions are the following:

IOException

`c:\notWriteableDir` is a directory, but it is not writable. This causes `createTempFile()` to throw an `IOException` (checked).

IllegalArgumentException

The first argument passed to `createTempFile` should be three or more characters long. If it is not, the method throws an `IllegalArgumentException` (unchecked).

ArrayIndexOutOfBoundsException

As in previous examples, trying to access a nonexistent index of an array throws an `ArrayIndexOutOfBoundsException` (unchecked).

Catching IOException

```
01 public static void main(String[] args) {
02     try {
03         createFile();
04     } catch (IOException ioe) {
05         System.out.println(ioe);
06     }
07 }
08
09 public static void createFile() throws IOException {
10     File testF = new File("c:/notWriteableDir");
11     File tempF = testF.createTempFile("te", null, testF);
12     System.out.println("Temp filename: "+tempF.getPath());
13     int myInt[] = new int[5];
14     myInt[5] = 25;
15 }
```



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

The example in the slide shows the minimum exception handling (the compiler insists on at least the IOException being handled).

With the directory is set as shown at c:/notWriteableDir, the output of this code is:

java.io.IOException: Permission denied

However, if the file is set as c:/writeableDir (a writable directory), the output is now:

Exception in thread "main" java.lang.IllegalArgumentException: Prefix string
too short
at java.io.File.createTempFile(File.java:1782)
at MultipleExceptionExample.createFile(MultipleExceptionExample.java:34)
at MultipleExceptionExample.main(MultipleExceptionExample.java:18)

The argument "te" causes an IllegalArgumentException to be thrown, and because it is a RuntimeException, it gets thrown all the way out to the console.

Catching IllegalArgumentException

```
01 public static void main(String[] args) {
02     try {
03         createFile();
04     } catch (IOException ioe) {
05         System.out.println(ioe);
06     } catch (IllegalArgumentException iae){
07         System.out.println(iae);
08     }
09 }
10
11 public static void createFile() throws IOException {
12     File testF = new File("c:/writeableDir");
13     File tempF = testF.createTempFile("te", null, testF);
14     System.out.println("Temp filename: "+tempF.getPath());
15     int myInt[] = new int[5];
16     myInt[5] = 25;
17 }
```



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

The example in the slide shows an additional `catch` clause added to catch the potential `IllegalArgumentException`.

With the first argument of the `createTempFile` method set to "te" (fewer than three characters), the output of this code is:

`java.lang.IllegalArgumentException: Prefix string too short`

However, if the argument is set to "temp", the output is now:

```
Temp filename is /Users/kenny/writeableDir/temp938006797831220170.tmp
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
... < some code omitted > ...
```

Now the temporary file is being created, but there is still another argument being thrown by the `createFile` method. And because `ArrayIndexOutOfBoundsException` is a `RuntimeException`, it is automatically thrown all the way out to the console.

Catching Remaining Exceptions

```
01 public static void main(String[] args) {
02     try {
03         createFile();
04     } catch (IOException ioe) {
05         System.out.println(ioe);
06     } catch (IllegalArgumentException iae){
07         System.out.println(iae);
08     } catch (Exception e){
09         System.out.println(e);
10     }
11 }
12 public static void createFile() throws IOException {
13     File testF = new File("c:/writeableDir");
14     File tempF = testF.createTempFile("te", null, testF);
15     System.out.println("Temp filename: "+tempF.getPath());
16     int myInt[] = new int[5];
17     myInt[5] = 25;
18 }
```



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

The example in the slide shows an additional `catch` clause to catch all the remaining exceptions.

For the example code, the output of this code is:

Temp filename is /Users/kenny/writeableDir/temp7999507294858924682.tmp
java.lang.ArrayIndexOutOfBoundsException: 5

Finally, the `catch exception` clause can be added to catch any additional exceptions.

Summary

In this lesson, you should have learned how to:

- Describe the different kinds of errors that can occur and how they are handled in Java
- Describe what exceptions are used for in Java
- Determine what exceptions are thrown for any foundation class
- Write code to handle an exception thrown by the method of a foundation class



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

Practices Overview

- 14-1: Adding exception handling



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

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

Deploying and Maintaining the Soccer Application

15



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



ORACLE®

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

Interactive Quizzes



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

Before you start today's lessons, test your knowledge by answering some quiz questions that relate to yesterday's lessons. Open the Quiz files by clicking the quizzes.html shortcut from the desktop of your VM. In the welcome page, JavaSEProgrammingI.html, click the links for Lessons 12, 13, and 14.

Objectives

After completing this lesson, you should be able to:

- Deploy a simple application as a JAR file
- Describe the parts of a Java application, including the user interface and the back end
- Describe how classes can be extended to implement new capabilities in the application



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

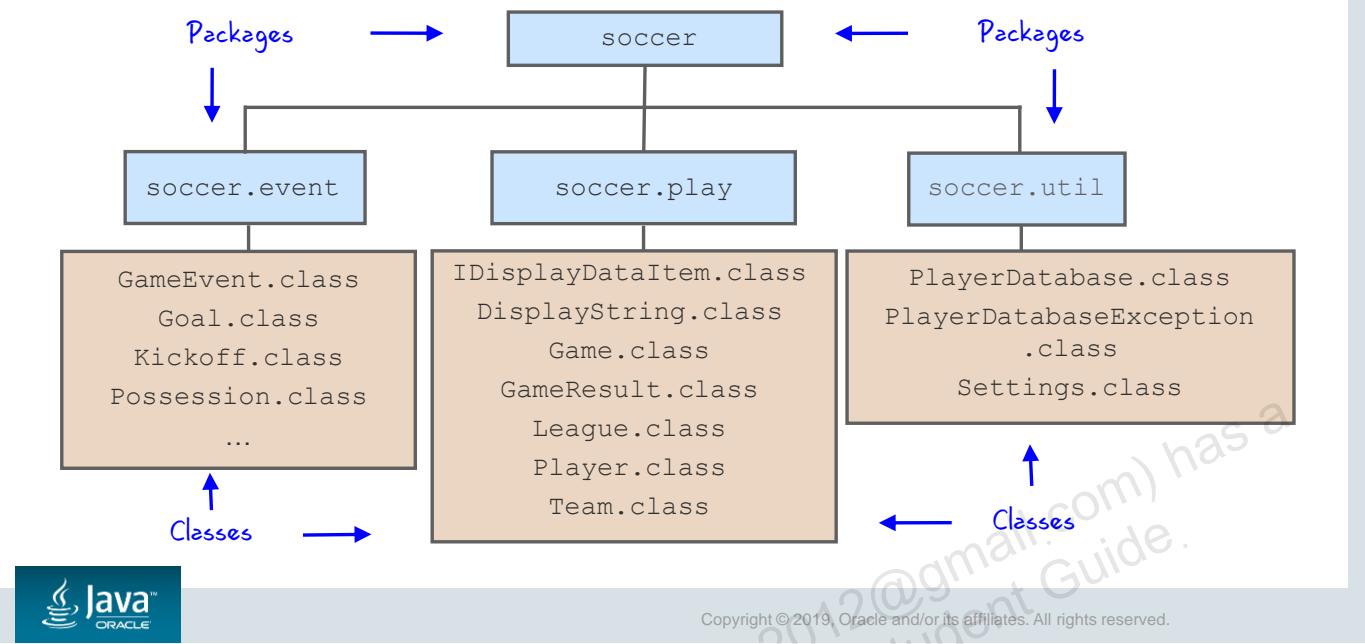
Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements



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

Packages

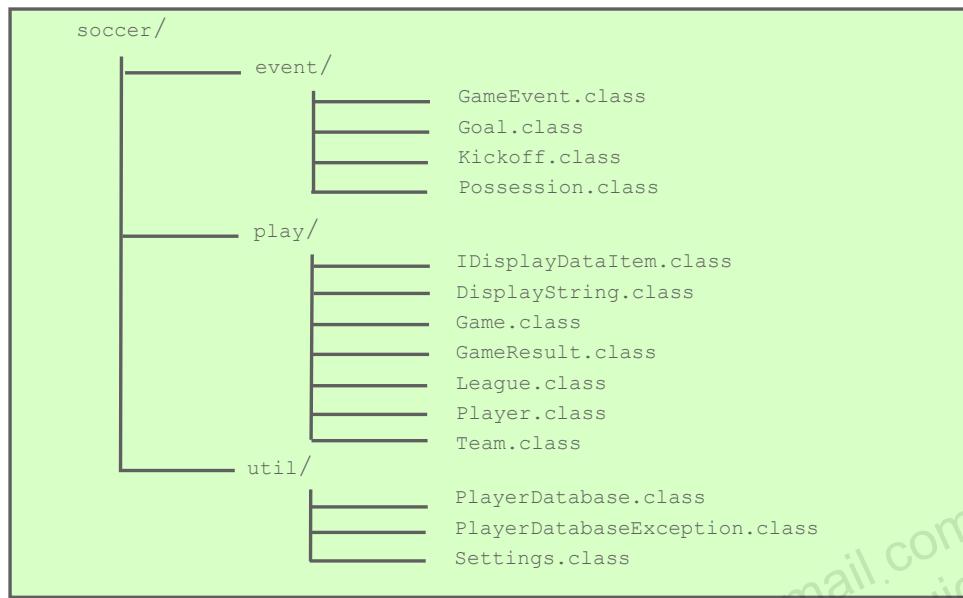


Classes are grouped into packages to ease management of the system.

There are many ways to group classes into meaningful packages. There is no right or wrong way, but a common technique is to group classes into a package by semantic similarity.

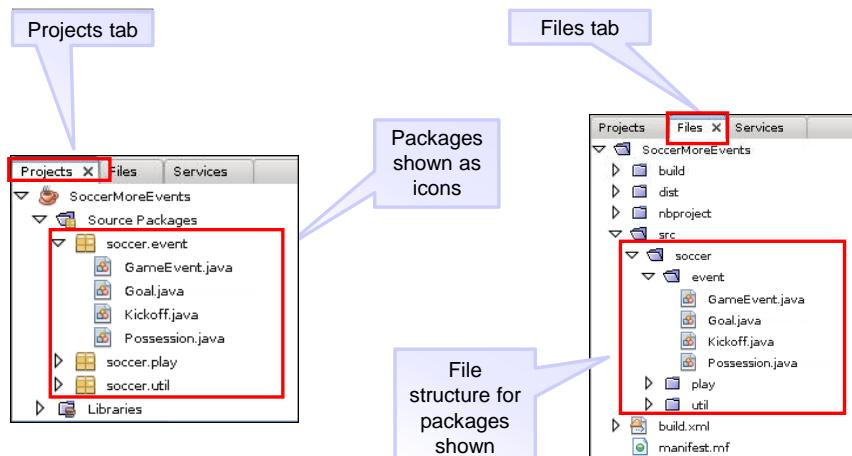
For example, the software for the soccer application could contain a set of event classes (the superclass `GameEvent`, with subclasses `Goal`, `Kickoff`, and so on), a set of classes that use these event classes to model the playing of a game, and a set of utility classes. All these packages are contained in the top-level package called `soccer`.

