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

13

Using Interfaces

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

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

Interactive Quizzes

ORACLE

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

Before you start today's lessons, test your knowledge by answering some quiz questions that relate to yesterday's lessons. Open your quiz file from labs > Quizzes > Java SE 8 Fundamentals Quiz.html. Click the links for lessons titled "Working with Arrays, Loops, and Dates" and "Using Inheritance."

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Override the `toString` method of the `Object` class
- Implement an interface in a class
- Cast to an interface reference to allow access to an object method
- Write a simple lambda expression that consumes a `Predicate`

ORACLE

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Topics

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

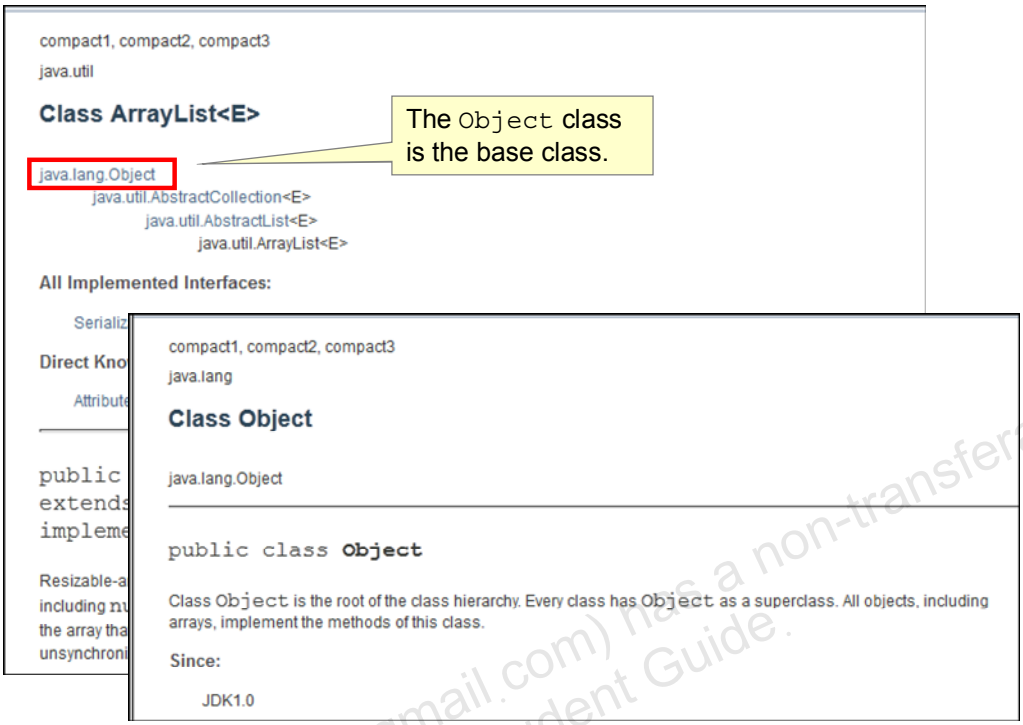
Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

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

The `Object` Class



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

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

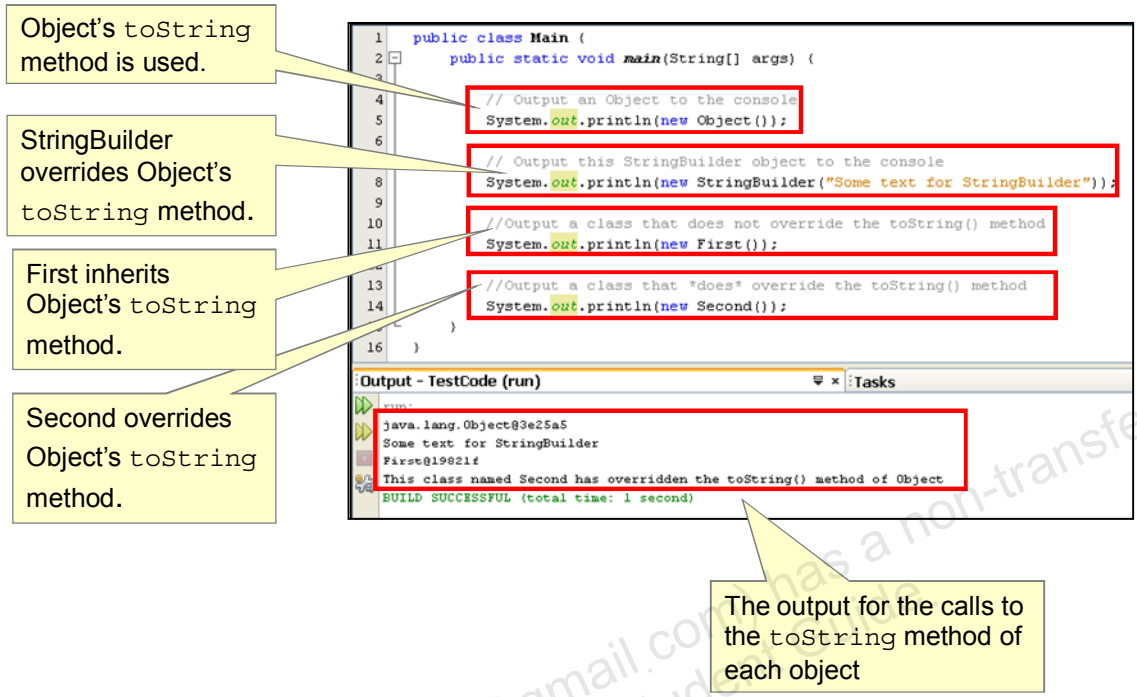
So all classes have `Object` at the root of their hierarchy. This means that all classes have access to the methods of `Object`. Being the root of the object hierarchy, `Object` does not have many methods—only very basic ones that all objects must have.

