

Java Fundamentals

Student Guide—Volume II

X95176GC10

Edition 1.0 | May 2016

Learn more from Oracle University at oracle.com/education/

ORACLE

Authors

Nick Ristuccia
Eric Jendrock
Michael Williams
Luz Elena Peralta Ayala
Eduardo Moranel Rosales

Editors

Aishwarya Menon
Nikita Abraham
Raj Kumar
Vijayalakshmi Narasimhan
Aju Kumar

Graphic Designer

Maheshwari Krishnamurthy

Publishers

Giri Venugopal
Srividya Rameshkumar
Raghunath M

Copyright © 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

- 0 Lesson 0: Course Introduction**
- 1 Lesson 1: Introduction to Modern Software Development**
- 2 Lesson 2: Java Programming and Git**
- 3 Lesson 3: Creating a Java Main Class**
- 4 Lesson 4: Storing and Managing Local Data**
- 5 Lesson 5: Managing Multiple Items**
- 6 Lesson 6: Describing Objects and Classes**
- 7 Lesson 7: Manipulating and Formatting the Data in Your Program**
- 8 Lesson 8: Creating and Using Methods**
- 9 Lesson 9: Project Management**
- 10 Lesson 10: Using Encapsulation**
- 11 Lesson 11: Using Conditionals**
- 12 Lesson 12: Working with Arrays, Loops, and Dates**
- 13 Lesson 13: Using Inheritance**
- 14 Lesson 14: Using Interfaces**
- 15 Lesson 15: Handling Exceptions**
- 16 Lesson 16: Introduction to HTTP, REST, and JSON**
- 17 Lesson 17: Creating RESTWeb Services with Java**
- 18 Lesson 18: Deploying RESTWeb Services to theCloud**

Oracle University Student Learning Subscription Use Only



Lesson 10: Using Encapsulation

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

Oracle University Student Learning Subscription Use Only

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 University Student Learning Subscription Use Only

ORACLE®

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

Topics

- Access control
- Encapsulation
- Overloading constructors



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Oracle University Student Learning Subscription Use Only

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 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



Oracle University Student Learning Subscription Use Only

```
public class Item {  
    private double price = 15.50;  
    public void setPrice(Customer cust){  
        if (cust.hasLoyaltyDiscount()) {  
            price = price*.85; }  
    }  
}
```

```
public class Order{  
    public static void main(String args[]){  
        Customer cust = new Customer(10);  
        Item item = new Item();  
        item.price = 10.00; // Won't compile  
        item.setPrice(cust); // You don't need to know how  
        // setPrice works in order to use it.  
    }  
}
```

ORACLE®

Copyright © 2016, 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 }
```

ORACLE®

Copyright © 2016, 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();  
9         desc = "--description required--";  
10        price = 0.00;  
11    }  
12  
13    private void setId() {  
14        id = Item.nextId++;  
15    }  
}
```

Called from within a public method

Private method



Copyright © 2016, 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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 the interface remains the same.
- Encapsulation encourages good object-oriented (OO) design.

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 © 2016, 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 © 2016, 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, 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



Oracle University Student Learning Subscription Use Only

```
15  public void setColorCode(char newCode) {  
16      if (newCode == 'R') {  
17          colorCode = newCode;  
18          return;  
19      }  
20      if (newCode == 'G') {  
21          colorCode = newCode;  
22          return;  
23      }  
24      if (newCode == 'B') {  
25          colorCode = newCode;  
26          return;  
27      }  
28      System.out.println("Invalid colorCode. Use R, G, or B");  
29  }
```



Copyright © 2016, 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.setStatusCode('Z'); Not a valid color code  
8         System.out.println("Color Code: " + theShirt.getColorCode());  
9     }  
}
```

Output:

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



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

Building on the previous slides, before the call to `setStatusCode`, the default color value of U (unset) is printed. If you call `setStatusCode` 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.

Practice 10-1: Encapsulating a Class



In this exercise, you encapsulate the `Customer` class.

1. Open the project **Practice_10-1**.
2. Change access modifiers so that fields must be read or modified through public methods.
3. Allow the `name` field to be read and modified.
4. Allow the `ssn` field to be read but not modified (read only).



ORACLE®

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

Oracle University Student Learning Subscription Use Only

In this exercise, you encapsulate the `Customer` class.

Topics



- Access control
- Encapsulation
- Overloading constructors

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Java Fundamentals 10 - 16

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.setCode('R');  
7         theShirt.setDescription("Outdoors shirt");  
8         theShirt.setPrice(39.99);  
9     }  
10 }
```

Using a constructor:

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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Shirt Constructor with Arguments



Oracle University Student Learning Subscription Use Only

```
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    }  
}
```

ORACLE®

Copyright © 2016, 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.

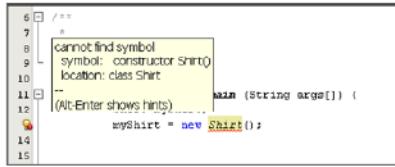
Default Constructor and Constructor with Args



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

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

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



A screenshot of an IDE showing a compilation error. The code is as follows:

```
6 // **  
7 *  
8 * cannot find symbol  
9 *   symbol: constructor Shirt()  
10 *   location: class Shirt  
11 * (Alt-Enter shows hints) ... main (String args[]) {  
12 *     myShirt = new Shirt();  
13 * }  
14 *  
15 *
```

The line at position 8, which contains the declaration of the no-argument constructor, is highlighted with a red box and has a tooltip: "cannot find symbol symbol: constructor Shirt() location: class Shirt". The line at position 12, where the constructor is called, is also highlighted with a red box.

ORACLE®

Copyright © 2016, 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 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 © 2016, 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.

Practice 10-2: Creating a Constructor



1. Continue editing **Practice_10-1** or open **Practice_10-2**.

In the Customer class:

2. Add a custom constructor that initializes the fields.

In the ShoppingCart class:

3. Declare, instantiate, and initialize a new Customer object by calling the custom constructor.
4. Test it by printing the Customer object name
(call the getName method).



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

Oracle University Student Learning Subscription Use Only

In this exercise, you add a constructor to the `Customer` class and create a new `Customer` object by calling the constructor.

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 University Student Learning Subscription Use Only

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Practice 10-3: Using Encapsulation



- In this practice, you are asked to encapsulate code in the soccer league application. This includes creating constructors. You are told what must be encapsulated, but it's up to you to figure out the implementation.
- You are also asked to remove a method from GameUtils.java and to test code by printing four games instead of one. The code should be able to accommodate any number of games.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Oracle University Student Learning Subscription Use Only



Lesson 11: Using Conditionals

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

Oracle University Student Learning Subscription Use Only

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 University Student Learning Subscription Use Only

ORACLE®

Copyright © 2016, 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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Review: Relational Operators



Oracle University Student Learning Subscription Use Only

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



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

Oracle University Student Learning Subscription Use Only

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

Oracle University Student Learning Subscription Use Only

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

Oracle University Student Learning Subscription Use Only

- 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."); ✓  
        }  
    }  
}
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

- 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 © 2016, 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



Oracle University Student Learning Subscription Use Only

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

Equivalent statements

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

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

ORACLE®

Copyright © 2016, 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 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"`);



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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.

Java Fundamentals 11 - 11

Practice 11-1: 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 © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle University Student Learning Subscription Use Only

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Java Puzzle Ball



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

Consider the following:

[What happens if the ball strikes the blade?](#)



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 University Student Learning Subscription Use Only

ORACLE®

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

The logic driving the behavior in the game is conditional upon what type of object the ball (or blade) strikes. This lesson introduces some new conditional constructs and some new ways of using the constructs that you already know.

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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Determining the Number of Days in a Month



Oracle University Student Learning Subscription Use Only

```
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.");
```

ORACLE®

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

- The code example in the slide 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 © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle University Student Learning Subscription Use Only

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 example in the slide, 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.

Practice 11-2: Chaining if Statements



1. Open the project **Practice_11-2**.

In the Order class:

2. Complete the `calcDiscount` method so it determines the discount for three different customer types:

- Nonprofits get a discount of 10% if total > 900, else 5%.
- Private customers get a discount of 7% if total > 900, else 0%.
- Corporations get a discount of 8% if total < 500, else 5%.

In the ShoppingCart class:

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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Topics



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

Oracle University Student Learning Subscription Use Only

ORACLE®

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

Handling Complex Conditions with a switch Statement



The `switch` statement:

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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 }
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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_coe  
10 }
```



Copyright © 2016, 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 or enum (enumerated types)
 - Wrapper classes (special classes that wrap certain primitive types): Integer, Short, Byte, and Character

Only a single variable can be tested.

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

Known values



Copyright © 2016, 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.

Practice 11-3: Using switch Constructs



1. Continue editing **Practice_11-2** or open **Practice_11-3**.

In the Order class:

2. Rewrite calcDiscount to use a switch statement:

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

In the ShoppingCart class:

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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Oracle University Student Learning Subscription Use Only

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

Oracle University Student Learning Subscription Use Only

Setting Breakpoints



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

```
3  public class DebugTestIfElse {
4      public static void main(String[] args) {
5          int month =1;
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 }
```



Copyright © 2016, 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 © 2016, 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

Breakpoint

Current line of execution

Name	Type	Value
args	String[]	["2016"]
month	int	2
isLeapYear	boolean	true

Value of variables

ORACLE®

Copyright © 2016, 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 University Student Learning Subscription Use Only

ORACLE®

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

Practice 11-4: Using Conditionals



In this practice, you are asked to enhance the soccer league application to keep track of the points and goals of each team. Features are best implemented using conditional statements, but it's up to you to figure out the implementation details.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Oracle University Student Learning Subscription Use Only



A group of four students are gathered around a desk in a classroom, looking at two computer monitors. One student is pointing at the screens. The Oracle logo is visible in the bottom left corner of the slide.

Lesson 12: Working with Arrays, Loops, and Dates

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



Oracle University Student Learning Subscription Use Only

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 University Student Learning Subscription Use Only

ORACLE®

Copyright © 2016, 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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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.



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

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 © 2016, 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 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 the classes in the package
`java.time.*;`

Working with Dates



`java.time`

- Is the main package for the 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);
```

ORACLE®

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

In the example in the slide, `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);`
- Static methods:
 - `of(int year, Month month, int dayOfMonth)`
 - `parse(CharSequence text, DateTimeFormatter formatter)`
 - `now()`



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

LocalDate has many methods and fields. A few of the instance and static methods that you might use are as follows:

- `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 by 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);
```

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

ORACLE®

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

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

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

Practice 12-1: Declaring a LocalDateTime Object



1. Open the project Practice_12-1 or create your own project with a Java Main Class named TestClass.
2. Declare a LocalDateTime object to hold the order date.
3. Initialize the object to the current date and time by using the now() static method of the class.
4. Print the orderDate object with a suitable label.
5. Format orderDate by using the ISO_LOCAL_DATE static constant field of the DateTimeFormatter class.
6. Add the necessary package imports.
7. Print the formatted orderDate with a suitable label.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

In this exercise, you print and format today's date.

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Using the args Array in the main Method



- The `main` method contains a String array, `args`.
- You can pass arguments to this array as a property of the NetBeans project.
- The code for retrieving the parameters is:

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

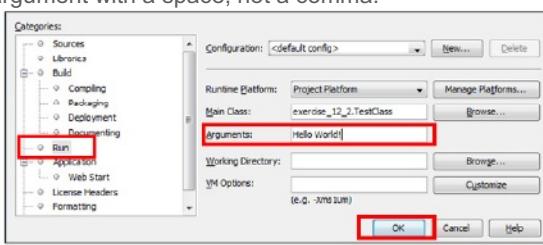
The command line options are passed to the Main method using the args array. These are sometimes known as command-line arguments.

How to Pass Arguments to the args Array



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

args[0] is Hello
args[1] is World!



ORACLE®

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

Oracle University Student Learning Subscription Use Only

You can set arguments as a property of the NetBeans project.

When you pass arguments to your program, they are put in the `args` array as strings. 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`).

Converting String Arguments to Other Types



- Numbers can be typed as parameters:

Arguments:

Total is: **23** Concatenation,
Total is: 5 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));  
    }  
}
```

Note the parentheses.

ORACLE®

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

The `main` method treats everything that 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 that the numbers 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`.

Practice 12-2: Parsing the args Array



1. Open the project **Practice_12-2** or create your own project with a **Java Main Class** named TestClass.
2. Parse the args array to populate name and age.
 - If args contains fewer than two elements, print a message telling the user that two arguments are required.
 - Remember that the age argument will have to be converted to an int.
3. Print the name and age values with a suitable label.
Hint: Use a static method of the Integer class to convert it.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Topics



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

ORACLE®

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

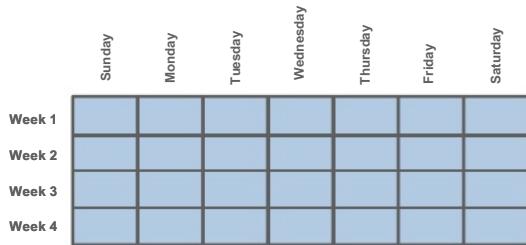
Oracle University Student Learning Subscription Use Only

Java Fundamentals 12 - 16

Describing Two-Dimensional Arrays



Oracle University Student Learning Subscription Use Only



ORACLE®

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

You can store matrixes 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;
```

ORACLE®

Copyright © 2016, 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 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 in the slide 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  
int[][] yearlySales = new int[5][4];
```

Syntax:

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

	Quarter1	Quarter2	Quarter3	Quarter4
Year 1				
Year 2				
Year 3				
Year 4				
Year 5				



Copyright © 2016, 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;
```

	Quarter1	Quarter2	Quarter3	Quarter4
Year 1	1000	1500	1800	
Year 2	1000			
Year 3				
Year 4			2000	
Year 5				



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

Oracle University Student Learning Subscription Use Only

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, the 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]`.

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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.

ORACLE®

Copyright © 2016, 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 in the slide.

Repeating Behavior



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

ORACLE®

Copyright © 2016, 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 in the slide 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     }  
}
```

Annotations on the code:

- A blue bracket underlines the condition `(currentFloor != targetFloor)` with the label "Boolean expression".
- A blue bracket encloses the code block from `if(currentFloor < targetFloor)` to `goDown();` with the label "Body of the loop".



Copyright © 2016, 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;  
}
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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:

```
/*
 *
 *
 *
 */

```

ORACLE®

Copyright © 2016, 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 by 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 }
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 in the slide, the loop iterates four times.

Standard for Loop Compared to a while loop



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

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



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

Oracle University Student Learning Subscription Use Only

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



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

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

The example in the slide 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 in the slide could be rewritten as a while loop as follows:

```
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 © 2016, 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 University Student Learning Subscription Use Only

ORACLE®

Copyright © 2016, 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 © 2016, 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 in the slide. 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.

Practice 12-3: Processing an Array of Items



1. Open the project **Practice_12-3**.

In the `ShoppingCart` class:

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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

In this exercise, you code the `displayTotal` method of the `ShoppingCart` class so that it iterates through the `items` array and prints out the total for the Shopping Cart.

Topics



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

Oracle University Student Learning Subscription Use Only

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 © 2016, 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 the 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 10 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 10 columns.

Nested while Loop



Example:

```
01 String name = "Lenny";
02 String guess = "";
03 int attempts = 0;
04 while (!guess.equalsIgnoreCase()) {
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!