Packages Directory Structure



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

Packages in NetBeans



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

The left panel in NetBeans has three tabs. Two of these tabs, Projects and Files, show how packages relate to the file structure.

The Projects tab shows the packages and libraries for each project. The source package shown is the one containing the packages and classes for the Soccer application, and the screenshot shows the three packages: `soccer.event`, `soccer.play`, and `soccer.util`. Each of these packages can be expanded to show the source files within, as has been done for the `soccer.event` package in the screenshot.

The Files tab shows the directory structure for each project. In the screenshot, you can see how the packages listed on the Projects tab have a corresponding directory structure. For example, the `soccer.event` package has the corresponding file structure of the `duke` directory just under the `src` directory and contains the `item` directory, which in turn contains all the source files in the package.

Packages in Source Code

```
package soccer.event;

public class Goal extends GameEvent {

    public String toString() {
        return "GOAL! ";
    }
    ... < remaining code omitted > ...
}
```

This class is in the package soccer.event.

The package that a class belongs to is defined in the source code.



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

The example code in the slide shows the package statement being used to define the package that the Goal class is in. Just as the class itself must be in a file of the same name as the class, the file (in this case, Goal.java) must be contained in a directory structure that matches the package name.

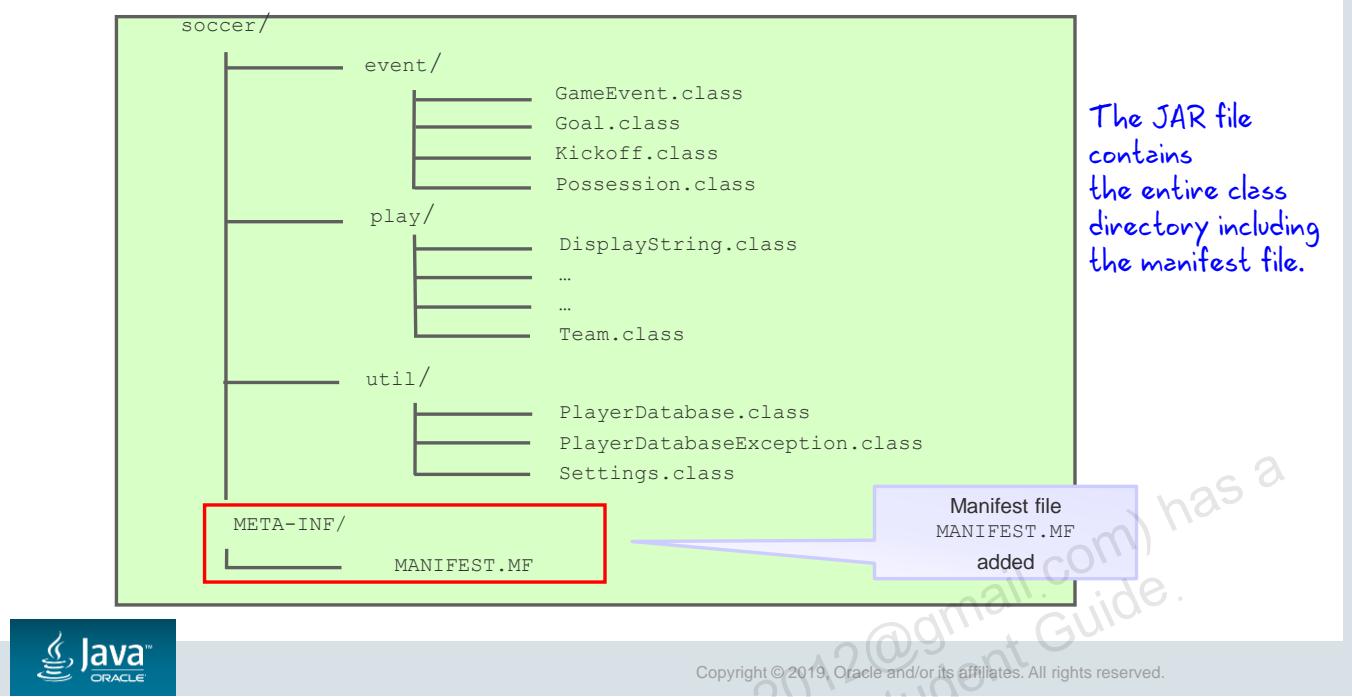
Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements



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

SoccerEnhanced.jar

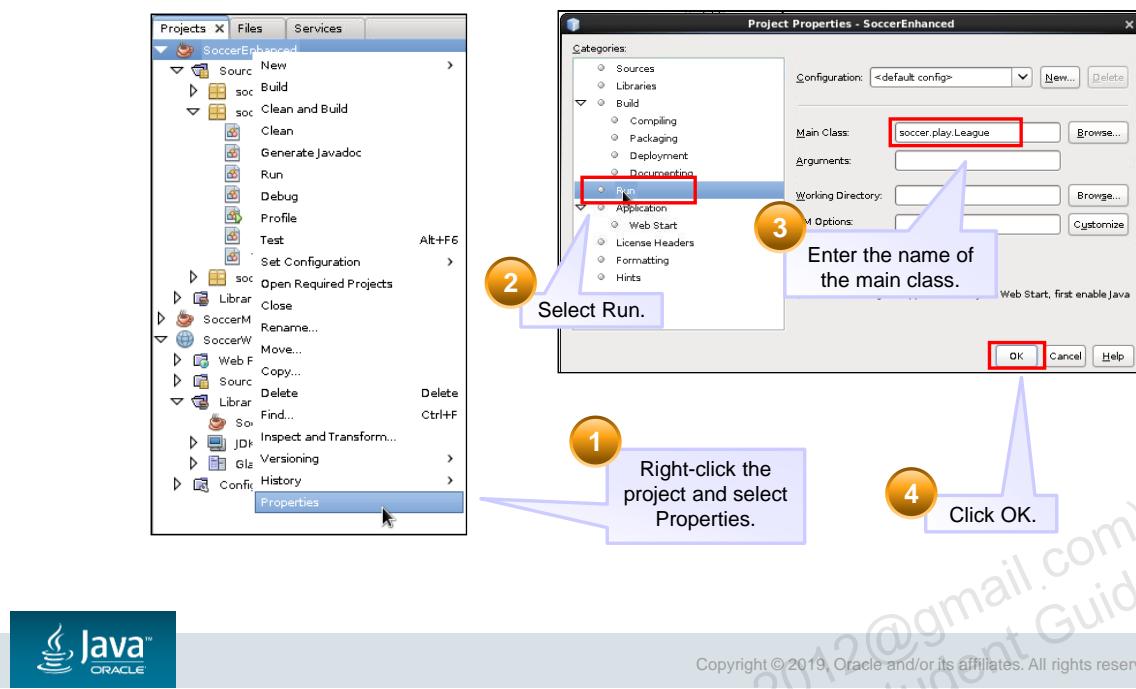


To deploy a Java application, you typically put the necessary files into a JAR file. This greatly simplifies running the application on another machine.

A JAR file is much like a zip file (or a tar file on UNIX) and contains the entire directory structure for the compiled classes plus an additional **MANIFEST.MF** file in the **META-INF** directory. This **MANIFEST.MF** file tells the Java runtime which file contains the **main** method.

You can create a JAR file by using a command-line tool called **jar**, but most IDEs make the creation easier. In the following slides, you see how to create a JAR file using NetBeans.

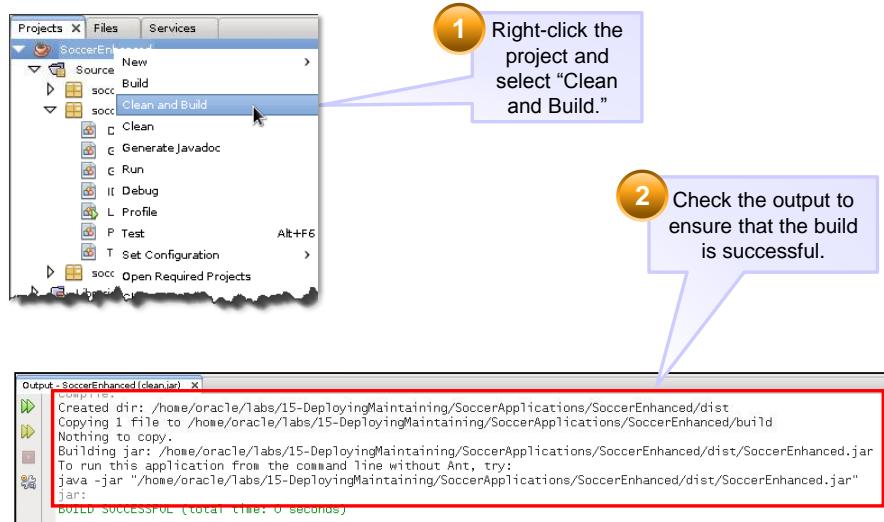
Set Main Class of Project



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

Before you create the JAR file, you need to indicate which file contains the `main` method. This is subsequently written to the `MANIFEST.MF` file.

