



Integrated Cloud Applications & Platform Services



Java SE 8 Fundamentals

Student Guide - Volume II

D83527GC10

Edition 1.0 | December 2016 | D87050

Learn more from Oracle University at education.oracle.com

Authors

Kenneth Somerville
Jill Moritz
Cindy Church
Nick Ristuccia

Technical Contributors and Reviewers

Kenneth Somerville
Peter Fernandez
Jill Moritz
Cindy Church
Nick Ristuccia
Tom McGinn
Mike Williams
Matt Heimer

Editors

Anwesha Ray
Raj Kumar

Graphic Designers

Divya Thallap
Rajiv Chandrabhanu
Daniel Milne

Publishers

Michael Sebastian
Veena Narasimhan
Nita Brozowski
Sumesh Koshy

Copyright © 2014, 2015, 2016 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

- About This Course 1-2
- Audience 1-3
- Course Objectives 1-4
- Schedule 1-6
- Course Environment 1-9
- Test Your Lab Machines 1-11
- How Do You Learn More After the Course? 1-12
- Quiz 1-13
- Summary 1-14

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 (Linux Example): The Tools 2-16
- Examining the Installed JDK (Windows Example): The Libraries 2-17
- Topics 2-18
- Compiling and Running a Java Program 2-19
- Compiling a Program 2-20
- Executing (Testing) a Program 2-21
- Output for a Java Program 2-22
- Exercise 2-1 2-23

Quiz 2-24

Summary 2-25

3 Creating a Java Main Class

Objectives 3-2

Topics 3-3

Java Classes 3-4

Program Structure 3-5

Java Packages 3-6

Using the Java Code Console 3-7

Using the Java Code Console: Creating a New Java Class 3-8

Using the Java Code Console: Creating a New Java Class for an Exercise 3-9

Exercise 3-1: Creating a Class 3-10

Topics 3-11

The main Method 3-12

A main Class Example 3-13

Output to the Console 3-14

Fixing Syntax Errors 3-15

Exercise 3-2: Creating a main Method 3-16

Quiz 3-17

Summary 3-18

4 Data in a Cart

Objectives 4-2

Topics 4-3

Variables 4-4

Variable Types 4-5

Naming a Variable 4-6

Uses of Variables 4-7

Topics 4-8

Variable Declaration and Initialization 4-9

String Concatenation 4-10

String Concatenation Output 4-11

Exercise 4-1: Using String Variables 4-12

Quiz 4-13

Topics 4-14

int and double Values 4-15

Initializing and Assigning Numeric Values 4-16

Topics 4-17

Standard Mathematical Operators 4-18

Increment and Decrement Operators (++ and --) 4-19

Operator Precedence 4-20
Using Parentheses 4-22
Exercise 4-2: Using and Manipulating Numbers 4-23
Quiz 4-24
Summary 4-26

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
Play Time! 5-29
About Java Puzzle Ball 5-30
Tips 5-31

6 Describing Objects and Classes

Interactive Quizzes 6-2
Objectives 6-3
Topics 6-4
Java Puzzle Ball 6-5

Java Puzzle Ball Debrief	6-6
Object-Oriented Programming	6-7
Duke's Choice Order Process	6-8
Characteristics of Objects	6-9
Classes and Instances	6-10
Quiz	6-11
Topics	6-12
The Customer Properties and Behaviors	6-13
The Components of a Class	6-14
Modeling Properties and Behaviors	6-15
Exercise 6-1: Creating the Item Class	6-16
Topics	6-17
Customer Instances	6-18
Object Instances and Instantiation Syntax	6-19
The Dot (.) Operator	6-20
Objects with Another Object as a Property	6-21
Quiz	6-22
Topics	6-23
Accessing Objects by Using a Reference	6-24
Working with Object References	6-25
References to Different Objects	6-28
References and Objects in Memory	6-30
Assigning a Reference to Another Reference	6-31
Two References, One Object	6-32
Exercise 6-2: Modify the ShoppingCart to Use Item Fields	6-33
Topics	6-34
Arrays Are Objects	6-35
Declaring, Instantiating, and Initializing Arrays	6-36
Storing Arrays in Memory	6-37
Storing Arrays of Object References in Memory	6-38
Quiz	6-39
Topics	6-41
Java IDEs	6-42
The NetBeans IDE	6-43
Creating a Java Project	6-44
Creating a Java Class	6-45
Avoiding Syntax Problems	6-46
Compile Error: Variable Not Initialized	6-47
Runtime Error: NullPointerException	6-48
Compiling and Running a Program by Using NetBeans	6-49
Topics	6-50

Soccer Application	6-51
Creating the Soccer Application	6-52
Soccer Web Application	6-53
Summary	6-54
Challenge Questions: Java Puzzle Ball	6-55
Practice 6-1 Overview: Creating Classes for the Soccer League	6-56
Practice 6-2 Overview: Creating a Soccer Game	6-57

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
Java Platform SE 8 Documentation	7-12
Java Platform SE 8: Method Summary	7-13
Java Platform SE 8: Method Detail	7-14
indexOf Method Example	7-15
Topics	7-16
StringBuilder Class	7-17
StringBuilder Advantages over String for Concatenation (or Appending)	7-18
StringBuilder: Declare and Instantiate	7-19
StringBuilder Append	7-20
Quiz	7-21
Exercise 7-1: Use indexOf and substring Methods	7-22
Exercise 7-2: Instantiate the StringBuilder object	7-23
Topics	7-24
Primitive Data Types	7-25
Some New Integral Primitive Types	7-26
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
Play Time!	7-54
Practice 7-1 Overview: Manipulating Text	7-55

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
Java Puzzle Ball	8-25
Java Puzzle Ball Debrief	8-26
Static Methods and Variables	8-27
Example: Setting the Size for a New Item	8-28
Creating and Accessing Static Members	8-29
When to Use Static Methods or Fields	8-30
Some Rules About Static Fields and Methods	8-31
Static Fields and Methods vs. Instance Fields and Methods	8-32
Static Methods and Variables in the Java API	8-33
Examining Static Variables in the JDK Libraries	8-34
Using Static Variables and Methods: System.out.println	8-35
More Static Fields and Methods in the Java API	8-36
Converting Data Values	8-37
Topics	8-38
Passing an Object Reference	8-39
What If There Is a New Object?	8-40
A Shopping Cart Code Example	8-41
Passing by Value	8-42
Reassigning the Reference	8-43
Passing by Value	8-44
Topics	8-45
Method Overloading	8-46
Using Method Overloading	8-47
Method Overloading and the Java API	8-49
Exercise 8-2: Overload a setItemFields Method	8-50
Quiz	8-51
Summary	8-52
Challenge Questions: Java Puzzle Ball	8-53
Practice 8-1 Overview: Using Methods	8-54
Practice 8-2 Overview: Creating Game Data Randomly	8-55
Practice 8-3 Overview: Creating Overloaded Methods	8-56

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
Play Time!	9-26
Practice 9-1 Overview: Encapsulating Fields	9-27
Practice 9-2 Overview: Creating Overloaded Constructors	9-28

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
Java Puzzle Ball	10-14
Java Puzzle Ball Debrief	10-15
Handling Complex Conditions with a Chained if Construct	10-16
Determining the Number of Days in a Month	10-17
Chaining if/else Constructs	10-18
Exercise 10-2: Chaining if Statements	10-19
Topics	10-20
Handling Complex Conditions with a switch Statement	10-21
Coding Complex Conditions: switch	10-22
switch Statement Syntax	10-23

When to Use switch Constructs 10-24
Exercise 10-3: Using switch Construct 10-25
Quiz 10-26
Topics 10-27
Working with an IDE Debugger 10-28
Debugger Basics 10-29
Setting Breakpoints 10-30
The Debug Toolbar 10-31
Viewing Variables 10-32
Summary 10-33
Challenge Question: Java Puzzle Ball 10-34
Practice 10-1 Overview: Using Conditional Statements 10-35
Practice 10-2 Overview: Debugging 10-36

11 Working with Arrays, Loops, and Dates

Interactive Quizzes 11-2
Objectives 11-3
Topics 11-4
Displaying a Date 11-5
Class Names and the Import Statement 11-6
Working with Dates 11-7
Working with Different Calendars 11-9
Some Methods of LocalDate 11-10
Formatting Dates 11-11
Exercise 11-1: Declare a LocalDateTime Object 11-12
Topics 11-13
Using the args Array in the main Method 11-14
Converting String Arguments to Other Types 11-15
Exercise 11-2: Parsing the args Array 11-16
Topics 11-17
Describing Two-Dimensional Arrays 11-18
Declaring a Two-Dimensional Array 11-19
Instantiating a Two-Dimensional Array 11-20
Initializing a Two-Dimensional Array 11-21
Quiz 11-22
Topics 11-23
Some New Types of Loops 11-24
Repeating Behavior 11-25
A while Loop Example 11-26
Coding a while Loop 11-27
while Loop with Counter 11-28

Coding a Standard for Loop	11-29
Standard for Loop Compared to a while loop	11-30
Standard for Loop Compared to an Enhanced for Loop	11-31
do/while Loop to Find the Factorial Value of a Number	11-32
Coding a do/while Loop	11-33
Comparing Loop Constructs	11-34
The continue Keyword	11-35
Exercise 11-3: Processing an Array of Items	11-36
Topics	11-37
Nesting Loops	11-38
Nested for Loop	11-39
Nested while Loop	11-40
Processing a Two-Dimensional Array	11-41
Output from Previous Example	11-42
Quiz	11-43
Topics	11-45
ArrayList Class	11-46
Benefits of the ArrayList Class	11-47
Importing and Declaring an ArrayList	11-48
Working with an ArrayList	11-49
Exercise 11-4: Working with an ArrayList	11-50
Summary	11-51
Play Time!	11-52
Practice 11-1 Overview: Iterating Through Data	11-53
Practice 11-2 Overview: Working with LocalDateTime	11-54

12 Using Inheritance

Objectives	12-2
Topics	12-3
Java Puzzle Ball	12-4
Java Puzzle Ball Debrief	12-5
Inheritance in Java Puzzle Ball	12-6
Implementing Inheritance	12-8
More Inheritance Facts	12-9
Topics	12-10
Duke's Choice Classes: Common Behaviors	12-11
Code Duplication	12-12
Inheritance	12-13
Clothing Class: Part 1	12-14
Shirt Class: Part 1	12-15
Constructor Calls with Inheritance	12-16

Inheritance and Overloaded Constructors	12-17
Exercise 12-1: Creating a Subclass	12-18
Topics	12-19
More on Access Control	12-20
Overriding Methods	12-21
Review: Duke's Choice Class Hierarchy	12-22
Clothing Class: Part 2	12-23
Shirt Class: Part 2	12-24
Overriding a Method: What Happens at Run Time?	12-25
Exercise 12-2: Overriding a Method in the Superclass	12-26
Topics	12-27
Polymorphism	12-28
Superclass and Subclass Relationships	12-29
Using the Superclass as a Reference	12-30
Polymorphism Applied	12-31
Accessing Methods Using a Superclass Reference	12-32
Casting the Reference Type	12-33
instanceof Operator	12-34
Exercise 12-3: Using the instanceof Operator	12-35
Topics	12-36
Abstract Classes	12-37
Extending Abstract Classes	12-39
Summary	12-40
Challenge Questions: Java Puzzle Ball	12-41
Practice 12-1 Overview: Creating a Class Hierarchy	12-43
Practice 12-2 Overview: Creating a GameEvent Hierarchy	12-44

13 Using Interfaces

Interactive Quizzes	13-2
Objectives	13-3
Topics	13-4
The Object Class	13-5
Calling the <code>toString</code> Method	13-6
Overriding <code>toString</code> in Your Classes	13-7
Topics	13-8
The Multiple Inheritance Dilemma	13-9
The Java Interface	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
The Collections Framework	13-19
ArrayList Example	13-20
List Interface	13-21
Example: Arrays.asList	13-22
Exercise 13-1: Converting an Array to an ArrayList	13-24
Topics	13-25
Example: Modifying a List of Names	13-26
Using a Lambda Expression with replaceAll	13-27
Lambda Expressions	13-28
The Enhanced APIs That Use Lambda	13-29
Lambda Types	13-30
The UnaryOperator Lambda Type	13-31
The Predicate Lambda Type	13-32
Exercise 13-2: Using a Predicate Lambda Expression	13-33
Summary	13-34
Practice 13-1 Overview: Overriding the toString Method	13-35
Practice 13-2 Overview: Implementing an Interface	13-36
Practice 13-3 (Optional) Overview: Using a Lambda Expression for Sorting	13-37

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
Interactive Quizzes	14-37
Practice 14-1 Overview: Adding Exception Handling	14-38

15 Deploying and Maintaining the Soccer Application

Objectives	15-2
Topics	15-3
Packages	15-4
Packages Directory Structure	15-5
Packages in NetBeans	15-6
Packages in Source Code	15-7
Topics	15-8
SoccerEnhanced.jar	15-9
Set Main Class of Project	15-10
Creating the JAR File with NetBeans	15-11
Topics	15-13
Client/Server Two-Tier Architecture	15-14
Client/Server Three-Tier Architecture	15-15
Topics	15-16
Client/Server Three-Tier Architecture	15-17
Different Outputs	15-19
The Soccer Application	15-20
IDisplayDataItem Interface	15-21
Running the JAR File from the Command Line	15-22
Text Presentation of the League	15-23

Web Presentation of the League	15-24
Topics	15-25
Enhancing the Application	15-26
Adding a New GameEvent Kickoff	15-27
Game Record Including Kickoff	15-28
Summary	15-29
Course Summary	15-30

16 Oracle Cloud

Agenda	16-2
What is Cloud?	16-3
What is Cloud Computing?	16-4
History – Cloud Evolution	16-5
Components of Cloud Computing	16-6
Characteristics of Cloud	16-7
Cloud Deployment Models	16-8
Cloud Service Models	16-9
Industry Shifting from On-Premises to the Cloud	16-13
Oracle IaaS Overview	16-15
Oracle PaaS Overview	16-16
Oracle SaaS Overview	16-17
Summary	16-18

17 Oracle Application Container Cloud Service Overview

Objectives	17-2
Oracle Application Container Cloud Service	17-3
Oracle Application Container Cloud	17-4
Polyglot Platform	17-5
Open Platform	17-6
Container-based Application Platform as a Service	17-7
Elastic Scaling	17-8
Profiling	17-9
Manageable	17-10
Deploy—Application Archive (Zip)	17-12
Application Deployment	17-13
Application Container Cloud Architecture	17-14
Load Balancer	17-15
Oracle Developer Cloud Service	17-16
Developer Cloud Service – Easy Adoption/Integration	17-17
Application Container Cloud Service Advantages	17-19
Summary	17-20

A Java Puzzle Ball Challenge Questions Answered

- Topics A-2
- Question 1 A-3
- Question 2 A-4
- Question 3 A-5
- Question 4 A-6
- Topics A-7
- Question 1 A-8
- Topics A-9
- Question 1 A-10
- Topics A-12
- Question 1 A-13
- Topics A-14
- Question 1 A-15
- Question 2 A-16

B Introducing the Java Technology

- Java's Place in the World B-2
- Java Desktops B-3
- Java Mobile Phones B-4
- Java TV and Card B-5
- The Story of Java B-6
- Identifying Java Technology Product Groups B-7
- Java SE B-8
- Java EE B-9
- Java ME B-10
- Java Card B-11
- Product Life Cycle (PLC) Stages B-12

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.

Ramon Robles Garcia (ramonrobles22@yahoo.com.mx) has a
non-transferable license to use this Student Guide.

Using Encapsulation

9

ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Interactive Quizzes



ORACLE®

Copyright© 2014, 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 your quiz file from labs > Quizzes > Java SE 8 Fundamentals Quiz.html. Click the links for the lessons titled "Describing Objects and Classes," "Manipulating and Formatting the Data in Your Program," and "Creating and Using Methods."

Objectives

After completing this lesson, you should be able to:

- Use an access modifier to make fields and methods private
- Create get and set methods to control access to private fields
- Define encapsulation as “information hiding”
- Implement encapsulation in a class using the NetBeans refactor feature
- Create an overloaded constructor and use it to instantiate an object



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Topics

- Access control
- Encapsulation
- Overloading constructors

What Is Access Control?

Access control allows you to:

- Hide fields and methods from other classes
- Determine how internal data gets changed
- Keep the implementation separate from the public interface
 - Public interface:
`setPrice(Customer cust)`
 - Implementation:
`public void setPrice(Customer cust) {
 // set price discount relative to customer
}`



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Access control allows you to hide internal data and functionality in a class. In this lesson, you distinguish between the public interface of a class and the actual implementation of that interface.

- The public interface is what you see when you look up a class in the JDK API documentation. You get just the information you need in order to use a class. That is, the signatures for public methods, and data types of any public fields.
- The implementation of a class is the code itself, and also any private methods or fields that are used by that class. These are the internal workings of a class and it is not necessary to expose them to another class.

Access Modifiers

- `public`: Accessible by anyone
- `private`: Accessible only within the class

```
1 public class Item {  
2     // Base price  
3     private double price = 15.50;  
4  
5     public void setPrice(Customer cust){  
6         if (cust.hasLoyaltyDiscount()) {  
7             price = price*.85; }  
8     }  
9 }
```



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

When a field is declared as public, any other class can access and potentially change the field's value. This is often problematic. It could be that the field represents sensitive data, such as a social security number, or that some type of logic or manipulation of the data may be required in order to safely modify the data. In the code example, the shirt price is declared in a private method. You would not want outside objects, such as a customer, to be able to freely manipulate the price of an item.

Access from Another Class

```
1 public class Item {  
2     private double price = 15.50;  
3     public void setPrice(Customer cust){  
4         if (cust.hasLoyaltyDiscount ()) {  
5             price = price*.85; }  
6     }  
7 }  
8 public class Order{  
9     public static void main(String args[]){  
10         Customer cust = new Customer(int ID);  
11         Item item = new Item();    └─ Won't compile  
12         item.price = 10.00;      └─ You don't need to know  
13         item.setPrice(cust);   how setPrice works in  
14     }  
15 }
```

Annotations:

- item.price = 10.00; └─ Won't compile
- item.setPrice(cust); └─ You don't need to know how setPrice works in order to use it.



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Another Example

The data type of the field does not match the data type of the data used to set the field.

```
1 private int phone;
2 public void setPhoneNumber(String s_num) {
3     // parse out the dashes and parentheses from the
4     // String first
5     this.phone = Integer.parseInt(s_num);
6 }
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

It may be that the data representing someone's phone number may be collected as a string, including spaces, dashes, and parentheses. If the phone number is represented internally as an `int`, then the setter method for the phone number will need to parse out spaces, dashes, and parentheses first, and then convert the `String` to an `int`. The `parseInt` method of `Integer` is covered in the “Using Encapsulation” lesson.

Using Access Control on Methods

```
1 public class Item {  
2     private int id;  
3     private String desc;  
4     private double price;  
5     private static int nextId = 1;  
6  
7     public Item(){  
8         setId();    Called from within a  
public method  
9         desc = "--description required--";  
10        price = 0.00;  
11    }  
12  
13    private void setId() {    Private method  
14        id = Item.nextId++;  
15    }
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Here you see a private method that sets a new unique ID for an item. It is not necessary to expose this functionality to another class. The `setId` method is called from the public constructor method as part of its implementation.

Topics

- Access control
- Encapsulation
- Overloading constructors

Encapsulation

- Encapsulation means hiding object fields. It uses access control to hide the fields.
 - Safe access is provided by getter and setter methods.
 - In setter methods, use code to ensure that values are valid.
- Encapsulation mandates programming to the interface:
 - A method can change the data type to match the field.
 - A class can be changed as long as interface remains same.
- Encapsulation encourages good object-oriented (OO) design.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Get and Set Methods

```
1 public class Shirt {
2     private int shirtID = 0;          // Default ID for the shirt
3     private String description = "-description required-"; // default
4     private char colorCode = 'U'; //R=Red, B=Blue, G=Green, U=Unset
5     private double price = 0.0;      // Default price for all items
6
7     public char getColorCode() {
8         return colorCode;
9     }
10    public void setColorCode(char newCode) {
11        colorCode = newCode;
12    }
13    // Additional get and set methods for shirtID, description,
14    // and price would follow
15
16 } // end of class
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

If you make attributes private, how can another object access them? One object can access the private attributes of a second object if the second object provides public methods for each of the operations that are to be performed on the value of an attribute.

For example, it is recommended that all fields of a class should be private, and those that need to be accessed should have public methods for setting and getting their values.

This ensures that, at some future time, the actual field type itself could be changed, if that were advantageous. Or the getter or setter methods could be modified to control how the value could be changed, such as the value of the `colorCode`.

Why Use Setter and Getter Methods?

```
1 public class ShirtTest {  
2     public static void main (String[] args) {  
3         Shirt theShirt = new Shirt();  
4         char colorCode;  
5         // Set a valid colorCode  
6         theShirt.setColorCode ('R');  
7         colorCode = theShirt.getColorCode();  
8         System.out.println("Color Code: " + colorCode);  
9         // Set an invalid color code  
10        theShirt.setColorCode ('Z');  Not a valid color code  
11        colorCode = theShirt.getColorCode();  
12        System.out.println("Color Code: " + colorCode);  
13    }  
14 ...
```

Output:

```
Color Code: R  
Color Code: Z
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Though the code for the `Shirt` class is syntactically correct, the `setColorCode` method does not contain any logic to ensure that the correct values are set.

The code example in the slide successfully sets an invalid color code in the `Shirt` object.

However, because `ShirtTest` accesses a private field on `Shirt` using a setter method, `Shirt` can now be recoded without modifying any of the classes that depend on it.

In the code example above, starting with line 6, the `ShirtTest` class is setting and getting a valid `colorCode`. Starting with line 10, the `ShirtTest` class is setting an invalid `colorCode` and confirming that invalid setting.

Setter Method with Checking

```
15 public void setColorCode(char newCode) {  
16     if (newCode == 'R') {  
17         colorCode = newCode;  
18         return;  
19     }  
16     if (newCode == 'G') {  
17         colorCode = newCode;  
18         return;  
19     }  
16     if (newCode == 'B') {  
17         colorCode = newCode;  
18         return;  
19     }  
19     System.out.println("Invalid colorCode. Use R, G, or B");  
20 }  
21}
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

In the slide is another version of the `Shirt` class. However, in this class, before setting the value, the setter method ensures that the value is valid. If it is not valid, the `colorCode` field remains unchanged and an error message is printed.

Note: Void type methods can have return statements. They just cannot return any values.

Using Setter and Getter Methods

```
1 public class ShirtTest {  
2     public static void main (String[] args) {  
3         Shirt theShirt = new Shirt();  
4         System.out.println("Color Code: " + theShirt.getColorCode());  
5  
6         // Try to set an invalid color code  
7         Shirt1.setColorCode('Z');    ————— Not a valid color code  
8         System.out.println("Color Code: " + theShirt.getColorCode());  
9     }  
}
```

Output:

```
Color Code: U ————— Before call to setColorCode() – shows default value.  
Invalid colorCode. Use R, G, or B ————— call to setColorCode prints error message  
Color Code: U ————— colorCode not modified by invalid argument passed to setColorCode()
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Building on the previous slides, before the call to `setColorCode`, the default color value of U (unset) is printed. If you call `setColorCode` with an invalid code, the color code is not modified and the default value, U, is still the value. Additionally, you receive an error message that tells you to use the valid color codes, which are R, G, and B.