ORACLE®

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

The nested `while` loop in the example in the slide 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 }
```

ORACLE®

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

The example in the slide 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, and so on) 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 60745  
Q4 11975  
  
Q1 72535  
Q2 37360  
Q3 20527  
Q4 36670  
  
Q1 3195  
Q2 98608  
Q3 21433  
Q4 98519
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Topics



Oracle University Student Learning Subscription Use Only

ORACLE®

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

`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 elements. Some of these are listed in the slide, 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 in the slide.

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.

- Has an option to designate the object type it contains:

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

Contains only String objects

- Can call methods on an ArrayList or its elements:

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

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



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

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

- 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 by 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 in the slide 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 © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle University Student Learning Subscription Use Only

- If you forget to import `java.util.ArrayList`, NetBeans will complain but also correctly suggest that you add the `import` statement.
- In the example in the slide, 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  = 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);
```

ORACLE®

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

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

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

There are a number of methods to add data to theArrayList. This example uses theadd 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.

- 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 theArrayList to System.out.println. The resulting output is: [Prashant, Alisa]

More Methods in the ArrayList Class



- **get(int index)** : Returns the element at the specified position in this list

```
ArrayList<String> names = new ArrayList();
names.add("Jamie");
names.add("Gustav");
System.out.println(names.get(0)); //Prints "Jamie"
```

- **toArray()** : Returns an Object array containing all elements in the list

```
ArrayList<String> names = new ArrayList();
names.add("Jamie");
names.add("Gustav");
String[] namesArray = (String[]) names.toArray();
```

Must cast from an Object array to a String array



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

Oracle University Student Learning Subscription Use Only

Consult Oracle's Java documentation to see every method in the `ArrayList` class.

Even More Methods in the ArrayList Class



- **contains(Object o)**: Returns true if the list contains the specified element

```
ArrayList<String> names = new ArrayList();
names.add("Jamie");
names.add("Gustav");
boolean b = names.contains("Damien");           //false
```

Oracle University Student Learning Subscription Use Only



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

Consult Oracle's Java documentation to see every method in the `ArrayList` class.

Practice 12-4: Working with an ArrayList



1. Open the project **Practice_12-4**.
2. Create a String ArrayList with at least 3 elements.
 - Be sure to add the correct import statement.
 - Print the ArrayList and test your code.
3. Add a new element to the middle of the list.
 - **Hint:** Use the overloaded add method that takes an index number as one of the arguments.
 - Print the list again to see the effect.
4. Test for a particular value in the ArrayList and remove it.
 - **Hint:** Use the contains method. It returns a boolean and takes a single argument as the search criterion.
 - Print the list again.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

In this exercise, you create an ArrayList with at least three elements, add an element, and then remove an element.

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 University Student Learning Subscription Use Only

ORACLE®

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

Play Time!

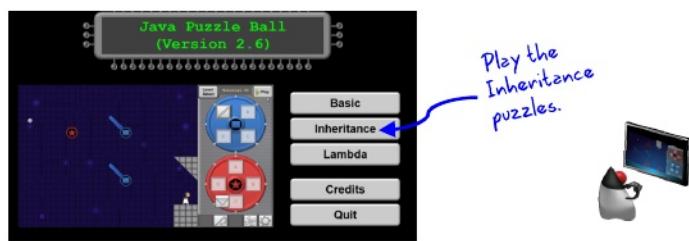


Play **Inheritance Puzzles 1, 2, and 3** before the next lesson titled “Using Inheritance.”

Consider the following:

What is inheritance?

Why are these considered “Inheritance” puzzles?



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Practice 12-5: Iterating Through Data and Working with LocalDateTime



- In this practice, you are asked to enhance the soccer league application to create teams from a database of names, and have all teams play each other. The program must also assign a date to each game and print the length of the season. Features are best implemented by iterating through arrays and ArrayLists, and using LocalDateTime. But it's up to you to figure out the implementation details.
- The practice recommends you consult the Java documentation on the StringTokenizer and Period classes, which were not discussed in lecture.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Oracle University Student Learning Subscription Use Only



Lesson 13: Using Inheritance

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

Oracle University Student Learning Subscription Use Only

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



Oracle University Student Learning Subscription Use Only

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Java Puzzle Ball



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

Consider the following:

[What is inheritance?](#)

[Why are these considered “Inheritance” puzzles?](#)



ORACLE®

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

Oracle University Student Learning Subscription Use Only

What Is Inheritance?



- Did you notice blue shapes appearing on green bumpers?
- **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*.
- The parent class is known as the **superclass**.
- The child class is known as the **subclass**.
- From playing the game, you may have observed and come to understand three key aspects of inheritance and the relationship between superclasses and subclasses.



ORACLE®

Copyright © 2016, 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 Puzzle 1



In the game:

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

In Java:

- The subclass shares the same methods as the superclass.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Both Blue Bumpers and Green Bumpers share the Triangle Wall.

Inheritance Puzzle 2

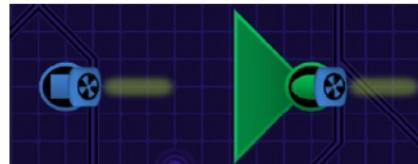


In the game:

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

In Java:

- A subclass may have additional methods which aren't found in the superclass.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Inheritance Puzzle 3



In the game:

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

In Java:

- A subclass may override the methods it inherits from the superclass.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Oracle University Student Learning Subscription Use Only

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



A subclass:

- Has access to all of the public fields and methods of the superclass
- May also have unique fields and methods not found in the superclass

```
subclass      superclass
public class Shirt extends Clothing {
    private int neckSize;

    public int getNeckSize() {
    }
    public void setNeckSize(int nSize) {
        this.neckSize = nSize;
    }
}
```

ORACLE®

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

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

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Duke's Choice Classes: Common Behaviors



Oracle University Student Learning Subscription Use Only

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



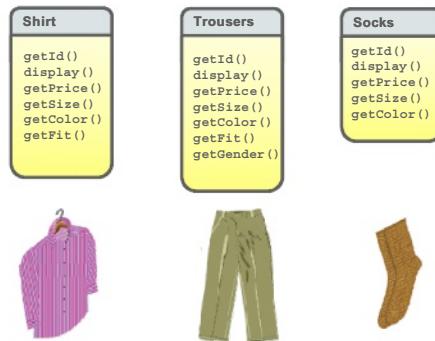
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

The table in the slide shows a set of behaviors for some classes belonging to the Duke's Choice shopping cart application, the Shirt class, and the Trousers class. The classes are shown fully encapsulated so that all field values are accessible only through setter and getter methods. Notice how both classes use many of the same methods; this may result in code duplication, making maintenance and further expansion more difficult and error-prone.

Code Duplication



Oracle University Student Learning Subscription Use Only

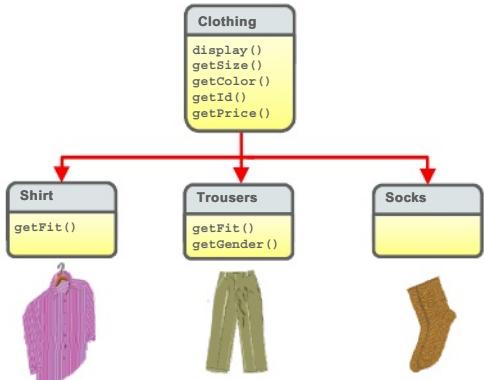


ORACLE®

Copyright © 2016, 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



Benefits:

1. There is less code duplication.
2. Code modification can be done once for all subclasses.

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 {  
02     private char fit = 'U';  
03  
04     public Shirt(int itemID, String description, char  
05                 colorCode, double price, char fit) {  
06         super(itemID, description, colorCode, price);  
07         this.fit = fit; Reference to the  
superclass constructor  
08     }  
09     public char getFit() { Reference to  
this object  
10         return fit;  
11     }  
12     public void setFit(char fit) {  
13         this.fit = fit;  
14     }  
15 }
```



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

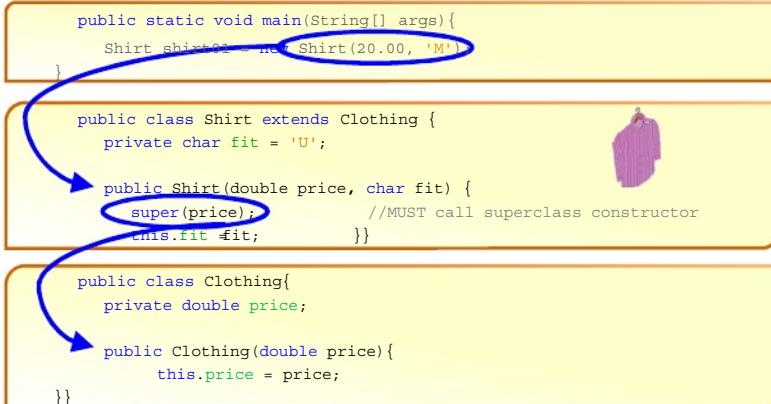
The slide shows the code of the `Shirt` subclass. As you have seen in an earlier example, the `extends` keyword enables the `Shirt` class to inherit all the members of the `Clothing` class. The code declares attributes and methods that are unique to this class. Attributes and methods that are common 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.

Constructor Calls with Inheritance



Oracle University Student Learning Subscription Use Only



Copyright © 2016, 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;  
        }  
    }  
}
```



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

Oracle University Student Learning Subscription Use Only

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.

Practice 13-1: Creating a Subclass, Part 1



1. Open the project **Practice_13-1**.
2. Examine the `Item` class. Pay close attention to the overloaded constructor and also the `display` method.
3. In the **practice_13_1** package, create a new class called `Shirt` that inherits from `Item`.
4. Declare two private `char` fields: `size` and `colorCode`.
5. Create a constructor method that takes three args (`price`, `size`, `colorCode`). The constructor should:
 - Call the two-arg constructor in the superclass:
 - Pass a String literal for the `desc` arg ("Shirt").
 - Pass the `price` argument from this constructor.
 - Assign the `size` and `colorCode` fields.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Practice 13-1: Creating a Subclass, Part 2



In the ShoppingCart class:

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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

Overriding Methods



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

- The new method replaces the method from the superclass (parent).
- For example, the `display` method in `Item` could be overridden for `Shirt`:

```
public void display() {  
  
    System.out.println("Item description: "+getDesc());  
    System.out.println("ID: "+getId());  
    System.out.println("Price: "+getPrice());  
    System.out.println("Size: " + size); // new  
    System.out.println("Color: " + colorCode); // new  
}
```

ORACLE®

Copyright © 2016, 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.

Note: Only nonprivate fields and methods can be accessed by a subclass.

Access Modifiers



Access Modifiers:

public: Accessible by anyone

protected: Accessible only within the class and subclasses in the same package

private: Accessible only within the class

✓ **public** and **protected** methods and data can be accessed or overridden from subclasses.

🚫 **private** methods and data cannot be accessed or overridden from subclasses.

ORACLE®

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

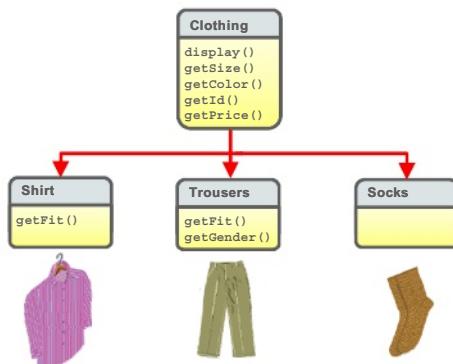
Oracle University Student Learning Subscription Use Only

The access modifier applied to a method determines if it can be overridden.

Review: Duke's Choice Class Hierarchy



Now consider these classes in more detail.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 © 2016, 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 }
```

Call the superclass's version of `setColorCode`.

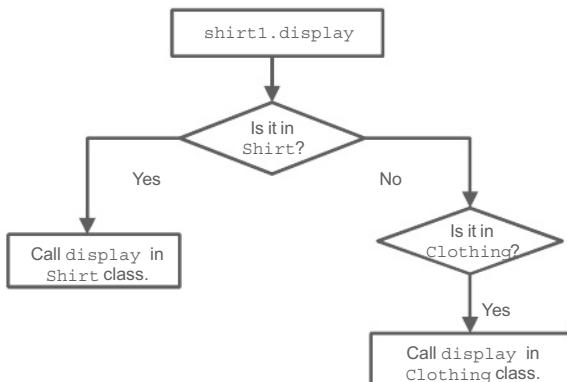


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

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 © 2016, 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 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 understanding which method implementation gets called in various situations.

Practice 13-2: Overriding a Method in the Superclass



1. Open **Practice_13-2** or continue editing **Practice_13-1**.

In the `Shirt` class:

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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Superclass and Subclass Relationships



Use inheritance only when it is completely valid or unavoidable.

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



A Shirt *is a* piece of Clothing.

– A Hat *is a* Sock.

– Equipment *is a* piece of Clothing.



Clothing and Equipment *are* Items.



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

Oracle University Student Learning Subscription Use Only

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



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

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

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

This could be a Trouser or Shirts object.



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

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

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

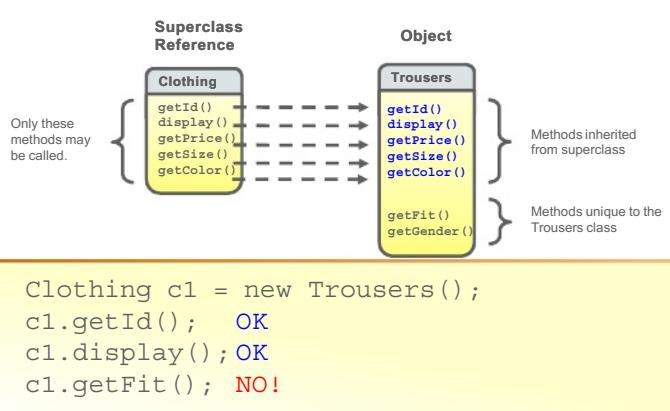
In the example in the slide, the object could be any subclass of `Clothing`. Recall that some of the subclasses of `Clothing` implemented the `display` and `setColorCode` methods, thereby overriding those methods in the `Clothing` class.

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

Accessing Methods Using a Superclass Reference



Oracle University Student Learning Subscription Use Only



ORACLE®

Copyright © 2016, 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

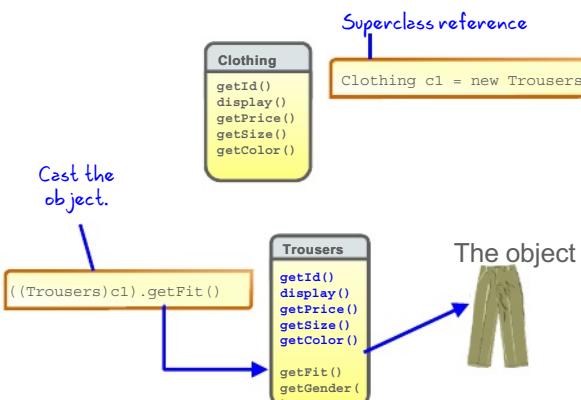


ORACLE®

Copyright © 2016, 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) cl` 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 lesson titled “Using Interfaces.”



instanceof Operator



Possible casting error:

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

What if cl is not a Trousers object?

instanceof operator used to ensure there is no casting error:

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

instanceof returns true if cl is a Trousers object.

ORACLE®

Copyright © 2016, 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.

Practice 13-3: Using the instanceof Operator, Part 1



1. Open **Practice_13-3** or continue editing **Practice_13-2**.

In the Shirt class:

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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Practice 13-3: Using the instanceof Operator, Part 2



In the ShoppingCart class:

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



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

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

Topics



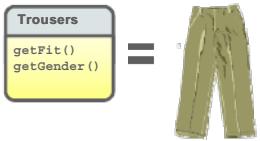
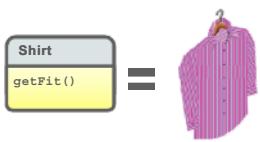
- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

Oracle University Student Learning Subscription Use Only

ORACLE®

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

Abstract Classes



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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
- 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 © 2016, 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());  
    }  
}
```

ORACLE®

Copyright © 2016, 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 that:

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

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

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

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

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

Practice 13-4: Creating a GameEvent Hierarchy



In this practice, you are asked to enhance the soccer league application so that it supports an abstract GameEvent superclass with Goal and Possession subclasses. It's up to you to figure out the implementation details.