Creating the JAR File with NetBeans



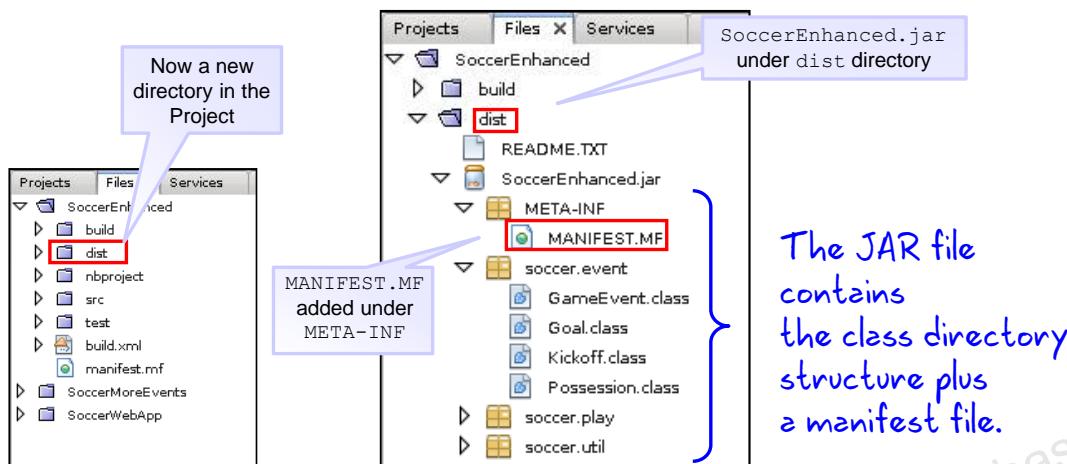
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

You create the JAR file by right-clicking the project and selecting “Clean and Build.” For a small project such as SoccerEnhanced, this should take only a few seconds.

- Clean removes any previous builds.
- Build creates a new JAR file.

You can also run “Clean” and “Build” separately.

Creating the JAR File with NetBeans



By default, the JAR file will be placed in the dist directory. (This directory is removed in the clean process and re-created during build.) Using the Files tab of NetBeans, you can look inside the JAR file and make sure that all the correct classes have been added.

Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements



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



Client/Server Two-Tier Architecture

Client/server computing involves two or more computers sharing tasks:

- Each computer performs logic appropriate to its design and stated function.
- The front-end client communicates with the back-end database.
- The client requests data from the back end.
- The server returns the appropriate results.
- The client handles and displays data.



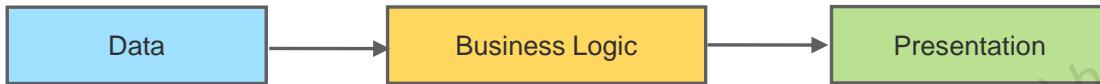
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

A major performance penalty is paid in two-tier client/server. The client software ends up larger and more complex because most of the logic is handled there. The use of server-side logic is limited to database operations. The client here is referred to as a *thick client*.

Thick clients tend to produce frequent network traffic for remote database access. This works well for intranet-based and local area network (LAN)-based network topologies, but produces a large footprint on the desktop in terms of disk and memory requirements. Also, not all back-end database servers are the same in terms of server logic offered, and all of them have their own API sets that programmers must use to optimize and scale performance.

Client/Server Three-Tier Architecture

- Three-tier client/server is a more complex, flexible approach.
- Each tier can be replaced by a different implementation:
 - The data tier is an encapsulation of all existing data sources.
 - Business logic defines business rules.
 - Presentation can be GUI, web, smartphone, or even console.



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

The three components or tiers of a three-tier client/server environment are *data, business logic or functionality*, and *presentation*. They are separated so that the software for any one of the tiers can be replaced by a different implementation without affecting the other tiers.

For example, if you want to replace a character-oriented screen (or screens) with a GUI (the presentation tier), you write the GUI using an established API or interface to access the same functionality programs in the character-oriented screens.

The business logic offers functionality in terms of defining all of the business rules through which the data can be manipulated. Changes to business policies can affect this layer without having an impact on the actual databases.

The third tier, or data tier, includes existing systems, applications, and data that have been encapsulated to take advantage of this architecture with minimal transitional programming effort.

Topics

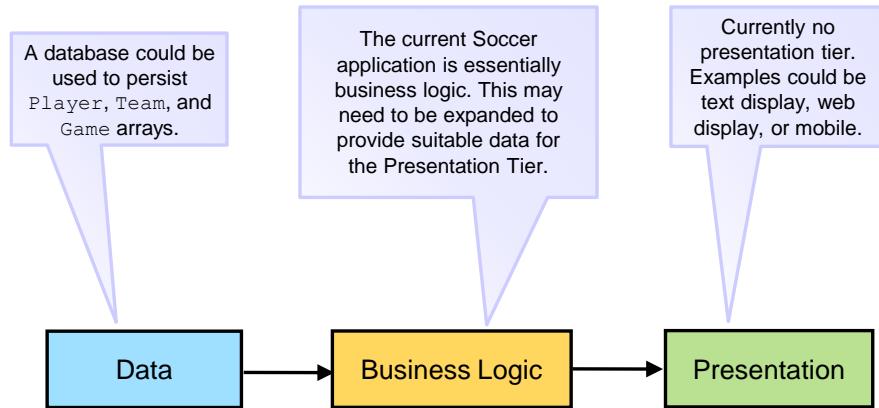
- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- **The Soccer application**
- Application modifications and enhancements



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



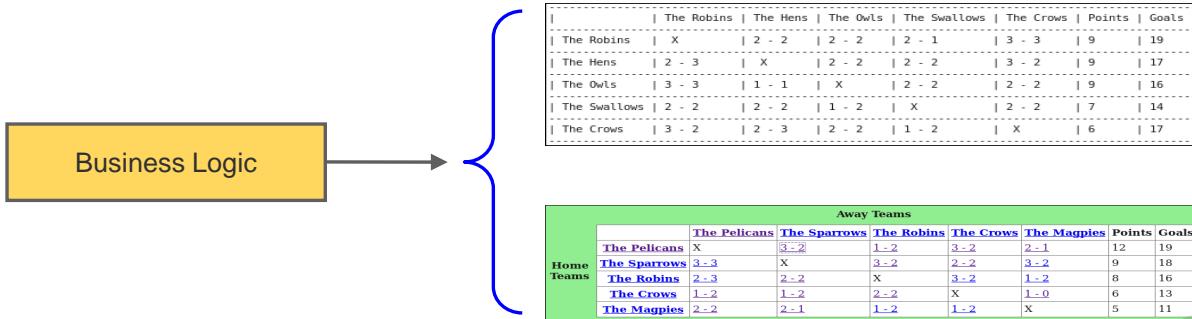
Client/Server Three-Tier Architecture



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

In the Soccer application, the Data tier does not really exist—there is currently no way to persist the data. But a Data tier could be added by saving the Player, Team, and Game arrays to a database. The current code of the Soccer application is business logic code. There is currently no presentation tier—at the moment the only presentation of data is the console of the application. To support a presentation tier, the business logic tier may need to present the data in some fashion where it can be consumed easily by many different types of presentation tier.

Client/Server Three-Tier Architecture



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

To support different presentations of the grid view of the league, the business logic tier should provide the data in a way that it can be queried to produce the individual values needed.

Different Outputs

A two-dimensional String array could provide the String output for each element of the grid, but this is inflexible:

- The presentation can only display the String provided.
- The presentation cannot access other useful information—for example, the data required to allow users to click on the score for more details.

	The Robins	The Hens	The Owls	The Swallows	The Crows	Points	Goals
The Robins	X	2 - 2	2 - 2	2 - 1	3 - 3	9	19
The Hens	2 - 3	X	2 - 2	2 - 2	3 - 2	9	17
The Owls	3 - 3	1 - 1	X	2 - 2	2 - 2	9	16
The Swallows	2 - 2	2 - 2	1 - 2	X	2 - 2	7	14
The Crows	3 - 2	2 - 3	2 - 2	1 - 2	X	6	17

Away Teams							
Home Teams	The Pelicans	The Sparrows	The Robins	The Crows	The Magpies	Points	Goals
	X	3 - 2	1 - 2	3 - 2	2 - 1	12	19
	3 - 3	X	3 - 2	2 - 2	3 - 2	9	18
	2 - 3	2 - 2	X	3 - 2	1 - 2	8	16
	1 - 2	1 - 2	2 - 2	X	1 - 0	6	13
	2 - 2	2 - 1	1 - 2	1 - 2	X	5	11

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



Ideally for each element of data in the two-dimensional array, the presentation should have access to the data the display is based on, or at least a useful subset of that data. For example, for the text that displays a team name, the presentation might wish to query further data about the team—for example to provide a pop-up list of the players in the team or a pop-up of the details of a game for any of the game scores.

Note that using a two-dimensional array is of course not the only way to package the data; you could use a List of List objects or create a custom class.

But assuming a two-dimensional array—it can only be of one type, so you cannot put references to Team objects and a Game objects into the same array. Or can you?

The Soccer Application

- Abstract classes
 - GameEvent
 - _ Extended by Goal and other GameEvent classes
- Interfaces
 - Comparable
 - _ Implemented by Team and Player so that they can be ranked
 - IDisplayDataItem
 - _ Implemented by Team, Game, and DisplayString



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

An enhanced version of the Soccer application has been created to illustrate object-oriented programming in Java. The IDisplayDataItem is a new Interface that is implemented by Team and Game, and a new class, DisplayString. Any class that implements this interface can be used in a two-dimensional array of type IDisplayDataItem. So you can have references that access both Team objects and Game objects in the same array after all!

IDisplayDataItem Interface

```
package soccer.play;

public interface IDisplayDataItem {

    public boolean isDetailAvailable ();
    public String getDisplayDetail();
    public int getID();
    public String getDetailType();

}
```



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

Running the JAR File from the Command Line

```
Output - SoccerEnhanced(clean.jar) X
[oracle@EDBSR2P14 ~]$ java -jar "/home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar"
Created dir: /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist
Copying 1 file to /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist
Nothing to copy.
Building jar: /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar
To run this application from the command line without Ant, try:
java -jar "/home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar"
jar:
BUILD SUCCESSFUL (total time: 0 seconds)
```

The command to run the JAR file

```
[oracle@EDBSR2P14 ~]$ java -jar "/home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar"
-----
| The Robins | The Hens | The Owls | The Swallows | The Crows | Points | Goals |
-----
| The Robins | X | 2 - 2 | 2 - 2 | 2 - 1 | 3 - 3 | 9 | 19 |
-----
| The Hens | 2 - 3 | X | 2 - 2 | 2 - 2 | 3 - 2 | 9 | 17 |
-----
| The Owls | 3 - 3 | 1 - 1 | X | 2 - 2 | 2 - 2 | 9 | 16 |
-----
| The Swallows | 2 - 2 | 2 - 2 | 1 - 2 | X | 2 - 2 | 7 | 14 |
-----
| The Crows | 3 - 2 | 2 - 3 | 2 - 2 | 1 - 2 | X | 6 | 17 |
```



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

Running the command-line application using the JAR file is very straightforward and the instructions are actually given in the output window for the build process. (If the JAR were a GUI application, it would be run the same way.)