Exercise 9-1: Encapsulate a Class

In this exercise, you encapsulate the Customer class.

- Change access modifiers so that fields can be read or changed only through public methods.
- Allow the `ssn` field to be read but not modified.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- Open the Java Code Console and access 09-Encaps > Exercise1.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

- Access control
- Encapsulation
- Overloading constructors

Initializing a Shirt Object

Explicitly:

```
1 public class ShirtTest {  
2     public static void main (String[] args) {  
3         Shirt theShirt = new Shirt();  
4  
5         // Set values for the Shirt  
6         theShirt.setColorCode('R');  
7         theShirt.setDescription("Outdoors shirt");  
8         theShirt.price(39.99);  
9     }  
10 }
```

Using a constructor:

```
Shirt theShirt = new Shirt('R', "Outdoors shirt", 39.99);
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Assuming that you now have setters for all the private fields of `Shirt`, you could now instantiate and initialize a `Shirt` object by instantiating it and then setting the various fields through the setter methods.

However, Java provides a much more convenient way to instantiate and initialize an object by using a special method called a *constructor*.

Constructors

- Constructors are usually used to initialize fields in an object.
 - They can receive arguments.
 - When you create a constructor with arguments, it removes the default no-argument constructor.

ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

All classes have at least one constructor.

If the code does not include an explicit constructor, the Java compiler automatically supplies a no-argument constructor. This is called the default constructor.

Shirt Constructor with Arguments

```
1 public class Shirt {  
2     public int shirtID = 0;          // Default ID for the shirt  
3     public String description = "-description required-"; // default  
4     private char colorCode = 'U'; //R=Red, B=Blue, G=Green, U=Unset  
5     public double price = 0.0;      // Default price all items  
6  
7     // This constructor takes three argument  
8     public Shirt(char colorCode, String desc, double price ) {  
9         setColorCode(colorCode);  
10        setDescription(desc);  
11        setPrice(price);  
12    }  
13}
```



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

The Shirt example shown in the slide has a constructor that accepts three values to initialize three of the object's fields. Because `setColorCode` ensures that an invalid code cannot be set, the constructor can just call this method.

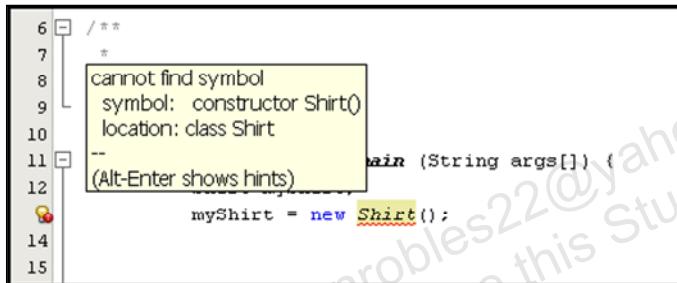
Default Constructor and Constructor with Args

When you create a constructor with arguments, the default constructor is no longer created by the compiler.

```
// default constructor  
public Shirt ()
```

This constructor is not in the source code. It only exists if no constructor is explicitly defined.

```
// Constructor with args  
public Shirt (char color, String desc, double price)
```



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

When you explicitly create an overloaded constructor, it replaces the default no-argument constructor.

You may be wondering why you have been able to instantiate a `Shirt` object with `Shirt myShirt = new Shirt()` even if you did not actually create that no-argument constructor. If there is no explicit constructor in a class, Java assumes that you want to be able to instantiate the class, and gives you an *implicit* default no-argument constructor. Otherwise, how could you instantiate the class?

The example above shows a new constructor that takes arguments. When you do this, Java removes the implicit default constructor. Therefore, if you try to use `Shirt myShirt = new Shirt()`, the compiler cannot find this constructor because it no longer exists.

Overloading Constructors

```
1 public class Shirt {  
2     ... //fields  
3  
4     // No-argument constructor  
5     public Shirt() {  
6         setColorCode('U');  
7     }  
8     // 1 argument constructor  
9     public Shirt(char colorCode) {  
10        setColorCode(colorCode);  
11    }  
12    // 2 argument constructor  
13    public Shirt(char colorCode, double price) {  
14        this(colorCode);  
15        setPrice(price);  
16    }  
}
```

If required, must be added explicitly

Calling the 1 argument constructor



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

The code in the slide shows three overloaded constructors:

- A default no-argument constructor
- A constructor with one parameter (a `char`)
- A constructor with two parameters (a `char` and a `double`)

This third constructor sets both the `colorCode` field and the `price` field. Notice, however, that the syntax where it sets the `colorCode` field is one that you have not seen yet. It would be possible to set `colorCode` with a simple call to `setColorCode()` just as the previous constructor does, but there is another option, as shown here.

You can chain the constructors by calling the second constructor in the first line of the third constructor using the following syntax:

```
this(argument);
```

The keyword `this` is a reference to the current object. In this case, it references the constructor method from this class whose signature matches.

This technique of chaining constructors is especially useful when one constructor has some (perhaps quite complex) code associated with setting fields. You would not want to duplicate this code in another constructor and so you would chain the constructors.

Quiz

What is the default constructor for the following class?

```
public class Penny {  
    String name = "lane";  
}
```

- a. public Penny(String name)
- b. public Penny()
- c. class()
- d. String()
- e. private Penny()

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b

Exercise 9-2: Create an Overloaded Constructor

In this exercise, you:

- Add an overloaded constructor to the `Customer` class
- Create a new `Customer` object by calling the overloaded constructor



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- Open the Java Code Console and access 09-ManipulateFormat > Exercise2.
- Follow the instructions below the code editor to modify the `Customer` class and then instantiate a `Customer` object using the new constructor from the `ShoppingCart` class.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Summary

In this lesson, you should have learned how to:

- Use public and private access modifiers
- Restrict access to fields and methods using encapsulation
- Implement encapsulation in a class
- Overload a constructor by adding method parameters to a constructor



ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Play Time!

Play **Basic Puzzle 12** before the next lesson titled “More on Conditionals.”

Consider the following:

What happens if the ball strikes the blade?



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You will be asked this question in the lesson titled “More on Conditionals.”

Practice 9-1 Overview: Encapsulating Fields

This practice covers using the NetBeans refactor feature to encapsulate the fields of several classes from the Soccer application.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 9-2 Overview: Creating Overloaded Constructors

This practice covers the following topics:

- Creating overloaded constructors for several classes of the Soccer application
- Initializing fields within the custom constructor methods



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

10

More on Conditionals

ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Correctly use all of the conditional operators
- Test equality between string values
- Chain an `if/else` statement to achieve the desired result
- Use a `switch` statement to achieve the desired result
- Debug your Java code by using the NetBeans debugger to step through code line by line and view variable values



ORACLE

Copyright © 2014, 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

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© 2014, Oracle and/or its affiliates. All rights reserved.

By way of review, here you see a list of all the relational operators. Previously, you used the `==` operator to test equality for numeric values. However, String variables are handled differently because a String variable is an object reference, rather than a primitive value.

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.");  
        }  
    }  
}
```



ORACLE

Copyright© 2014, 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.");  
        }  
    }  
}
```



ORACLE

Copyright© 2014, 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.");  
        }  
    }  
}
```



ORACLE

Copyright© 2014, 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© 2014, 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© 2014, Oracle and/or its affiliates. All rights reserved.

Relational operators are often used in conjunction with conditional operators. You might need to make a single decision based on more than one condition. Under such circumstances, you can use conditional operators to evaluate complex conditions as a whole.

The table in the slide lists the common conditional operators in the Java programming language. For example, all of the examples in the table yield a boolean result of `false`.

Discussion: What relational and conditional operators are expressed in the following paragraph?

- If the toy is red, I will purchase it. However, if the toy is yellow and costs less than a red item, I will also purchase it. If the toy is yellow and costs the same as or more than another red item, I will not purchase it. Finally, if the toy is green, I will not purchase it.

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.	<code>? :</code>	<code>condition ? value1 : value2</code> Example: <code>int x = 2, y = 5, z = 0;</code> <code>z = (y < x) ? x : y;</code>

Equivalent statements

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

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

ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

The ternary operator is a conditional operator that takes three operands. It has a more compact syntax than an `if/else` statement.

Use the ternary operator instead of an `if/else` statement if you want to make your code shorter. The three operands shown in the example above are described here:

- `(y < x)`: This is the boolean expression (condition) being evaluated.
- `? x`: If `(y < x)` is true, `z` will be assigned the value of `x`.
- `: y`: If `(y < x)` is false, `z` will be assigned the value of `y`.

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© 2014, 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;
```



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 10-MoreConditions > Exercise1
- Follow the instructions below the code editor to write a `ternary` statement that solves the same problem as the `if/else` statement in this Java class (and shown above).
- Print the result.
- Run the file to test your code.

Note: If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

Java Puzzle Ball

Have you played through **Basic Puzzle 12**?

Consider the following:

What happens if the ball strikes the blade?



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This is the question you were asked to think about before this lesson began. What conclusions did you reach? In this topic, some Java concepts and principles will be discussed that can help explain this behavior.

Java Puzzle Ball Debrief

- What happens if the ball strikes the blade?
 - if the ball strikes the blade:
 - Transform the ball into a blade
 - if the ball is a blade && it strikes the fan:
 - The ball is blown in the direction of the fan
 - if the ball is a blade && it strikes any object other than the fan || blade:
 - Destroy that object
 - Transform the ball back into a ball



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Handling Complex Conditions with a Chained if Construct

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

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© 2014, 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© 2014, 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

In this exercise, you write a `calcDiscount` method that determines the discount for three different customer types:

- Nonprofits get a discount of 10% if total > 900, else 8%.
- Private customers get a discount of 7% if total > 900, else no discount.
- Corporations get a discount of 8% if total > 500, else 5%.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 10-MoreConditions > Exercise2.
- Click the Order tab and follow the instructions below the code editor to code the body of the `calcDiscount` method as described above.
- Click the ShoppingCart tab and click Run to test your code.

Note: If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

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

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© 2014, 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© 2014, Oracle and/or its affiliates. All rights reserved.

The `switch` construct helps you avoid confusing code because it simplifies the organization of the various branches of code that can be executed.

The syntax for the `switch` construct is shown in the slide, where:

- The `switch` keyword indicates a `switch statement`
- `variable` is the variable whose value you want to test. Alternatively, you could use an `expression`. The `variable` (or the result of the `expression`) can be only of type `char`, `byte`, `short`, `int`, or `String`.
- The `case` keyword indicates a value that you are testing. A combination of the `case` keyword and a `literal value` is referred to as a `case label`.
- `literal value` is any valid value that a variable might contain. You can have a `case label` for each value that you want to test. Literal values can be constants (final variables such as `CORP`, `PRIVATE`, or `NONPROFIT` used in the previous exercise), literals (such as '`A`' or `10`), or both.
- The `break` statement is an optional keyword that causes the code execution to immediately exit the `switch statement`. Without a `break` statement, all `code block` statements following the accepted `case statement` are executed (until a `break` statement or the end of the `switch construct` is reached).

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 value 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 }
```

Known values

ORACLE

Copyright© 2014, 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

In this exercise, you modify the `calcDiscount` method to use a `switch` construct, instead of a chained `if` construct:

- Use a ternary operator instead of a nested `if` within each case block.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 10-MoreConditions > Exercise3.
- Follow the instructions below the code editor to change the `calcDiscount` method of the `Order` class to use a `switch` construct instead of the chained `if` construct. You may wish to just comment out the chained `if` statement so that you will be able to reference it in order to duplicate the logic.
- Use a ternary statement in each `switch` block to replace the nested `if` statement logic.
- Test it by running the `ShoppingCart` class.

Note: If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

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© 2014, 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

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?)



ORACLE

Copyright© 2014, 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.

 ORACLE

Copyright© 2014, 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 7 and 20.

ORACLE

Copyright© 2014, 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



ORACLE

Copyright© 2014, 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

The screenshot shows a Java debugger interface. A switch statement is being debugged. A red circle highlights the first line of the switch block (line 9). A green arrow points to the current line of execution (line 15). The code is as follows:

```
switch(month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        System.out.println("31 days in the month.");
        break;
    case 2:
        if(!isLeapYear){
            System.out.println("28 days in the month.");
        }
        else{
            System.out.println("29 days in the month.");
        }
        break;
    case 4: case 6: case 9: case 11:
}
```

The 'Breakpoints' tab is selected in the bottom navigation bar. The 'Variables' tab is also visible. In the variables table, the 'isLeapYear' variable is highlighted with a red box and the label 'Value of variables'. The value is shown as 'true'.

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

ORACLE

Copyright© 2014, 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

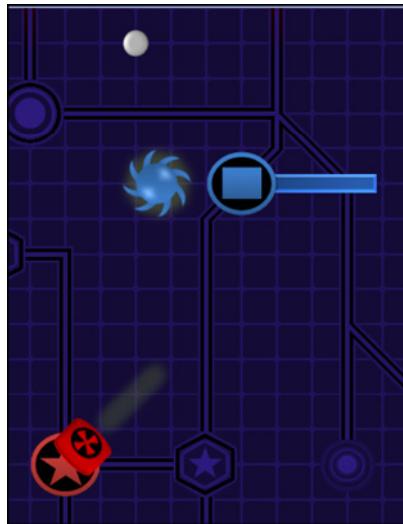


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Challenge Question: Java Puzzle Ball

What type of conditional construct would you use to handle the behavior of the blade?



ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

When you have an opportunity to play the game, see whether you can “pseudocode” the logic needed to control the behavior of the ball when it has been turned into a blade. You are now familiar with several conditional constructs. Choose the one that you think works best and is easiest to read.

For some possible answers to these questions and more discussion, see “Appendix A: Java Puzzle Ball Challenge Questions Answered.”

Practice 10-1 Overview: Using Conditional Statements

This practice covers enhancing the `getDescription` method of the Game class to announce the name of the winning team.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10-2 Overview: Debugging

This practice covers the following topics:

- Enhancing the `showBestTeam` method to differentiate between teams with the same number of points
- Using the NetBeans debugger to step through the code line by line

11

Working with Arrays, Loops, and Dates

ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Interactive Quizzes



ORACLE®

Copyright© 2014, 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 your quiz file from labs > Quizzes > Java SE 8 Fundamentals Quiz.html. Click the links for the lessons titled "Using Encapsulation" and "More on Conditionals."

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
- Correctly declare and instantiate a two-dimensional array
- Code a nested `while` loop to achieve the desired result
- Use a nested `for` loop to process a two-dimensional array
- Code a `do/while` loop to achieve the desired result
- Use an `ArrayList` to store and manipulate lists of Objects
- Evaluate and select the best type of loop to use for a given programming requirement



ORACLE

Copyright © 2014, 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



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: 2013-12-20

- `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.



ORACLE

Copyright© 2014, 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 2013-12-20 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© 2014, 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© 2014, 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.

- `java.time.format.DateTimeFormatter` provides static and nonstatic methods to format dates using a specific format style. It also provides static constants (variables) that represent specific formats.
- The example in the slide uses a static constant variable, `ISO_LOCAL_DATE`, from the `DateTimeFormatter` class. It is passed as an argument into the `format` method of `Date` object:
`myDate.format(DateTimeFormatter.ISO_LOCAL_DATE)`
- A formatted `String` representing the `LocalDateTime` is returned from the `format` method. For a more complete discussion of date formatting, see “Some Methods of `LocalDate`” later in the lesson.

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© 2014, 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);`
 - `myDate.plusDays (8);`
 - `(long monthsToSubtract)`
 - `(long daysToAdd)`
- Static methods:
 - `of(int year, Month month, int dayOfMonth)`
 - `parse(CharSequence text, DateTimeFormatter formatter)`
 - `now()`



Copyright© 2014, 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);  
7 System.out.println("Date in ISO_DATE_TIME format: "  
8 + sdate);  
9  
10 String fdate =  
11    today.format(DateTimeFormatter.ofLocalizedDateTime  
12        (FormatStyle.MEDIUM));  
13 System.out.println("Formatted with MEDIUM FormatStyle: "  
14 + fdate);  
15  
16
```

Format the date in standard ISO format.

Localized date time in Medium format

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 © 2014, Oracle and/or its affiliates. All rights reserved.

Exercise 11-1: Declare a LocalDateTime Object

1. Declare and initialize a `LocalDateTime` object.
2. Print and format the `OrderDate`.
3. Run the code.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- Open the Java Code Console and access 11-ArraysLoopsDates > Exercise1.
- Follow the instructions below the code editor.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

ORACLE

Copyright © 2014, 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!
```

Goes into args[1]
Goes into args[0]

- 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© 2014, 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© 2014, Oracle and/or its affiliates. All rights reserved.

The `main` method treats everything you type as a literal string. If you want to use the string representation of a number in an expression, you must convert the string to its numerical equivalent.

The `parseInt` static method of the `Integer` class is used to convert the `String` representation of each number to an `int` so they can be added.

Note that the parentheses around `arg1 + arg2` are required so that the `+` sign indicates addition rather than concatenation. The `System.out.println` method converts any argument passed to it to a `String`. We want it to add the numbers first, and *then* convert the total to a `String`.

Exercise 11-2: Parsing the args Array

In this exercise, you parse the `args` array in the `main` method to get the command-line arguments and assign them to local variables.

- To enter command-line arguments, click the Configure button and enter the values, separated by a space.



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 11-ArraysLoopsDates > Exercise2.
- Follow the instructions below the code editor to parse the `args` array in the `main` method, saving the arguments to local variables and then printing them.
- Enter command arguments as described in the slide above.
- Run the file to test your code.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Describing Two-Dimensional Arrays

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 1							
Week 2							
Week 3							
Week 4							



Copyright© 2014, 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

Example:

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

Syntax:

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



Copyright© 2014, 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

Example:

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

Syntax:

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

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

ORACLE

Copyright© 2014, 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© 2014, 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



ORACLE

Copyright © 2014, 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

ORACLE

Copyright © 2014, 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© 2014, 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;
```

ORACLE®

Copyright© 2014, 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.

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     }  
}
```

Boolean expression

Body of the loop



Copyright© 2014, 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.

Coding a while Loop

Syntax:

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



Copyright© 2014, 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.

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© 2014, 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.

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;`

Syntax:

```
01 for (<type> counter = n;  
02      <boolean_expression>;  
03      <counter_increment>){  
04          code_block;  
05 }
```



Copyright© 2014, 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;                                boolean expression
02 while (i < 3) {                            }
03     System.out.println(" * ");
04     i++;                                     Increment counter
05 }
```

Initialize counter

Increment counter

`for` loop

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

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this slide, you see a `while` loop example at the top of the slide. At the bottom, you see the same logic implemented using a standard `for` loop.

The three essential elements of a `while` loop are also present in the `for` loop, but in different places.

1. The counter (`i`) is declared and initialized outside the `while` loop on line 1.
2. The counter is incremented in the `while` loop on line 4.
3. The boolean expression that determines the number of loop iterations is within the parentheses for the `while` loop on line 2.
4. In the `for` loop, all three elements occur within the parentheses as indicated in the slide.

The output for each statement is the same.

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

ORACLE

Copyright © 2014, 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

ORACLE

Copyright © 2014, 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.

ORACLE

Copyright© 2014, 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.

ORACLE

Copyright© 2014, 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© 2014, 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

In this exercise you:

- Process an array of items to calculate the Shopping Cart total
- Skip any items that are back ordered
- Display the total

```
Item.java ShoppingCart.java
1 package ex11_2_exercise;
2
3 public class ShoppingCart {
4     Item[] items = {new Item("Shirt",25.60),
5                     new Item("WristBand", 1.00),
6                     new Item("Pants",35.99)};
7
8     public static void main(String[] args){
9         ShoppingCart cart = new ShoppingCart();
10        cart.displayTotal();
11    }
12
13
14     // Use a standard for loop to iterate through the items array, adding up the total price
15     // Skip any items that are back ordered. Display the Shopping Cart total.
16     public void displayTotal(){
17
18    }
19
20 }
```



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 11-ArraysLoopsDates > Exercise3.
- Follow the instructions below the code editor. Code the `displayTotal` method of the `ShoppingCart` class so that it will iterate through the `items` array and print out the total for the Shopping Cart.
- Skip any items that are back ordered. (You might want to examine the `Item` class to see how this setting is determined.)
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

ORACLE

Copyright © 2014, 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...

ORACLE

Copyright© 2014, 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)
```

ORACLE

Copyright© 2014, 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© 2014, 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© 2014, Oracle and/or its affiliates. All rights reserved.

This example illustrates the process of a two-dimensional array called `sales`. The `sales` array has three rows of four columns. The rows represent years and the columns represent the quarters of the year.

- The `initArray` method is called on line 2 to initialize the array with values.
- An opening message is printed on lines 3 and 4.
- On line 5, you see the outer `for` loop defined. The outer loop iterates through the three years. Notice that `i` is used as the counter for the outer loop.
- On line 6, you see the inner `for` loop defined. The inner loop iterates through each quarter of the year. It uses `j` as a counter. For each quarter of the year, it prints the quarter number (Q1, Q2, ...) plus the quarterly sales value in the element of the array indicated by the row number (`i`) and the column number (`j`).

Note: The “\t” character combination creates a tab indent.

- Notice that the quarter number is calculated as `j+1`. Because array elements start with 0, the index number of the first element will be 0, the second element index will be 1, and so on. To translate this into a quarter number, you add 1 to it.
- The output can be seen in the next slide.

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© 2014, 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

 ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: c

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

 ORACLE

Copyright © 2014, 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

ORACLE

Copyright © 2014, 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© 2014, 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 8 Programming* 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
```

ORACLE

Copyright© 2014, 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© 2014, 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;           | Declare an ArrayList of  
02 names = new ArrayList();          | Strings.  
03  
04 names.add("Jamie");             | Instantiate the ArrayList.  
05 names.add("Gustav");  
06 names.add("Alisa");  
07 names.add("Jose");  
08 names.add(2, "Prashant");        | Initialize it.  
09  
10 names.remove(0);                | Modify it.  
11 names.remove(names.size() - 1);  
12 names.remove("Gustav");  
13  
14 System.out.println(names);
```



Copyright© 2014, 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

In this exercise, you:

- Declare, instantiate, and initialize an `ArrayList` of `Strings`
- Use two different `add` methods
 - `add(E element)`
 - `add(int index, E element)`
- Use the following methods to find and remove a specific element if it exists
 - `contains (Object element)`
 - `remove (Object element)`



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 11-ArraysLoopsDates > Exercise4.
- Follow the instructions below the code editor. Initialize and manipulate an `ArrayList`, using various methods of the `ArrayList` class.
- Note that, in this case, the `ArrayList` contains `String` objects, so the `Object` reference in the above method signatures can be a `String` literal.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

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



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Play Time!

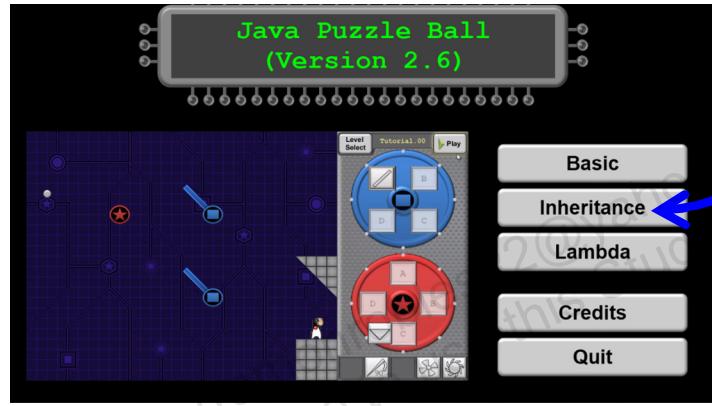


Play **Inheritance Puzzles 1, 2, and 3** before the next lesson titled “Using Inheritance.”