ORACLE®

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

Oracle University Student Learning Subscription Use Only



Oracle University Student Learning Subscription Use Only

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 an interface reference to allow access to an object method
- Write a simple lambda expression that consumes a `Predicate`



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Topics



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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

The Object Class



Copyright © 2016, 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



The diagram shows a Java code editor with annotations pointing to specific lines of code. The annotations are:

- Object to String method is used.
- StringBuilder overrides Object's `toString` method.
- First inherits Object's `toString` method.
- Second overrides Object's `toString` method.
- The output for the calls to the `toString` method of each object

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

Output - TestCode (run)

```
java.lang.Object@3e25a5  
Some text for StringBuilder  
First@4f9214  
This class named Second has overridden the toString() method of Object  
BUILD SUCCESSFUL (total time: 1 second)
```

ORACLE®

Copyright © 2016, 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 that 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 reimplementing 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`

ORACLE®

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

Topics



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

Oracle University Student Learning Subscription Use Only

ORACLE®

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

The Multiple Inheritance Dilemma



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

- 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 © 2016, 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 }  
  
1 public class Shirt implements Printable {  
2     ...  
3     public void print(){  
4         System.out.println("Shirt description");  
5     }  
6 }
```

Implicitly abstract

Implements the print() method.



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

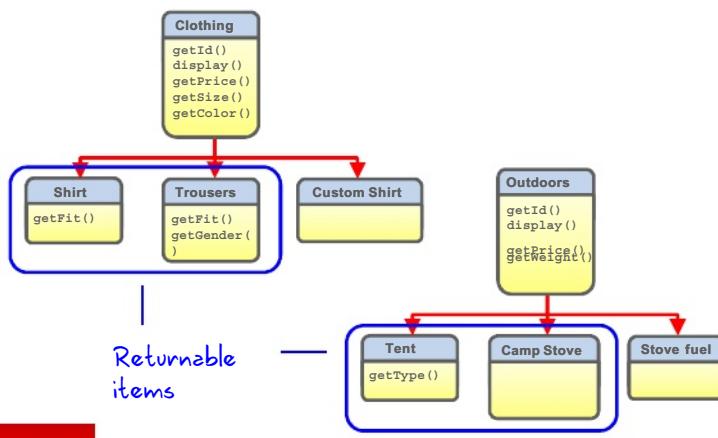
When a class implements an interface, it enters into a contract with the interface to implement all of its abstract methods. Therefore, using an interface lets you enforce a particular public interface (set of public methods).

- In first example in the slide, 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 more depth in the *Java SE8 New Features* course.

Multiple Hierarchies with Overlapping Requirements



ORACLE®

Copyright © 2016, 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.

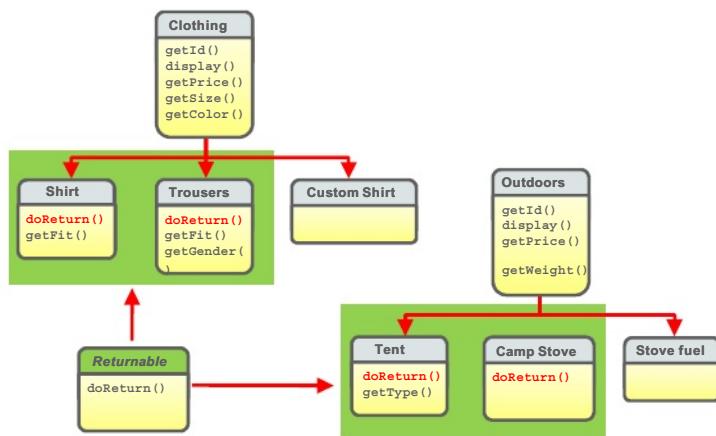
How can this be modeled? Here are some things to consider:

- 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 University Student Learning Subscription Use Only



ORACLE®

Copyright © 2016, 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();  
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() {  
08         // See notes below  
09         return "Suit returns must be within 3 days";  
10     }  
11     ...< other methods not shown > ...  
 } // end of class
```

Shirt implements the method declared in Returnable.



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

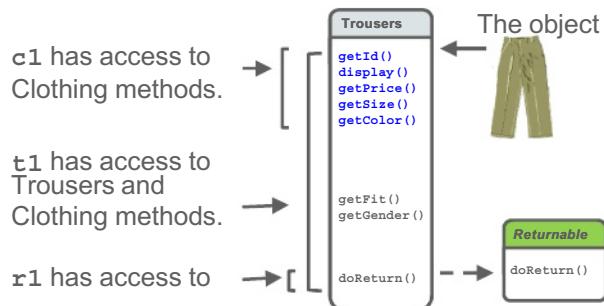
The code in this example shows the `Returnable` interface and the `Shirt` class. Notice that the abstract methods in the `Returnable` class are stub methods (that is, they contain only the method signature).

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

Access to Object Methods from Interface



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



ORACLE®

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

Casting an Interface Reference



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

- The Returnable interface does not know about Trousers methods:

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

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

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

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

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

ORACLE®

Copyright © 2016, 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.

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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`)

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 depth in the *Java SE 8 Programming* course.

ArrayList Example

The screenshot shows the Java API documentation for the `ArrayList<E>` class. It highlights several key points:

- Inheritance:** `ArrayList` extends `AbstractList`, which in turn extends `AbstractCollection`.
- Implementation:** `ArrayList` implements a number of interfaces.
- Interfaces Implemented:** `Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`
- Subclasses:** `AttributeList, RoleList, RoleUnresolvedList`
- Description:** The `List` interface is principally what is used when working with `ArrayList`.
- Implementation Details:** 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 © 2016, 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.

The `ArrayList` class extends the `AbstractList` class, which itself extends `AbstractCollection`.

`AbstractList` implements the `List` interface, which means that `ArrayList` also implements the `List` interface. An `ArrayList` object reference has access to the methods of all of the interfaces and the superclasses shown in the slide.

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;
```



Copyright © 2016, 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 in the slide.

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

Example: Arrays.asList



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

- Converting an array to a List:

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

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

`mylist.replaceAll()`

This works! `replaceAll` comes from `List`.

`mylist.removeIf()`

Error! `removeIf` comes from `Collection` (superclass of `ArrayList`).



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

As you saw in 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 in the slide 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 © 2016, 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 in 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 in the previous slide.

Practice 14-1: Converting an Array to an ArrayList, Part 1



1. Open the project **Practice_14-1** or create your own project with a **Java Main Class** named TestClass.
2. Convert the days array to an ArrayList.
 - Use Arrays.asList to return a List.
 - Use that List to initialize a new ArrayList.
 - Preferably do this all on one line.
3. Iterate through the ArrayList, testing to see if an element is "sunday".
 - If it is a "sunday" element, print it out, converting it to uppercase.

Use String class methods:

- public Boolean equals (Object o);
- public void toUpperCase();
- Otherwise, print the day anyway, but not in uppercase.



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

Oracle University Student Learning Subscription Use Only

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

Practice 14-1: Converting an Array to an ArrayList, Part 2



4. After the `for` loop, print out the `ArrayList`.
 - While within the loop, was "sunday" printed in uppercase?
 - Was the "sunday" array element converted to uppercase?
 - Your instructor will explain what is going on in the next topic.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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

Topics



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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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);
```

Returns a new
String to print

Output:

NED, FRED, JESSIE, ALICE, RICK,

After for loop: [Ned, Fred, Jessie, Alice, Rick]

The list
elements are
unchanged.

ORACLE®

Copyright © 2016, 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 code in the slide 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 in the slide) 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 © 2016, 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();});
```

Diagram labels for the long version:

- Declare input parameter: Points to the parameter `s`.
- Arrow token: Points to the arrow token `->`.
- Method body: Points to the code block `{return s.toUpperCase();}`.

Short version:

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



Copyright © 2016, 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. In fact, a lambda has everything a method has. 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 © 2016, 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<E>` type expression.

Method Summary	
All Methods	Instance Methods
Modifier and Type	Method and Description
default void	<code>replaceAll(UnaryOperator<E> operator)</code> 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 © 2016, Oracle and/or its affiliates. All rights reserved.

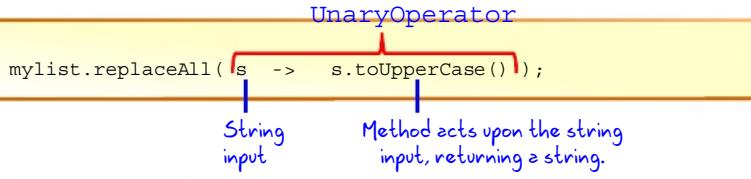
Oracle University Student Learning Subscription Use Only

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:



Copyright © 2016, 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 © 2016, 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 five characters. The long version of this Predicate expression would look like this:

```
mylist.removeIf ( (String s) -> {return (s.length() < 5); } )
```

Practice 14-2: Using a Predicate Lambda Expression



1. Open the project **Practice_14-2**.

In the ShoppingCart class:

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



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

Oracle University Student Learning Subscription Use Only

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

Summary



In this lesson, you should have learned how:

- 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 © 2016, 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 14-3: Overriding and Interfaces



In this practice, you are asked to enhance the soccer league application by finishing the implementation of features mentioned in previous practices. It's up to you to figure out the implementation details.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Oracle University Student Learning Subscription Use Only

Lesson 15: Handling Exceptions

ORACLE®

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

Oracle University Student Learning Subscription Use Only

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 University Student Learning Subscription Use Only

ORACLE®

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

Topics



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



ORACLE®

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

Oracle University Student Learning Subscription Use Only

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.

ORACLE®

Copyright © 2016, 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 © 2016, 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 © 2016, 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 such as 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 }
21 }
```

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 © 2016, 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 © 2016, 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. It can be said that 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 }
}
```

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
```

ORACLE®

Copyright © 2016, 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)
```

## Topics



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

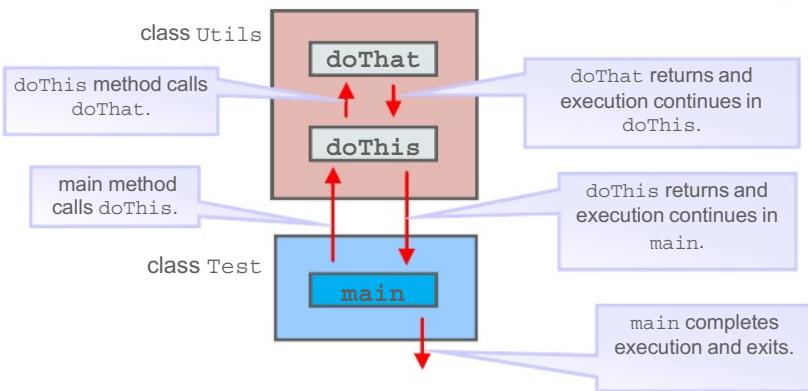
ORACLE®

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

Oracle University Student Learning Subscription Use Only

**Java Fundamentals 15 - 10**

## Normal Program Execution: The Call Stack



ORACLE®

Copyright © 2016, 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 © 2016, 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 `System.out.err` (possibly the console) as you saw previously.

## Topics



Oracle University Student Learning Subscription Use Only

ORACLE®

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

**Java Fundamentals 15 - 13**

## Working with Exceptions in NetBeans



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

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

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

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

unreported exception java.lang.Exception;  
must be caught or declared to be thrown  
--  
Alt+Enter shows hints

ORACLE®

Copyright © 2016, 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.

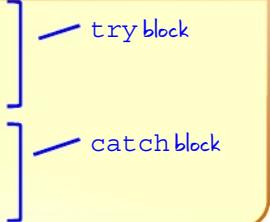
In these early examples, the `Exception` superclass is used for simplicity. However, as you will see later, you should not throw so general an exception. Where possible, when you catch an exception, you should try to catch a specific exception.

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

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

- **Option 1: Catch the exception.**

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

- **Option 2: Declare the method to throw the exception:** In this case, the method declaration includes "throws `Exception`" (or it could be a specific exception, such as `ClassCastException`).

## Program Flow when an Exception Is Caught



main method:

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

3

Output

Utils class methods:

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

2

1

Copyright © 2016, 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:

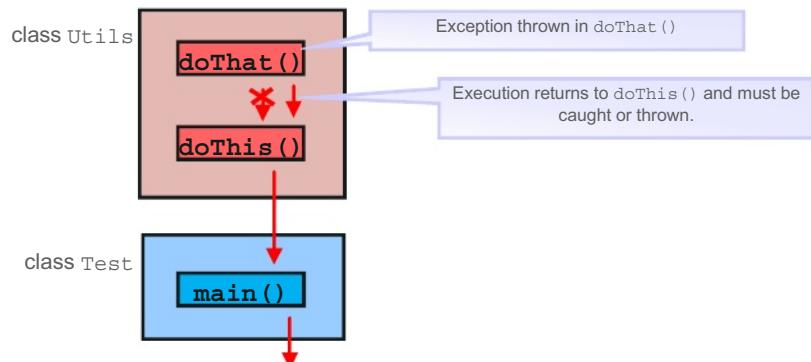
- In line 14, doThat explicitly throws a newException object.

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

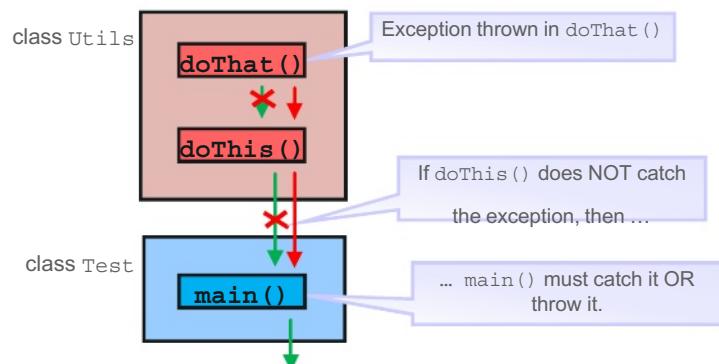
As mentioned previously, when a method finishes executing, the *normal* flow (on completion of the method or on a return statement) goes back to the calling method and continues execution at the next line of the calling method.

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 University Student Learning Subscription Use Only

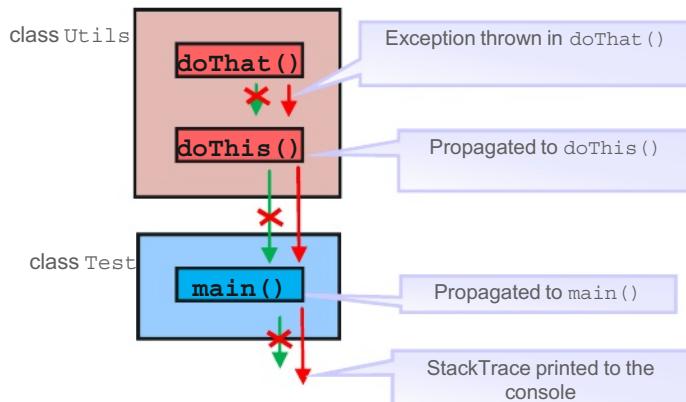


ORACLE®

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

The diagram in the slide illustrates an exception 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 © 2016, 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 thrown all the way up the call stack. But what does it mean to throw an exception from the `main` method? 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 an IDE interface with a window titled "Output - ClassExercises (run)". Inside the window, a red box highlights a stack trace:

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

In the example, you can see what happens when the exception is thrown 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 thrown up the call stack without the `throw` 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 subclasses:
  - `Error`
    - Automatically thrown to the calling method if created
  - `Exception`
    - Must be explicitly thrown to the calling method
    - OR
    - Caught using a `try/catch` block
- `RuntimeException` is a subclass of `Exception`:
  - Is automatically thrown to the calling method

Must be  
explicitly  
handled



Copyright © 2016, 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 actually 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 `Exception`.

## Practice 15-1: Catching an Exception



1. Open the project **Practice\_15-1**.

In the Calculator class:

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

In the TestClass class:

3. Surround the code that calls the divide method with a try/catch block.  
Handle the exception object by printing it to the console.
4. Run the TestClass to view the outcome.



**ORACLE®**

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

Oracle University Student Learning Subscription Use Only



## Exceptions in the Java API Documentation

**ORACLE®**

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

Method Summary

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

createNewFile

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

Nonrecursively 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.checkCreate("java.io.createFile") method denies write access to the file
```

Since:

1.2

Click to get the detail of `createNewFile`.