If the application were an early command-line version of the software, you might run it as shown in the slide. Even though the output is simply to a terminal, the display code is iterating through a two-dimensional array of type `IDisplayDataItem` to build this display.

Text Presentation of the League

	The Robins	The Hens	The Owls	The Swallows	The Crows	Points	Goals
The Robins	X	2 - 2	2 - 2	2 - 1	3 - 3	9	19
The Hens	2 - 3	X	2 - 2	2 - 2	3 - 2	9	17
The Owls	3 - 3	1 - 1	X	2 - 2	2 - 2	9	16
The Swallows	2 - 2	2 - 2	1 - 2	X	2 - 2	7	14
The Crows	3 - 2	2 - 3	2 - 2	1 - 2	X	6	17

The object type behind these data elements is Team.

The object type behind these data elements (except for the output Xs) is Game.

The object type behind these data elements is DisplayString.



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

Note that there is a new class, `DisplayString`, that also implements `IDisplayDataItem`. It is used for data that is not represented by any of the current core classes.

Web Presentation of the League

		Away Teams						
		The Pelicans	The Sparrows	The Robins	The Crows	The Magpies	Points	Goals
Home Teams	The Pelicans	X	3 - 2	1 - 2	3 - 2	2 - 1	12	19
	The Sparrows	3 - 3	X	3 - 2	2 - 2	3 - 2	9	18
	The Robins	2 - 3	2 - 2	X	3 - 2	1 - 2	8	16
	The Crows	1 - 2	1 - 2	2 - 2	X	1 - 0	6	13
	The Magpies	2 - 2	2 - 1	1 - 2	1 - 2	X	5	11

The object type behind these data elements is Team.

The object type behind these data elements (except for the output Xs) is Game.

The object type behind these data elements is DisplayString.



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

Given that there is a two-dimensional array of type IDisplayDataItem, it can just as easily be used to create the output for a web display.

Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements



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

Enhancing the Application

- Well-designed Java software minimizes the time required for:
 - Maintenance
 - Enhancements
 - Upgrades
- For the Soccer application, it should be easy to:
 - Add new GameEvent subclasses (business logic)
 - Develop new clients (presentation)
 - Take the application to a smartphone (for example)
 - Change the storage system (data)



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

In the following slides, you see what is involved in adding another class to represent a new GameEvent.

Adding a New GameEvent Kickoff

It is possible to add a new GameEvent to record kickoffs by:

- Creating a new Kickoff class that extends the GameEvent class
- Adding any new unique features for the item
- Modifying any other classes that need to know about this new class



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

Game Record Including Kickoff

The Magpies vs. The Sparrows (2 - 3)			
Event	Team	Player	Time
Kickoff	The Sparrows	Dorothy Parker	0
Possession	The Sparrows	Jane Austin	15
Possession	The Sparrows	J. M. Synge	19
Possession	The Sparrows	Brendan Behan	20
Possession	The Sparrows	Dorothy Parker	26
GOAL!	The Sparrows	Dorothy Parker	32
Kickoff	The Magpies	G. K. Chesterton	34
Possession	The Magpies	Oscar Wilde	35
Possession	The Magpies	G. K. Chesterton	41
GOAL!	The Magpies	G. K. Chesterton	43
Kickoff	The Sparrows	Dorothy Parker	50
Possession	The Sparrows	J. M. Synge	54
GOAL!	The Sparrows	J. M. Synge	55
Kickoff	The Magpies	Wilkie Collins	59
Possession	The Magpies	G. K. Chesterton	62
Possession	The Magpies	Arthur Conan Doyle	63
Possession	The Magpies	Oscar Wilde	64
GOAL!	The Magpies	Oscar Wilde	74
Kickoff	The Sparrows	Frank O'Connor	75
Possession	The Sparrows	Frank O'Connor	81
GOAL!	The Sparrows	Frank O'Connor	83

The new event,
Kickoff, has
been added.



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

Summary

In this lesson, you should have learned how to:

- Deploy a simple application as a JAR file
- Describe the parts of a Java application, including the user interface and the back end
- Describe how classes can be extended to implement new capabilities in the application



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

Understanding Modules



ORACLE®



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

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

Objectives

After completing this lesson, you should be able to:

- Understand Java modular design principles
- Define module dependencies
- Expose module content to other modules



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



Topics

- Module system: Overview
- JARs
- Module dependencies
- Modular JDK

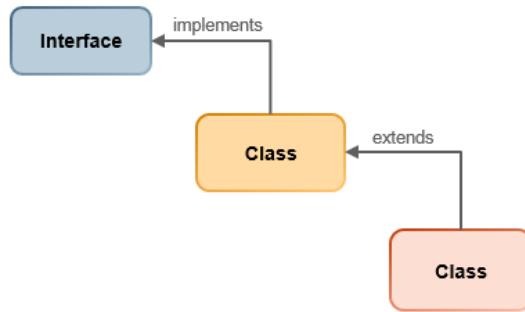


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Reusing Code in Java: Classes

One of the core features of any programming language is its ability to reuse code.

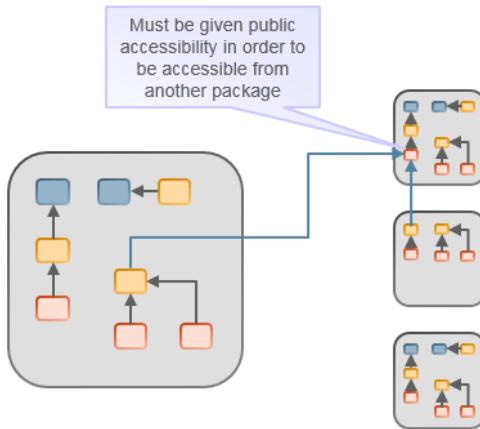
- So that “large” programs can be built from “small” programs.
- In Java the basic unit of reuse has been a class, that is, **programs are classes**.
- Java has good mechanisms for promoting reuse of a class:
 - Inheritance for reusing behavior
 - Interfaces for reusing abstractions



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

Reusing Code in Java: Packages

- Java also has packages for grouping together similar classes, that is, **programs are packages**
- Packages are grouped in JARs, and JARs are the unit of distribution.



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



Reusing Code in Java: Programming in the Large

- In a large Java codebase, when the application uses several packages and is distributed in many JARs, then it becomes difficult to:
 - Control which classes and interfaces are re-using the code
- The only way to share code between packages is through the `public` modifier. But then the code is shared with everyone.
- Packages are a great way to organize classes, but is there a way to organize packages?



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

What Is a Module?

A module is a set of packages that is designed for reuse.

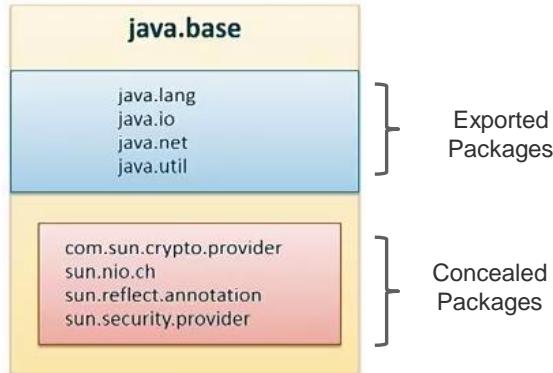
- Modularity was introduced in JDK 9.
- Modularity adds a higher level of aggregation above packages.
- A module is a reusable group of related packages, as well as resources (such as images and XML files) and a module descriptor, that is, **programs are modules**.
- In a module, some of the packages are:
 - Exported packages: Intended for use by code outside the module
 - Concealed packages: Internal to the module; they can be used by code inside the module but not by code outside the module



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

Example: `java.base` Module

- A module is a set of exported packages and concealed packages.
- This is strong encapsulation.



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

`java.base` module is the foundation module for all other modules similar to `java.lang` class, which is the foundation for every Java class. You'll learn more about `java.base` later in this lesson.

In blue are the packages in `java.base` module, which are intended to be used outside the module. These are its exported packages.

In red are the packages that are internal to the module and are meant to be used by code inside the module and not by code outside the module. These are its concealed packages.

Module System

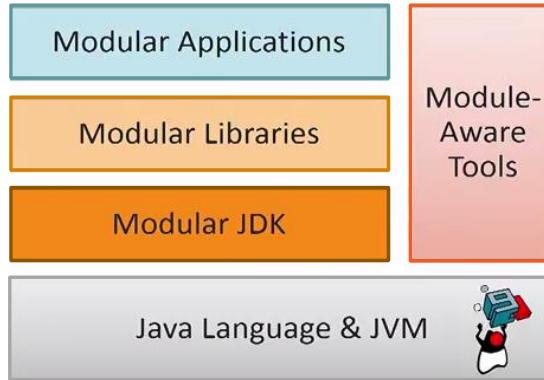
- The module system:
 - Is usable at all levels:
 - Applications
 - Libraries
 - The JDK itself
 - Addresses reliability, maintainability, and security
 - Supports creation of applications that can be scaled for small computing devices



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

Modular Development in JDK 9

JDK 9 enables modular development all the way down.



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

The module system enables modular development all the way down to the JDK. It is built into the Java language and the Java Virtual Machine so that the applications you write and the libraries you consume and even the JDK itself can all be developed and tested and packaged and deployed as modules managed by the module system. Making all levels of an application play by the same rules for modularity has great benefits for reliability, maintainability, and security. It's not essential to use the module system. The previous method of structuring applications by using JARs is still supported. But managing dependencies is an issue as evidenced by current approaches, such as maven. The module system addresses this dependency issue as well as encapsulation at the "large" level.

In JDK 9, the Java language and the Java virtual machine understand modules very well. So the application you run, the libraries you consume, and the JDK itself enable you to develop, package, test, and deploy modules. The modular development also makes your code more reliable and secure.