Consider the following:

What is inheritance?

Why are these considered “Inheritance” puzzles?



Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

You will be asked this question in the lesson titled “Using Inheritance.”

Practice 11-1 Overview: Iterating Through Data

This practice covers the following topics:

- Converting a comma-separated list of names to an array of names
- Processing the array using a `for` loop
- Using a nested loop to populate an `ArrayList` of games



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 11-2 Overview: Working with `LocalDateTime`

This practice covers working with a few classes and methods from the `java.time` package in order to show date information for games played.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

12

Using Inheritance

ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

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 a 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 nonabstract classes
- Create a class hierarchy by extending an abstract class



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Topics

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

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Puzzle Ball

Have you played through **Inheritance Puzzle 3?**

Consider the following:

What is inheritance?

Why are these considered “Inheritance” puzzles?



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Puzzle Ball Debrief

What is inheritance?

- **Inheritance** allows one class to be derived from another.
 - A child inherits properties and behaviors of the parent.
 - A child *class* inherits the fields and method of a parent *class*.
- In the game:
 - Blue shapes also appear on *green* bumpers



ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, you will examine the object-oriented concept of inheritance.

- Inheritance is a mechanism by which a class can be derived from another class, just as a child derives certain characteristics from the parent.
- You can see this reflected in the game. When you drag an icon to the blue wheel, it affects green objects as well as blue objects in the field of play. The green wheel derives its characteristics from the blue wheel.

Inheritance in Java Puzzle Ball

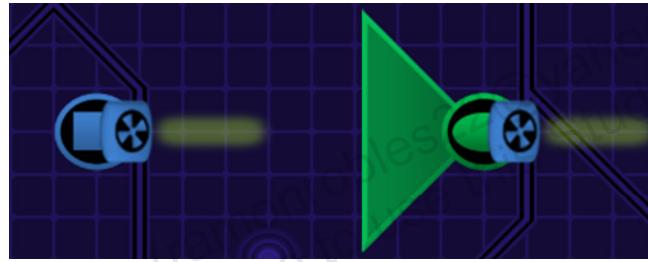
Inheritance Puzzle 1:

- Methods for deflecting the ball that were originally assigned to Blue Bumpers are also found on Green Bumpers.



Inheritance Puzzle 2:

- Green Bumpers contain methods from Blue Bumpers, PLUS methods unique to Green Bumpers.



ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

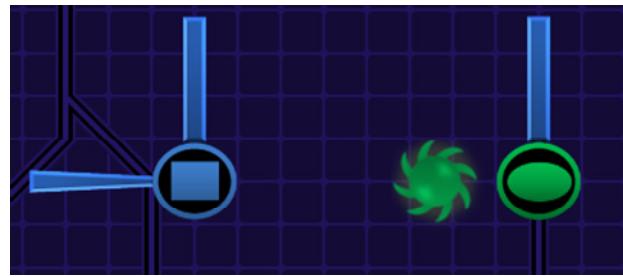
In the first image, both Blue Bumpers and Green Bumpers share the Triangle Wall.

In the second image, both Blue Bumpers and Green Bumpers share the fan. In addition, Green Bumpers uniquely have a Triangle Wall.

Inheritance in Java Puzzle Ball

Inheritance Puzzle 3:

- If Green Bumpers inherit unwanted Blue Bumper methods, it is possible to **override**, or replace those methods.



ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Green Bumpers have overridden the rotation wall with a blade.

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!

ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

In the code example above, an inheritance relationship between the `Shirt` class and its parent, the `Clothing` class, is defined. The keyword `extends` creates the inheritance relationship:

```
public class Shirt extends Clothing...
```

As a result, `Shirt` objects share the `display` and `setSize` methods of the `Clothing` class. Although these methods are not actually written in the `Shirt` class, they may be used by all `Shirt` objects. Therefore, the following code can be successfully compiled and run:

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

More Inheritance Facts

- The parent class is the **superclass**.
- The child class is the **subclass**.
- 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;
    }
}
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Topics

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

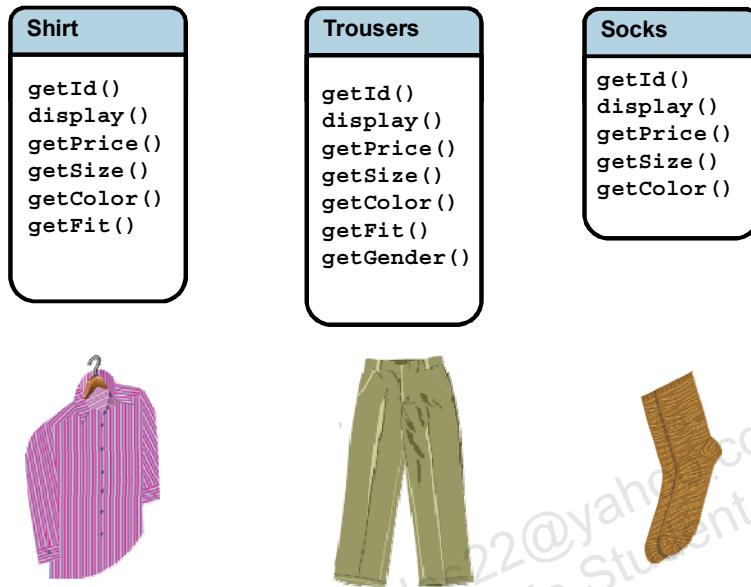
Duke's Choice Classes: Common Behaviors

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 © 2014, Oracle and/or its affiliates. All rights reserved.

Code Duplication

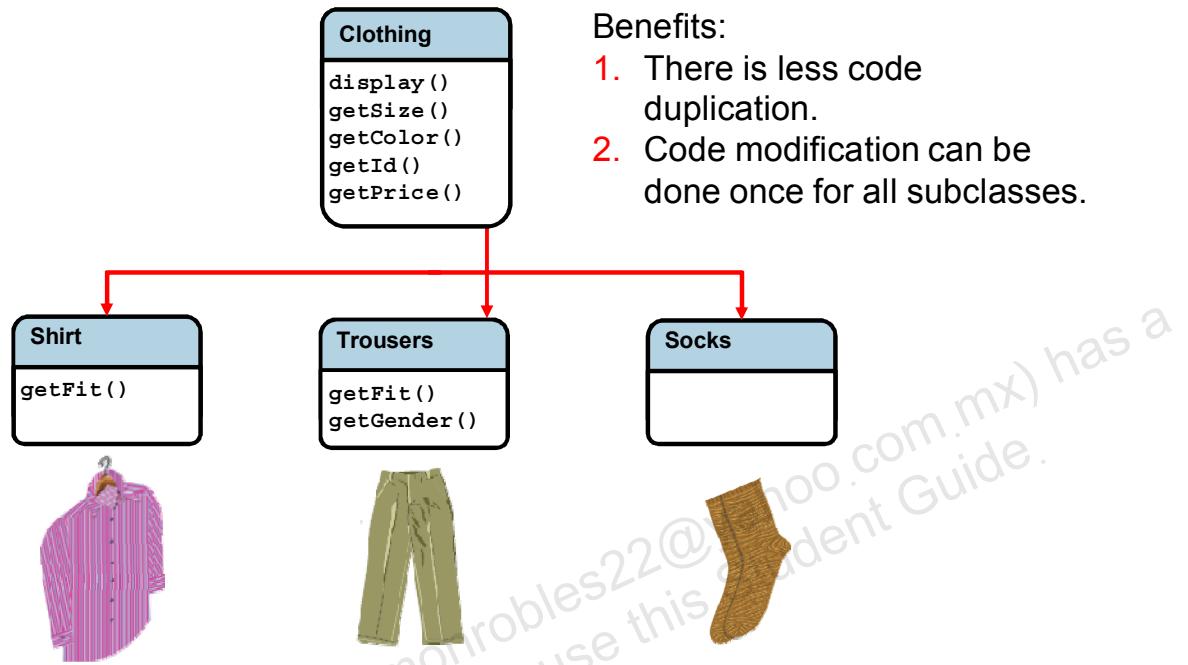


ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

If Duke's Choice decides to add a third item, socks, as well as trousers and shirts, you may find even greater code duplication. The diagram in the slide shows only the getter methods for accessing the properties of the new objects.

Inheritance



ORACLE

Copyright© 2014, 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.

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 © 2014, 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;                                Reference to the  
09     }                                              superclass constructor  
10    public char getFit() {                         Reference to  
11        return fit;                               this object  
12    }  
13    public void setFit(char fit) {  
14        this.fit = fit;  
15    }  
16}
```

The slide shows the code of the `Shirt` subclass. As you have seen in an earlier example, the `extends` keyword enables the `Shirt` class to inherit all the members of the `Clothing` class. The code declares attributes and methods that are unique to this class. Attributes and methods that are common with the `Clothing` class are inherited and do not need to be declared. It also includes two useful keywords and shows a common way of implementing constructors in a subclass.

- `super` refers to the superclass. In the example in the slide, the `super` keyword is used to invoke the constructor on the superclass. By using this technique, the constructor on the superclass can be invoked to set all the common attributes of the object being constructed. Then, as in the example here, additional attributes can be set in following statements.
- `this` refers to the current object instance. The only additional attribute that `Shirt` has is the `fit` attribute, and it is set after the invocation of the superclass constructor. The `this` keyword is necessary here as the constructor method parameter is also named `fit`). However, even when not strictly necessary, it is good programming practice because it makes the code more readable.

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;  
    } }
```

ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Within the constructor of a subclass, you must call the constructor of the superclass. If you call a superclass constructor, the call must be the first line of your constructor. This is done using the keyword `super`, followed by the arguments to be passed to the superclass constructor.

The constructor of the subclass sets variables that are unique to the subclass. The constructor of the superclass sets variables that originate from the superclass.

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;  
    }  
}
```

ORACLE

Copyright© 2014, 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

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

- Add two fields that are unique to the `Shirt` class.
- Invoke the superclass constructor from the `Shirt` constructor.
- Instantiate a `Shirt` object and call the `display` method.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 12-Inheritance > Exercise1.
- Follow the instructions below the code editor to create the `Shirt` class and then instantiate it and call the `display` method (in the `Item` class).
- Run the `ShoppingCart` class to test your code. Your output should look similar to the screenshot above.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

ORACLE

Copyright © 2014, 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).



Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier	Y	Y	N	N
private	Y	N	N	N



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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



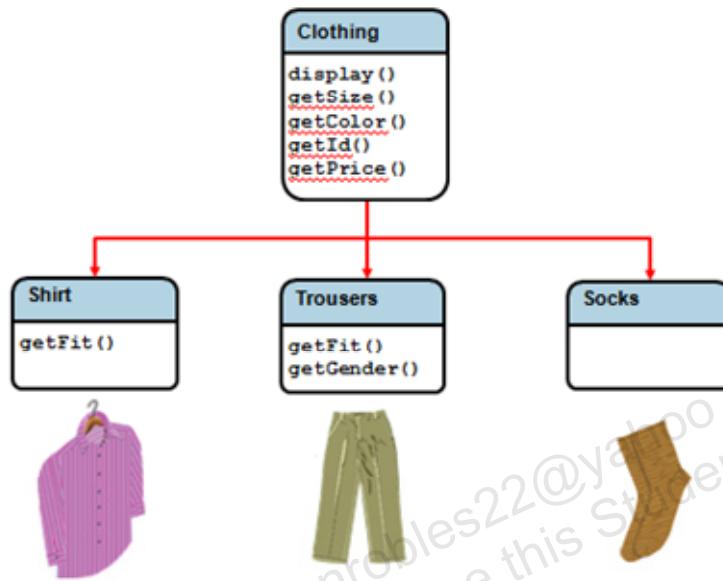
Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

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.



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

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 © 2014, 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.

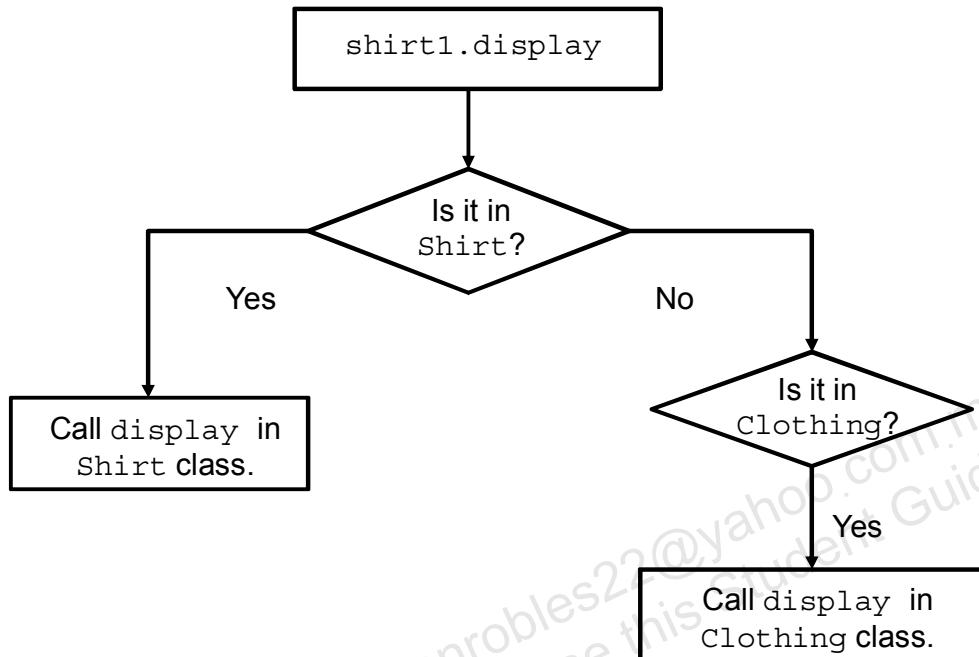


Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Notice that the `display` method overrides the `display` method of the superclass and is more specific to the `Shirt` class because it displays the shirt's fit property.

- The `Shirt` class does not have access to the private fields of the `Clothing` class such as `itemID`, `desc`, and `price`. The `Shirt` class's `display` method must, therefore, call the public getter methods for these fields. The getter methods originate from the `Clothing` superclass.
- The `setColorCode` method overrides the `setColorCode` method of the superclass to check whether a valid value is being used for this class. The method then calls the superclass version of the very same method.

Overriding a Method: What Happens at Run Time?



ORACLE

Copyright© 2014, 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

In this exercise, you override a method in the Item class.

- Override the `display` method to show the additional fields from the `Shirt` class.
- Run the `ShoppingCart` to see the result.

```
Output ✎  
Item description: Shirt  
ID: 1  
Price: 25.99  
Size: M  
Color Code: P
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 12-Inheritance > Exercise12.
- Follow the instructions below the code editor to override the `display` method.
- Run the `ShoppingCart` class to test your code. Your output should look similar to the screenshot above.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

ORACLE

Copyright © 2014, 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.



ORACLE

Copyright© 2014, 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.

ORACLE®

Copyright© 2014, 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.

ORACLE

Copyright© 2014, 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.

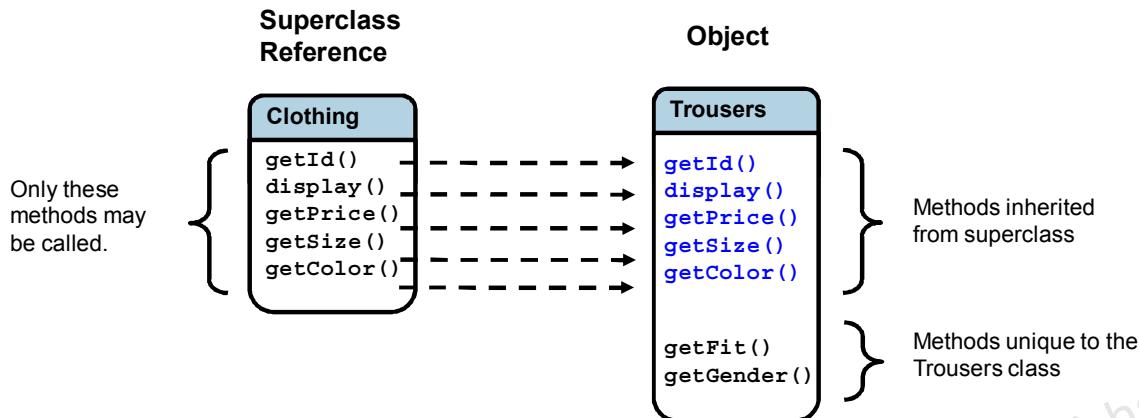
The method will be implemented differently on different types of objects. For example:

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

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Accessing Methods Using a Superclass Reference



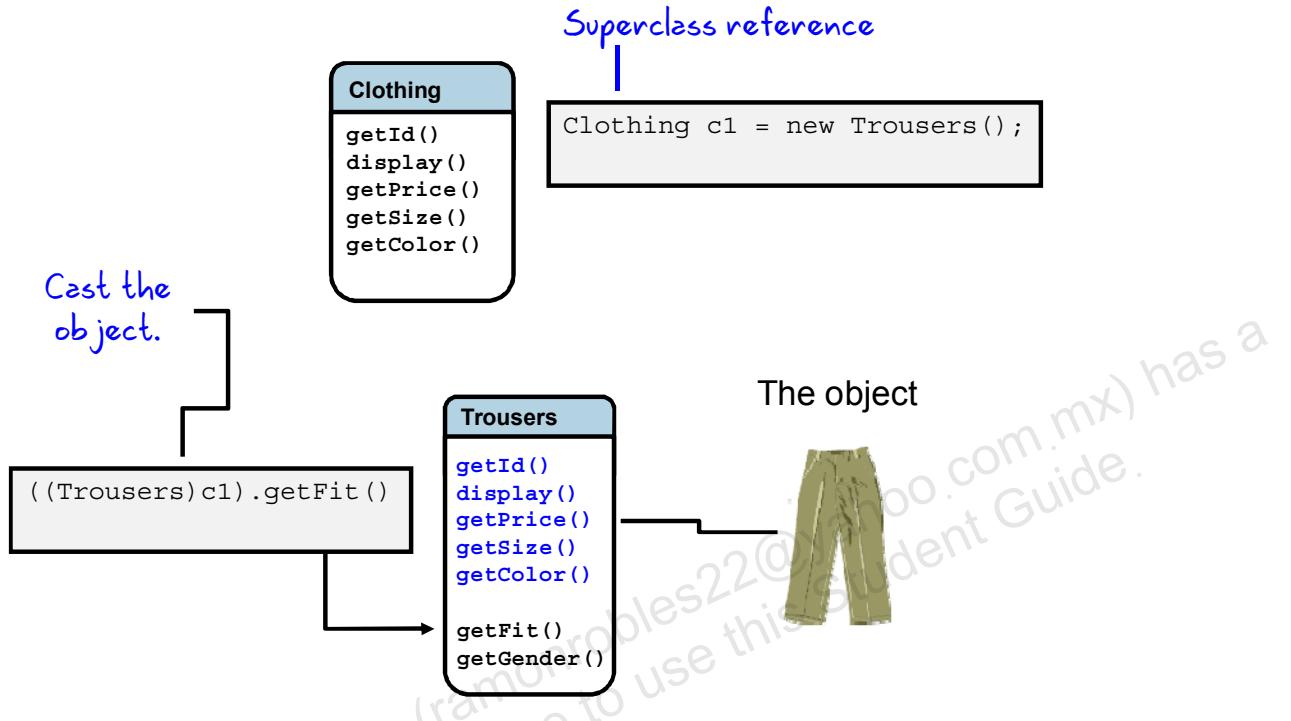
```
Clothing c1 = new Trousers();  
c1.getId(); OK  
c1.display(); OK  
c1.getFit(); NO!
```

ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Using a reference type `Clothing` does not allow access to the `getFit` or `getGender` method of the `Trousers` object. Usually this is not a problem, because you are most likely to be passing `Clothing` references to methods that do not require access to these methods. For example, a purchase method could receive a `Clothing` argument because it needs access only to the `getPrice` method.

Casting the Reference Type



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Given that a superclass may not have access to all the methods of the object it is referencing, how can you access those methods? The answer is that you can do so by replacing the superclass reference with:

- A reference that is the same type as the object. The code in this example shows a `Clothing` reference being cast to a `Trousers` reference to access the `getFit` method, which is not accessible via the `Clothing` reference. Note that the inner parentheses around `Trousers` are part of the cast syntax, and the outer parentheses around `(Trousers) c1` are there to apply the cast to the `Clothing` reference variable. Of course, a `Trousers` object would also have access to the nonprivate methods and fields in its superclass.
- An Interface that declares the methods in question and is implemented by the class of the object. Interfaces are covered in the next lesson.

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 }  
}
```

ORACLE

Copyright © 2014, 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

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

- Add a `getColor` method to the `Shirt` class.
- Instantiate a `Shirt` object using an `Item` reference type.
- Call the `display` method on the object.
- Cast the `Item` reference as a `Shirt` and call `getColor`.