Note the exceptions that can be thrown.

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

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

## Calling a Method That Throws an Exception



The diagram shows two code snippets from NetBeans. The first snippet shows a constructor call to `new File("testfile.txt")`. A callout bubble states: "Constructor causes no compilation problems." The second snippet shows a call to `testFile.createNewFile()`. A callout bubble states: "createNewFile can throw a checked exception, so the method must throw or catch." The code is as follows:

```
53 public void testCheckedException(){
54 File testFile = new File("//testfile.txt");
55
56 System.out.println("testFile exists: "+ testFile.exists());
57 testFile.delete();
58 System.out.println("testFile exists: "+ testFile.exists());
59 }
60
61 }
```

```
53 public void testCheckedException() {
54 File testFile = new File("//testfile.txt");
55
56 System.out.println("testFile exists: "+ testFile.exists());
57 testFile.delete();
58 System.out.println("testFile exists: "+ testFile.exists());
59
60 testFile.createNewFile();
61 }
```



Copyright © 2016, 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 © 2016, 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 University Student Learning Subscription Use Only

ORACLE®

Copyright © 2016, 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) {
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 }
```

*Catching superclass?*

*No processing of exception class?*

ORACLE®

Copyright © 2016, 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) {
05 System.out.println(e);
06 //<other actions>
07 }
08 }
09 public static void createFile(String fname)
10 {
11 File f = new File(fname);
12 System.out.println(name+" exists? "+f.exists());
13 f.createNewFile();
14 System.out.println(name+" exists? "+f.exists());
15 int[] intArray = new int[5];
16 intArray[5] = 27;
17 }
```

What is the object type?  
toString() is called on this object.

ORACLE®

Copyright © 2016, 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

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## 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;
10 }
```

Directory must be writable:  
IOException

Arg must be greater than 3  
characters:  
IllegalArgumentException

Array index must be valid:  
ArrayIndexOutOfBoundsException



Copyright © 2016, 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 }
```

ORACLE®

Copyright © 2016, 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 © 2016, 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 }
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Practice 15-2: Adding Exception Handling



In this practice, you are asked to enhance the soccer league application to anticipate a scenario where there aren't enough players to fully field every team. You must create a custom exception. When an exception occurs, it should be thrown all the way up to the main method. It's up to you to figure out the implementation details.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Oracle University Student Learning Subscription Use Only



## Lesson 16: Introduction to HTTP, REST, and JSON

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Objectives



After completing this lesson, you should be able to:

- Define HTTP and HTML
- Describe the steps in an HTTP transaction
- List key HTTP headers in request and response
- List the HTTP methods
- Define the structure of a URL and a URI
- Describe a REST Web Service
- Describe the characteristics of JSON



Oracle University Student Learning Subscription Use Only

ORACLE®

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

## Topics



- The Internet and World Wide Web
- HTTP (HyperText Transfer Protocol)
- REST Introduction
- REST in Action
- Testing REST



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## The Internet and World Wide Web



Oracle University Student Learning Subscription Use Only

- Internet
  - ARPANET
  - First two nodes connected October 29, 1969
- World Wide Web
  - Tim Berners-Lee released proposal November 1990
    - HTTP and HTML
  - Netscape released first web browser November 1994
  - Google incorporates 1998
  - First Facebook site is launched Feb 2004
  - iPhone unveiled Jan 2007
- Internet, web, and mobile phones are fundamental to life.
  - But how does it all work?

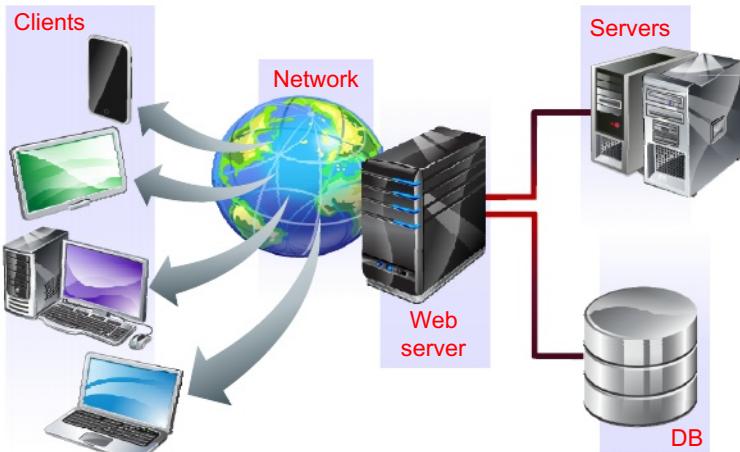


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

- First two nodes of what would become the ARPANET were interconnected between UCLA (Los Angeles) and SRI International in Menlo Park, CA on 29 October 1969.
- Tim Berners-Lee published a formal proposal for HTML and HTTP in November 1990. He proposed building a “Hypertext project,” which he called “WorldWideWeb” as a “web” of “hypertext documents” to be viewed by “browsers” using a client/server architecture.
  - HTTP: HyperText Transfer Protocol
  - HTML: HyperText Markup Language
- Netscape released the first commercial version of its browser “Mosaic Netscape 0.9” on October 13, 1994.
- Mark Zuckerberg launched his first version of “thefacebook” on February 4, 2004.
- iPhone unveiled by Steve Jobs at MacWorld January 9, 2007.

**Note:** All information referenced from Wikipedia

## HTTP Web Architecture



ORACLE®

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

On the web, clients connect to servers by using a network, commonly the Internet. Servers can be connected with one another to request information to complete requests from clients.

Oracle University Student Learning Subscription Use Only

## Web Servers



Web applications are usually stored in web servers. They:

- Handle web requests
- Store application files
- Provide access to resources
- Typically return HTML



**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

## How Web Servers Work



ORACLE®

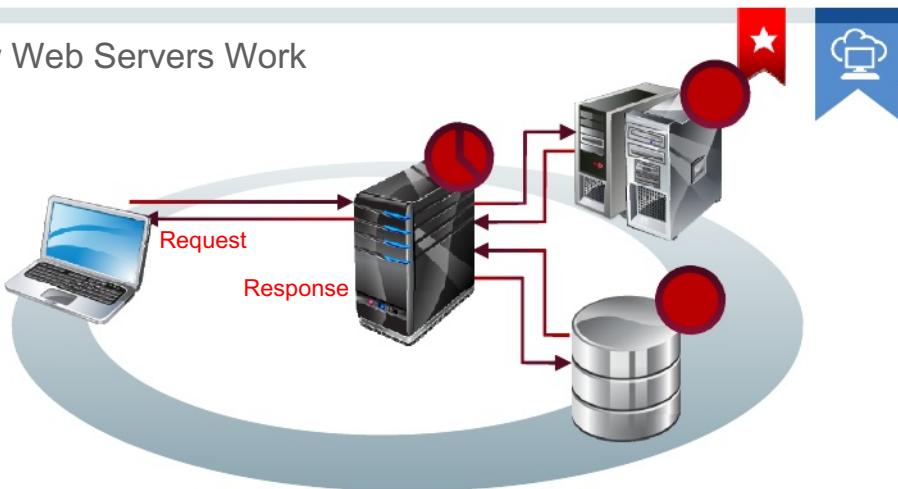
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Web servers work in three stages:

1. A request is always initiated by the client.
2. The server locates the resource and constructs a response.
3. The response with the content of the resource is sent back to the client. Typically, the content is

Note that these three steps may be repeated multiple times for a given HTML page. Not only is the page text loading into the browser, but also things such as images and media items may be required for the page.

## How Web Servers Work



ORACLE®

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

Oracle University Student Learning Subscription Use Only

When external services are added, the process remains very similar.

1. A client makes a request.
2. The server tries to locate the resource and finds that it requires additional services to process the request.
3. For example, a database service or another web service may provide the required information that the server needs to respond to the client.
4. The web server constructs the response and sends it back to the client.

## What Is a Client?



A client is an application that runs on a machine that can make requests from a web server and HTML response data.

- Web browsers
  - Firefox, Chrome, Internet Explorer, and Safari
- Other applications
  - Weather app on mobiles
  - News readers



**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

In this course, the target client applications are web browsers.

## HTML Files



Hypertext Markup Language (HTML):

- It is a language used to define documents.
  - Structure, layout, and content
- Documents contain hyperlinks.
  - To navigate to other documents
  - To include resources

In a web application, HTML:

- Defines the page layout
- Links to other pages
- References other files
- Contains forms



ORACLE®

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

HTML files are the core of the web applications that are run in a browser.

They define the layout of a page, and most importantly, define the additional resources that need to be loaded on the page, such as style sheets, images, and scripts.

## HTML File Structure



```
<!DOCTYPE html>
<html>
 <head>
 <title>TODO supply a title</title>
 <meta charset="UTF-8">
 </head>
 <body>
 <h1>Title</h1>
 <p>Paragraph</p>
 Some text

 Text in another line.
 </body>
</html>
```

HTML5 files have a `<!DOCTYPE html>` directive.

The whole document is enclosed in an `<html>` tag.

An HTML document has two sections:

- The head: Contains information about the document
- The body: Contains the content of the document

ORACLE®

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

An HTML document begins with a `<!DOCTYPE html>` declaration, followed by opening the `<html>` root element. The entire document is contained within the `<html>` element.

An HTML document contains two mandatory sections:

- `<head>` contains information about the document. Everything in the head section is loaded even before the document renders. It can contain scripts, style sheets, meta information, and the title of the document.
- `<body>` contains the actual document. The elements inside the body are parsed and loaded sequentially.

## Topics



- The Internet and World Wide Web
- HTTP (HyperText Transfer Protocol)
- REST Introduction
- REST in Action
- Testing REST



ORACLE®

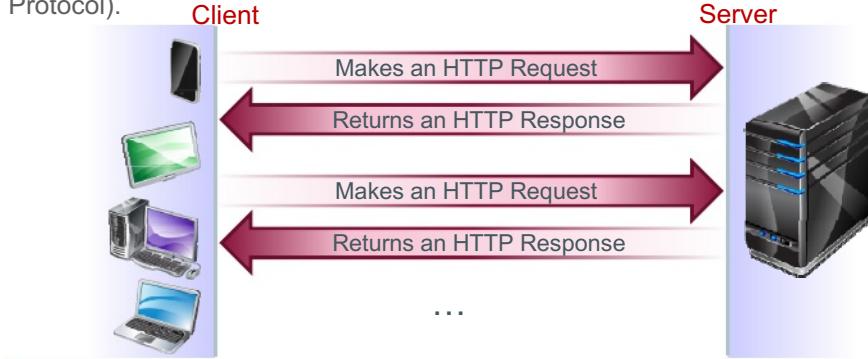
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle University Student Learning Subscription Use Only

## HTTP Protocol



Clients communicate with the server by using HTTP (HyperText Transfer Protocol).



Oracle University Student Learning Subscription Use Only

- HTTP is a communication protocol, which relies on TCP connections over IP to connect the client and the server.
- In the HTTP protocol, all requests are independent.
- The client might trigger additional requests but each one of these requests is completely new to the server, and is handled independent of previous requests.
- There is no session management as part of the HTTP protocol.
- HTTP is stateless.
- Session management is accomplished by sending an identifier as part of the request that can relate to a specific session.

## HTTP Request



### Request Line

- Method
- URI
- Version

Essential request information

### Headers

- Name–Value pairs

Request options and additional client information

### Message Body

- Additional data

Extra information (Optional)

ORACLE®

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

A request contains some information that helps the server generate a response.

Every request contains a Request Line. This contains the Uniform Resource Identifier (URI), which is the name of the requested resource, and the method used. HTTP defines the following methods: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, and CONNECT. Web browsers use only GET or POST.

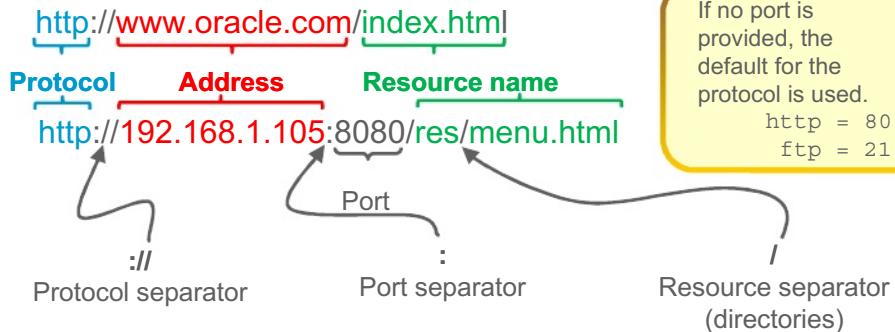
The HTTP request headers are key–value pairs that are used to provide additional information.

- accept-encoding: Defines what encoding is expected in the response
- user-agent: Specifies the client application being used. When you are using a browser, the name and version of the browser are supplied in this header.
- Session IDs and session information, such as cookies, can be contained in request headers.

For more information about HTTP headers, refer to <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>.

The Message Body can contain additional information supplied by the user, such as form data when using the POST method.

## HTTP Request: URL



Oracle University Student Learning Subscription Use Only

ORACLE®

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

The slide shows the different components that compose a URL.

Remember that if no port is specified, the default is used. For HTTP, the default port is 80; for FTP, it is 21.

## HTTP Response



### Status Line

- Version
- Status Code
- Reason phrase

Status of the response

### Headers

- Name–Value pairs

Additional response information

### Message Body

- Response body

Response content



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

The HTTP response must contain the status line.

Status codes are grouped in the following way:

- **1xx:** Informational, for example, 101 Switching Protocols
- **2xx:** Success, for example, 200 OK
- **3xx:** Redirection. Further action is required to complete the request, for example, 301 Moved Permanently.
- **4xx:** Client Error. The request contains a problem and could not be fulfilled, for example, 404 Not Found.
- **5xx:** Server Error. There was a problem in the server, for example, 500 Server Error.