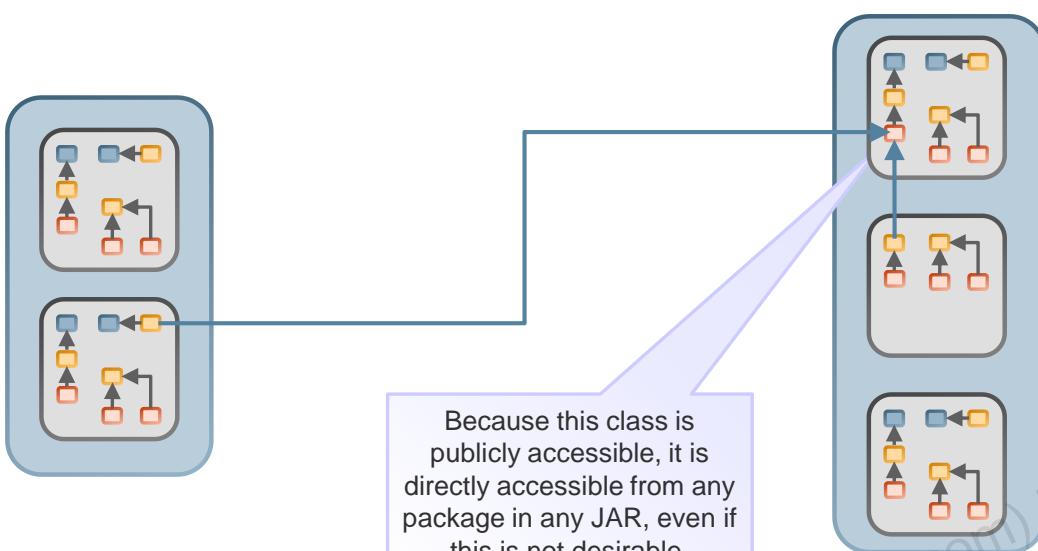
Topics

- Module system: Overview
- JARs
- Module declarations
- Modular JDK



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

JARs



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

JARs are the main unit of distribution. But JARs do not add any access or dependency information to the application.

JAR Files and Distribution Issues

- JAR files are:
 - Typically used for packaging the class files for:
 - The application
 - The libraries
 - Composed of a set of packages with some additional metadata
 - For example: main class to run, class path entries, multi-release flag
 - Added to the class path in order that their contents (classes) are made available to the JDK for compilation and running
 - Some applications may have hundreds of JARs in the class path.



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

Most non-trivial applications comprise the application itself and multiple libraries. Typically the application will be in a JAR file, and then each of the libraries will be in its own JAR file.

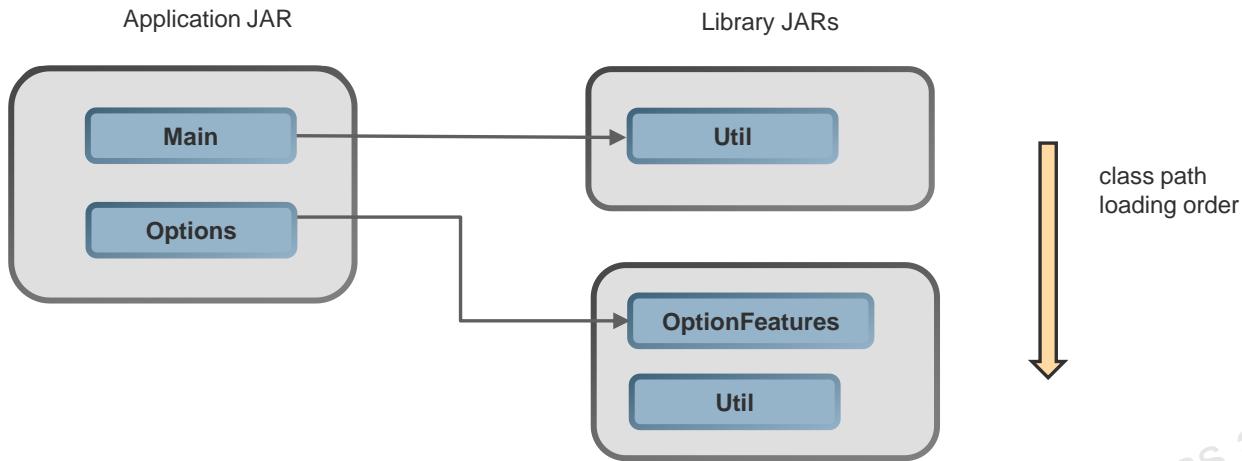
Class Path Problems

- JARs in the class path can have duplicate classes and/or packages.
- Java runtime tries to load each class as it finds it.
 - It uses the first class it finds in class path, even if another similarly named class exists.
 - The first class could be the wrong class if several versions of a library exist in the class path.
 - Problems may occur only under specific operation conditions that require a particular class.



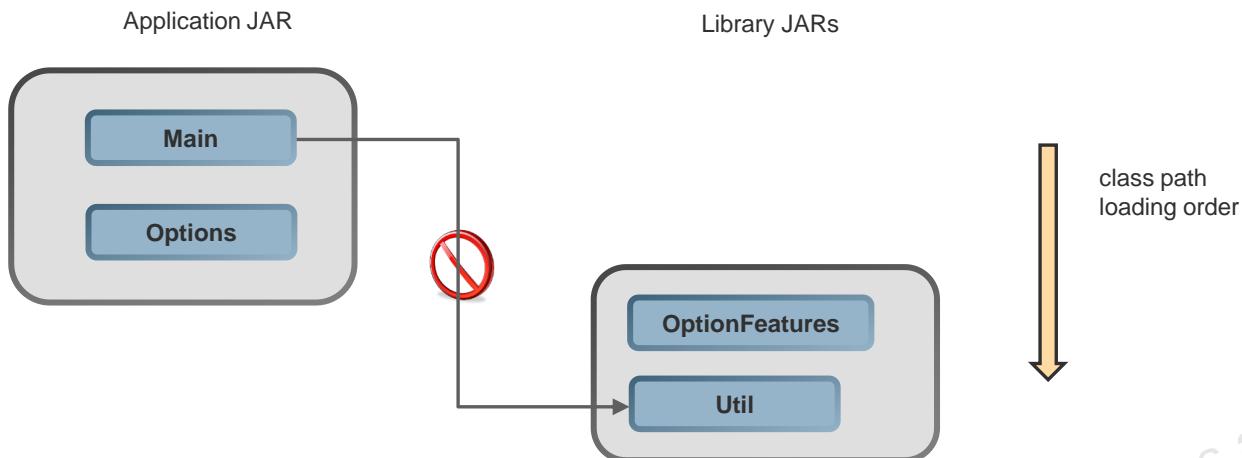
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Example JAR Duplicate Class Problem 1



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

Example JAR Duplicate Class Problem 2



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

The diagram is a continuation of the one on the previous slide and shows what might happen if the first library JAR is omitted from the class path (or placed later in the class path than the second JAR). In this case, the Main class searches the class path for the Util class it needs, but the first Util class it finds is not the one it was designed for. This is a very simple example, but it is always a potential problem as applications are enhanced and as further libraries that support the enhancements are added to the class path.

Module System: Advantages

- Addresses the following issues at the unit of distribution/reuse level:
 - Dependencies
 - Encapsulation
 - Interfaces
- The unit of reuse is the module.
 - It is a full-fledged Java component.
 - It explicitly declares:
 - Dependencies on other modules
 - What packages it makes available to other modules
 - Only the public interfaces in those available packages are visible outside the module.



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

Accessibility Between Classes

Accessibility (JDK 1 – JDK 8)

- `public`
- `protected`
- `<package>`
- `private`



Accessibility (JDK 9 and later)

- `public` to everyone
- `public`, but only to specific modules
- `public` only within a module
- `protected`
- `<package>`
- `private`



- `public` no longer means "accessible to everyone."
- You must edit the `module-info` classes to specify how modules read from each other.

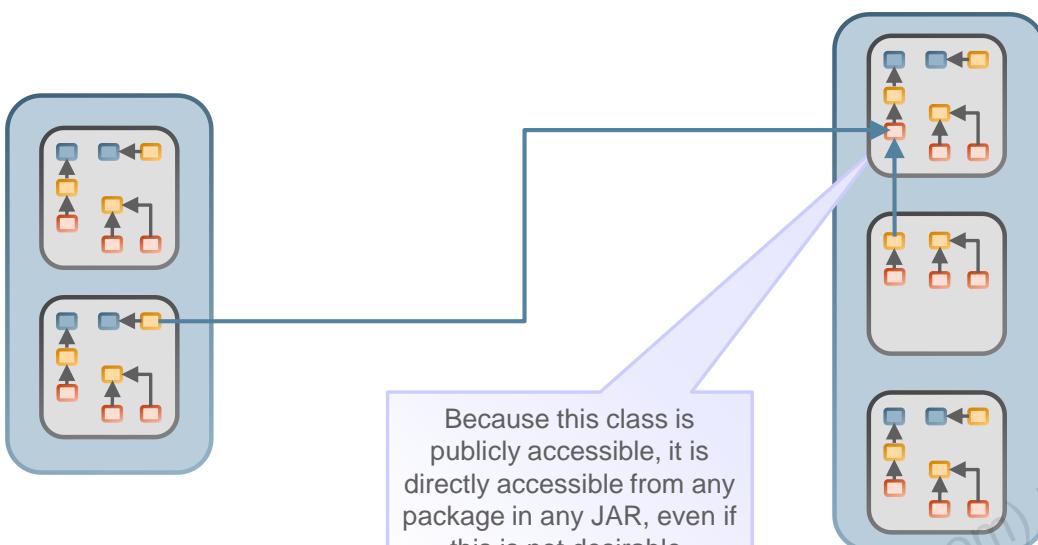


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

By default, modules cannot access each other. They can't instantiate each other's classes. They can't call each other's methods. This is true even if classes, fields, and methods use the access modifier. In a sense, "public" no longer has the same definition after JDK 9.