```
Output ✎
Item description: Shirt
ID: 1
Price: 25.99
Size: M
Color Code: G
Color: Green
```



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 13-Interfaces > Exercise1.
- Follow the instructions below the code editor to modify the `Shirt` class with a new method, and the `ShoppingCart` to instantiate a `Shirt` object with an `Item` reference. Then call the new method by casting the reference as a `Shirt`. You will need to test the object type with the `instanceof` operator first.
- Run the `ShoppingCart` class to test your code. Test the use case where `Item` is not a `Shirt`, too. Your output should look similar to the screenshot above.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

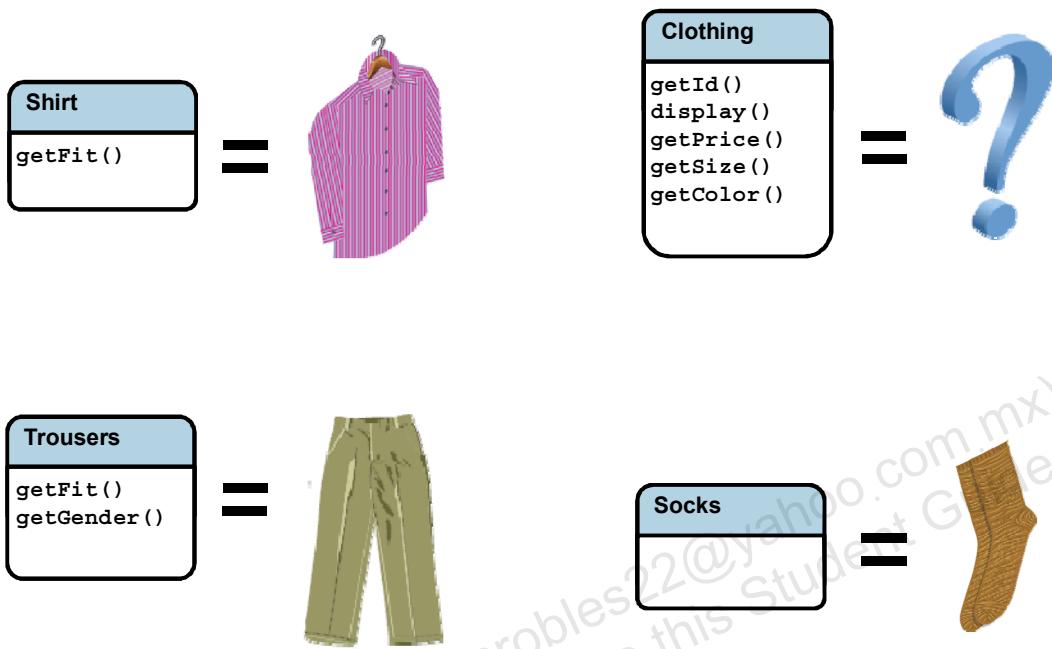
Topics

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

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Abstract Classes



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Sometimes a superclass makes sense as an object, and sometimes it does not. Duke's Choice carries shirts, socks, and trousers, but it does not have an individual item called "clothing." Also, in the application, the superclass `Clothing` may declare some methods that may be required in each subclass (and thus can be in the superclass), but cannot really be implemented in the superclass.

Abstract Classes

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

- Cannot be instantiated  `Clothing cloth01 = new Clothing()`
- May contain concrete methods
- May contain abstract methods that **must** be implemented later by any nonabstract subclasses

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

    public int getId() {
        return id;
    }

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

Concrete method

Abstract methods

ORACLE

Copyright © 2014, 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© 2014, 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:

- Creating class hierarchies with subclasses and superclasses helps to create extensible and maintainable code by:
 - Generalizing and abstracting code that may otherwise be duplicated
 - Allowing you to override the methods in the superclass
 - Allowing you to use less-specific reference types
- An abstract class cannot be instantiated, but it can be used to impose a particular interface on its descendants.



ORACLE

Copyright © 2014, 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.

Challenge Questions: Java Puzzle Ball



Bumpers share many of the same properties and behaviors. Is there a way to design code for these objects that minimizes duplicate code and take advantage of polymorphism?

- Common properties: color, shape, x-position, y-position...
- Common behaviors: `flash()`, `chime()`, `destroy()` ...



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

When you have an opportunity to play the game, run any of the inheritance puzzles and answer the questions above.

For some possible answers to these questions and more discussion, see “Appendix A: Java Puzzle Ball Challenge Questions Answered.”

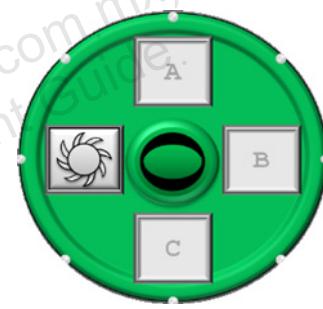
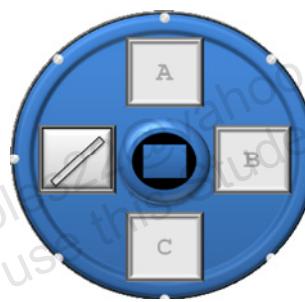
Challenge Questions: Java Puzzle Ball



For a method to be overridden, a subclass must provide a method with the same name and signature as the superclass's method. Only the implementation may differ.

To make overriding possible, which game components best represent:

- A method name and signature?
- A method implantation?



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

When you have an opportunity to play the game, run any of the inheritance puzzles and answer the questions above.

For some possible answers to these questions and more discussion, see “Appendix A: Java Puzzle Ball Challenge Questions Answered.”

Practice 12-1 Overview: Creating a Class Hierarchy

This practice covers rewriting the playGame method so that it will eventually be able to work with GameEvent objects and not just Goal objects, which is the case at present.

Practice 12-2 Overview: Creating a GameEvent Hierarchy

This practice covers adding game events to the application. To do this, you create a hierarchy of game event classes that extend an abstract superclass. Some of the game events are:

- Goal
- Pass
- Kickoff



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A game event is something that happens and that can be reported as game statistics by the soccer league application. You create an abstract superclass from which all of the game event classes are inherited.

13

Using Interfaces

ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Interactive Quizzes



ORACLE®

Copyright© 2014, 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 your quiz file from labs > Quizzes > Java SE 8 Fundamentals Quiz.html. Click the links for lessons titled "Working with Arrays, Loops, and Dates" and "Using Inheritance."

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
- Write a simple lambda expression that consumes a Predicate



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Topics

- Polymorphism in the JDK foundation classes
- Using interfaces
- Using the `List` interface
- Introducing lambda expressions

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

The Object Class

The screenshot shows the JavaDoc for the `ArrayList<E>` class. It highlights the inheritance path from `java.lang.Object` through `AbstractCollection<E>`, `AbstractList<E>`, and finally to `ArrayList<E>`. A callout box points to the `Object` link with the text "The Object class is the base class." Below this, it lists all implemented interfaces, which include `Serializable`, `Cloneable`, and `Iterable<E>`. The `Object` class itself is shown with its single method, `toString()`. A note states that `Object` is the root of the class hierarchy and that all objects implement its methods. The `Object` class is marked as being since JDK 1.0.

ORACLE

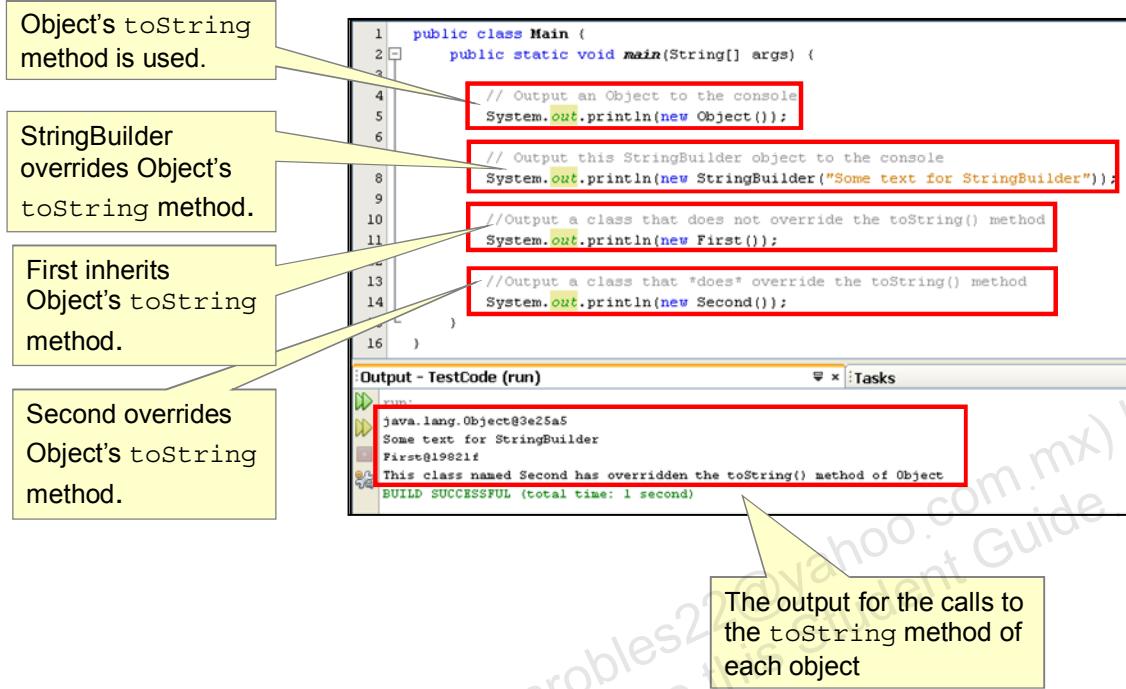
Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

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



Copyright© 2014, 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© 2014, 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 the List interface
- Introducing lambda expressions

The Multiple Inheritance Dilemma

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

- Class Red:

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

- 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?

ORACLE

Copyright© 2014, 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 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.



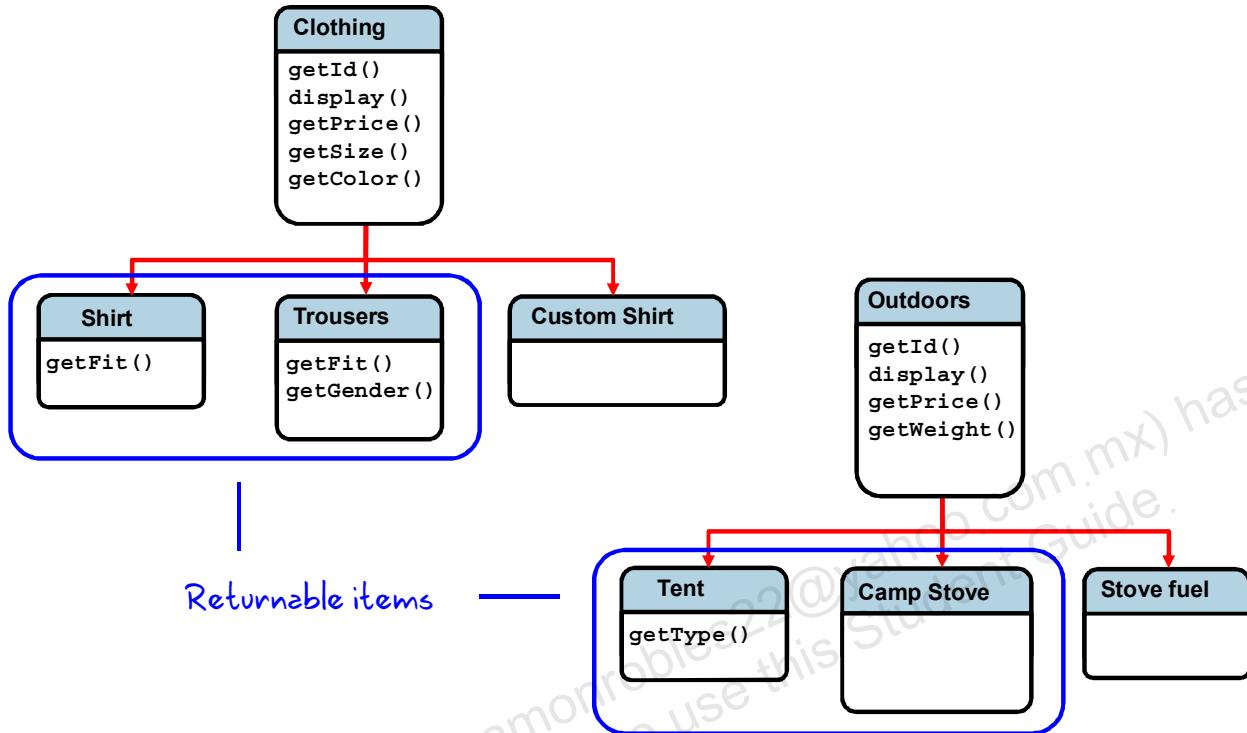
Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

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` keyword. Default methods in an interface are new with SE 8. They are used with lambda expressions. You will learn about lambda expressions a little later in this lesson; however, default methods and lambda expressions are covered in even more depth in the *Java SE8 New Features* course.

Multiple Hierarchies with Overlapping Requirements



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

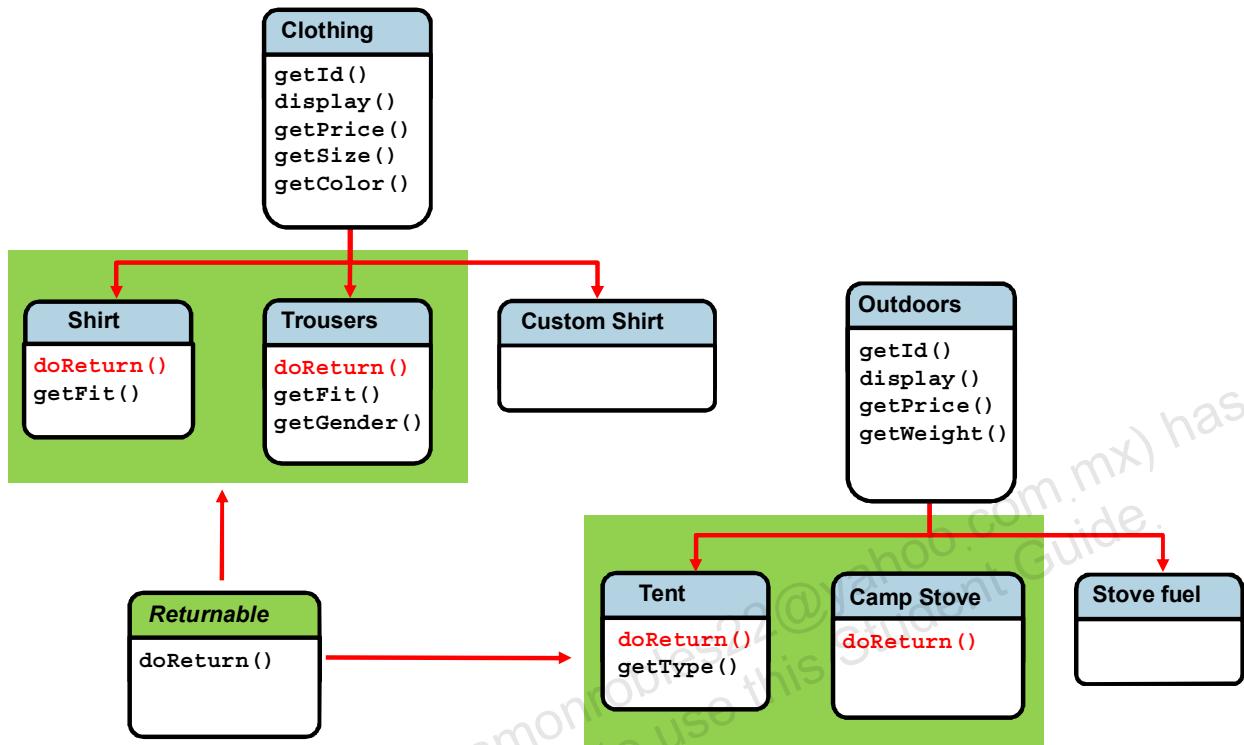
A more complex set of classes may have items in two different hierarchies. If Duke's Choice starts selling outdoor gear, it may have a completely different superclass called `Outdoors`, with its own set of subclasses (for example, `getWeight` as an `Outdoors` method).

In this scenario, there may be some classes from each hierarchy that have something in common. For example, the custom shirt item in `Clothing` is not returnable (because it is made manually for a particular person), and neither is the `Stove fuel` item in the `Outdoors` hierarchy. All other items are returnable.

How can this be modeled? Here are some things to consider:

- A new superclass will not work because a class can extend only one superclass, and all items are currently extending either `Outdoors` or `Clothing`.
- A new field named `returnable`, added to every class, could be used to determine whether an item can be returned. This is certainly possible, but then there is no single reference type to pass to a method that initiates or processes a return.
- You can use a special type called an *interface* that can be implemented by any class. This interface type can then be used to pass a reference of any class that implements it.

Using Interfaces in Your Application



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

The diagram in the slide shows all returnable items implementing the `Returnable` interface with its single method, `doReturn`. Methods can be declared in an interface, but they cannot be implemented in an interface. Therefore, each class that implements `Returnable` must implement `doReturn` for itself. All returnable items could be passed to a `processReturns` method of a `Returns` class and then have their `doReturn` method called.

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
08         // See notes below
09         return "Suit returns must be within 3 days";
10     }
11     ...< other methods not shown >... } // end of class
```

ORACLE

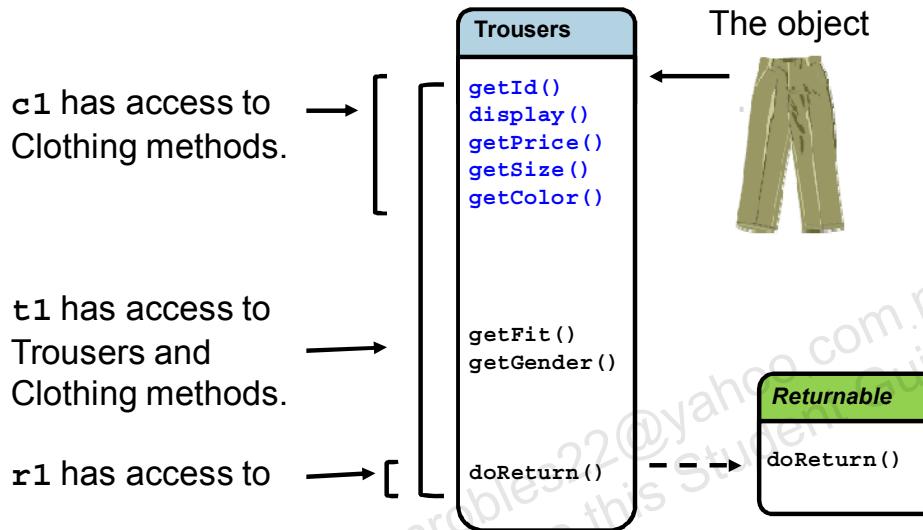
Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

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 Trouzers();  
Trousers t1 = new Trouzers();  
Returnable r1 = new Trouzers();
```



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

The reference used to access an object determines the methods that can be called on it. So in the case of the interface reference shown in the slide (`r1`), only the `doReturn` method can be called.

The `t1` reference has access to all of the methods shown above. This is because of the “is a” relationship. The `Trousers` class extends `Clothing`; therefore, a `Trousers` object is a (type of) `Clothing`. It implements `Returnable` and, therefore, it is a `Returnable`. `Clothing` is the root class and, consequently, the least specific. A reference of this type can only access the methods of the `Clothing` class (and, of course `Object`, which is the root of all classes).

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© 2014, 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 © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: c

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 © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b, c

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using the List interface
- Introducing lambda expressions

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© 2014, 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 depth in the *Java SE 8 Programming* course.

ArrayList Example

compact1, compact2, compact3
java.util

Class ArrayList<E>

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

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

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

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

ORACLE

Copyright© 2014, 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

compact1, compact2, compact3
java.util

Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

Many classes implement the List interface.

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

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

ORACLE

Copyright© 2014, 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()
mylist.removeIf()

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



Copyright© 2014, 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);
```

Shortcut:

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



Copyright© 2014, 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 13-1: Converting an Array to an ArrayList

In this exercise, you:

- Convert a String array to an ArrayList
- Work with the ArrayList reference to manipulate list values



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 13-Interfaces > Exercise2.
- Follow the instructions below the code editor to convert the given array to an ArrayList and then manipulate the values in the list.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using the List interface
- Introducing lambda expressions



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(String s: mylist) {
6     System.out.print(s.toUpperCase() + ", ");
7 }
8 System.out.println("After for loop: " + mylist);

```

Output:

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

Returns a new
String to print

The list
elements are
unchanged.



Copyright© 2014, 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]
```

ORACLE

Copyright© 2014, 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());
```

ORACLE

Copyright© 2014, 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` – *New*
 - Provides target types for lambda expressions
- `java.util.stream` – *New*
 - Provides classes that support operations on streams of values
- `java.util` – *Enhanced*
 - Interfaces and classes that make up the collections framework
 - Enhanced to use lambda expressions
 - Includes List and ArrayList



Copyright© 2014, 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 Programming* 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.



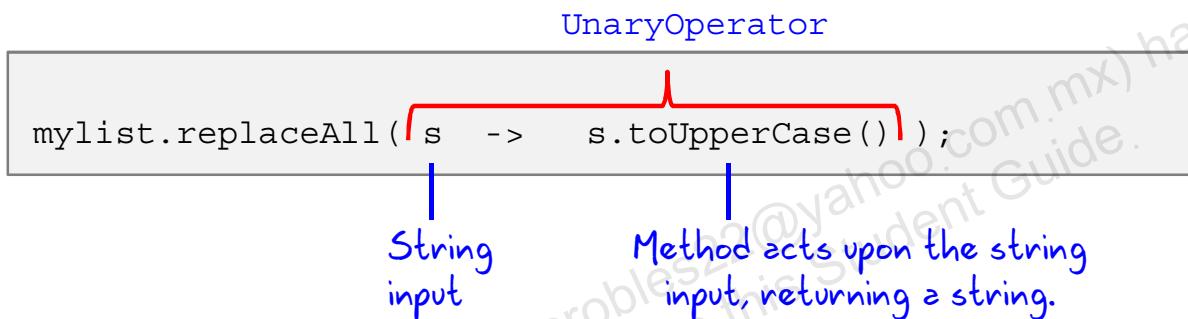
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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:



ORACLE

Copyright© 2014, 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);
```

ORACLE

Copyright© 2014, 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 13-2: Using a Predicate Lambda Expression

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

- Code the `removeItemFromCart()` method of `ShoppingCart`.
- Create a `Predicate` lambda expression that takes an `Item` object as input to the expression.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 13-Interfaces > Exercise2.
- Follow the instructions below the code editor to code the `removeItemFromCart` method.
- Run the `ShoppingCart` class to test your code.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Summary

In this lesson, you should have learned the following:

- Polymorphism provides the following benefits:
 - Different classes have the same methods.
 - Method implementations can be unique for each class.
- Interfaces provide the following benefits:
 - You can link classes in different object hierarchies by their common behavior.
 - An object that implements an interface can be assigned to a reference of the interface type.
- Lambda expressions allow you to pass a method call as the argument to another method.



ORACLE

Copyright © 2014, 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 13-1 Overview: Overriding the `toString` Method

This practice covers overriding the `toString` method in `Goal` and `Possession`.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 13-2 Overview: Implementing an Interface

This practice covers implementing the Comparable interface so that you can order the elements in an array.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 13-3 (Optional) Overview: Using a Lambda Expression for Sorting

This practice covers using a lambda expression to sort the players.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.

Ramon Robles Garcia (ramonrobles22@yahoo.com.mx) has a
non-transferable license to use this Student Guide.

14

Handling Exceptions

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



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Topics

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

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 © 2014, 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 © 2014, 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 8 Programming* 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)
```

ORACLE

Copyright © 2014, 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 © 2014, 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 © 2014, 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 © 2014, 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



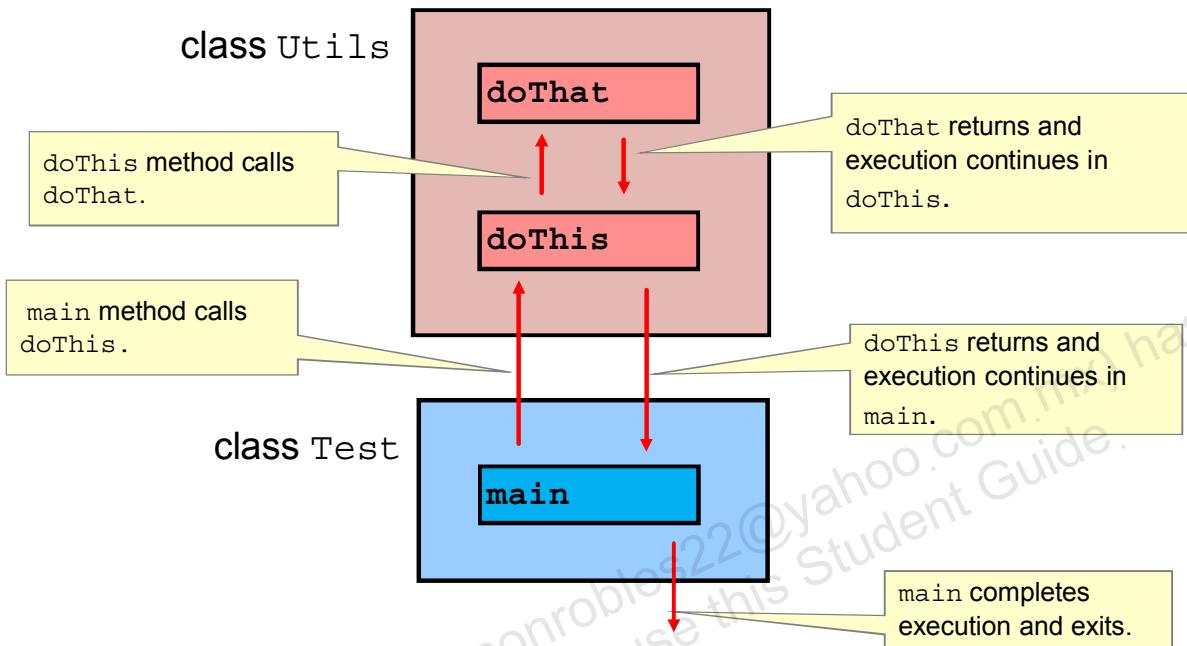
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: c

Topics

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

Normal Program Execution: The Call Stack



ORACLE

Copyright © 2014, 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 © 2014, 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

Working with Exceptions in NetBeans

The image contains two side-by-side screenshots of a Java code editor in NetBeans. The code defines a class named `Utils` with two methods: `doThis()` and `doThat()`. Both methods print messages to `System.out`.

Screenshot 1 (Left): Shows the code without any exception handling. A blue arrow points from the text "No exceptions thrown; nothing needs be done to deal with them." to the code area.