You can find the list of response codes at <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

Response headers contain additional information about the response, such as the content type, content size, user cookies, session ID, and so on.

Between the headers and the body, there needs to be an empty line.

The body contains the actual content of the requested resource. The response content is optional.

## Response Bodies



A response body contains the content of a resource, including:

- Documents
- Images
- Audio
- Video
- JavaScript files

The client (web browser) usually knows how to handle or display the contents of the response.

**The JavaScript code in web applications is run by the web browser on the client machine.**

**ORACLE®**

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

The response body contains the content of the requested resource.

The web browser displays the response content depending on the content-type that the server returns. Sometimes, browsers can infer from the content based on its actual structure or how it is being used in an application.

Note that the JavaScript code is returned to the client web browser, and then executed on the client machine. Some machines and browser configurations might block JavaScript code from being executed.

## An HTTP Response



```
GET http://localhost/test.html
-- response -
200 OK
Date: Tue, 02 Feb 2016 18:21:20 GMT
Server: Apache/2.4.16 (Unix) PHP/5.5.30
Last-Modified: Tue, 02 Feb 2016 18:13:47 GMT
Etag: "115-52acd766becc0"
Accept-Ranges: bytes
Content-Length: 277
Content-Type: text/html

<!DOCTYPE html>
<html>
<head>
<title>TODO supply a title</title>
...
</html>

@Path("message")
public class Message {
 private static String message =
 "Today's word is JAVA!";
 @GET
 public String getMessage() {
 return message;
 }
}
```

Response code

Content-Type

ORACLE®

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

Oracle University Student Learning Subscription Use Only

An HTTP response includes header information along with the body.

- The body includes HTML text.
- Notice that a response code of 200 is returned which indicates the HTTP transaction returned OK.
- The Content-Type header indicates what is included in the HTTP response. In this case, it is HTML text.

## Practice 16-1: Examining HTTP Headers



In this practice, connect to some popular websites and examine the HTTP headers.



**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

- Use a Curl to examine HTTP headers and data from popular websites.

## Topics



- The Internet and World Wide Web
- HTTP (HyperText Transfer Protocol)
- REST Introduction
- REST in Action
- Testing REST



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Representational State Transfer



Oracle University Student Learning Subscription Use Only

REST is the architecture that the entire web uses. It is based on the following design principles:

- Client/server interactions
- Uniform interface
- Layered system
- Cache
- Stateless
- Code-on-demand



ORACLE®

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

REST is a term that has existed for a long time. Roy Fielding describes REST in his doctoral dissertation at UC Irvine, and defines the web architecture for servicing multiple clients in a flexible, scalable, and uniform way.

REST takes full advantage of the HTTP protocol. The complete HTTP protocol complies entirely with the REST proposed architecture:

- Multiple and diverse clients exist in the form of web browsers and applications.
- Uniform Resource Locators and HTTP methods are present to access and modify resources.
- It can be a system that may contain many layers in between the client and the server.
- Caches can be implemented on resources because often the same operations performed over the same resources produce the same result.
- Stateless operations establish no sequence or dependency in REST calls.
- Finally, servers may provide code that needs to be executed. Most often, instead of providing code, the server provides further possible operations in the form of HyperMedia in such a way that clients know what REST operations are possible.

## REST on the Web



You are already using REST.

- Client/server interactions: Browser – Web Server
- Uniform interface: HTTP, URL
- Layered System: IP: Proxies, Gateways, and so on
- Cache: Web browser cache, intermediary caches
- Stateless: HTTP request-and-response model
- Code-on-demand: JavaScript code



ORACLE®

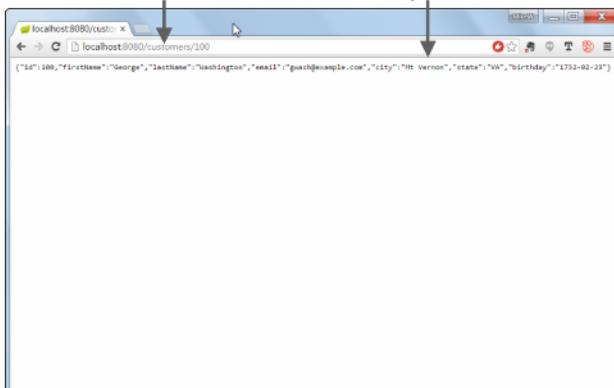
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Oracle University Student Learning Subscription Use Only

## Practical REST

Address: Uniform Resource Identifier

Pressing Enter executes a “GET” request on the resource. A JSON response is returned.



ORACLE®

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

The example makes a request from a sample customer REST web service. A REST GET call is made when an address is entered into the browser (<http://localhost:8080/customers/100>). A JSON text response is returned for the user with an ID of 100.

The browser makes an HTTPGET request to get the resource from a server that is specified by the Uniform Resource Identifier provided in the address bar. The JSON response is returned in the Response text area in the form.

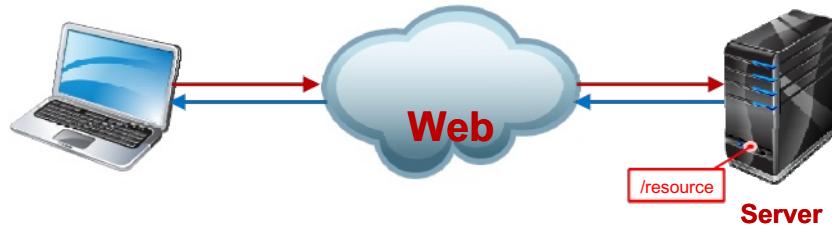
### Uniform Resource Identifier or Uniform Resource Locator

- A URI is composed of the name of the resource but does not specify the way to access such resource.
- A URL has the protocol or way to get a particular resource.

In the end, all URLs are also URI but the term URI is more widely accepted when referring to addresses because sometimes the protocol is omitted, for example:

- <http://www.oracle.com/education> is a valid URL and URI.
- <www.oracle.com/education> is a valid URI only.

Thus, using URI for both is correct and more “universal.”

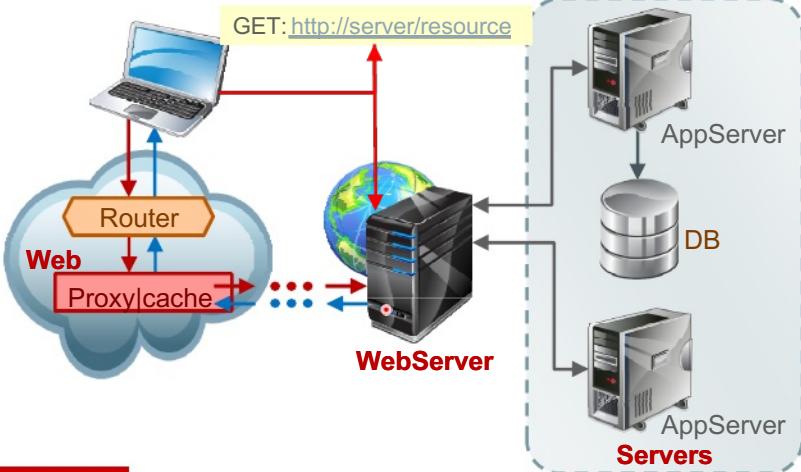
GET : <http://server/resource>

ORACLE®

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

The request is sent through the network to the server, which locates the resources and applies the method. In the example in the slide, the method is a GET request. Thus, the server will create a representation of the resource and send it back to the client.

## Practical Rest



ORACLE®

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

In a real-world scenario, multiple layers are involved.

- The client communicates with the web by using routers and other devices.
- There can be proxies that will process the request and redirect it accordingly.
- Whether a usually static response has changed in the server and if it is not the case, a cached version is sent back. This is possible because of the stateless nature of HTTP.
- The web server may delegate processing to Application Servers to get the information required to construct a resource or to modify data in a database.

Note that sometimes the WebServer is a module of a bigger Application Server.

## Topics



- The Internet and World Wide Web
- HTTP (HyperText Transfer Protocol)
- REST Introduction
- REST in Action
- Testing REST



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Exchanging Data with REST



To create REST applications, you need to define a way to exchange information between clients and servers.

- HTML provides no easy way to identify data types.
- Typically, the choice is between two text formats.

### Extensible Markup Language (XML)

- A text file format that marks up text using tags
- Designed to be both human-readable and machine-readable

### JavaScript Object Notation (JSON)

- A text file format that stores data in attribute value pairs
- Designed to be both human-readable and machine-readable



ORACLE®

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

Oracle University Student Learning Subscription Use Only

### Extensible Markup Language (XML)

A text file format that marks up text using tags. For example:

```
<root>
 <name>John Doe</name>
 <email>jdoe@example.com</email>
</root>
```

## XML Versus JSON in Web Services



Both formats have advantages and disadvantages.

- **Extensible Markup Language (XML)**
  - A large number of processing libraries exist to support XML in almost every language (SAX, DOM, StAX, and JAXB).
  - Representation of data is verbose.
  - It may require additional system resources to parse and process.
- **JavaScript Object Notation (JSON)**
  - A subset of JavaScript
  - Less verbose than XML
  - More lightweight to transport and process
  - Support evolving in other languages



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

Oracle University Student Learning Subscription Use Only

JSON has become a much more popular option given its compactness and ease to process.



- JavaScript Object Notation is a simple format to represent objects, arrays, and values as strings.
- JavaScript includes functions to convert objects, arrays, and values to JSON strings and vice versa.



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

The JavaScript Object Notation (JSON) is a format to represent JavaScript objects, arrays, and values as strings. It is one of the preferred ways of sending and receiving data using REST web services.

JSON allows you to represent objects as key–pair values and arrays as sequential lists of items.

Values can also be represented as JSON, allowing you to create all sorts of objects by using strings, numbers, and Booleans.

JSON is often compared to XML because both can be used to represent structured data. The main difference between JSON and XML is that JSON contains only data without any schema information. Therefore, JSON has no direct validation mechanism.

Also JSON contains only string, number, and Boolean value types and Object and Array data structures. You do not need to define custom data types or nodes as in XML. In JSON, data is represented as it would be inside a JavaScript object, making it extremely flexible at the exchange of validation and schema facilities.

## JSON Strings



Arrays:

```
[1, 2, 3]
["a", "b", "c"]
```

Objects:

```
{"name": "john", "age": 31}
{"itemId": ["a", "b"], "total": 2, "active": true}
```

• JSON is very similar to declaring JavaScript literal values.

- JSON does not have a representation for functions.
- Only values are allowed in JSON.

ORACLE®

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

JSON values are represented in the following manner:

- **Numbers:** Use the number literal.
- **Strings:** Enclose the string literal inside double quotation marks.
- 

• **Booleans:** These are either true or false.

JSON structures are represented in the following manner:

- **Arrays:** Must be enclosed in brackets “[]” and elements must be comma-separated. They can contain arrays, objects, numbers, strings, or Boolean as elements.
- **Objects:** Must be enclosed in braces “{}.” Properties are comma-separated key–value pairs. The key for the property must be a string; therefore, it must be enclosed in double quotation marks. Values for properties can be arrays, objects, numbers, strings, or Boolean.

There is no way to represent functions in JSON because it is used to represent only data.

## JSON Example



- Customer example shows JSON data.
  - Note that the ID is shown as a number.
  - Notice pairs of fieldname/value.
  - All other data pairs are strings.

```
{ "id":100,"firstName":"George","lastName":"Washington", "email": "George.Washington@example.com", "city": "New Vernon", "state": "VA", "birthday": "1732-02-23"}
```



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

## REST in Action



To use REST in web applications, you need to address the following:

- The HTTP method that you need
- The resource identified by its URI
- The content of the request and parsing of the response accordingly

The preferred format for the data sent from and to servers is JSON.

ORACLE®

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

Oracle University Student Learning Subscription Use Only

To know how to use a particular REST service, you need to know what resources you need to access, what methods you can ask for, and what data is going to be exchanged.

The preferred format for the data sent from and to servers is JSON, although the specific format is service dependent.

## HTTP Methods



### GET

#### Get a resource

Used to get a resource from the server.  
For example, Get user list, or get the details of a user.

### POST

#### Add a resource

Used to create a new resource in the server.  
For example, add a new user.

### PUT

#### Update a resource

Used to update a resource in the server.  
For example, update the user details.

### DELETE

#### Delete a resource

Used to delete a resource in the server.  
For example, delete or disable a user.

**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

- The GET method is used to obtain the resource.
- PUT is used to update the resource.
- POST is used to add a new resource.
- DELETE

HTTP defines the OPTIONS, HEAD, TRACE, and CONNECT methods as well but their usage in RESTful APIs is not as wide.

Sometimes, you will find the OPTIONS method returning a representation of the possible uses of the other methods as well as the URI compositions available. This is not standard and should not be relied upon.

## Topics



- The Internet and World Wide Web
- HTTP (HyperText Transfer Protocol)
- REST Introduction
- REST in Action
- Testing REST



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Testing with cURL



cURL is a command-line tool for downloading files from a URL and supports a number of internet protocols.

- Pronounced k-er-l, originally stood for “see URL”
- Makes an excellent tool for testing REST operations
- Very scriptable
- Some of the protocols supported include HTTP, HTTPS, FTP, FTPS, SCP, SFTP, LDAP, TELNET, IMAP, POP3, and SMTP
- Free software with an MIT License

Oracle University Student Learning Subscription Use Only

ORACLE®

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

## cURL Request and Sample Response Headers



Example command:

```
curl -X GET -I https://www.google.com
```

Sample Response Headers:

```
HTTP/1.1 200 OK
Date: Wed, 24 Feb 2016 16:57:59 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See
https://www.google.com/support/accounts/answer/151657?hl=en for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alt-Svc: quic="www.google.com:443"; ma=2592000; v="30,29,28,27,26,25",quic=:443";
ma=2592000; v="30,29,28,27,26,25"
```

**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

cURL can return all of the data from an HTTP request or just the HTTP headers. This example shows a headers-only response. All HTTP methods can be used in a request.

## cURL Switch Options



Here are some commonly used command-line options.

Switch	Description
-X	Specifies the HTTP method to be used in the HTTP request
-i	Includes the HTTP header in the output. This downloads the header along with any other data that is normally returned with the request.
-I	Returns the HTTP header only. In this case, an HTML page will not be included in the result.
-H	Adds an HTTP header to the request. You can specify as many of this option as you need.
-d	Specifies data to upload for methods such as POST or PUT



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

## Testing with Postman



Postman is a Chrome web browser extension for creating and testing HTTP requests.