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Calling the `toString` Method



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

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

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

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide

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

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

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide

Topics

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

The Java Interface

- An interface is similar to an abstract class, except that:
  - Methods are implicitly abstract (except default methods)
  - A class does not *extend* it, but *implements* it
  - A class may implement more than one interface
- All abstract methods from the interface must be implemented by the class.

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

Implicitly abstract

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

Implements the print() method.

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

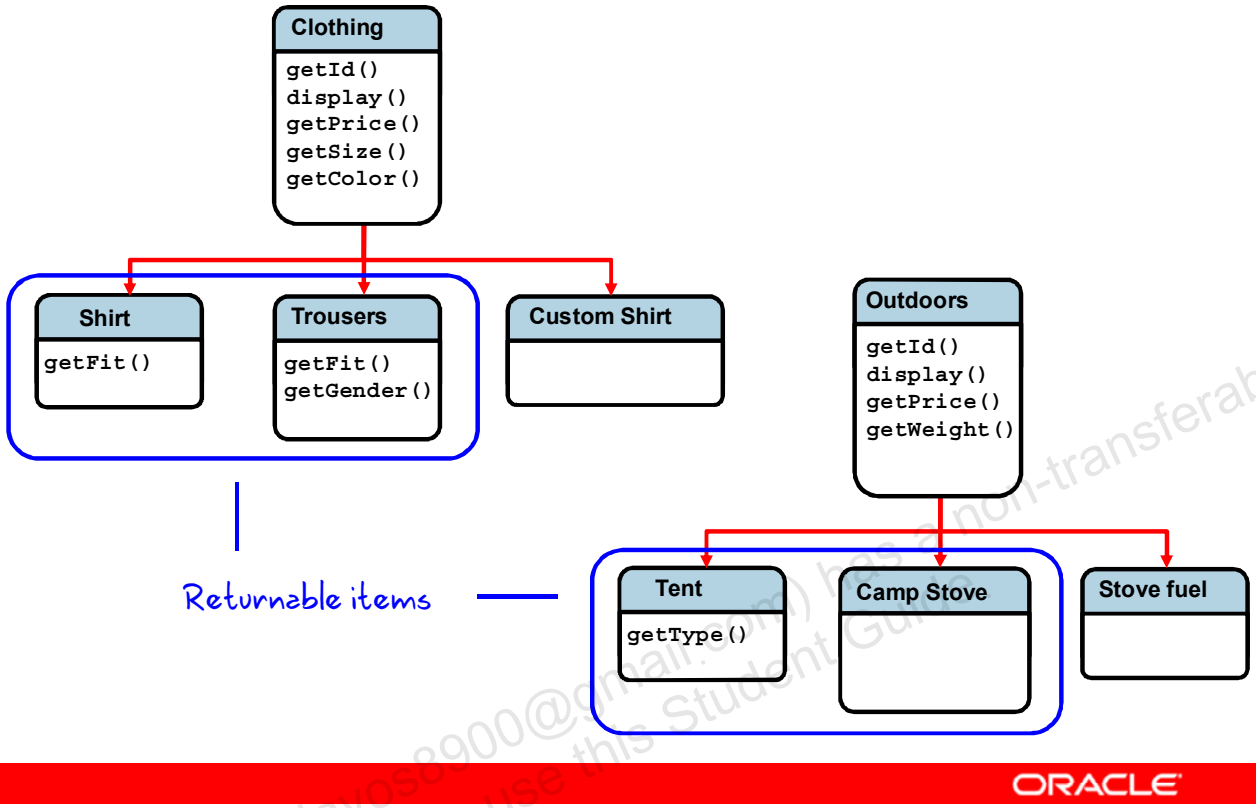
When a class implements an interface, it enters into a contract with the interface to implement all of its abstract methods. Therefore, using an interface lets you enforce a particular public interface (set of public methods).