```
10 public class Utils {  
11     public void doThis() {  
12         System.out.println("Arrived in doThis()");  
13         doThat();  
14         System.out.println("Back in doThis()");  
15     }  
16     public void doThat() {  
17         System.out.println("In doThat()");  
18     }  
19 }  
20  
21  
22  
23  
24  
25
```

Screenshot 2 (Right): Shows the same code, but the `doThat()` method now contains a `throw new Exception();` statement. A red box highlights this line. A tooltip appears over the line, reading: "unreported exception java.lang.Exception; must be caught or declared to be thrown. -- (Alt-Enter shows hints)". A blue arrow points from the text "When you throw an exception, NetBeans gives you two options." to the tooltip.

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

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

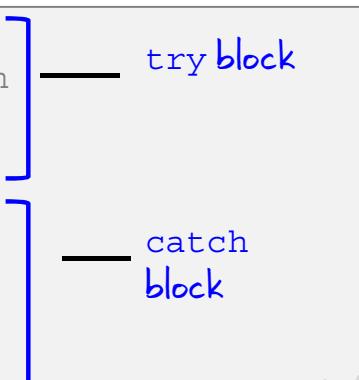
- In the first screenshot, no exceptions are thrown, so NetBeans shows no syntax or compilation errors.
- In the second screenshot, `doThat` explicitly throws an exception, and NetBeans flags this as something that needs to be dealt with by the programmer. As you can see from the tooltip, it gives the 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();  
}
```

ORACLE

Copyright © 2014, 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");
```

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 }
```

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

2

1

ORACLE

Copyright © 2014, 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:

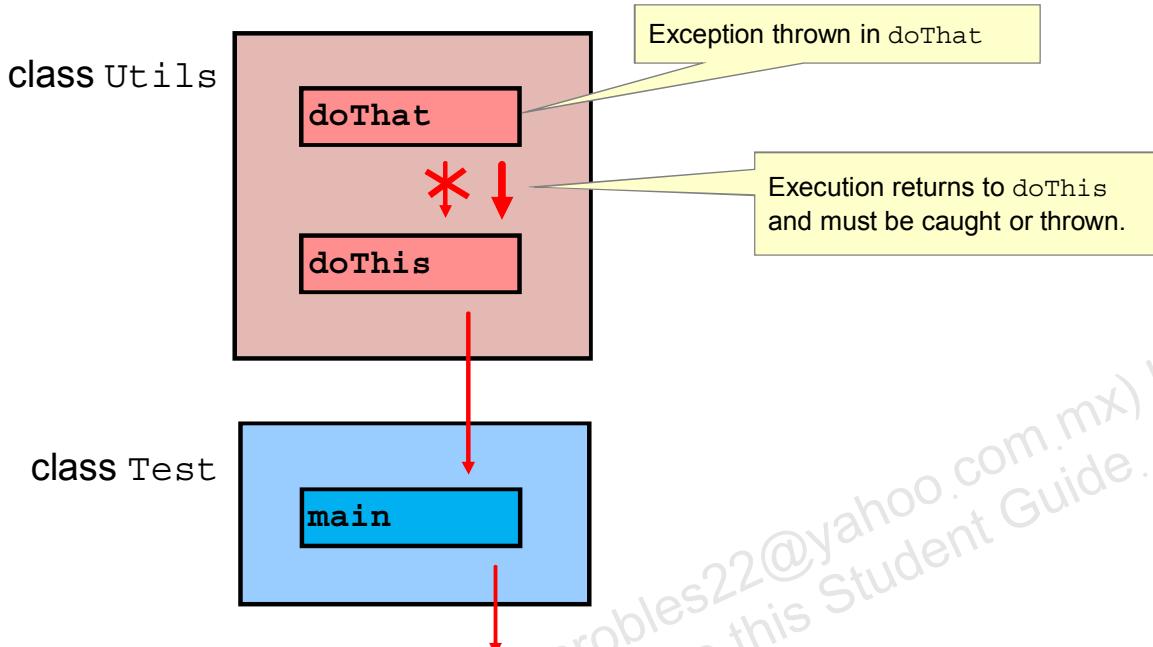
- In line 14, `doThat` explicitly throws a new `Exception` object.

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



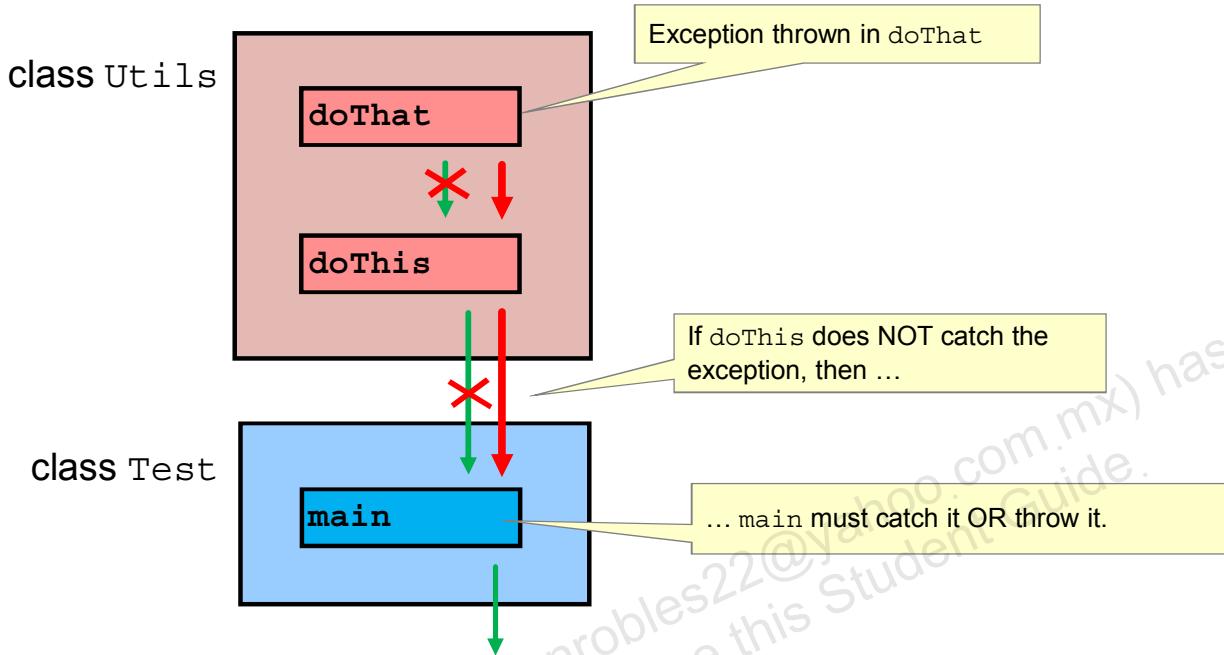
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As mentioned previously, when a method finishes executing, the *normal* flow (on completion of the method or on a return statement) goes back to the calling method and continues execution at the next line of the calling method.

When an exception is thrown, program flow returns to the calling method, but *not* to the point just after the method call. Instead, if there is a `try/catch` block, program flow goes to the `catch` block associated with the `try` block that contains the method call. You will see in the next slide what happens if there is no `try/catch` block in `doThis`.

Throwing Throwable Objects

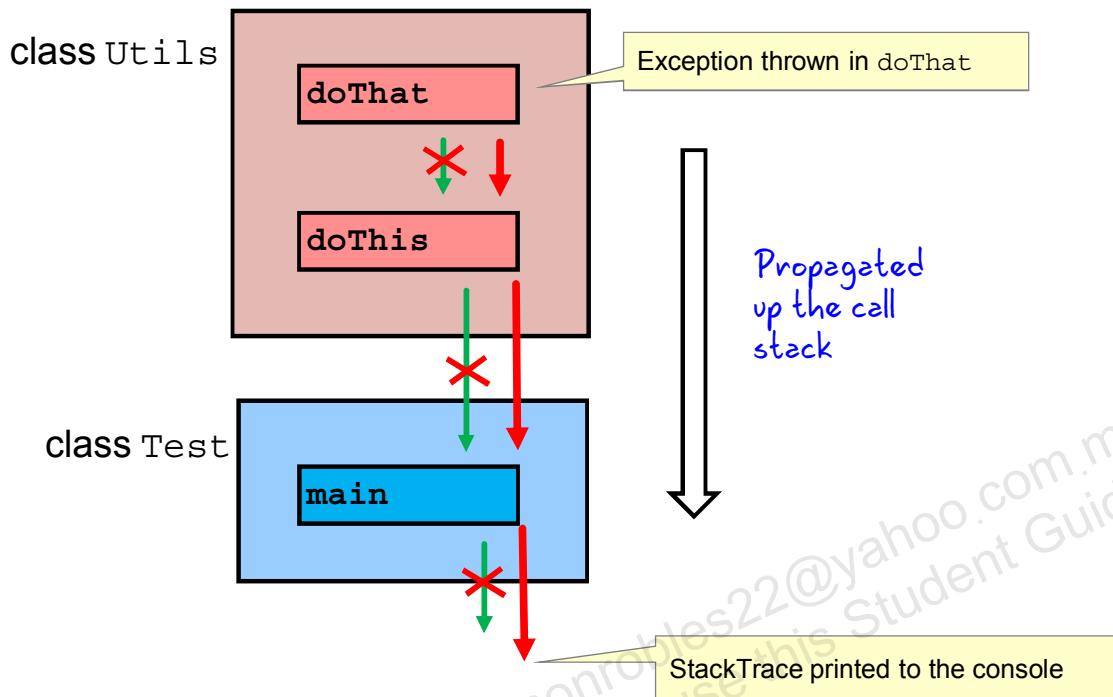


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The diagram in the slide illustrates an exception originally thrown in `doThat` being thrown to `doThis`. The error is not caught there, so it is thrown to its caller method, which is the `main` method. The thing to remember is that the exception will continue to be thrown back up the call stack until it is caught.

Uncaught Exception



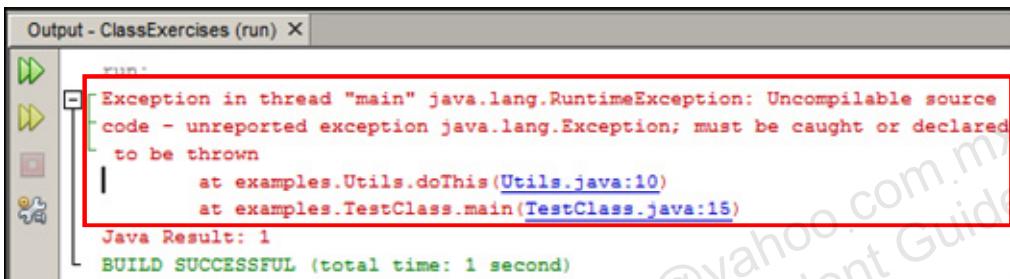
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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 'Output' window from the Oracle Java Development Kit (JDK). The window title is 'Output - ClassExercises (run)'. The output text is:
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)

ORACLE

Copyright © 2014, 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 © 2014, 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

In this exercise, you work with the `ShoppingCart` class and a `Calculator` class to implement exception handling.

- Change a method signature to indicate that it throws an exception.
- Catch the exception in the class that calls the method.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- In the Java Code Console, access Lessons > 13-Exceptions > Exercise1.
- Follow the instructions below the code editor to add exception handling to catch an `ArithmaticException` when division by zero is attempted.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

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.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: d

Exceptions in the Java API Documentation

These are methods of the **File Class**.

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

Click to get the detail of **createNewFile**.

Note the exceptions that can be thrown.

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

ORACLE

Copyright © 2014, 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

```
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 }
```

Constructor causes no compilation problems.

```
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 }
```

createNewFile can throw a checked exception, so the method must throw or catch.

ORACLE

Copyright © 2014, 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 8 Programming*), 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 © 2014, Oracle and/or its affiliates. All rights reserved.

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

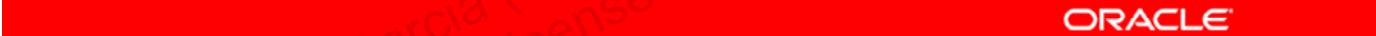
- Throwing the exception from the `testCheckedException` method
- Catching the exception in the caller method

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?”



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Bad Practices

```
01 public static void main(String[] args){  
02     try {  
03         createFile("c:/testFile.txt");  
04     } catch (Exception e) {           —————— Catching superclass?  
05         System.out.println("Error creating file.");  
06     }  
07 }  
08 public static void createFile(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 }
```

No processing of exception class?



Copyright © 2014, 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         createFile("c:/testFile.txt");
04     } catch (Exception e) {           What is the
05         System.out.println(e);       object type?
06         //<other actions>
07     }
08 }
09 public static void createFile(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 }
```

tostring() is called
on this object.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Putting in a few `System.out.println` calls in the `createFile` 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 `createFile`.

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

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());  
08     int myInt[] = new int[5];  
09     myInt[5] = 25;  
11 }
```

Directory must be writeable:
IOException

Arg must be greater than
3 characters:
IllegalArgumentException

Array index must be valid:
ArrayIndexOutOfBoundsException

ORACLE

Copyright © 2014, 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` File 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 © 2014, 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 © 2014, 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 © 2014, 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



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Interactive Quizzes



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Before you take a break to perform the practices, test your knowledge by answering some quiz questions. Open your quiz file from labs > Quizzes > Java SE 8 Fundamentals Quiz.html. Click the links for this lesson as well as the lesson titled “Using Interfaces.”

Practice 14-1 Overview: Adding Exception Handling

This practice covers the following topics:

- Investigating how the Soccer application can break under certain circumstances
- Modifying your code to handle the exceptions gracefully



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

15

Deploying and Maintaining the Soccer Application

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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



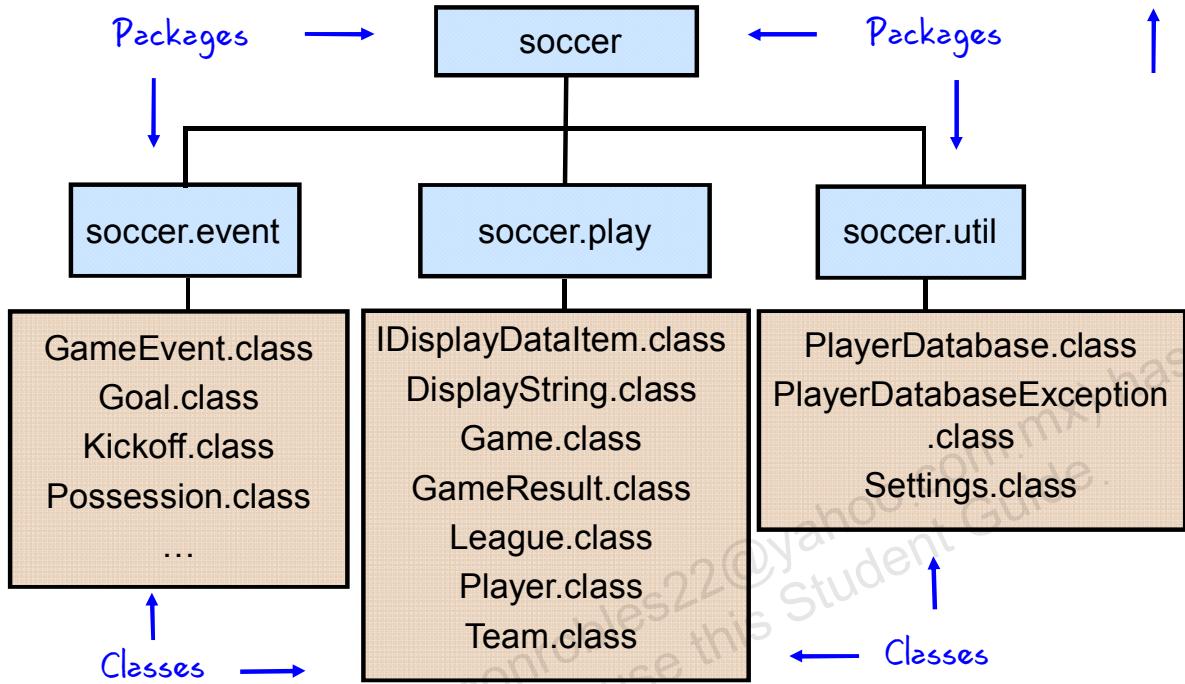
ORACLE®

Copyright © 2014, 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

Packages



ORACLE

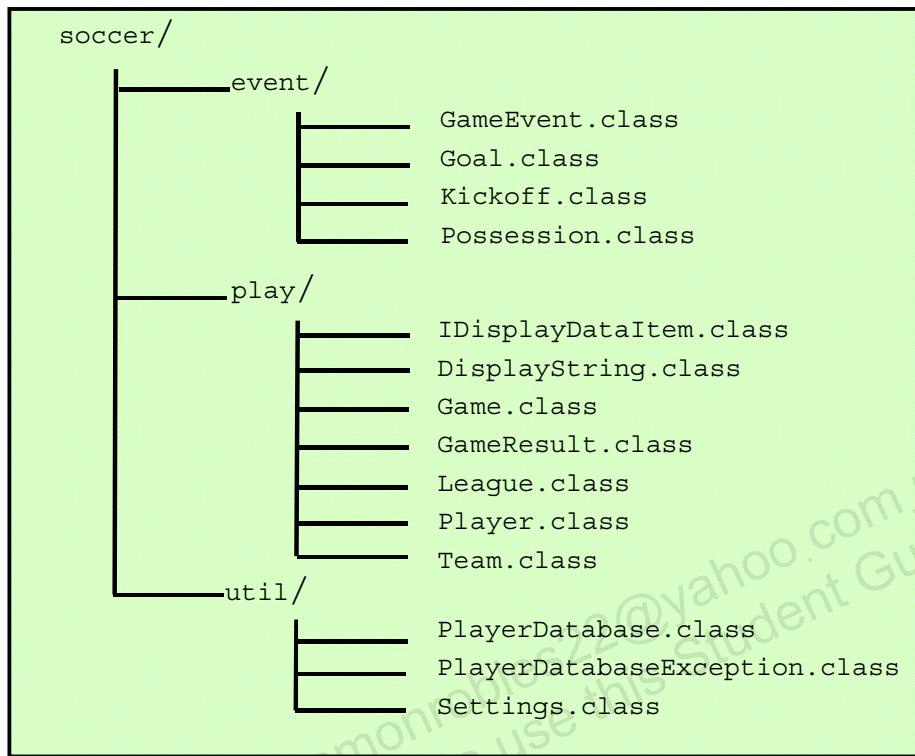
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

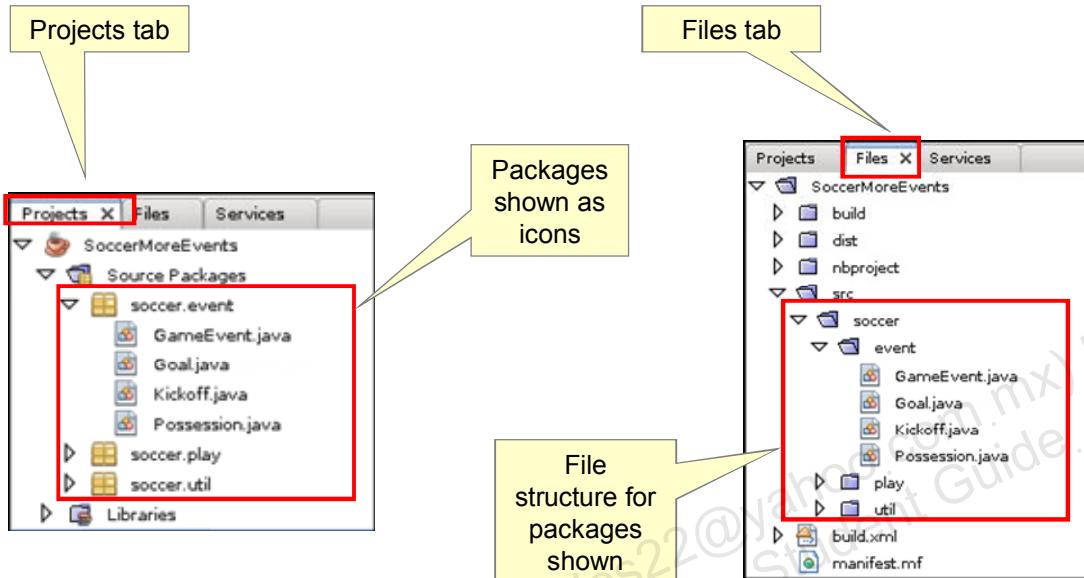


ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Packages are stored in a directory tree containing directories that match the package names. For example, the `Goal.class` file should exist in the directory `event`, which is contained in the directory `soccer`.

Packages in NetBeans



ORACLE

Copyright © 2014, 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

This class is in the package soccer.event.

```
package soccer.event;

public class Goal extends GameEvent {

    public String toString(){
        return "GOAL! ";
    }
    ... < remaining code omitted > ...
}
```

The package that a class belongs to is defined in the source code.