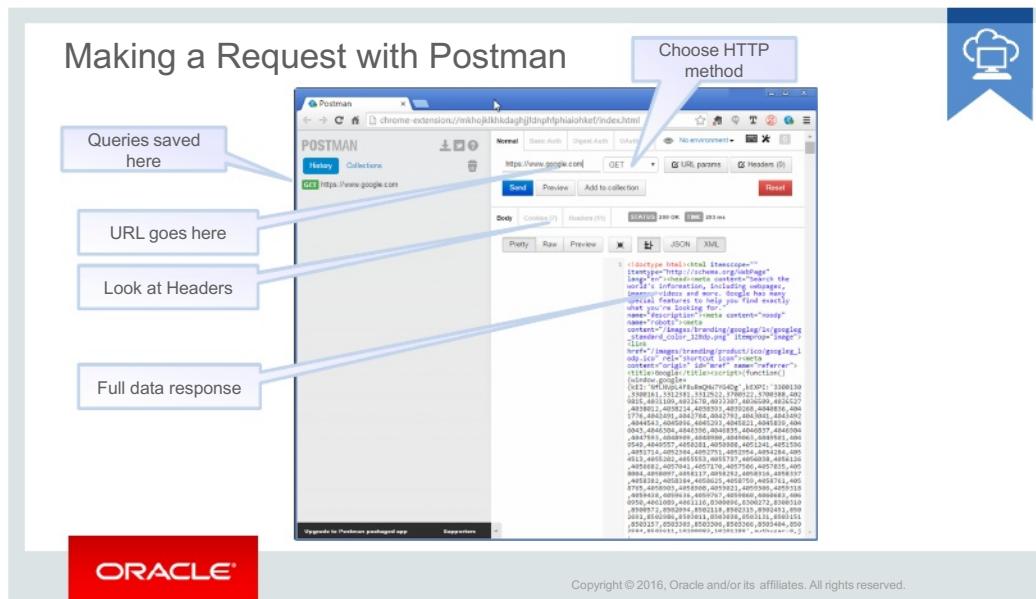
- Specify HTTP method to use
- Can create POST data or parameters
- Request history saves requests for reuse
- Features include:
  - Create requests quickly
  - Replay and organize
  - Switch context quickly
  - Built-in authentication helpers
  - Customize with scripts
  - Robust testing framework
  - Automate collections



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

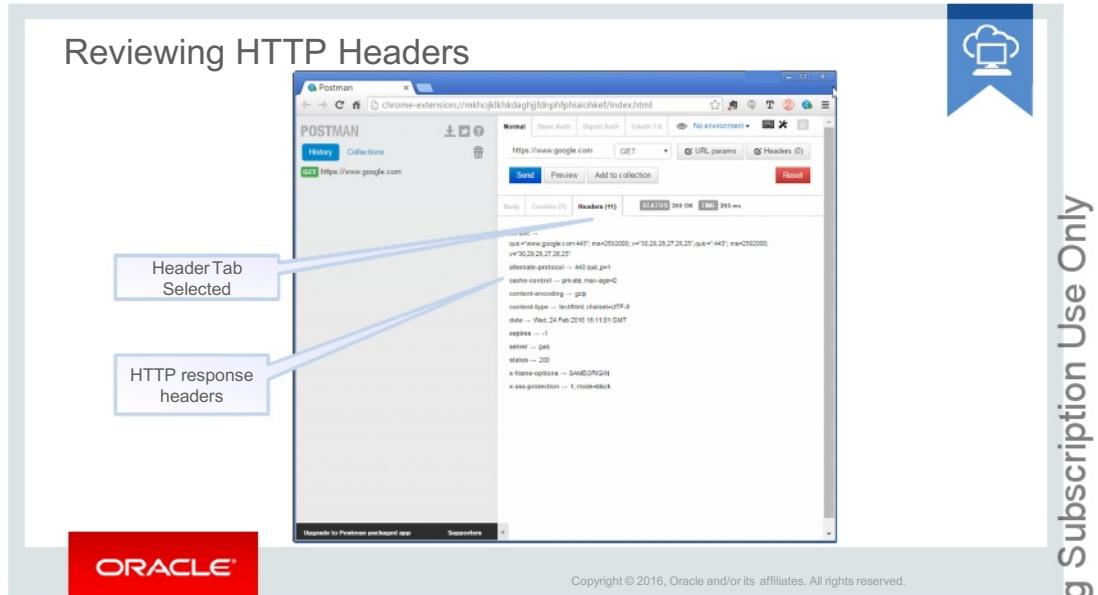
Oracle University Student Learning Subscription Use Only

Postman is a Chrome browser extension. It provides a powerful user interface for testing HTTP requests.



The picture shows the response from making a GET request to <https://www.google.com>.

## Reviewing HTTP Headers



This picture shows the headers returned from the previous request.

Oracle University Student Learning Subscription Use Only

## RESTful URI



URIs should describe access to a resource, for instance.

- GET <http://www.example.com/users/>
  - Lists all the users
- GET <http://www.example.com/users/john>
  - Gives the details of a user identified as “john”
- POST <http://www.example.com/users/>
  - Creates a new user
- PUT <http://www.example.com/users/john>
  - Updates the user identified as “john”



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

In the examples in the slide, you can see the common structure of the RESTful API's URIs.

URIs are usually constructed progressively as you need more details.

For an application that stores places and reviews of such places:

- /nearby/places → Refers to the places that are nearby to the user
- /places → Refers to the list of places registered
- /places/toms\_pizza → Refers to the place called Tom's Pizza
- /places/toms\_pizza/reviews → Refers to the reviews of the place called Tom's Pizza

You can see how methods would apply:

- GET /nearby/places → Gets a list of the nearby places
- POST /places → Adds a new place
- PUT /places/toms\_pizza → Updates the place called Tom's Pizza
- DELETE /places/toms\_pizza → Deletes the place called Tom's Pizza
- DELETE /nearby/places → NOT ALLOWED

## Practice 16-2: Working with Spring Boot and JSON Data



In this practice, use REST tools to retrieve JSON data from a REST micro web service.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Summary



In this lesson, you should have learned how to:

- Define HTTP and HTML
- Describe the steps in an HTTP transaction
- List key HTTP headers in requests and responses
- List the HTTP methods
- Define the structure of a URL and a URI
- Describe a REST Web Service
- Describe the characteristics of JSON



Oracle University Student Learning Subscription Use Only

ORACLE®

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

## Practice 16-3: Using Postman to Read HTTP Headers (Optional)



In this practice, use the Postman Chrome browser plug-in to read the HTTP headers of several common web sites.



**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

## Practice 16-4: Working with Spring Boot and JSON Data (Optional)



In this practice, you use Postman to test a Spring Boot REST application. Instead of returning HTML, a Spring Boot REST application works with JSON data when communicating with an HTTP client.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Practice 16-5: Testing a Rest Application



With your newly acquired REST testing knowledge, perform some REST operations on the sample Spring Boot application.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

A photograph showing four people in an office environment. Three individuals are visible in the foreground, looking at computer monitors. A fourth person is partially visible behind them. The scene is overlaid with a dark blue rectangular area containing white text and a logo.

**Lesson 17: Creating REST Web Services with Java**

 ORACLE

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

Oracle University Student Learning Subscription Use Only

## Objectives



After completing this lesson, you should be able to:

- Design a RESTful web service and its collection of resource URLs
- Create methods that follow the prescribed rules of HTTP method behavior, including idempotence
- Create a Spring Boot resource and application classes
- Consume query and other parameter types
- Identify and return appropriate HTTP status codes



Oracle University Student Learning Subscription Use Only

ORACLE®

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

## Topics



- Designing a Java REST Web Service
- Implementing a Java REST Web Service

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## REST Resources



REST is centered around an abstraction known as a “resource.” Any named piece of information can be a resource.

- A resource is identified by a uniform resource identifier (URI).
- Clients request and submit representations of resources.
  - There can be many different representations available to represent the same resource.
- A representation consists of data and metadata.
  - Metadata often takes the form of HTTP headers.
- Resources are interconnected by hyperlinks.
- A RESTful web service is designed by identifying the resources.



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

## Resource Design



A RESTful web service contains multiple resources.

- Resources are linked together (hypermedia).
- You can design a tree of resources with a single base resource.
- It is comparable to an object graph in Java or a hierarchy of elements in an XML document.
- Resources are (usually) nouns or things.
- Each resource has a limited number of general-purpose operations that it may support (GET, PUT, POST, and DELETE).
- A resource is uniquely identified by a URL:

<http://localhost:8080/customers>

A collection is a resource.

<http://localhost:8080/customers/matt>

A specific resource in a collection



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

Sometimes when designing noun-only resources, you may have a use-case that spans resources or does not map to a standard operation. An action resource can be used in this situation.

- <http://localhost/bank/accounts/matt>
- <http://localhost/bank/accounts/tom>

Matt wants to give money to Tom. With two separate resources, you cannot atomically transfer money.

- <http://localhost/bank/transfers>

The verb in this use-case is “transfer;” by using the verb as a noun, you can correctly identify the needed resource—a record of account transfers, which can be added to in order to create a new transfer.

## HTTP Methods



HTTP 1.1 methods are outlined by RFC 2616.

Method	Purpose
GET	Read, possibly cached
POST	Update or create without a known ID
PUT	Update or create with a known ID
DELETE	Remove
HEAD	Read headers
OPTIONS	List the "Allow"ed methods



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

In practice, Java-based UI web applications use only the `GET` and `POST` requests. JavaScript-based, client-side UI web applications make AJAX Web Service requests against a Java back end and potentially use the full range of HTTP methods.

The `GET` and `POST` requests have similar capabilities, because they both supply a request that contains a URI, and they both expect some data to be returned in the response body. The differences between `GET` and `POST` requests can best be seen in the way that form data is sent from the browser to the server. For example, if a user submits a login form by using `GET`, the browser might display the form data with the URL, as shown in the following example:

```
GET /FormExample/ReadForm?user=weblogic&password=welcome1 HTTP/1.1
```

The browser issues a `POST` request when it submits a form with a definition in HTML that contains the `method="POST"` attribute as follows:

```
<form action="ReadForm" method="POST">
```

When submitting a `POST` request, all form data is placed in the request body instead of being included in the URL.

RESTful web services that use HTTP should use HTTP as the application-level protocol. This means that HTTP methods are the only methods that clients can call on your URLs or resources. The small number of general-purpose methods is one reason why a RESTful service will have a large number of resources.

## Idempotence



Idempotence: An operation that when applied multiple times results in the same return value

- **Idempotent methods**

- GET: Read only. Never changes the state of the resource
- PUT: An update of a resource has same result if repeated
- DELETE: Resource removed and result does not change with multiple calls

- **Non-idempotent methods**

- POST: Creates something new that changes state



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

Oracle University Student Learning Subscription Use Only

Idempotence is a term that is used often when discussing REST methods.

## Resource Collections



It is very common to represent collections of resources; the collection itself is a resource.

- GET /collection: Return a listing of hyperlinks to the elements in the collection.
- PUT /collection: Replace everything.
- POST /collection: Create one new resource in the collection; return the generated ID of that new resource.
- DELETE /collection/{id}: Delete an item from the collection.
- PUT /collection/{id}: Update an item in the collection or create a new item with this ID.



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

Oracle University Student Learning Subscription Use Only

The example used in this lesson follows the same basic pattern.

## Using HTTP Response Status Codes



REST web services return a response code for each request.

- **2xx – Success:** Action was successfully received, understood, and accepted.
  - 200 OK: Returned for successful GET, PUT, DELETE
  - 201 Created: Returned for successful POST
- **4xx – Client Error:** Request contains bad syntax or cannot be fulfilled.
  - 400 Bad Request: Returned when POST fails (for example, duplicate ID)
  - 404 Not Found: Returned for GET, PUT if resource lookup fails
- **5xx – Server Error:** Server failed to fulfill an apparently valid request.
  - 500 Internal Server Error: General error response



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

RFC 2616 and subsequent updates (RFC 7230 - 7235) provide a number of response codes for HTTP requests. This slide shows only the most commonly used responses for a REST web service.

## Topics



- Designing a Java REST Web Service
- Implementing a Java REST Web Service

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Introduction to Spring Boot



- Is a framework designed to create production-ready stand-alone applications
  - Released under the Apache 2.0 license
- Prefers convention to configuration
- Can embed Tomcat or Jetty with your applications
- Creates stand-alone uber JARs by default
- Provides quick starter templates to make application development fast
  - Includes RESTful templates
- Generates no code and does not rely on XML for configuration
- Built on the Spring Framework
  - Data oriented framework based on POJOs
  - Often seen as an alternative to Java EE

ORACLE®

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

Spring Boot is a popular Java framework that is used to quickly create web applications and RESTful applications by using Java. Given its design, Spring Boot is a good choice for creating cloud-based Java applications.

## Customer Web Service



A simple web service for customer data

- Each customer is represented by the Customer class.
- CopyOnWriteArrayList is used to store customer data.
  - This is a thread safe version of an ArrayList.
- All web service operations take place on this ArrayList.

The following fields are included in the Customer class:

```
public class Customer {
 private final long id;
 private final String firstName;
 private final String lastName;
 private final String email;
 private final String city;
 private final String state;
 private final String birthday;
```



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

Oracle University Student Learning Subscription Use Only

The Customer class and data set is used for all the examples in this lesson. See the examples for this lesson to completely review the source code.

## Implementing Resources in Java



The primary component of a RESTful web service is the resource.

- A resource is a “thing” or noun.
- A resource has a unique URL.
- A Spring Boot REST application is created by annotating a controller class.
- A Spring Boot–annotated controller contains a Java method for each HTTP method supported.
  - A method is implemented for each operation.
  - CRUD operations (Create, Read, Update, Delete) are supported.
- Information is exchanged by using JSON.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

The CRUD operations are usually a minimum requirement for a data-oriented service.

## A Root Resource Controller Class



To begin, you must create a controller class.

- Annotate the class with the `@RestController` annotation:

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

 private final CopyOnWriteArrayList<Customer> cList =
 MockCustomerList.getInstance();

 // Your Code here

}
```

Annotation makes this a REST class.

Collection URL for this service

ArrayList that functions as the data source

ORACLE®

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

This class is now designated as the controller for the requests that are made to the `/customers` URL. `CopyOnWriteArrayList` is a thread safe version of `ArrayList`. Functionally, they are the same.

**Note:** With NetBeans, you can right-click and select **Fix Imports** to add the Spring libraries that are required for your class.

## The Resource URL



With the annotations specified, this class will handle REST calls for this URL:

`http://localhost:8080/customers`

- `/customers` is the root resource name.
- Java methods and HTTP methods will be linked in this class.
- HTTP GET and POST can be associated with this URL.
- Specifying an ID will require additional information.



**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

## Resource Method for GET



A @RequestMapping-annotated method maps a Java method to an HTTP method.

- The getAll method is mapped to the GET HTTP method.
- By default, data is returned in JSON format.

```
@RequestMapping(method = RequestMethod.GET)
public Customer[] getAll() {
 return cList.toArray(new Customer[0]
}
```



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

This method will return the complete customer list in JSON format.

```
[{"id":100,"firstName":"George","lastName":"Washington","email":"gwash@example.com","city":"Boston","state":"MA","birthday":"1732-02-11"}, {"id":101,"firstName":"John","lastName":"Adams","email":"jadams@example.com","city":"Braintree","state":"MA","birthday":"1735-10-30"}, {"id":102,"firstName":"Thomas","lastName":"Jefferson","email":"tjeff@example.com","city":"Charlottesville","state":"VA","birthday":"1743-04-13"}, {"id":103,"firstName":"James","lastName":"Madison","email":"jmad@example.com","city":"Orange","state":"VA","birthday":"1751-03-16"}, {"id":104,"firstName":"James","lastName":"Monroe","email":"jmo@example.com","city":"New York","state":"NY","birthday":"1758-04-28"}]
```

## Adding a Path Variable for a Lookup



To look up a single customer, the `@PathVariable` annotation must be added to the method signature. The `GET` method looks up a single customer by ID.

- The `value` element specifies a pattern to match.
- The `@PathVariable` annotation passes the value from the URL to the method.