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

Note: A method within an interface is assumed to be abstract unless it uses the `default` keyword. Default methods in an interface are new with SE 8. They are used with lambda expressions. You will learn about lambda expressions a little later in this lesson; however, default methods and lambda expressions are covered in even more depth in the *Java SE8 New Features* course.

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Multiple Hierarchies with Overlapping Requirements



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

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

Daniel Hurtado (dhvos8900@gmail.com) has a non-transferable license to use this Student Guide.

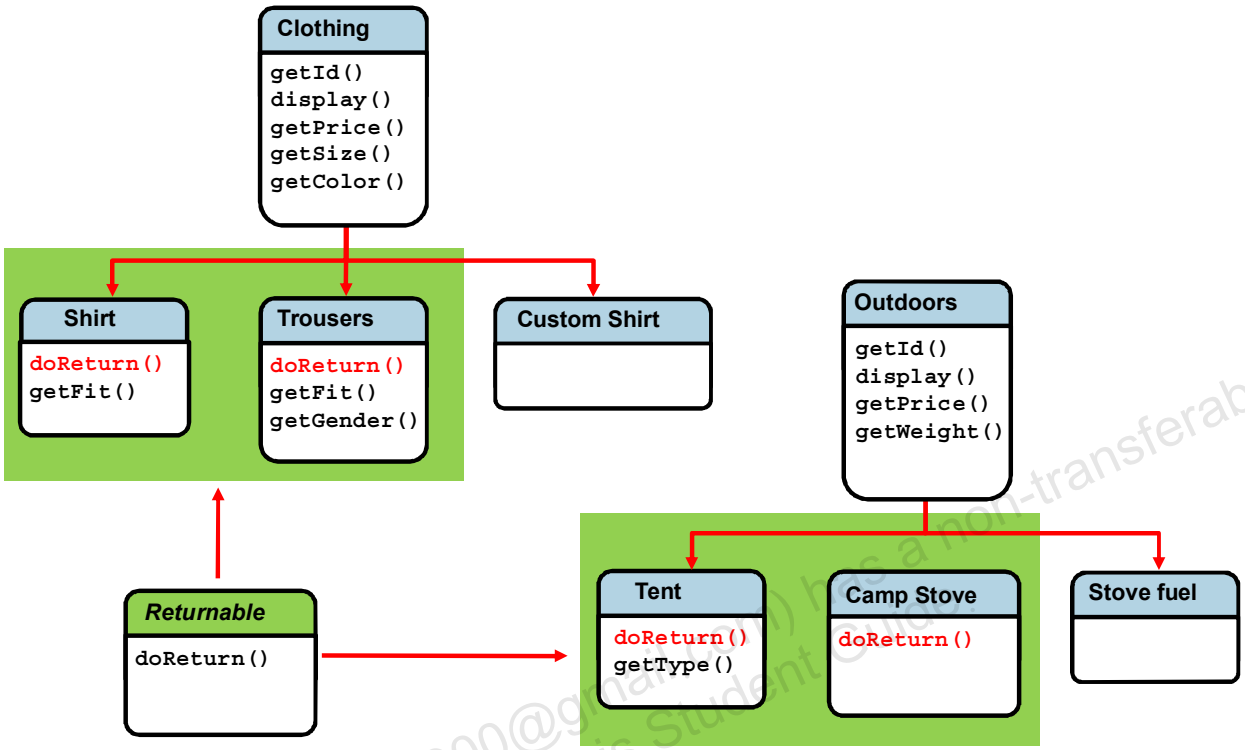
A more complex set of classes may have items in two different hierarchies. If Duke's Choice starts selling outdoor gear, it may have a completely different superclass called `Outdoors`, with its own set of subclasses (for example, `getWeight` as an `Outdoors` method).

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



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

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Implementing the `Returnable` Interface

`Returnable` interface

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

Implicitly abstract method

`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.