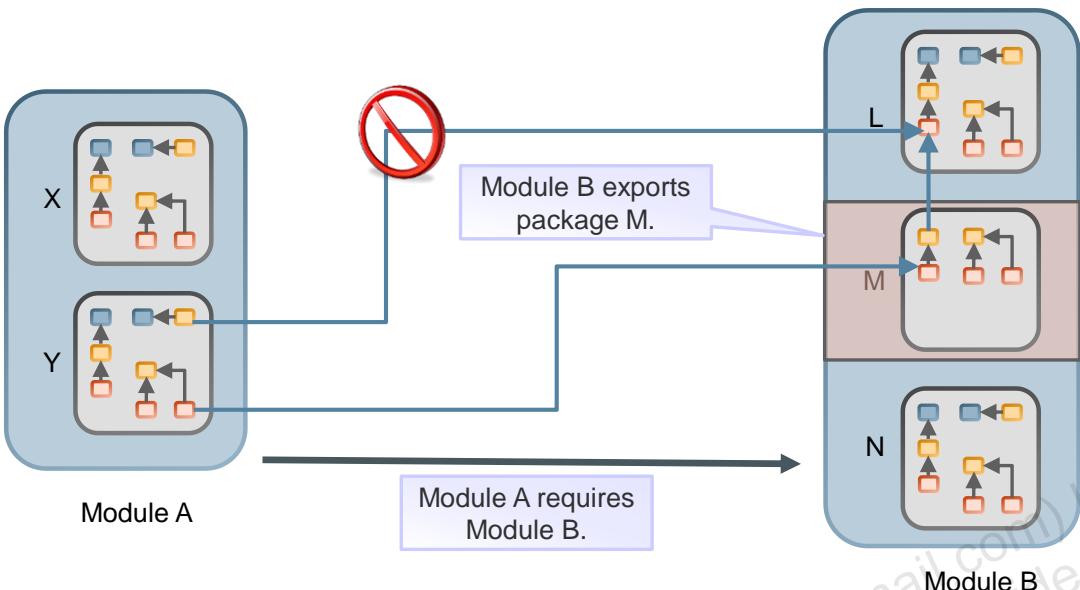
Readability is a new term used to describe the relationship between modules. Making one module readable by another is a multi-step process. You must edit each module's `module-info` class to specify how readability should occur. A module declares which of its packages can be readable by other modules. A module declares what other modules it is required to read from.

Access Across Non-Modular JARs



If a class needs to be made accessible to a different package in the same JAR, it must be made public. This makes it also accessible to any class in any package.

Access Across Modules



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

Topics

- Module system: Overview
- JARs
- **Module dependencies**
- Modular JDK



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

module-info.java

- A module must be declared in a **module-info.java** file.
 - Metadata that specifies the module's dependencies, the packages the module makes available to other modules, and more.
- Each module declaration begins with the keyword `module`, followed by a unique module name and a module body enclosed in braces, as in:

```
module modulename
{
}
```

- The module declaration's body can be empty or may contain various module directives, such as `requires`, `exports`.
- Compiling the module declaration creates the **module descriptor**, which is stored in a file named **module-info.class** in the module's root folder.



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

Example: module-info.java

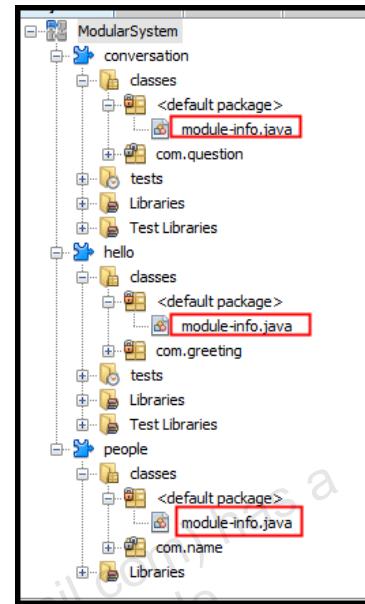
```
module soccer {  
  
    requires competition;  
    requires gameapi;  
    requires java.logging;  
    exports soccer to competition;  
}
```



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

Creating a Modular Project

- Name of the project
- Place `module-info.java` in the root directory of the packages that you want to group as a module.
- NetBeans marks this as the default package
- One modular JAR is produced for every module.
 - Modular JARs become the unit of release and reuse.
 - They're intended to contain a very specific set of functionality.



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

The image in the slide shows several modules. Each module contains its own module-info class. Real-world applications may contain many modules. This allows modules to be reused and swapped between programs.

This example modular project, `ModularSystem` can be accessed in the lab environment in `/home/oracle/labs/16_ModularSystem/practices`

exports Module Directive

- An `exports` module directive specifies one of the module's packages whose public types (and their nested `public` and `protected` types) should be accessible to code in all other modules.
- For example:
 - The `conversation` module's `module-info` class explicitly states which packages it's willing to let other modules read, using the `exports` keyword.

```
module conversation {  
  
    exports com.question;  
}
```



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

exports...to Module Directive

- An `exports...to` directive enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package; this is known as a qualified export.
- Consider this, the conversation module's `module-info` class explicitly states:
 - Which packages it's willing to allow to be read
 - Which modules are allowed to read a particular package
- This is done with the `exports` and `to` keywords.

```
module conversation {  
    exports com.question to people;  
}
```



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

requires Module Directive

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.
- To specify a dependency on another module, use `requires`, as in:

```
requires modulename;
```

- Example: The `main` module's `module-info` class explicitly lists which modules it depends on.

```
module hello {  
    requires people;  
}
```

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



The Effects of Requiring

Everything in the `game` package is now readable by the `main` module.

The `competition` module provides an API to the `main` module.

Classes in the `main` module can now:

- Instantiate object types defined in the `game` package
- Call methods of classes in the `game` package

Nothing with,, or package access is readable.

Nothing in the `util` package is readable.

requires transitive Module Directive

- To specify a dependency on another module and to ensure that other modules reading your module also read that dependency known as implied readability, use requires transitive, as in:

```
requires transitive modulename;
```



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

Implementing a Requires Transitively Relationship

```
module hello {  
    requires people;  
}
```

```
module people {  
    exports com.name;  
    requires transitive conversation;  
}
```



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

Summary of Keywords

Keywords and Syntax	Description
<code>export <package></code>	Declares which package is eligible to be read
<code>export <package> to <module></code>	Declares which package is eligible to be read by a specific module
<code>requires <module></code>	Specifies another module to read from
<code>requires transitive <module></code>	Specifies another module to read from. The relationship is transitive in that indirect access is given to modules requiring the current module.

- These are restricted keywords.
- Their creation won't break existing code.
- They're only available in the context of the `module-info` class.



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

For example, complications won't arise if your code has a variable called `export` because these keywords are restricted to the `module-info` class.

Compiling Modules

- When compiling a module, specify all of your java sources from various packages that you want this module to contain.
- Make sure to include packages that are exported by this module to other modules and a module-info.

```
javac -d <compiled output folder> <list of source code file paths  
including module-info>
```

- For example:

```
javac -d mods --module-source-path src $(find src -name "*.java")
```



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

2 - 31

3
1

Running a Modular Application

- Running a modular application:

```
java --module-path <path to compiled module>
      --module <module name>/<package name>. <main class name>
```

- For example:

```
java -p mods -m hello/com.greeting.Main
```

Note: To execute a modular application, don't use CLASSPATH !



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

2 - 32

32

Before the introduction of modules java runtime, search for classes not by analysing packaged module dependencies, but rather by scanning classpath. Classpath could be set on a system level as well as for a specific application, and sometimes this could have resulted in applications braking at runtime. 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. Executing these applications with common classpath definition, it would potentially result in NoClassDefFound or NoSuchMethod errors depending on the order in which different versions of the same API occurred in your classpath. When designing modules, you may include explicit version identity as part of the module name and avoid such errors.

Topics

- Module system: Overview
- JARs
- Module declarations
- Modular JDK



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

The JDK

Before JDK 9, the JDK was huge and monolithic, thus increasing the:

- Download time
- Startup time
- Memory footprint



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

The Modular JDK



- In JDK 9, the monolithic JDK is broken into several modules. It now consists of about 90 modules.
- Every module is a well-defined piece of functionality of the JDK:
 - All the various frameworks that were part of the prior releases of JDK are now broken down into their modules.
 - For example: Logging, Swing, and Instrumentation
- The modular JDK:
 - Makes it more scalable to small devices
 - Improves security and maintainability
 - Improves application performance



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

Listing the Modules in JDK 9

```
$java --list-modules
```

java.activation@9.0.1	java.xml@9.0.1	jdk.hotspot.agent@9.0.1	jdk.management.jfr@9.0.1
java.base@9.0.1	java.xml.bind@9.0.1	jdk.httpserver@9.0.1	jdk.management.resource@9
java.compiler@9.0.1	java.xml.crypto@9.0.1	jdk.incubator.httpclient@9.0.1	jdk.naming.dns@9.0.1
java.corba@9.0.1	java.xml.ws@9.0.1	jdk.internal.ed@9.0.1	jdk.naming.rmi@9.0.1
java.datatransfer@9.0.1	java.xml.ws.annotation@9.0.1	jdk.internal.jvmstat@9.0.1	jdk.net@9.0.1
java.desktop@9.0.1	javafx.base@9.0.1	jdk.internal.le@9.0.1	jdk.pack@9.0.1
java.instrument@9.0.1	javafx.controls@9.0.1	jdk.internal.opt@9.0.1	jdk.packager@9.0.1
java.jnlp@9.0.1	javafx.deploy@9.0.1	jdk.internal.vm.ci@9.0.1	jdk.packager.services@9.0
java.logging@9.0.1	javafx.fxml@9.0.1	jdk.jartool@9.0.1	jdk.plugin@9.0.1
java.management@9.0.1	javafx.graphics@9.0.1	jdk.javadoc@9.0.1	jdk.plugin.dom@9.0.1
java.management.rmi@9.0.1	javafx.media@9.0.1	jdk.javaws@9.0.1	jdk.plugin.server@9.0.1
java.naming@9.0.1	javafx.swing@9.0.1	jdk.jcmd@9.0.1	jdk.policytool@9.0.1
java.prefs@9.0.1	javafx.web@9.0.1	jdk.jconsole@9.0.1	jdk.rmic@9.0.1
java.rmi@9.0.1	jdk.accessibility@9.0.1	jdk.jdeps@9.0.1	jdk.scripting.nashorn@9.0.1
java.scripting@9.0.1	jdk.attach@9.0.1	jdk.jdi@9.0.1	jdk.scripting.nashorn.shell@9.0.1
java.se@9.0.1	jdk.charsets@9.0.1	jdk.jdwpxagent@9.0.1	jdk.sctp@9.0.1
java.se.ee@9.0.1	jdk.compiler@9.0.1	jdk.jfr@9.0.1	jdk.security.auth@9.0.1
java.security.jgss@9.0.1	jdk.crypto.cryptoki@9.0.1	jdk.jlink@9.0.1	jdk.security.jgss@9.0.1
java.security.sasl@9.0.1	jdk.crypto.ec@9.0.1	jdk.jsshell@9.0.1	jdk snmp@9.0.1
java.smartcardio@9.0.1	jdk.crypto.msccapi@9.0.1	jdk.jsobject@9.0.1	jdk.unsupported@9.0.1
java.sql@9.0.1	jdk.deploy@9.0.1	jdk.jstard@9.0.1	jdk.xml.bind@9.0.1
java.sql.rowset@9.0.1	jdk.deploy.controlpanel@9.0.1	jdk.localedata@9.0.1	jdk.xml.dom@9.0.1
java.transaction@9.0.1	jdk.dynalink@9.0.1	jdk.management@9.0.1	jdk.xml.ws@9.0.1
		jdk.management.agent@9.0.	jdk.zipfs@9.0.1
		jdk.management.cmm@9.0.1	oracle.desktop@9.0.1
			oracle.net@9.0.1



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

Using the java command from the JDK's bin folder with the --list-modules option, as in:

```
java --list-modules
```

Lists the JDK's set of modules, which includes the standard modules that implement the Java Language SE Specification (names starting with java), JavaFX modules (names starting with javafx), JDK-specific modules (names starting with jdk), and Oracle-specific modules (names starting with oracle).