```
@RequestMapping(method = RequestMethod.GET, value = "{id}")
public Customer get(@PathVariable long id) {
 // Code that performs lookup
 return match; // match is a Customer
}
```



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

This method will return a single customer. For example, the URL  
`http://localhost:8080/customers/100` will return a single entry for that ID.

```
{"id":100,"firstName":"George","lastName":"Washington","email":"gwash@example.com","city":"Mt Vernon","state":"VA","birthday":"1732-02-23"}
```

## Creating a Spring Boot Rest Application



To create a Spring Boot REST application, you need a sample Maven pom.xml file.

- Get the file from: <https://spring.io/guides/gs/rest-service/>.
- The sample will contain references such as the following.
- See the source files for Exercise 1 for a complete example.

```
<!-- See sample for additional details -->

<dependencies>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
</dependencies>
```



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

Use a sample Maven pom.xml to download the required Spring Boot libraries that you need to build your application.

## Launching Your Spring Boot REST Web Service



A main method is needed to launch a Spring Boot application.

- The Application class is annotated with @SpringBootApplication.
- When the application is run, it will load any @RestController classes.

```
public static void main(String[] args) {
 myProps.setProperty("server.address", "localhost");
 myProps.setProperty("server.port", "8080");

 SpringApplication app = new SpringApplication(Application.class);
 app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 app.run(args);
}
```



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

Note that the server.address and server.port values are set programmatically. You could also set those values in a properties file. However, there is a reason for using this approach, which will be explained in the next lesson titled “Deploying REST Web Services to Oracle Application Container Cloud Service”.

## Practice 17-1: Create a Spring Boot REST Web Service



In this practice, you create a web service by using Spring Boot. Your service should be able to:

- Provide a list of all the customers in an `ArrayList`
- Look up a specific customer in an `ArrayList`



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Specifying Content Types



By default, Spring Boot uses JSON for all REST communication. This can be changed by adding a `produces` element to the annotation.

- The following example returns plain text.
- Notice the method return type must change as well.

```
@RequestMapping(method = RequestMethod.GET, value = "{id}", produces =
 "text/plain")
public String get(@PathVariable long id) {
 // Code that performs lookup
 return match.toString(); // match is a Customer
}
```



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

Oracle University Student Learning Subscription Use Only

Setting the return type to `String` works for the plain text content type.

## Handling Web Service Exceptions and Responses



Spring Boot provides mechanisms for handling exceptions and sending response codes back from a REST server.

- Send responses:
  - They are always sent for each request.
  - They return an appropriate HTTP code.
  - You use the `ResponseEntity` class to send codes and messages.
- Exceptions
  - Can throw exceptions if needed
  - Can catch thrown exceptions by using a method and annotations



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Throwing an Exception



For example, if the ID lookup for a customer fails, an exception could be thrown.

```
@RequestMapping(method = RequestMethod.GET, value = "{id}")
public Customer get(@PathVariable long id) {

 // code for lookup goes here
 if (match != null){
 return match;
 } else {
 throw new NotFoundException(id + " not found");
 }
}
```



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

Oracle University Student Learning Subscription Use Only

The `NotFoundException` was created for this example and extends `RuntimeException`.

## Handling Exceptions



The myError method is an exception handler for NotFoundException.

Define exception handler.

```
@ExceptionHandler(NotFoundException.class)
@ResponseBody

public ResponseEntity<?> myError(Exception exception) {
 new JsonError("ID not found error:",
 exception.getMessage(), HttpStatus.NOT_FOUND);
}
```

The ResponseEntity class  
is returned to the client.

Returns a JsonError object

Sets response code to 404

ORACLE®

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

Oracle University Student Learning Subscription Use Only

The ResponseEntity class uses generics. This syntax "<?>" just means that the class accepts any type. The class does two things: it converts some object into a JSON response and it sets the response code.

## The NotFoundException Class



The NotFoundException class is very basic.

- It is just a class definition with one constructor.
- The class extends RuntimeException.

```
public class NotFoundException extends RuntimeException {

 public NotFoundException(String message) {
 super(message);
 }
}
```

Oracle University Student Learning Subscription Use Only



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

## The JsonError Class



Oracle University Student Learning Subscription Use Only

```
public class JsonError {
 private String type;
 private String message;

 public JsonError(String type, String message) {
 this.type = type;
 this.message = message;
 }

 // get methods for type and message defined here
}
```



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

The class demonstrates how any type of JSON message could be designed and sent back in a response. The variable name and the value held in the variable become the JSON name/value pair that is returned in JSON format. You could define many more fields and provide as much information as you want in a response.

## Using POST to Add a Customer



Oracle University Student Learning Subscription Use Only

The add method takes the JSON data included in the request body and converts it into a `Customer` object.

- Note that `@RequestBody` tells Spring to look for the object in the body of the request.
- The `ResponseEntity` class is used to return a success message.

```
@RequestMapping(method = RequestMethod.POST)
public ResponseEntity add(@RequestBody Customer customer) {
 cList.add(customer);
 return new ResponseEntity<>(null, HttpStatus.CREATED);
}
```

Sets response code to  
201 created



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

The JSON for the `Customer` must be passed in the body of the request. Notice that an empty `ResponseEntity` object is returned. Only the response code needs to be sent. You are not required to send any additional information as in the first example.

The JSON text for customer 105 might look like the following:

```
{"id":105,
"firstName":"Abigail","lastName":"Adams","email":"aadams@example.com","city":"Braintree","state":"MA", "birthday":"1744-11-22"}
```

## Using PUT to Update a Customer



If a match is found, the update method replaces the current Customer data.

```
@RequestMapping(method = RequestMethod.PUT, value = "{id}")
public ResponseEntity update(@PathVariable long id, @RequestBody
Customer customer) {
 // Lookup code goes here
 if (matchIndex > -1){
 cList.set(matchIndex, customer);

 } else{
 return new ResponseEntity<>(null, HttpStatus.OK);
 }
}
```

**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

The class uses both the `@PathVariable` and `@RequestBody` annotations to get the data required for the method. If a match is found, a 200 OK status is returned to the client. If no match is found, a 404 NOT FOUND message is returned. Notice that in contrast to the earlier example, you can send only a response code. You are not required to send JSON data back to the calling application.

## Providing Cross Domain Headers



Oracle University Student Learning Subscription Use Only

Problem: When a web application tries to access REST web services on another domain, the browser will block some HTTP requests.

- Cross-domain details:
  - The web application is loaded on <http://www.example.com>.
  - The application needs to access services on <http://www.oracle.com>.
  - The browser blocks some HTTP requests because this is a security violation of the same srcin policy.
- Solution
  - Add Cross-Origin Resource Sharing (CORS).
  - Web service provides headers, which allow the web application to communicate with the server.



**ORACLE®**

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

The same srcin policy states that a web browser allows a page to load scripts on a second page only if the second page has the same srcin as the first. Origin is defined as a combination of the host name, URI scheme, and port number.

So when a browser loads an application at <http://localhost:8080>, the application would be prevented from accessing <http://example.com:80>. Adding CORS headers to a web service on the server side solves this problem.

## Cross Domain Headers



The HTTP headers that must be provided by the web service on the server side are as follows:

Header	Value
Access-Control-Allow-Origin	*
Access-Control-Allow-Methods	GET, POST, OPTIONS, PUT, PATCH, DELETE
Access-Control-Allow-Headers	X-Requested-With, content-type
Access-Control-Allow-Credentials	True



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

Oracle University Student Learning Subscription Use Only

## Providing Cross Domain Headers with Spring Boot



Oracle University Student Learning Subscription Use Only

Spring Boot provides CORS support by using an annotation.

- The `@CrossOrigin` annotation can be applied to a method or class.
- A sample updated `getAll` method is as follows:

```
@CrossOrigin(srcins = "http://localhost:8080")
@RequestMapping(method = RequestMethod.GET)
public Customer[] getAll() {
 return cList.toArray(new Customer[0]
}
```



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

The example in the slide would allow connections from a web application that is running on `localhost` on port 8080. See the API documentation for `@CrossOrigin` for details on what options are available.

## Additional Resources



Topic	Website
Architectural Styles and the Design of Network-based Software Architectures	<a href="http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm">http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm</a>
REST APIs must be hypertext-driven	<a href="http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven">http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven</a>
Hypertext Transfer Protocol -- HTTP/1.1	<a href="http://www.rfc-editor.org/rfc/rfc2616.txt">http://www.rfc-editor.org/rfc/rfc2616.txt</a>
Spring Boot	<a href="http://projects.spring.io/spring-boot/">http://projects.spring.io/spring-boot/</a>
Spring Boot API	<a href="http://docs.spring.io/spring-boot/docs/1.4.0.BUILD-SNAPSHOT/api/">http://docs.spring.io/spring-boot/docs/1.4.0.BUILD-SNAPSHOT/api/</a>
Web Linking	<a href="http://www.rfc-editor.org/rfc/rfc5988.txt">http://www.rfc-editor.org/rfc/rfc5988.txt</a>



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

Oracle University Student Learning Subscription Use Only

## Practice 17-2: Complete a Spring Boot REST Web Service



In this practice, you complete the web service that you started in the previous exercise. Your service should be able to:

- Provide a list of all the customers in the web service
- Look up a specific customer from the web service
- Add a new customer to the web service
- Update a customer in the web service
- Delete a customer from the web service



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Summary



In this lesson, you should have learned to:

- Create a Java application that uses packages
- Create a Java application by using a JAR file
- Design a RESTful web service and its collection of resource URLs
- Create methods that follow the prescribed rules of HTTP method behavior, including idempotence
- Create a Spring Boot resource and application classes
- Consume query and other parameter types
- Identify and return appropriate HTTP status codes



Oracle University Student Learning Subscription Use Only

ORACLE®

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

## Practice 17-3: Create a Soccer Data Model



In this practice, you create a Player class to represent soccer player stats. You will use an array list to store player classes as your data model in the Practice 17-4.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Practice 17-4: Creating a Spring Boot REST Application



In this practice, create a micro REST web service for the soccer league that will keep track of player statistics.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

**Lesson 18: Deploying REST Web Services to the Cloud**

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

ORACLE®

A blue arrow icon containing a white cloud and a computer monitor is positioned in the top right corner.

Oracle University Student Learning Subscription Use Only

## Objectives



After completing this lesson, you should be able to:

- Describe the key features of Oracle Application Container Cloud Service (OACCS)
- Configure a Java application to use environment variables at startup
- Create a manifest.json file
- Create an application archive for OACCS
- Deploy an application to OACCS
- Test an application that is deployed into OACCS
- Identify key characteristics for an OACCS application



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Topics



- Introduction to Oracle Application Container Cloud Service
- Creating an Application Archive for OACCS
- Deploying and Testing Your Application to OACCS
- Application Observations and Considerations

Oracle University Student Learning Subscription Use Only

ORACLE®

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

## Oracle Application Container Cloud Service



Oracle Application Container Cloud Service (OACCS) is a service for deploying Node.js and Java SE applications to the cloud. Features include:

- Quickly deploy applications to the Internet
- Supports on demand scaling of your applications
  - Can deploy multiple instances of each application
  - Can define memory size for each application
- Integration with Oracle Cloud Services
  - Oracle Database Cloud Service
  - Oracle Messaging Cloud Service
  - Oracle Developer Cloud Service
  - Oracle Java Cloud Service
- Supports open standards



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

Oracle University Student Learning Subscription Use Only

## Built Using Docker



Docker is an open source lightweight application deployment system for Linux.

- Each application runs in its container. It has its own:
  - Memory space
  - File system
- Containers are lightweight compared to OS virtual machines
  - There is less overhead to start and stop applications.
  - Certain files can be shared between instances for more efficient use of space.
- Applications are isolated and secure from each other.
- Open source under the Apache 2.0 license

## Oracle Application Container Cloud Service and Java Applications



OACCS provides fully functional Java containers.

- Supports latest versions of Java SE (7, 8, and later)
- Leverage any of the tens of thousands of Java libraries and frameworks
- Control size of container and the amount of memory allocated
- Complete control over JVM configuration
- Advanced Java feature support
  - Java Flight Recorder
  - Java Mission Control



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

With an OACCS Java application, you set the configuration of your JVM as desired. In addition, the OACCS provides built-in Java Mission Control and Flight Record for real-time monitoring and analysis of your application.

## Topics



- Introduction to Oracle Application Container Cloud Service
- Creating an Application Archive for OACCS
- Deploying and Testing Your Application to OACCS
- Application Observations and Considerations

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Overview of Steps to Deploy



To deploy a Java application to the Oracle Application Container Cloud Service (OACCS), you must do the following:

1. Create an application that sets the host name and network port at run time.
2. Package the application as a zip or Uber JAR.
  - Uber JAR: Executable JAR that includes all libraries and a runnable Main method
  - Will use Uber JAR in this example
  - May also provide zip containing required JAR files and simply point to the runnable JAR or executable
3. Combine your packaged application with the `manifest.json` file.
4. Upload your application archive to the OACCS.



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

## Configuring an Application with Environment Variables



For an application to run on OACCS, it must set the HOSTNAME and PORT values from the container's environment variables.

- In Java 8, this can be accomplished using the Optional class.

```
public static final Optional<String> host;
public static final Optional<String> port;
public static final Properties myProps = new Properties();

static {
 host = Optional.ofNullable(System.getenv("HOSTNAME"));
 port = Optional.ofNullable(System.getenv("PORT"));
}
// Code continues
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

The Optional class may store either a value or null. In this case, Optional classes of type String are defined. Then a static initializer block is used to read the HOSTNAME and PORT environment variables. If the values are not set, then null will be stored in the variables. Note a Properties object is defined here. It will be used in the next slide.

## Adding Environment Variables to Spring Boot Application



When an Spring Boot application starts, configuration values can be set using the Properties class.

- Setting values will add or overwrite the values for the application.

```
public static void main(String[] args) {
 myProps.setProperty("server.address", host.orElse("localhost"));
 myProps.setProperty("server.port", port.orElse("8080"));

 SpringApplication app = new
 SpringApplication(Application.class);
 app.setDefaultProperties(myProps);
 app.run(args);
}
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

The properties `server.address` and `server.port` are set using the `Optional` objects. The `orElse` method is used to set default values. If valid `String` values are stored in the variable, then that value is returned. However, if the value stored in the `Optional` is `null`, then the `orElse` method will return the value specified. In this case, if no environment variables are set, the host name is set to `localhost` and the port to `8080`.  
The properties are passed to the Spring application using the `setDefaultProperties` method. Those values are used to start the application when the `run` method is called.

## Creating a manifest.json File



The manifest.json file is a simple text file that specifies metadata about your application.

- Major Java version to use
- Command to execute your application
- Release information
  - Version
  - Build
  - Commit
- Notes

# JSON

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Example manifest.json File



This example uses Java 8 to execute a JAR file.