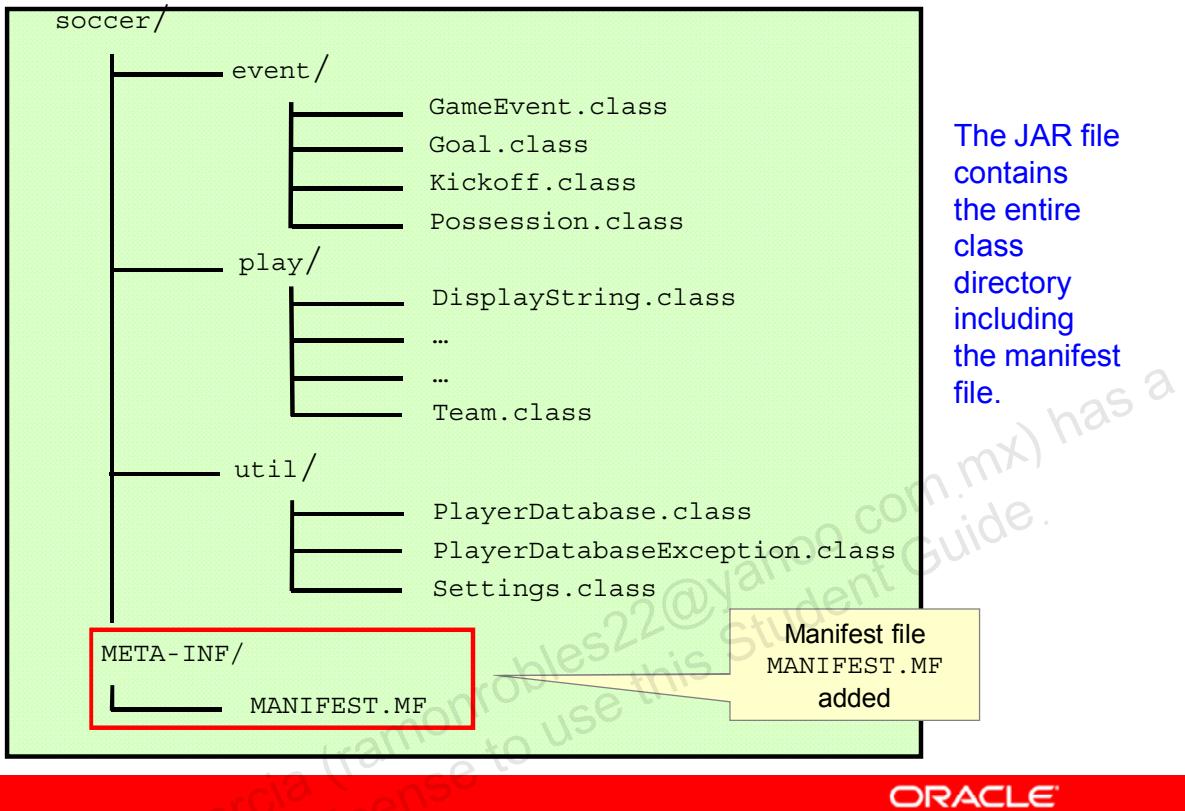
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

SoccerEnhanced.jar



ORACLE

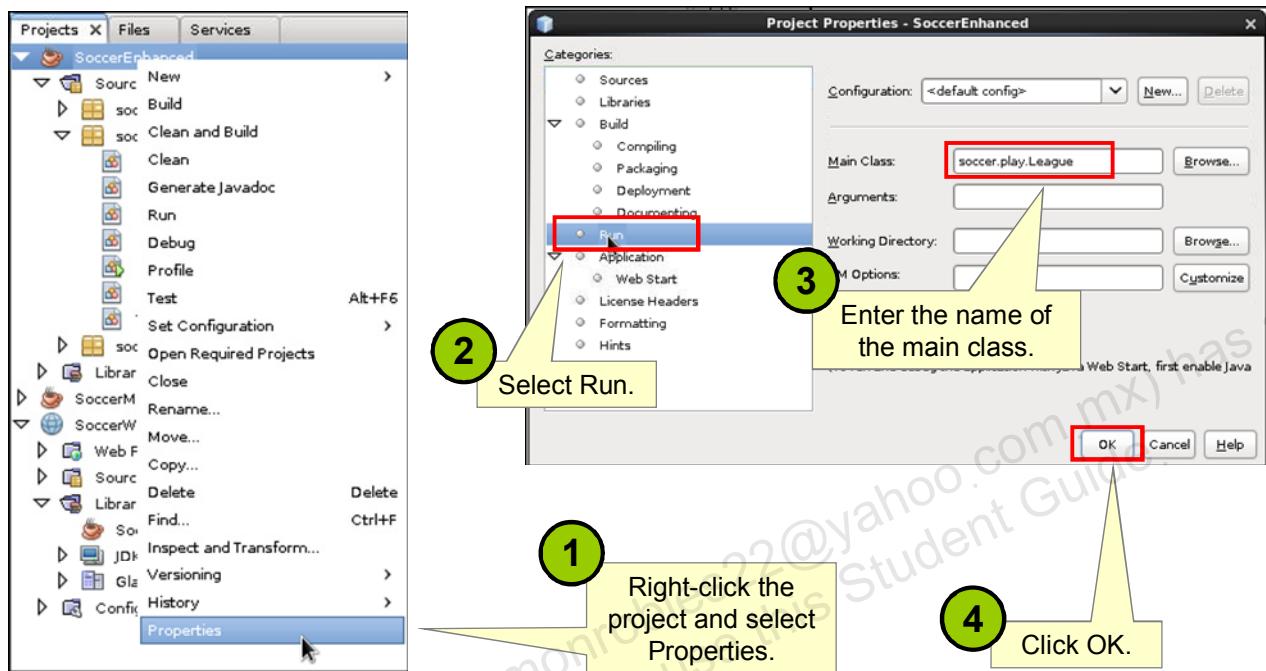
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

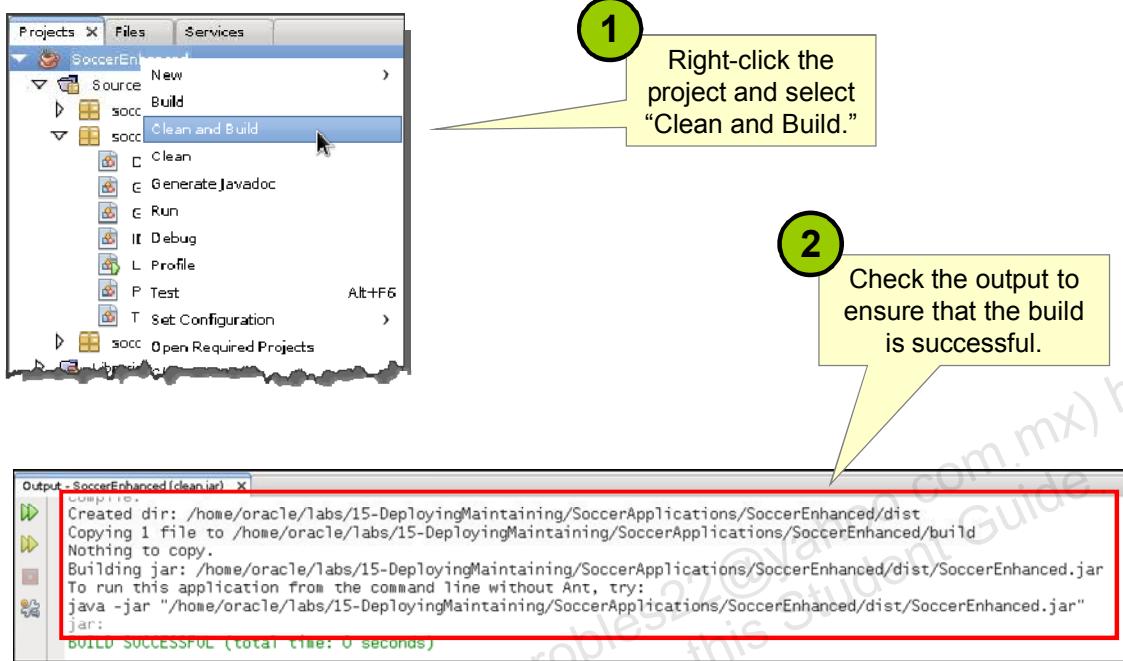


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

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



ORACLE®

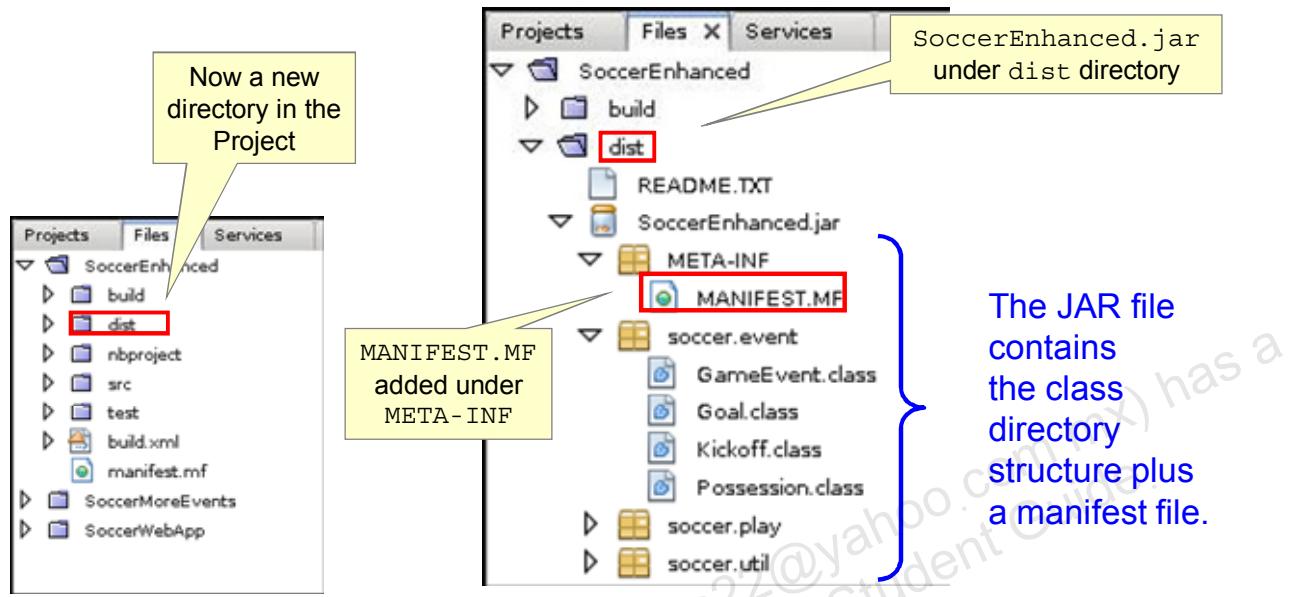
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

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.



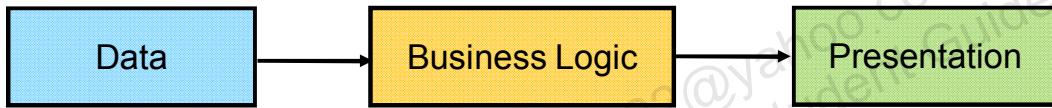
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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.



ORACLE

Copyright © 2014, 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.

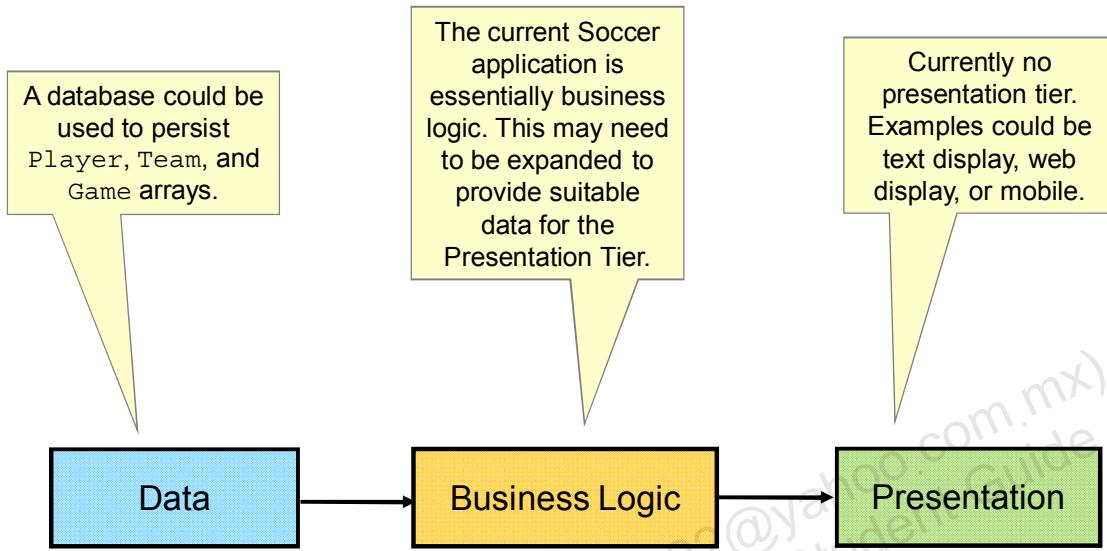
The business logic offers functionality in terms of defining all of the business rules through which the data can be manipulated. Changes to business policies can affect this layer without having an impact on the actual databases.

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

Client/Server Three-Tier Architecture

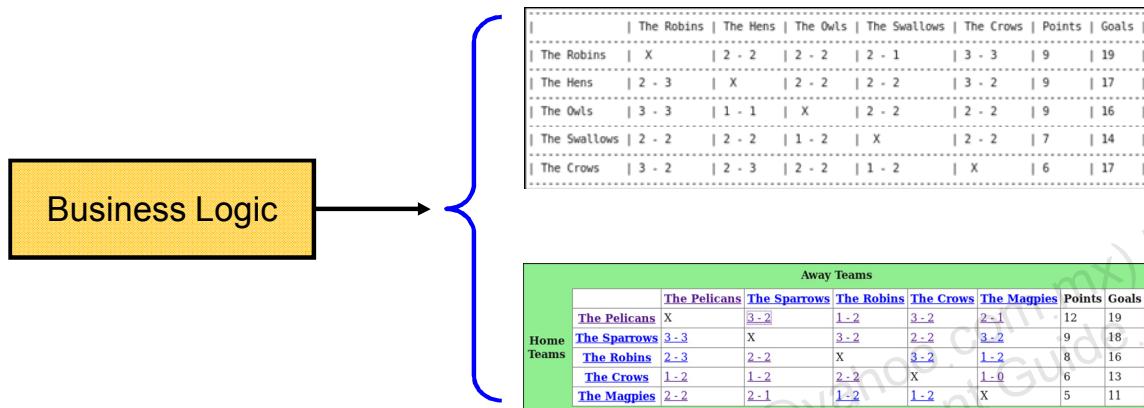


ORACLE®

Copyright © 2014, 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



ORACLE

Copyright © 2014, 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

Home Teams	Away Teams					Points	Goals
	The Pelicans	The Sparrows	The Robins	The Crows	The Magpies		
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



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

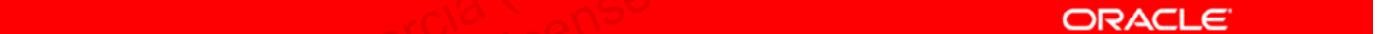
Ideally for each element of data in the two-dimensional array, the presentation should have access to the data the display is based on, or at least a useful subset of that data. For example, for the text that displays a team name, the presentation might wish to query further data about the team—for example 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



ORACLE

Copyright © 2014, 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 {

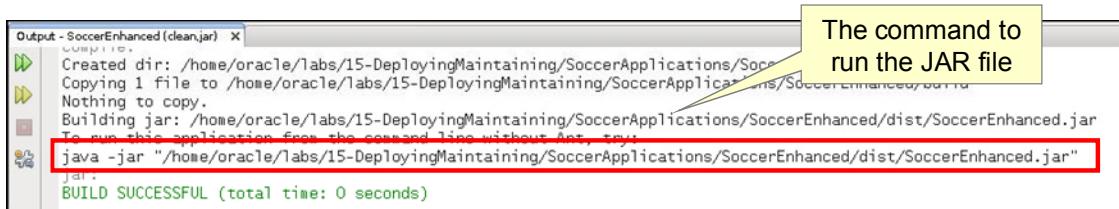
    public boolean isDetailAvailable ();
    public String getDisplayDetail();
    public int getID();
    public String getDetailType();

}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Running the JAR File from the Command Line



```
Output - SoccerEnhanced (clean.jar) X
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 |
```