Each module name is followed by a version string—@9 indicates that the module belongs to Java 9.

Java SE Modules

These modules are classified into two categories:

1. Standard modules (`java.*` prefix for module names):
 - Part of the Java SE specification.
 - For example: `java.sql` for database connectivity, `java.xml` for XML processing, and `java.logging` for logging
2. Modules not defined in the Java SE 9 platform (`jdk.*` prefix for module names):
 - Are specific to the JDK.
 - For example: `jdk.jshell`, `jdk.policytool`, `jdk.httpserver`



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

The Base Module

- The base module is `java.base`.
 - Every module depends on `java.base`, but this module doesn't depend on any other modules.
 - `java.base` module reference is implicitly included in all other modules.
 - The base module exports all of the platform's core packages.

```
// module-info.java
module java.base {
    exports java.lang;
    exports java.io;
    exports java.net;
    exports java.util;
}
```

```
module hello{
    requires java.base; //implied
    requires java.logging;
}
```



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

Summary

After completing this lesson, you should be able to:

- Understand Java modular design principles
- Define module dependencies
- Expose module content to other modules



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

Practice Overview

- 16-1: Creating a Modular Application from the Command Line
- 16-2: Compiling Modules from the Command Line
- 16-3: Creating a Modular Application from NetBeans



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

JShell



ORACLE®

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

17



Objectives

After completing this lesson, you should be able to:

- Explain the REPL process and how it differs from writing code in an IDE
- Launch JShell
- Create JShell scratch variables and snippets
- Identify available JShell commands and other capabilities
- Identify how an IDE enhances the JShell user experience



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

Topics

- Testing code and APIs
- JShell Basics
- JShell in an IDE

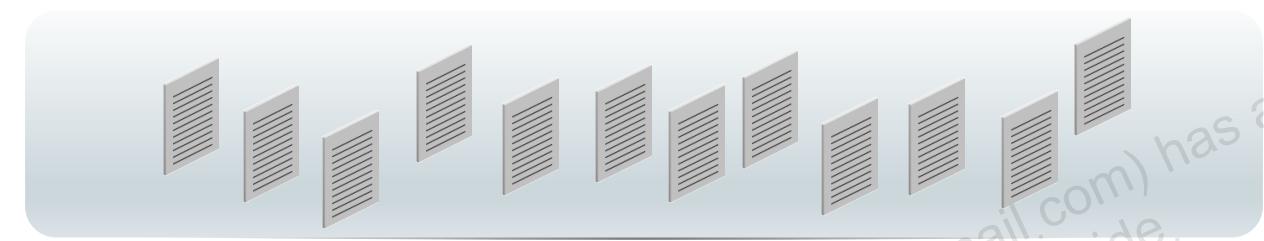


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.



A Million Test Classes and Main Methods

- Production code is dedicated to properly launching and running an application.
 - We'd complicate it by adding throwaway code.
 - It's a dangerous place for experimentation.
 - We'd alternatively clutter the IDE by creating little main methods or test projects.
- Creating a new main method or project sometimes feels like an unnecessary ceremony.
 - We're not necessarily interested in creating or duplicating a program.
 - We're interested in testing a few lines of code.



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

JShell Provides a Solution

- It's a command line interface.
- It avoids the ceremony of creating a new program and gets right into testing code.
- At any time you can:
 - Explore an API, language features, a class you wrote; do other experiments with logic, variables, or methods.
 - Prototype ideas and incrementally write more-complex code.
- You'll get instant feedback from the Read Evaluate Print Loop (REPL) process.



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

Topics

- Testing code and APIs
- **JShell Basics**
- JShell in an IDE



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



Comparing Normal Execution with REPL

- Normal Execution:
 - You enter all your code ahead of time.
 - Compile your code.
 - The program runs once in its entirety.
 - If after the first run you realize you've made a mistake, you need to run the entire program again.
- JShell's REPL:
 - You enter one line of code at a time.
 - You get feedback on that one line.
 - If the feedback proved useful, you can use that information to alter your next line of code accordingly.
- We'll look at simple examples to illustrate this.



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

Getting Started with JShell and REPL

- To launch JShell:
 - Open a terminal.
 - Enter `jshell`.
- Start entering code, for example:
 - R. The expression `Math.pow(2, 7)` is **read** into JShell.
 - E. The expression is **evaluated**.
 - P. Its value is **printed**.
 - L. The state of JShell **loops** back to where it began.
 - Repeat the process and enter more expressions.

```
[oracle@edvmr1p0 ~]$ jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> Math.pow(2,7);
$1 ==> 128.0

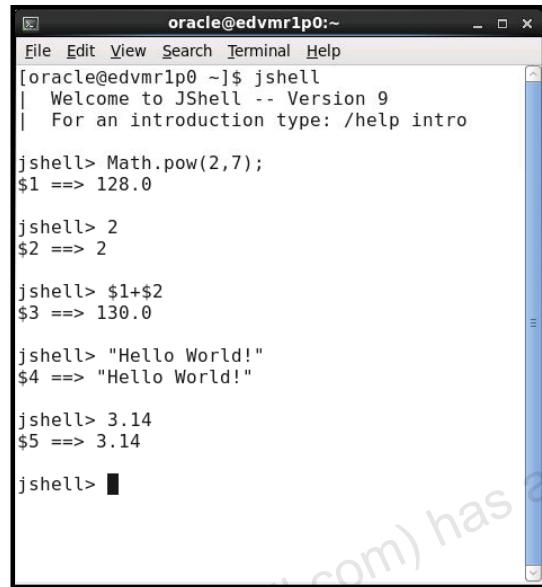
jshell>
```



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

Scratch Variables

- `Math.pow(2, 7)` evaluates to 128.
- 128 is reported back as `$1`.
- `$1` is a JShell **scratch variable**.
- Like most other variables, a scratch variable can:
 - Store the result of a method call
 - Be referenced later
 - Have its value changes
 - Be primitives or Object types
- Names are auto generated.
- Great for testing unfamiliar methods or other short experiments.



The screenshot shows a terminal window titled "oracle@edvmr1p0:~". It displays the following JShell session:

```
[oracle@edvmr1p0 ~]$ jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
jshell> Math.pow(2,7);
$1 ==> 128.0

jshell> 2
$2 ==> 2

jshell> $1+$2
$3 ==> 130.0

jshell> "Hello World!"
$4 ==> "Hello World!"

jshell> 3.14
$5 ==> 3.14

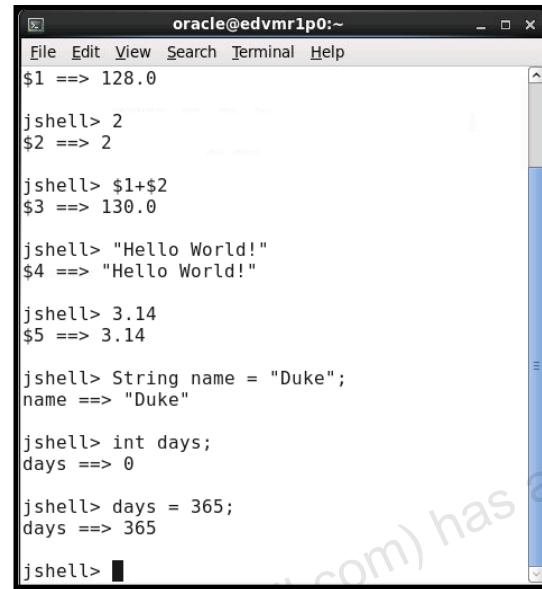
jshell> ■
```



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

Declaring Traditional Variables

- Too many scratch variables lead to confusion.
 - You may create an unlimited number of scratch variables.
 - Their names aren't descriptive.
 - It becomes hard to remember the purpose of each one.
- Traditional variables have names which provides context for their purpose.
- JShell allows you to declare, reference, and manipulate variables as you normally would.



```
oracle@edvmr1p0:~$1 ==> 128.0
jshell> 2
$2 ==> 2
jshell> $1+$2
$3 ==> 130.0
jshell> "Hello World!"
$4 ==> "Hello World!"

jshell> 3.14
$5 ==> 3.14
jshell> String name = "Duke";
name ==> "Duke"
jshell> int days;
days ==> 0
jshell> days = 365;
days ==> 365
jshell>
```



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

Code Snippets

The term **snippet** refers to the code you enter in a single JShell loop.

- Declarations

- String s = "hello"
- int twice(int x) {return x+x;}
- class Pair<T> {T a, b; Pair(...)}
- interface Reusable {}
- import java.nio.file.*

- Expressions

- Math.pow(2, 7)
- twice(12)
- new Pair<>("red", "blue")
- transactions.stream()
 - .filter(t->t.getType() ==trans.PIN)
 - .map(trans::getID)
 - .collect(toList())

- Statements

- while(mat.find()){...}
- if(x < 0){...}
- switch(val) {
 - case FMT:
 - format();
 - break;...

- Not Allowed

- package foo;
- Top-level access modifiers
 - static final
- Top-level statements of:
 - break continue return



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

A snippet can be one line. A snippet can be many lines including a loop, a method, a class... 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.

Completing a Code Snippet

- Some snippets are best written across many lines.

- Methods
- Classes
- for loop statements

- JShell waits for the snippet to be complete.
 - It detects the final closing curly brace.
 - Then it performs any evaluation.

A screenshot of a terminal window titled "oracle@edvmr1p0:~". The window has a menu bar with File, Edit, View, Search, Terminal, and Help. The main area shows the command "jshell> for(int i=0; i<5; i++){" followed by three ellipsis lines (...>) and a cursor. The scroll bar on the right is visible.