```
{
 "runtime": {
 "majorVersion": "8"
 },
 "command": "java -jar 18-01-customer-rs-18.1.0.jar",

 "release": {
 "version": "18.1.0",
 "build": "24",
 "commit": "1A2B345"
 },
 "notes": "Customer Spring Boot Web Service"
}
```

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Creating an Application Archive



Applications that are submitted to OACCS must be of a specific format.

- Zip file can be .zip format or .tar.gz format
- The compress file must be constructed in a specific way.
  - The manifest.json must be in the root of your zip archive.
  - Your uber JAR must also be in the root of your zip archive.

The structure should look like this:

```
/
|---> manifest.json
|---> customer-rs-18.1.0.jar
```



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

## Using Maven to Build Your Application Archive



Fortunately, the whole process can be automated with Maven.

- Using the assembly plugin to build final archive
- Supports both .zip and .tar.gz archive formats
- Specify a configuration file in the pom.xml
  - Points to bin.xml in the project root
  - Selecting archive type or types
  - Selecting files to include
  - Selecting directory where final archive is placed
- Can be linked to package stage so that the archive is created every time you create a final package

**maven**

**ORACLE®**

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

## Practice18-1: Create an Application Archive with Maven



In this practice, configure Maven to automatically generate both your uber JAR and an application archive zip file with `manifest.json` included.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Topics



- Introduction to Oracle Application Container Cloud Service
- Creating an Application Archive for OACCS
- Deploying and Testing Your Application to OACCS
- Application Observations and Considerations

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Using Oracle Application Container Cloud Service



To get started with Oracle Application Container Cloud Service, follow these steps:

- Log in to your account.
- Locate Oracle Application Container Cloud Service from your list of available cloud services.
- Click the **Service Console** button.



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

Oracle University Student Learning Subscription Use Only

## Create an Application



The first time you use OACCS you will see a screen such as this. Click the Create Application button and select Java Application.

The screenshot shows the Oracle Cloud My Services interface. At the top, there's a navigation bar with 'ORACLE CLOUD My Services', 'Dashboard', 'Notifications', and 'Users'. Below that, a sidebar for 'Application Container Cloud' has 'Applications' selected. The main content area is titled 'Applications' with a search bar. It displays a message: 'You don't have any applications. After meeting the [prerequisites](#), use this button to create an application.' Below this is a section for 'New to creating an application?' with links to 'Watch a video' and 'Step through a tutorial'. At the bottom right of the content area is a blue button labeled 'Create Application'. A large blue curved arrow points from the left towards this button. The footer contains the 'ORACLE' logo and copyright information: 'Copyright © 2016, Oracle and/or its affiliates. All rights reserved.'

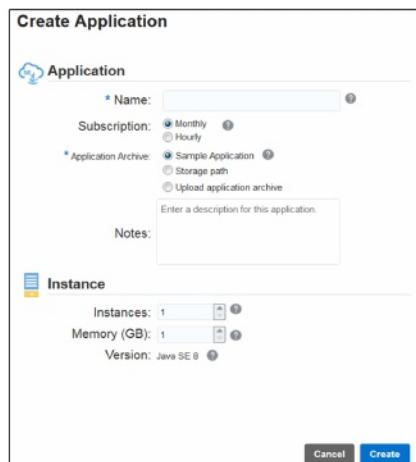
Click the **Create Application** button and select **Java Application**.

Oracle University Student Learning Subscription Use Only

## Create Application Dialog Box

In the Create Application dialog box, you identify your application and enter data about it.

- Default view when the form is loaded



ORACLE®

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

## Fill in Application Information

Fill in the form information for:

- Name
- Subscription Type
- Archive Type
- Notes
- Instance count
- Instance size

The screenshot shows the 'Create Application' dialog box. It has two main sections: 'Application' and 'Instance'. In the 'Application' section, the 'Name' field is set to 'SpringBootApp'. The 'Subscription' dropdown is set to 'Monthly'. Under 'Application Archive', the 'Upload application archive' option is selected, and a file named 'customer-ns-0.1.0-dist.zip' is chosen. A note 'Spring Boot App' is added. In the 'Instance' section, the 'Instances' field is set to '1', 'Memory (GB)' is set to '1', and the 'Version' is set to 'Java SE 8'. At the bottom right are 'Cancel' and 'Create' buttons.

ORACLE®

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

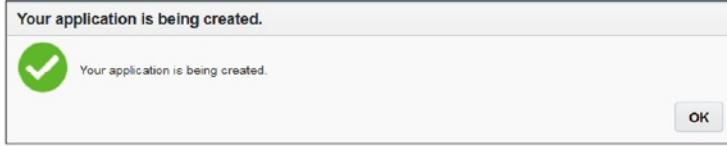


Oracle University Student Learning Subscription Use Only

## Confirmation



If your application archive is in the correct format, the archive is uncompressed and the application is deployed. This dialog box is displayed when the application deployment begins.



ORACLE®

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

## Overview Page

After your application is deployed, the **Overview** page is displayed.

The screenshot shows the Oracle Cloud Infrastructure (OCI) Overview page for an application named "SpringBootApp". The left sidebar lists sections: Overview (1 Instances), Deployments (0 Service Bindings, 0 Environment Variables), Administration (0 Updates Available), and Logs & Recordings (0 Logs, 0 Recordings). The main content area has tabs for Resources, Instances, and Activity. Under Resources, it shows 1 instance with 1 GB of memory. Under Instances, it shows one instance named "web.1" with 1 GB of memory. Under Activity, it shows additional information such as Last Deployed On (Feb 18, 2016 5:29:31 PM UTC), Current Version (0.1.0), Runtime (Java SE 8u60), Runtime Version (1.8.0\_60-b15), Notes (Spring Boot App), Created On (Feb 18, 2016 5:29:31 PM UTC), Subscription Type (MONTHLY), and Identity Domain (Identity Domain).

ORACLE®

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

The overview page displays information about your application. From this page you can determine the number of instances to run and the size of each image. The metadata related to the form you filled out is also displayed here.

Oracle University Student Learning Subscription Use Only

## Deployments Page

The deployments page is used to configure your application.

The screenshot shows the Oracle Cloud Deployments Page for an application named 'SpringBootApp'. The left sidebar has sections for Overview (1 Instances), Deployments (0 Service Bindings, 0 Environment Variables), Administration (0 Updates Available), and Logs & Recordings (0 Logs, 0 Recordings). The main content area is titled 'Deployments' and shows details for Current Version: 0.1.0, Last Deployed On: Feb 18, 2016 5:28:31 PM UTC, Archive Size: 12.79 MB, Archive Name: customer-rs-0.1.0-dist.zip, and Notes: Spring Boot App. It includes a 'Launch Command' field with 'java -jar customers-rs-0.1.0.jar' and a 'Deployment History' section. The bottom right of the page says 'Copyright © 2016, Oracle and/or its affiliates. All rights reserved.'

From the Deployments page, you can configure your application. For example, with the Service Bindings section, you can add connections to Oracle databases as your data store. You can also set any Environment Variables you might need for your application.

**Note:** These settings can also be set using the deployment .json file. Please see the documentation for further details.

Oracle University Student Learning Subscription Use Only

## Administration Page

The administration page manages versions for your application.

The screenshot shows the Oracle Cloud Administration interface for an application named 'SpringBootApp'. The left sidebar has a tree view with nodes: Overview (1 Instances), Deployments (0 Service Bindings, 0 Environment Variables), Administration (0 Updates Available), and Logs & Recordings (0 Logs, 0 Recordings). The main panel is titled 'Updates' and displays the message 'Current Version: Java SE 8u66' and 'Available Updates: No updates available at this time.' A timestamp 'As of Feb 18, 2016 5:39:10 PM UTC' is at the bottom right of the main panel. A blue ribbon banner at the top right says 'Cloud Computing' with a cloud icon.

ORACLE®

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

In this example, if a new version of Java 8 is released, the option to upgrade your instances to that version would be offered on this page.

Oracle University Student Learning Subscription Use Only

## Logs & Recordings Page - Logs Tab



The Logs tab provides the ability to download any logs from your instances.

The screenshot shows the Oracle Cloud interface for the application 'SpringBootApp'. On the left, there's a sidebar with sections: Overview (1 instance), Deployments (0 Service Bindings, 0 Environment Variables), Administration (0 Updates Available), and Logs & Recordings (0 Logs, 0 Recordings). The main content area is titled 'Logs' and shows a message: 'You can download logs from your application instances. The logs are kept in your cloud storage account.' Below this, it says 'Instance: web.1' and 'No logs'. There's a blue 'Get Log' button. At the bottom right of the main area, there's a link 'Logs capture history'.

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Logs & Recordings Page - Recordings Tab



From the Recordings tab, you can download any recordings from Java Flight Recorder sessions you initiated.

As of Feb 18, 2016 5:49:57 PM UTC C3

Overview

1 instances

Deployments

0 Service Bindings

0 Environment Variables

Administration

0 Updates Available

Logs & Recordings

0 Logs

0 Recordings

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Application List Page

If you click the Applications link on any of the pages, you are taken to the Applications list page. This page shows all the currently running applications for your account.

The screenshot shows the 'Applications' list page. At the top, there are tabs for 'Application Container Cloud' and 'Applications'. A search bar is present, and a timestamp 'As of Feb 18, 2016 5:41:23 PM UTC' is shown. A 'Create Application' button is at the top right. Below the header, there are two application entries:

Application	Version	Created On	Memory	Instances
SpringBootApp	0.1.0	Feb 18, 2016 5:29:31 PM UTC	1GB	1
node-app-db	1.0	Feb 3, 2016 5:26:13 PM UTC	1GB	1

Below the table, a link 'Application creation and deletion history' is visible.

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Test the Application

To test the application, look for the URL under your application name on the list page. The URL indicates the host name and port of where your application is running.



The screenshot shows the Oracle Application Container Cloud interface. At the top, there's a navigation bar with a cloud icon and the text "Application Container Cloud". Below it, a sub-menu bar has "Applications" selected. A search bar says "Search by application name" with a magnifying glass icon. On the right, a "Create Application" button is visible. The main content area is titled "Applications" and lists two entries:

Application	Version	Created On	Memory	Instances
SpringBootApp	0.1.0	Feb 18, 2016 5:29:31 PM UTC	1GB	1
node-app-db	1.0	Feb 3, 2016 5:26:13 PM UTC	1GB	1

Each entry includes a small cloud icon, the application name, its version, the creation date, memory allocation, and the number of instances. Below the list is a link "Application creation and deletion history".

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Topics



- Introduction to Oracle Application Container Cloud Service
- Creating an Application Archive for OACCS
- Deploying and Testing Your Application to OACCS
- Application Observations and Considerations

ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Application Observations and Considerations



The architecture of Oracle Application Container Cloud has implications for the applications written for it. Here are some issues to consider:

- Application configuration at startup
- Horizontal application scaling
- Application state
- Application network communication
- Application dependencies



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

## Vertical Scaling

Add more resources to a single system.

This scaling up:

- Increases all aspects of the system
- Renders the system very powerful
- Makes the system very expensive
- Adds limitations on the growth because it still is a single system
- Offers limited failover options

Example: Large Database System



Increase memory

Add disk space

Add CPUs

ORACLE®

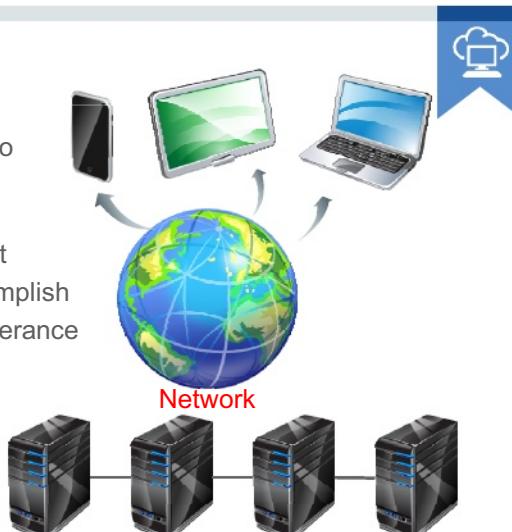
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

## Horizontal Scaling

Use identical machines to act as one to client machines. Scale out.

- Can handle more processing
- Complexity added to management
- Can use cheap hardware to accomplish
- Good failover options and fault tolerance

Example: Search engines



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Horizontal scaling refers to the practice of scaling by providing multiple instances of identical systems. This way any individual system can handle the request passed to it.

Horizontal scaling allows for a great deal of flexibility in scaling in or out. In addition, the system provides good failover options as a failed system can quickly and easily be replaced.

## What Is a Load Balancer?



On a computer network, a load balancer makes many servers look like a single system to a client requesting resources.

- Requests come to one network address.
- Load balancer allocates work to servers.
- Each server processes requests and returns data to calling client.

Load balancers make horizontal scaling work.

- OACCS includes load balancers for applications with more than one instance.
- This has implications for applications.

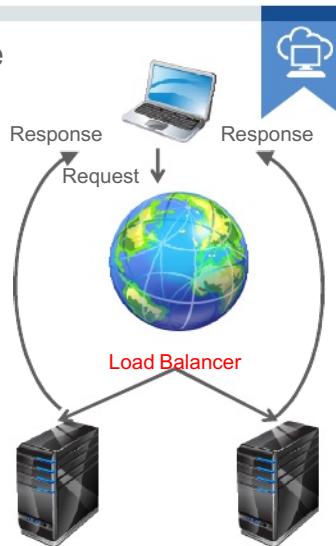
**ORACLE®**

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

The OACCS uses a load balancer when you create multiple instances of your application.

## Horizontal Scaling and Application State

- OACCS scales applications horizontally.
  - Adding instances makes duplicates of your application
  - Affects how an application deals with state
- The issue with state:
  - Load balancer randomly selects an instance to handle request.
  - A request changes the state of the instance.
  - The next request may or may not be the original instance.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

If an application is not stateless, changes made to one instance do not flow to the other instances. Therefore, application state will differ between instances of the application.

## Developing Stateless Applications



Oracle University Student Learning Subscription Use Only

OACCS applications should be stateless.

- State information is stored in a database:
  - Oracle or MySQL
  - No SQL
- Different instances have access to the same data.
- How does this affect the design of your application?
  - Is the application currently stateful or stateless?
  - What changes should you make to it?

**ORACLE®**

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

For an application to be stateless, any state information must be moved into an external system such as a relational database. That way, changes to state are shared by all instances of the application. Thus changes made to one instance will be reflected in all instances in the system.



Requirements and facts about an OACCS application

- Configuration is read from the environment.
  - Must read host name and port from the environment
  - Can set and read your own environment variables as well
  - Read service bindings to connect to other services (for example, database or messaging)
- Communication takes place over network ports.
  - Communicate to your application over network ports.
- Applications are self contained.
  - Required libraries must be included with the application
  - Uber JAR or use CLASSPATH



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

## Practice18-2: Complete a Spring Boot REST Web Service



In this practice, deploy the web service you completed in the previous exercise. Your service should be able to provide the following while deployed on OACCS:

- Provide a list of all the customers in the web service.
- Look up a specific customer from the web service.
- Add a new customer to the web service.
- Update a customer in the web service.
- Delete a customer from the web service.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Summary



In this lesson, you should have learned how to:

- Describe the key features of Oracle Application Container Cloud Service (OACCS)
- Configure a Java application to use environment variables at startup
- Create a manifest.json file
- Create an application archive for OACCS
- Deploy an application to OACCS
- Test an application that is deployed into OACCS
- Identify key characteristics for an OACCS application



**ORACLE®**

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

Oracle University Student Learning Subscription Use Only

## Practice18-3: Testing the Application on OACCS



In this practice, test the deployment of your application on OACCS.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Practice18-4: Scaling the Application



In this practice, scale your application out.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

## Practice18-5: Deploying Your Application to OACCS



Now that you have created a Spring Boot REST application, it is time to update the application for deployment on the Oracle Application Container Cloud Service. Make the required changes and deploy your application.



ORACLE®

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

Oracle University Student Learning Subscription Use Only

Oracle University Student Learning Subscription Use Only