ORACLE

Copyright © 2014, 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

```
[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

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.

ORACLE

Copyright© 2014, 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 © 2014, 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

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 © 2014, 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 © 2014, 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.

ORACLE®

Copyright © 2014, 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



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Summary

In this course, you should have learned how to:

- List and describe several key features of the Java technology: object-oriented, multithreaded, distributed, simple, and secure
- Identify different Java technology groups
- Describe examples of how Java is used in applications as well as in consumer products
- Describe the benefits of using an integrated development environment (IDE)
- Develop classes and describe how to declare a class
- Analyze a business problem to recognize objects and operations that form the building blocks of the Java program design



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Summary

- Define the term *object* and its relationship to a class
- Demonstrate Java programming syntax
- Write a simple Java program that compiles and runs successfully
- Declare and initialize variables
- List several primitive data types
- Instantiate an object and effectively use object reference variables
- Use operators, loops, and decision constructs
- Declare and instantiate arrays and ArrayLists and be able to iterate through them



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Summary

- Use Javadocs to look up Java foundation classes
- Declare a method with arguments and return values
- Use inheritance to declare and define a subclass of an existing superclass
- Describe how errors are handled in a Java program
- Describe how to deploy a simple Java application by using the NetBeans IDE



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.



16



Oracle Cloud

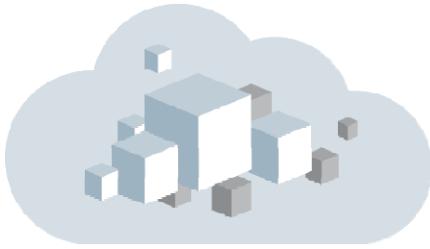
An Overview

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Ramon Robles Garcia (ramonroblesgarcia@yahoo.com.mx) has a
non-transferable license to use this document.

Agenda



- 1** What is Cloud Computing?
- 2** Cloud Evolution
- 3** Components of Cloud Computing
- 4** Characteristics and Benefits of Cloud
- 5** Cloud Deployment Models
- 6** Cloud Service Models
- 7** Oracle Cloud Services

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

What is Cloud?

The term Cloud refers to a Network or Internet.

It is a means to access any Software that is available remotely.



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

What is Cloud Computing?

- It is a means to access any Software that is available remotely.
- Refers to the practice of using remote Servers hosted on Internet to store, manage and process data
- When you store your photos online instead of on your home computer, or use webmail or a social networking site, you are using a “cloud computing” service.

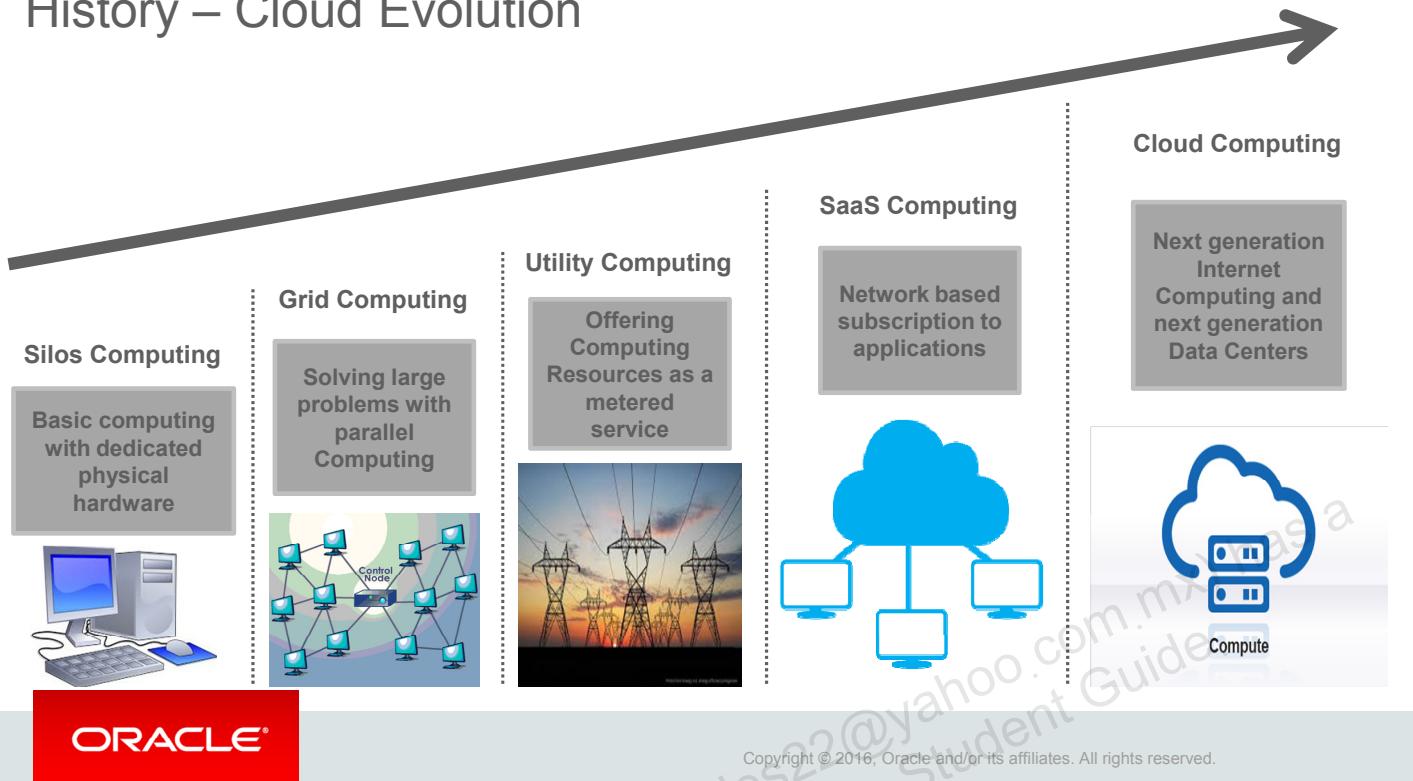


ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

History – Cloud Evolution

Unauthorized reproduction or distribution prohibited. Copyright © 2017, Oracle and/or its affiliates.



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Components of Cloud Computing

Client Computers



Devices that end user interact with cloud. Types of client Thick, Thin (Most popular), Mobile

Distributed Servers



Often Servers are in geographically different places, but server acts as if they are next to each other

Data Centers

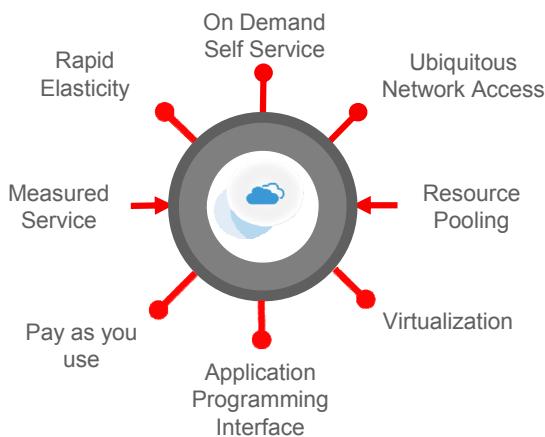


Collection of servers where application is placed and is accessed via Internet

ORACLE

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Characteristics of Cloud



Description

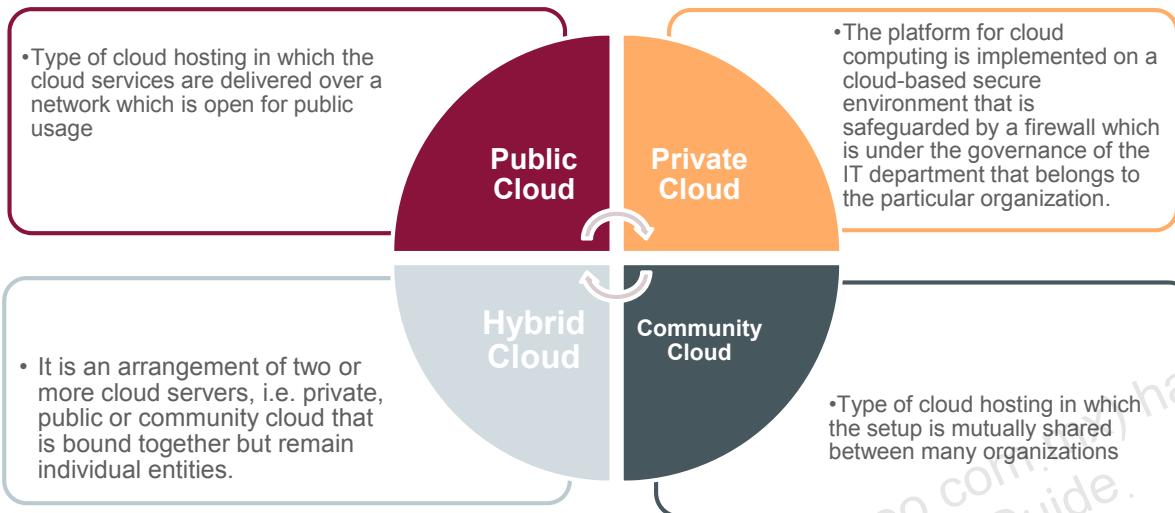
- Allows users to use the service on demand
- Anywhere, Anytime and Any Device
- Draw from a pool of computing resources, usually in remote data centers
- Request and manage own computing resources
- Service is measured and customers are billed accordingly
- Select a configuration of CPU, Memory and storage
- Services can be scaled larger or smaller

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Deployment Models

Deployment models define the type of access to the Cloud.



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models

All three tiers of computing delivered as Service via global network

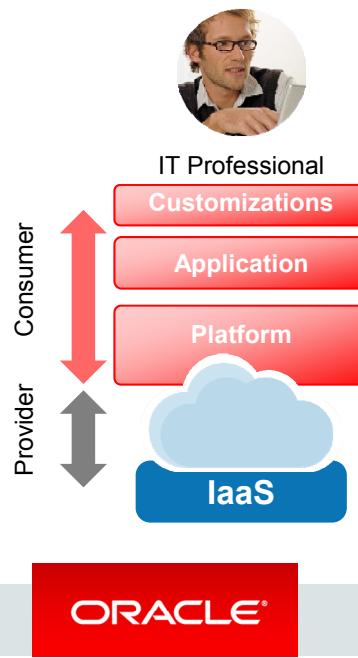
- **Applications:** Software as a Service - SaaS
- **Platform:** Database, Middleware, Analytics, Integration as a Service – Platform as a Service - PaaS
- **Infrastructure:** Storage, Compute, and Network as a service – Infrastructure as a Service - IaaS



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models

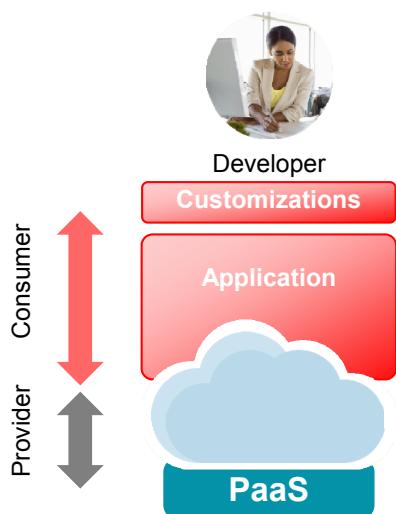


- Provides computer hardware (servers, networking technology, storage and data center space) as a web based service.
- Virtual Machines with pre-installed Operating System
- Target: Administrators
- Ready to Rent

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models



- Provides platform to develop and deploy applications
- Up to Date Software
- Target: Application Developers
- Ready to Use

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Cloud Service Models



Business End User

Customizations



- Allows usage of the software remotely as a web based service
- Software are automatically Upgraded and Updated
- All Users are running the same version of the Software
- Target: End Users
- Ready to Wear

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Industry Shifting from **On-Premises to the Cloud**

Transition to the Cloud is driven by a desire for:

- **Agility:** Self-service provisioning – deploy a database in minutes
- **Elasticity:** Scale on demand
- **Lower cost:** Reduction in management and total cost – pay for what is used
- **Back to core business:** Focus on core activities
- **More mobility:** Access from any device

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle IaaS Overview

IaaS

Designed for large enterprises, which allow them to scale up their computing, networking, and storage systems into the cloud, rather than expanding their physical infrastructure.

- Allows large businesses and organizations to run their workloads, replicate their network, and back up their data in the cloud.



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle PaaS Overview

PaaS

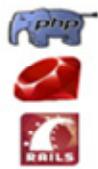
- Develop, deploy, integrate and manage applications on cloud.
- Seamless integration across PaaS and SaaS Applications.



Database Services



Java Services



Web Scripting Services



Mobile Services



Developer Services



Documents Services



Sites Services



Analytics Services

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle SaaS Overview

SaaS

Delivers modern cloud applications that connect business processes across the enterprise.

- Only Cloud integrating ERP, HCM, EPM, SCM
- Seamless co-existence with Oracle's On-Premise Applications



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have :

- Got an overview of Cloud Computing, its Characteristics, History and Technology
- Understood the various components , Deployment Models and Service Models of Cloud Computing
- Understood the Oracle Cloud Services



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.



17

Oracle Application Container Cloud Service Overview

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Get an overview of Oracle Application Container Cloud
- Understand the unique features of Oracle Application Container Cloud
- Understand how to build, zip, and deploy applications to the cloud



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle Application Container Cloud Service



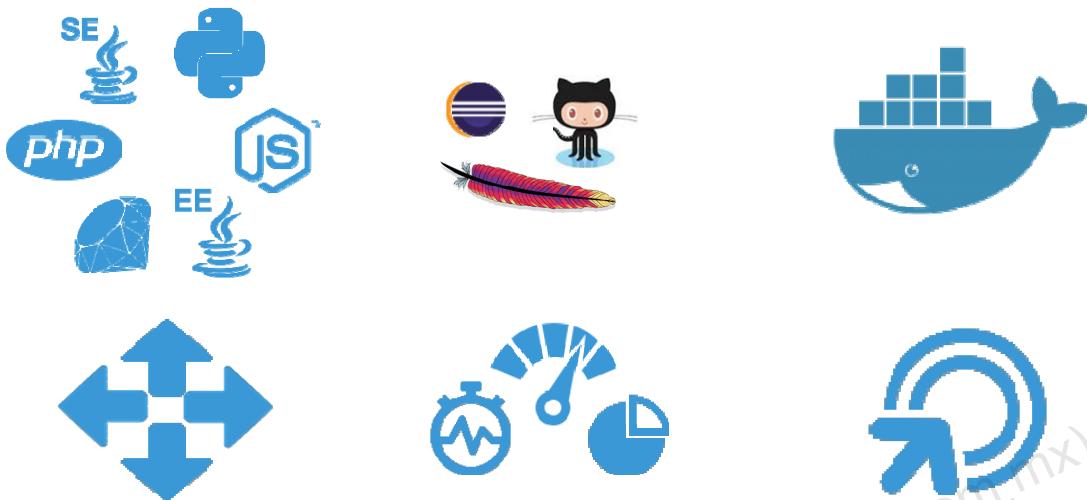
An open highly available
Docker container-based
elastic polyglot cloud
application platform

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle Application Container Cloud

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.

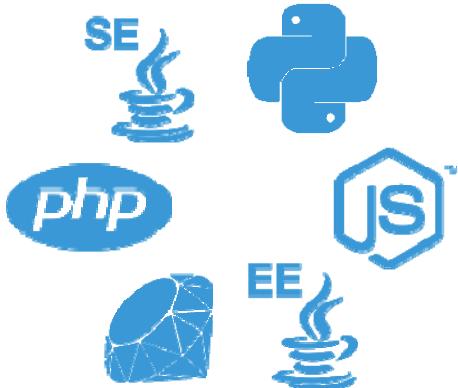


ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

- Simple and easy to use deployment platform for Java SE & Node applications
- Open platform—use any application frameworks and libraries
- Runs applications in Docker containers for reliability and scalability

Polyglot Platform



ORACLE®

Runtime releases regularly updated to the latest

Deploy applications to a selection of popular language runtimes supported

- Latest release supports Java SE, Java EE Web Apps, Node.js, and PHP

Leverage unique Oracle Java SE features

- Immediate access to platform upgrades, security, platform optimizations
- Continued commercial support for Java SE versions no longer receiving public updates

Node access to Oracle DB with open source database driver

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Open Platform

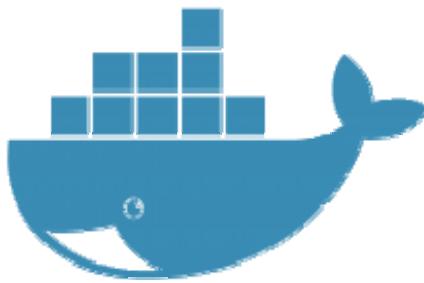
Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Use any of the thousands of open source or commercial Java or Node frameworks—no restrictions.

Container-based Application Platform as a Service



Applications run on Oracle Linux in Docker containers

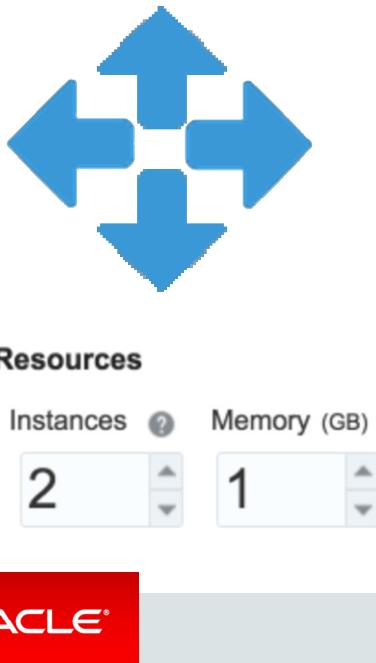
Stateless Applications

- Ephemeral disk
- Permanent storage through database or storage service

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Elastic Scaling



On demand elastic scaling either through the service console or using the service REST API

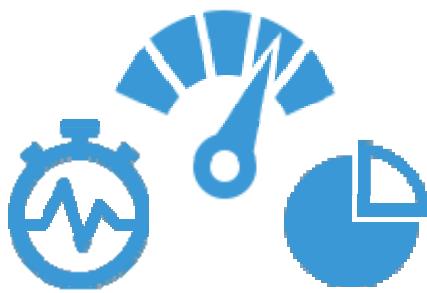
Scale out / in

- Add / remove application instances to handle workloads

Scale up / down

- Add / remove RAM to accommodate application memory requirements

Profiling



Java application can use Java Flight Recorder to monitor application and JVM behavior and analyze in Mission Control

Use Application Performance Monitoring Cloud Service for advanced use cases

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Manageable



New Java and Node releases published in the service console
One-click upgrade to the latest releases—applications are simply restarted to upgrade to new runtime

Updates 0

Current Version: Java SE 8u71

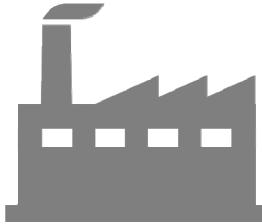
Available Updates

	Runtime: Java SE 8u91	Release Date: Jul 4, 2015 12:00:00 AM UTC	Update
<small>Description: This update contains new features as well as fix for critical issues. Refer to the 'Release Notes' for more details</small>			

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Build Zip Deploy!



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Build

- Use your favourite or corporate standard build system to produce binaries and deployable resources.

Zip

- Zip up all binaries, scripts, html files, images, etc. that make up your application. The structure of the zip is entirely up to the user—we have no opinion on structure.

Deploy

- Deploy the application archive (zip) to the platform and tell us how to start the application. This could be “java –jar”, “java –classpath ... <main>”, “node myapp.js”, or “sh bootmyapp.sh”.

Deploy—Application Archive (Zip)

- All application binaries
- All required libraries
- Binaries of any container/embedded container
- Images files
- HTML files

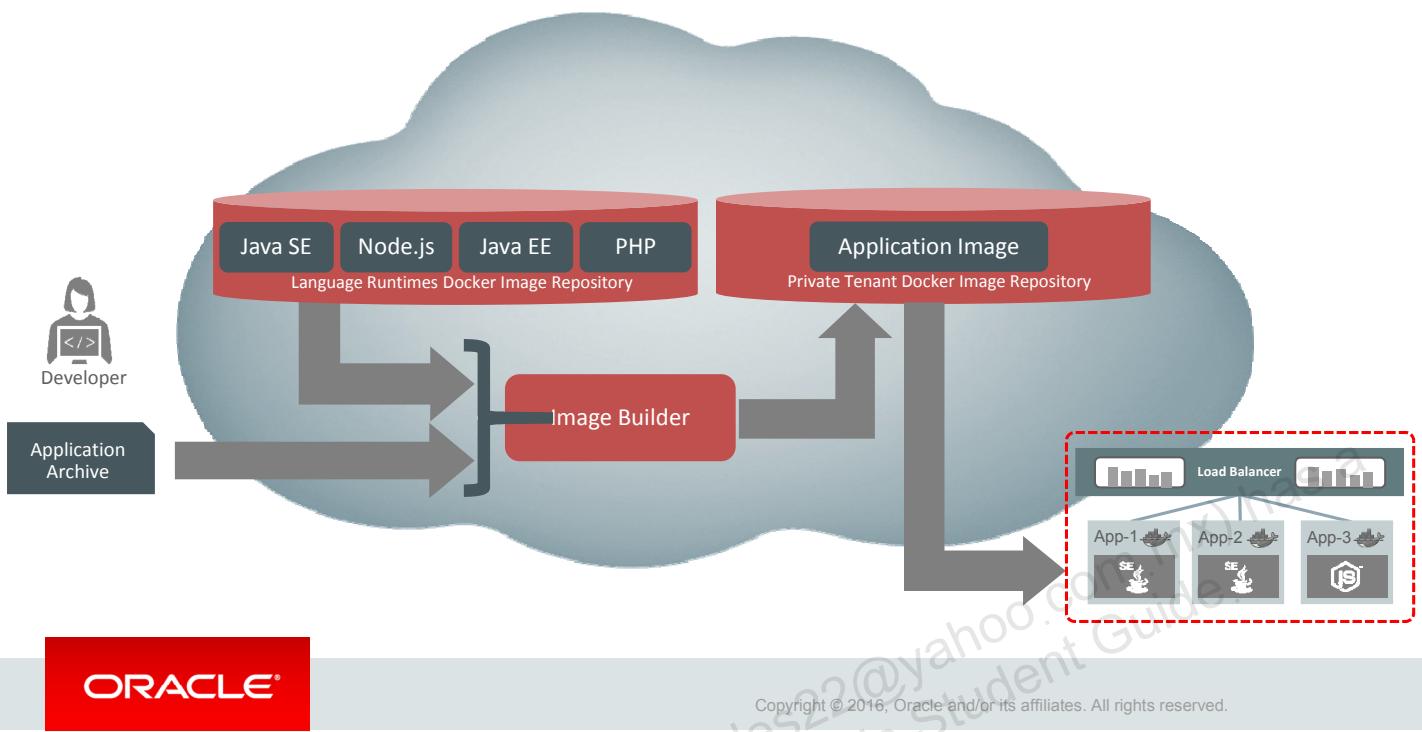
Everything you'd need to run your application on a virgin machine



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Application Deployment

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.

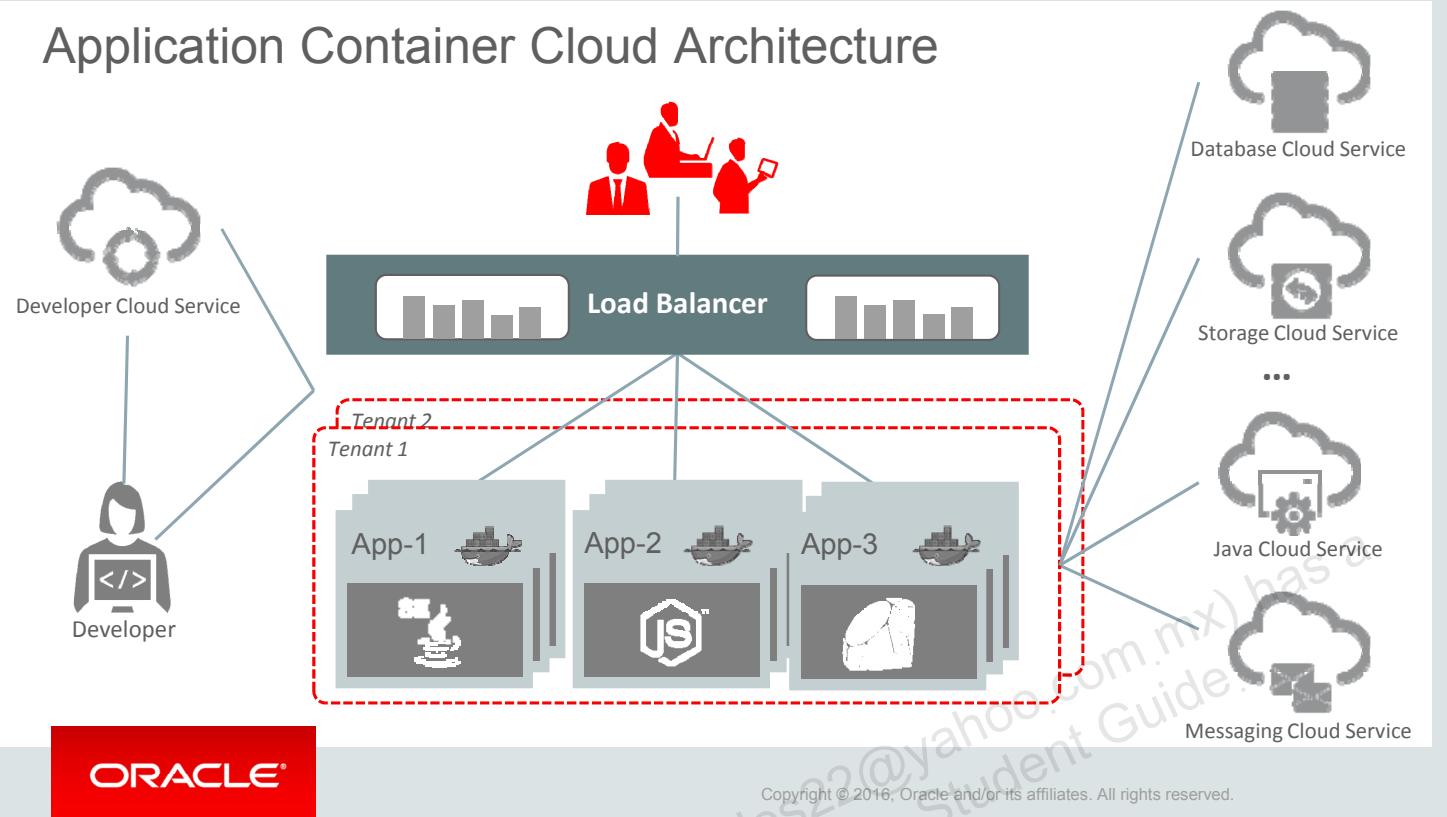


ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Application Container Cloud Architecture

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.



- Tenant Isolation
- Polyglot
- Integrated
- Developer Friendly

Load Balancer



Fully automated—no user management required

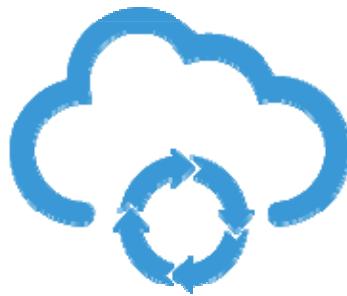
Scale out or in and application instances are automatically registered/unregistered

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Vanity URL support (upcoming) will allow installation of certificates

Oracle Developer Cloud Service



Source Control Management



Issue Tracking



Hudson Continuous Integration



Wiki Collaboration

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Complete, Integrated Development Platform—as a Service

Application Lifecycle Management

Team Management

Entitlement with all Application Container Cloud services

Developer Cloud Service – Easy Adoption/Integration

Pre-integrated development technologies in the cloud

Standards Based

- Git, Maven, Hudson, Ant, Grunt, Gulp, etc.

Built-in IDE Integration

- Eclipse, NetBeans, JDeveloper

Flexible Source Location

- Hosted Git or GitHub

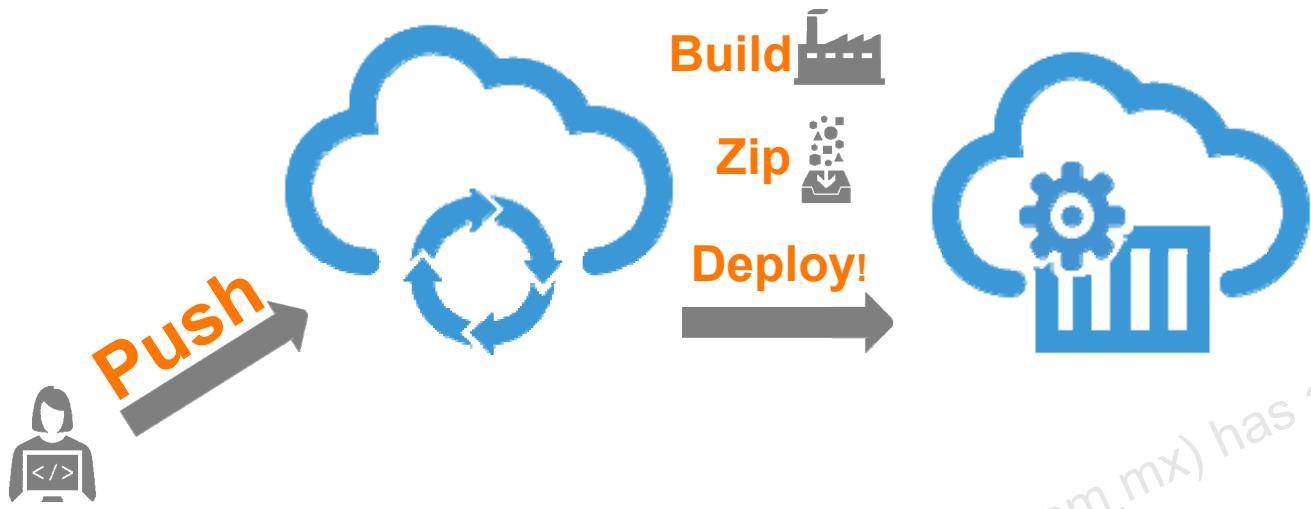
Choice of Deployment Target

- Oracle Cloud or on-premise



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

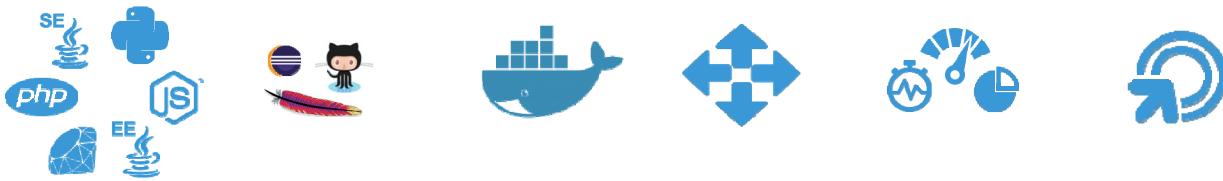


ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Rather than build on-premise, use DevCS to perform continuous build, test, and deployment.

Application Container Cloud Service Advantages



- Integrated **enterprise** ecosystem and services from IaaS to PaaS and SaaS
- Java SE Advanced – completely **unique** and unavailable on any other cloud platform
- Developer Cloud Service – **included** and **integrated**

ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have :

- Got an overview of Oracle Application Container Cloud
- Understood the unique features of Oracle Application Container Cloud
- Understood how to build, zip, and deploy applications to the cloud



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Java Puzzle Ball Challenge Questions Answered



ORACLE®

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Topics

- Lesson 6
- Lesson 8
- Lesson 10
- Lesson 11
- Lesson 12

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Question 1

How many objects can you identify in the game?

- Red wheels
- Blue wheels
- Ball
- Duke



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

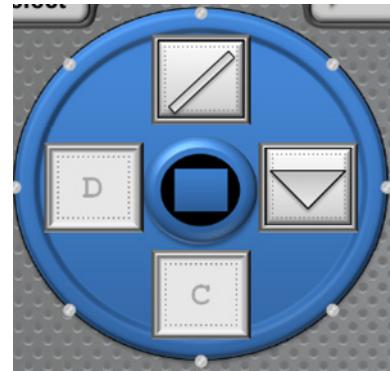
Discussion

You might also have answered that the walls are objects. Remember that objects are not always physical. For instance, you might suggest that Score is an object, or Level (the level of the game). The business of identifying objects for an object-oriented application is more art than science.

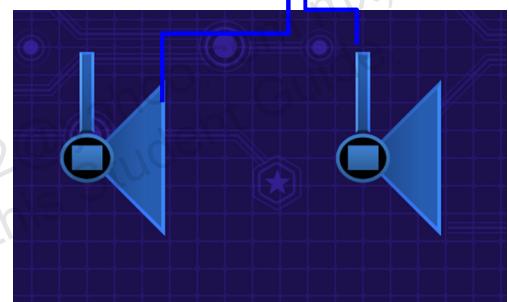
Question 2

Given that a class is a blueprint for an object, which game components best reflect the class-instance relationship?

- The blue objects are *instances* of the BlueWheel class.
- The objects share the properties and methods of the BlueWheel class.



BlueBumper object instances



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

Discussion

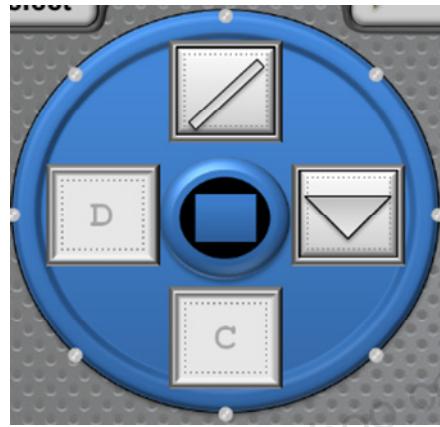
Here is an example of one of the game's objects, the blue bumper, and its function in the game. BlueBumper is the class, and a class is a blueprint or recipe for an object. The class describes an object's properties and behaviors. Classes are used to create object instances, such as the two BlueBumper object instances, as shown in the second image.

Initially, each object instance looks the same, but as you noticed, objects can change and differentiate themselves as play continues.

Question 3

How many object properties can you find?

- Color
- Shape
- Orientation
- X position
- Y position



ORACLE®

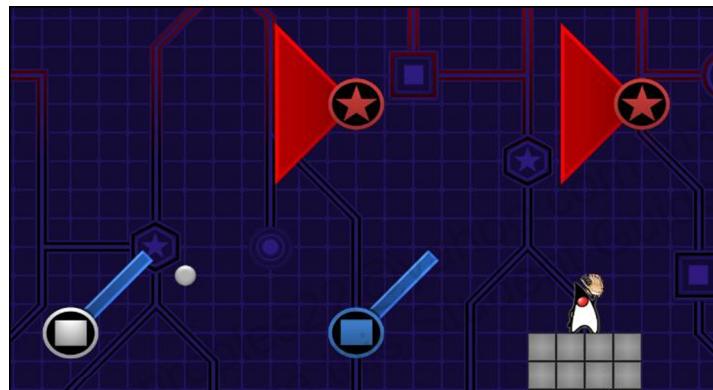
Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

There are many others you could have named as well.

Question 4

Can you guess what some of the methods might be?

- Behaviors:
 - divertCourseSimple
 - divertCourseTriangle
 - rotate
 - play



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are many possibilities of methods for this game. Methods are actions that occur in a program. Here are a few of the possible methods. You will, no doubt, think of many more.

- You know that when the ball strikes a wall, its course will be diverted, but the change in course will be different if the wall is a simple wall or if it is a triangle wall. Consequently, you assume that these are two different methods.
- You also know that when you rotate one of the wheels, the objects of the same color are also rotated, so you can assume that there must be a rotate method.
- When you click Play, the ball starts to move, so you can assume that a play method must exist.

Note: When you play the other puzzles of the game, you will see several other types of walls and different behaviors.

Topics

- Lesson 6
- Lesson 8
- Lesson 10
- Lesson 11
- Lesson 12

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Question 1

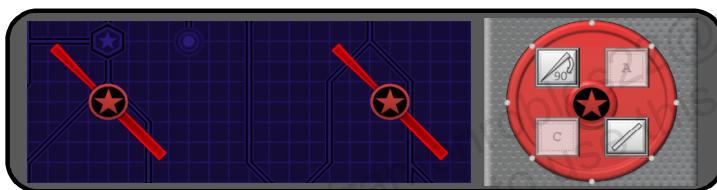
Which of the scenarios below reflects the behavior of:

- A static variable?
- An instance variable?

1. A single bumper rotates after being struck by the ball.



2. Rotating the red wheel changes the orientation of all red bumpers.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In Scenario 1, you see the behavior of an instance variable. When the rotation wall is struck, only that particular object changes its orientation. Other objects retain their previous orientation. This is the behavior you would expect to see with an instance variable.

In Scenario 2, you see the behavior of a static variable exhibited. When you change the orientation of a wheel, all objects of that same color are also rotated and, therefore, share the same orientation as the wheel.

Topics

- Lesson 6
- Lesson 8
- **Lesson 10**
- Lesson 11
- Lesson 12

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Question 1

- What type of conditional construct would you use to handle the behavior of the blade?

Option 1: Using an `if/else` construct

- Pseudocode example