A screenshot of a terminal window titled "oracle@edvmr1p0:~". The window has a menu bar with File, Edit, View, Search, Terminal, and Help. The main area shows the completed code "jshell> for(int i=0; i<5; i++){...> System.out.print(i + " ");...>}" followed by the output "0 1 2 3 4" and the command "jshell>". The scroll bar on the right is visible.



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

Tab Completion and Tab Tips

Confused about your options?

- After the dot operator, press tab to see a list of available fields, variables, or classes.
- Press tab as you call a method to view possible signatures.
- Press tab again to see documentation.



The image shows two terminal windows side-by-side. The top window, titled 'oracle@edvmr1po:~', displays the command 'jshell> System.' followed by a list of available methods: Logger, class, currentTimeMillis(), gc(), getProperty(), hashCode(), lineSeparator(), mapLibraryName(), runFinalization(), setIn(), setProperty(), LoggerFinder, clearProperty(), err, getLogger(), getSecurityManager(), in, load(), nanoTime(), runFinalizersOnExit(), setOut(), setSecurityManager(), arraycopy(), console(), exit(), getProperties(), getenv(), inheritedChannel(), loadLibrary(), out, setErr(), and setProperties(). A blue arrow points from the text 'After the dot operator, press tab to see a list of available fields, variables, or classes.' to the list of methods. The bottom window, also titled 'oracle@edvmr1po:~', shows the command 'jshell> System.exit()' followed by 'Signatures: void System.exit(int status)'. Below this, the text '<press tab again to see documentation>' is displayed, and the command 'jshell> System.exit()' is shown again with a cursor at the end. A blue arrow points from the text 'Press tab again to see documentation.' to the '<press tab again to see documentation>' text.

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

This is helpful if you forget or are curious about what options are available to you from a particular class.

Semicolons

A screenshot of a Java jshell terminal window titled "oracle@edvmlrp0:~". The terminal shows the following interactions:

```
jshell> Math.pow(2,7);
$1 ==> 128.0

jshell> Math.pow(2,7)
$2 ==> 128.0

jshell> for(int i=0; i<5; i++){
...>     System.out.print(i + " ");
...>
0 1 2 3 4
jshell> for(int i=0; i<5; i++){
...>     System.out.print(i + " ")
...>
| Error:
| ';' expected
| System.out.print(i + " ")
|
jshell> ■
```

The terminal window has a blue border and a vertical scroll bar on the right. Three blue arrows point from the explanatory text on the right to specific lines in the terminal output: one arrow points to the first line of code, another to the second line, and a third to the line containing the error message.

The semicolon which ends a snippet is optional.

All other semicolons are mandatory.

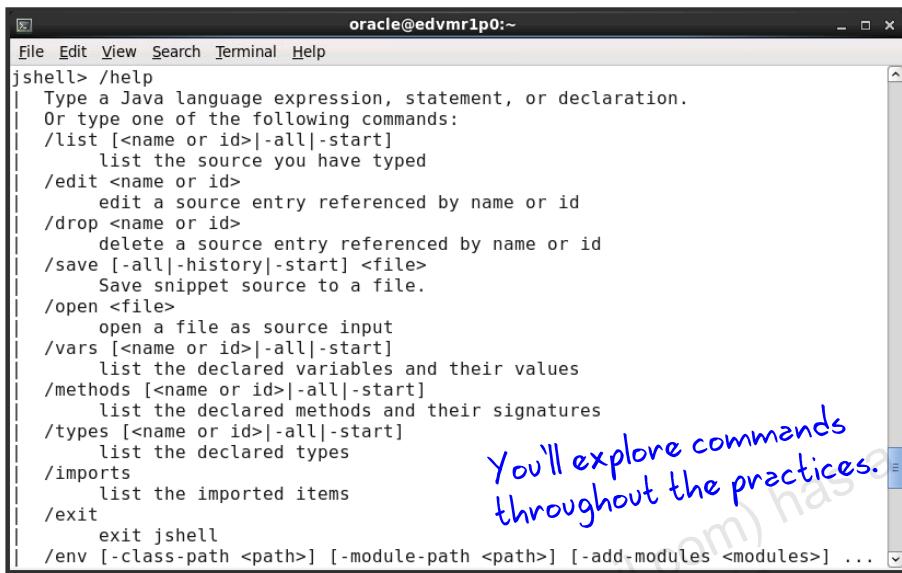


Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Omitting the final semicolon in a snippet is helpful if your snippets are very short, which allows you to work more quickly.

JShell Commands

- Commands allow you to do many things. For example:
 - Get snippet information.
 - Edit a snippet.
 - Affect the JShell session.
 - Show history.
- They're distinguished by a leading slash /
- Enter the /help command to reveal a list of all commands.



The screenshot shows a terminal window titled "oracle@edvmrlp0:~". The command "/help" is entered, and the output lists various JShell commands:

```
jshell> /help
Type a Java language expression, statement, or declaration.
Or type one of the following commands:
/list [<name or id>|-all|-start]
    list the source you have typed
/edit <name or id>
    edit a source entry referenced by name or id
/drop <name or id>
    delete a source entry referenced by name or id
/save [-all|-history|-start] <file>
    Save snippet source to a file.
/open <file>
    open a file as source input
/vars [<name or id>|-all|-start]
    list the declared variables and their values
/methods [<name or id>|-all|-start]
    list the declared methods and their signatures
/types [<name or id>|-all|-start]
    list the declared types
/imports
    list the imported items
/exit
    exit jshell
/env [-class-path <path>] [-module-path <path>] [-add-modules <modules>] ...
```

A handwritten note in blue ink says: "You'll explore commands throughout the practices."

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



Importing Packages

- Several packages are imported into JShell by default.
 - Type `/imports` to reveal the list.
- To test other APIs:
 - Write an import statement for the relevant packages.
 - Ensure the classpath is set appropriately.
 - JShell reports the classpath when it launches.
 - Use the `/classpath` command to set it manually.



A screenshot of a terminal window titled "oracle@edvmr1p0:~". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main area shows the command "jshell> /imports" followed by a list of imported packages:
import java.io.*
import java.math.*
import java.net.*
import java.nio.file.*
import java.util.*
import java.util.concurrent.*
import java.util.function.*
import java.util.prefs.*
import java.util.regex.*
import java.util.stream.*



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

Quiz 17-1

Do you need to make an import statement before creating an `ArrayList` in JShell?

- a. Yes
- b. No



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

Answer: No

Creating an `ArrayList` in an IDE normally requires an import statement. JShell is different because this import is done as part of automatically importing the package `java.util`.

Students may want to enter the `/imports` command or try creating an `ArrayList` in JShell themselves to uncover the answer.

Topics

- Testing code and APIs
- JShell Basics
- JShell in an IDE



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



Why Incorporate JShell in an IDE?

- IDEs perform a lot of work on behalf of developers.
- IDEs are designed to help developers with complex projects.
 - Precision code editing
 - Shortcuts (for example, `sout +Tab` for `System.out.println()`)
 - Auto-complete
 - Tips for fixing broken code
 - Java documentation integration
 - Matching curly braces
- Combine the benefits of two tools.
 - Quick feedback from JShell's REPL
 - Robust assistance from an IDE

```
12 | 1
13 | [1]-> for(int i=0; i<5; i++){
14 |     System.out.println(i++);
15 |
16 | [2]-> for(int i=0; i<5; i++){
17 |     System.out.println(i);
18 |
19 | [3]-> for(int i=0; i<10; i++){
20 |     System.out.println(i);
21 | }
```

Output - Java Shell - project SoI_11_02_JShell

0
2
4
0
1
2
3
4

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



Use Cases

- Experiment with unfamiliar code:
 - A class your colleague wrote
 - A Java API
 - A third party library or module
- Bypass and preserve the existing program.
 - Run quick tests without breaking existing code.
 - Simulate a scenario.
- Test ideas on how to build out your program.
 - Start with simple tests.
 - Gradually build up complexity.
 - Eventually integrate a workable solution with the rest of your program.



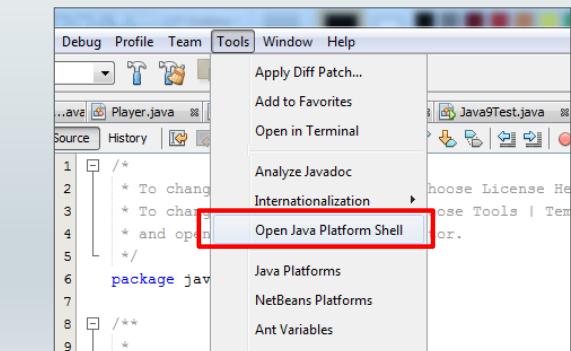
Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Run quick tests. Your existing code is already dedicated to initializing other components of your program. You don't need to break it to run a quick test.

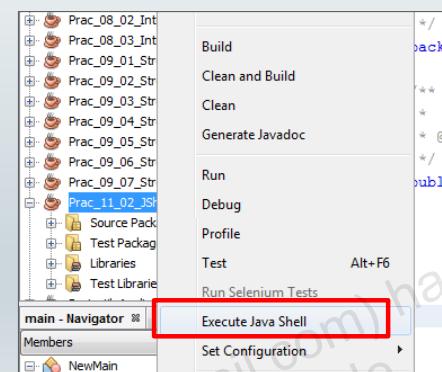
Simulate a scenario. For example, will your program need to run for a long time before it reaches a state where a particular class is instantiated and testable? Bypass this by instantiating the class and simulating the state you need.

Two Ways to Open JShell in NetBeans

1. Open a general JShell session.
 - Select **Tools**.
 - **Open Java Platform Shell**.



2. Open JShell on a specific project.
 - Right-click your project.
 - Select **Execute Java Shell**.
 - Make any necessary imports.



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

When you open JShell on a project, you're specifying a classpath. JShell tells you the classpath when it launches. You'll need to manually write import statements for any packages within the project you wish to experiment with.

Summary

In this lesson, you should have learned how to:

- Explain the REPL process and how it differs from writing code in an IDE
- Launch JShell
- Create JShell scratch variables and snippets
- Identify available JShell commands and other capabilities
- Identify how an IDE enhances the JShell user experience



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

Practices

- 17-1: Variables in JShell
- 17-2: Methods in JShell
- 17-3: Forward-Referencing



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

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

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