```
If object struck != fan
    destroy object struck
    change blade to a ball
else
    divert course
    destroy the next object struck
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As you saw in the game, there is some fairly complex logic implemented that determines both the trajectory of the ball and also the properties of the objects it strikes. When the appearance of the ball has changed to a blade, it destroys the object it strikes—but not if the object is a fan.

- There are several ways you could structure this logic. You will examine two options here.
- On this page, you see a pseudocode example of how you might handle the conditional logic using an `if/else` construct. It first tests for the exception (that is, not destroying the object) and then tests for all remaining conditions.

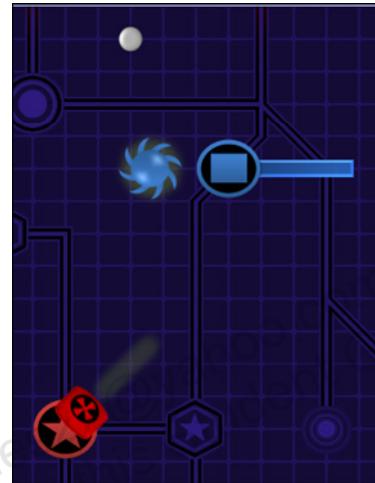
Question 1

- What type of conditional construct would you use to handle the behavior of the blade?

Option 2: Using a `switch` construct

- Pseudocode example

```
switch objectStruck
    case wall or rotation or triangle
        destroy objectStruck
        change blade to a ball
        break
    case fan
        divert course
        destroy the next object
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Here you see an example using a `switch` construct. You will notice that this logic is structured similarly to the `if/else` construct. The main difference is that it reverses the conditional test. It first tests all of the cases in which the object *should* be destroyed, then tests for the exception.
- Note that, because a `switch` statement can only evaluate a single value of type `int`, `short`, `byte`, `char`, or `String`, you would have to evaluate the object struck using some derived value (an object ID for instance).

Topics

- Lesson 6
- Lesson 8
- Lesson 10
- **Lesson 11**
- Lesson 12

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Question 1

- How might you structure the logic of the blade behavior using a `while` loop?
 - Pseudocode example

```
While ball image is "blade"
  if object struck == fan
    continue loop
  else
    destroy object struck
    change ball image to "ball"
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the pseudocode example, a `while` loop is executed as long as the image representing the ball is a blade. It continues to the next loop iteration if it encounters a fan. Thus, the loop will continue to iterate until it strikes something other than a fan. When this happens, the ball ceases to be a blade, so the loop exits.

You might have used a different approach to the `while` loop. Again, there are many approaches that would work.

Topics

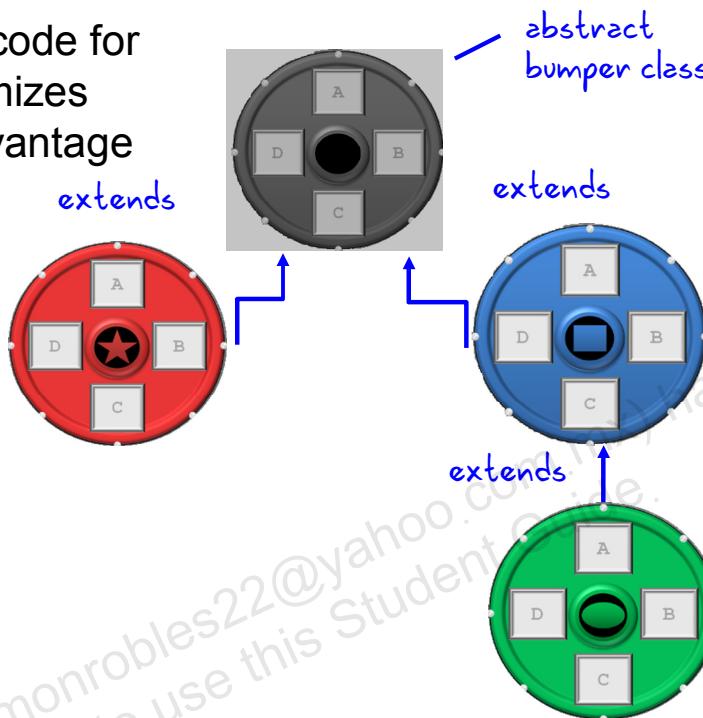
- Lesson 6
- Lesson 8
- Lesson 10
- Lesson 11
- Lesson 12

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Question 1

Is there a way to design code for these bumpers that minimizes duplication and takes advantage of polymorphism?



ORACLE

Copyright© 2014, Oracle and/or its affiliates. All rights reserved.

In the basic puzzles, you saw a red wheel and a blue bumper. In the inheritance puzzles of the game, green bumpers were introduced. You saw that the green bumper seems to inherit from (extend) the blue bumper. Obviously, all three of these bumpers share functionality and properties.

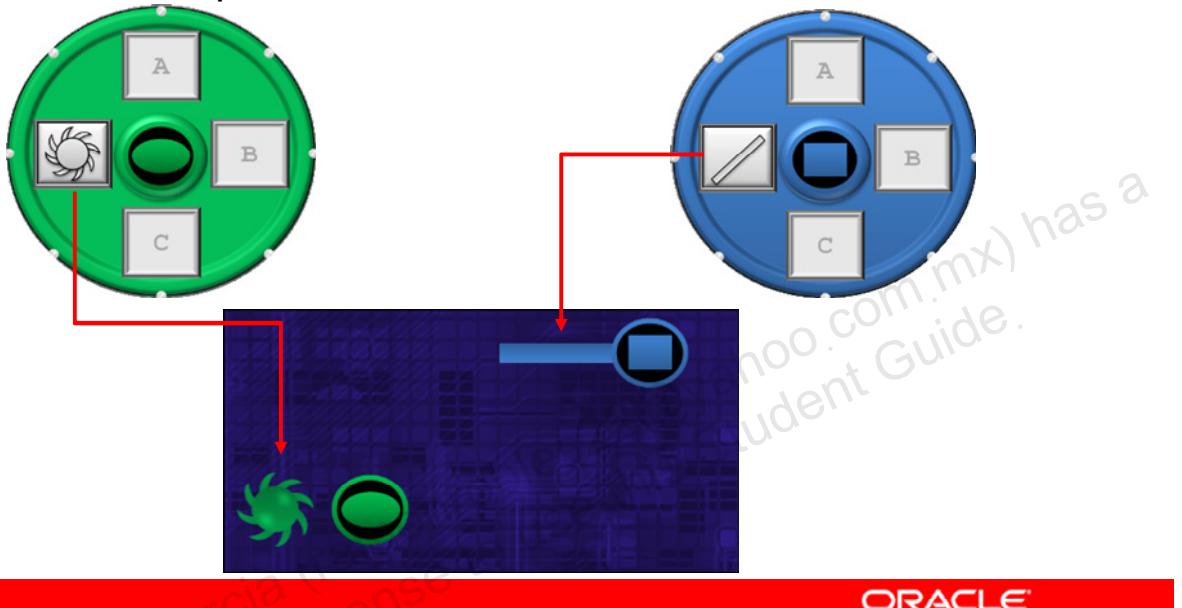
An abstract class can be used to impose the interface that we see in all of these classes. It might also include a concrete method or property.

The graphic above shows this modified hierarchy (Remember from the lesson titled “Describing Objects and Classes” how an instance was best represented by a bumper, and a class was best represented by the wheel). Assuming that the green bumper class extends the blue bumper class, it derives all of its behaviors and properties from the abstract bumper class through the blue bumper class. Only red and blue bumper classes directly extend the abstract bumper class.

Question 2

To make overriding possible, which game components best represent:

- A method name and signature?
- A method implementation?



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you drag an icon, such as the simple wall in the example above, to a slot in the blue wheel (slot D), all green objects show the simple wall in the D position as well.

If, however, you then drop a different icon (blade, in this example) to the D slot of the green wheel, the simple wall is replaced by the blade. The icon would therefore represent a new implementation of the method found in slot D. You could consider slot D to represent a method name and signature, such as `slot_D()` or `D_method()`.

JB

Introducing the Java Technology

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Ramon Robles Garcia (ramonrobles22@yahoo.com.mx) has a
non-transferable license to use this Student Guide.

Java's Place in the World

#1



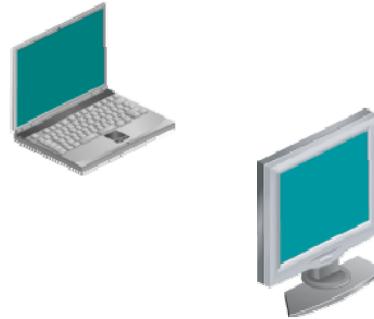
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To put things in perspective, Java is the single most widely used development language in the world today, with over 9 million developers saying they spend at least some of their time developing in Java, according to a recent Evans Data study. That's out of a world population of about 14 million developers.

Java Desktops

#1



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- 1.1 billion desktops run Java (Nielsen Online, Gartner 2010).
- 930 million JRE downloads a year (August 2009–2010): The Java Runtime Environment (JRE) is used by end users.
- 9.5 million JDK downloads a year (August 2009–2010): The Java Development Kit (JDK) is used by Java developers.

Java Mobile Phones

#1



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

All non-smart phones (“feature phones”) run Java.

Java TV and Card

#1



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- 100% of Blu-ray players run Java.
- 71.2 million people connect to the web on Java-powered devices (InStat 2010).
- 1.4 billion Java Cards are manufactured every year (InStat 2010).

The Story of Java

Once upon a time...



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

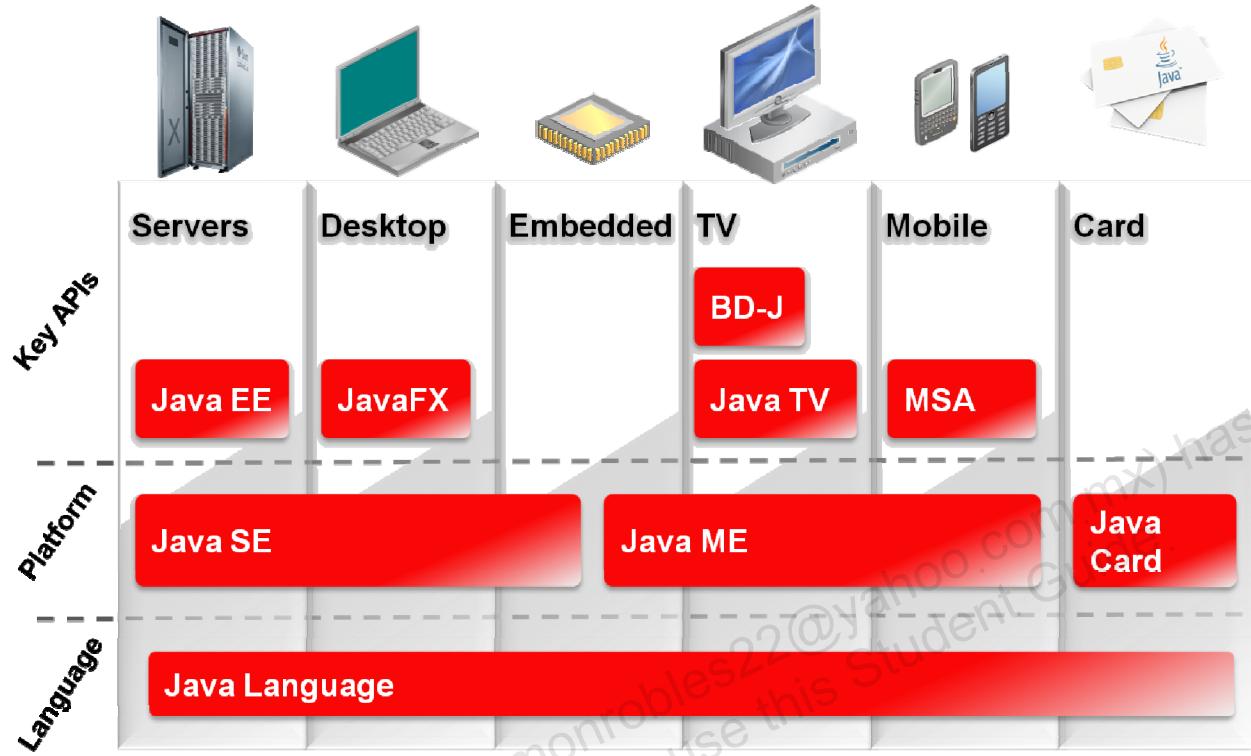
The Java programming language (formerly Oak) originated in 1991 as part of a research project to develop a programming language that would bridge the communication gap between many consumer devices, such as video cassette recorders (VCRs) and televisions. Specifically, a team of highly skilled software developers at Sun (the Green team, under the leadership of James Gosling) wanted to create a programming language that enabled consumer devices with different central processing units (CPUs) to share the same software enhancements.

This initial concept failed after several deals with consumer device companies were unsuccessful. The Green team was forced to find another market for their new programming language. Fortunately, the World Wide Web was becoming popular and the Green team recognized that the Oak language was perfect for developing web multimedia components to enhance webpages. These small applications, called applets, became the initial use of the Oak language, and programmers using the Internet adopted what became the Java programming language.

The turning point for Java came in 1995, when Netscape incorporated Java into its browser.

Did You Know? The character in the slide is Duke, Java's mascot. The original Duke was created by the Green team's graphic artist, Joe Palrang.

Identifying Java Technology Product Groups



ORACLE

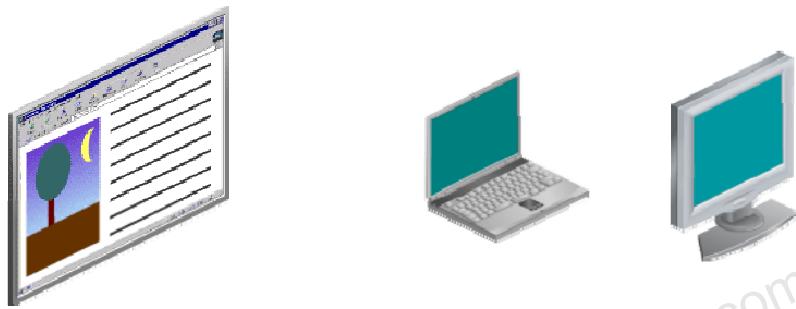
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle provides a complete line of Java technology products, ranging from kits that create Java technology programs to emulation (testing) environments for consumer devices such as cellular phones. As indicated in the graphic, all Java technology products share the foundation of the Java language. Java technologies, such as the Java Virtual Machine, are included (in different forms) in three different groups of products, each designed to fulfill the needs of a particular target market. The figure illustrates the three Java technology product groups and their target device types. Each edition includes a Java Development Kit (JDK) (also known as a Software Development Kit [SDK]) that allows programmers to create, compile, and execute Java technology programs on a particular platform.

Note: The JavaFX API is a rich client for creating user interfaces for your Java program. The MSA API is the mobile software application used to create user interfaces on portable devices.

Java SE

Is used to develop applets that run within web browsers and applications that run on desktop computers



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

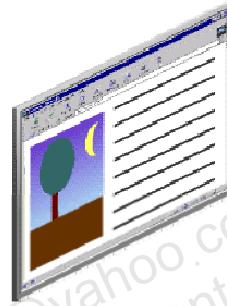
Java Platform, Standard Edition (Java SE) is used to develop applets and applications that run within web browsers and on desktop computers. For example, you can use the Java SE JDK to create a word processing program for a personal computer.

You use a Java desktop application in this course. It is an Integrated Development Environment (IDE) called NetBeans.

Note: Applets and applications differ in several ways. Primarily, applets are launched inside a web browser, whereas applications are launched within an operating system. Although this course focuses mainly on application development, most of the information in this course can be applied to applet development.

Java EE

Is used to create large enterprise, server-side, and client-side distributed applications



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Platform, Enterprise Edition (Java EE) is used to create large enterprise, server-side, and client-side distributed applications. For example, you can use the Java EE JDK to create a web shopping (eCommerce) application for a retail company's website.

Java EE is built on top of the Java SE Platform, extending it with additional APIs supporting the needs of large-scale, high-performance enterprise software. The APIs are packaged and grouped to support different kinds of containers, such as a web container for web-based applications, a client container for thick clients, and the EJB container to run workhorse Java components. Some of the kinds of functionality supported by the different APIs include objects, UI, integration, persistence, transactions, and security.

Java ME

Is used to create applications for resource-constrained consumer devices



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Platform, Micro Edition (Java ME) is used to create applications for resource-constrained consumer devices. For example, you can use the Java ME JDK to create a game that runs on a cellular phone. Blu-ray Disc Java applications and Java TV use the same SDK as Java ME.

Java Card

Java Card is typically used in the following areas (and many more):

- Identity
- Security
- Transactions
- Mobile phone SIMs



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Product Life Cycle (PLC) Stages

1. Analysis
2. Design
3. Development
4. Testing
5. Implementation
6. Maintenance
7. End-of-Life (EOL)



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The product life cycle is an iterative process used to develop new products by solving problems.

- **Analysis:** The process of investigating a problem that you want to solve with your product. Among other tasks, analysis consists of:
 - Clearly defining the problem you want to solve, the market niche you want to fill, or the system you want to create. The boundary of a problem is also known as the **scope** of the project.
 - Identifying the key subcomponents of your overall product

Note: Good analysis of the problem leads to a good design of the solution and to decreased development and testing time.

- **Design:** The process of applying the findings you made during the analysis stage to the actual design of your product. The primary task during the design stage is to develop blueprints or specifications for the products or components in your system.
- **Development:** Using the blueprints created during the design stage to create actual components
- **Testing:** Ensuring that the individual components, or the product as a whole, meet the requirements of the specification created during the design stage

Note: Testing is usually performed by a team of people other than those who actually developed the product. Such a team ensures that the product is tested without any bias on behalf of the developer.