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

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

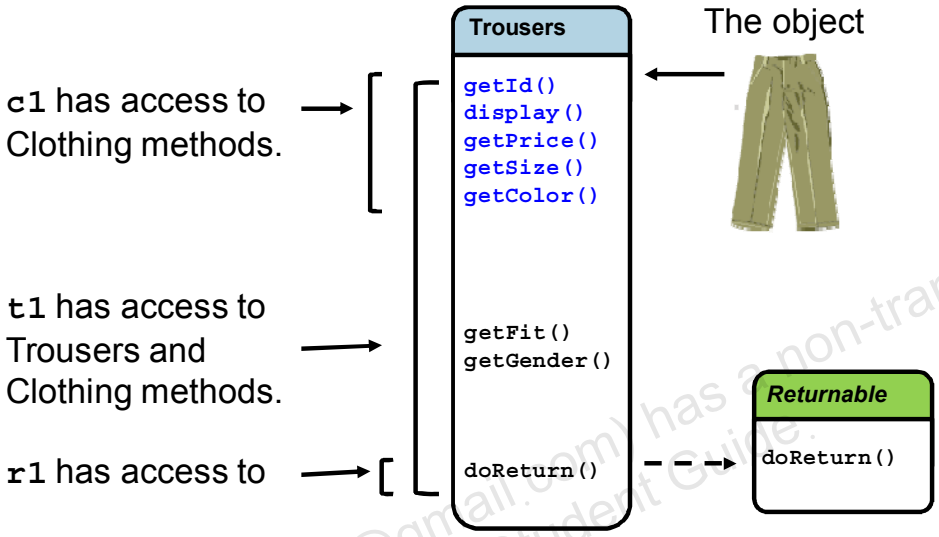
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Quiz

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

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

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

Answer: c

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Quiz

How can you change the reference type of an object?

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

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

Answer: b, c

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Topics

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

The Collections Framework

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

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

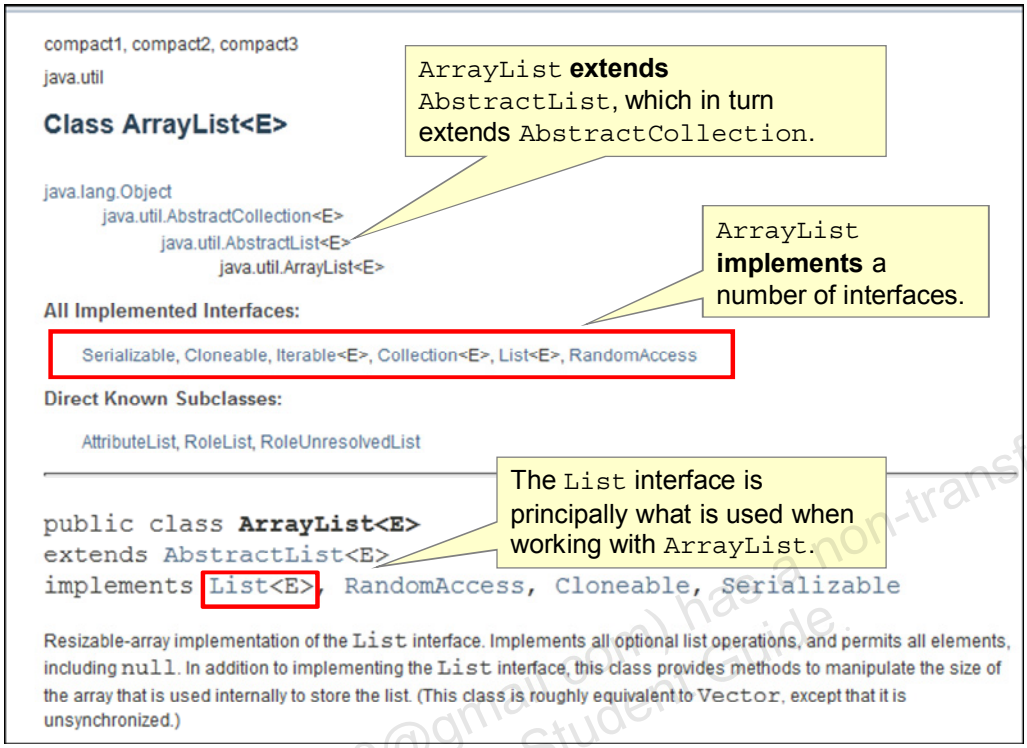
ORACLE

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

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

The collections framework is covered in much depth in the *Java SE 8 Programming* course.

`ArrayList` Example



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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
Dario Hurtado (dariohos8900@gmail.com) has a non-transferable license to use this Student Guide.

`List` Interface

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

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

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

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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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).

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

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

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

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

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

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

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));
```

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

Dario Hurtado (dariohs8900@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

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

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

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

Exercise 13-1: Converting an Array to an ArrayList

In this exercise, you:

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

ORACLE

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

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Topics

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

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

Example: Modifying a `List` of Names

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

```
1  String[] names = {"Ned","Fred","Jessie","Alice","Rick"};
2  List<String> mylist = new ArrayList(Arrays.asList(names));
3
4  // Display all names in upper case
5  for(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 © 2014, Oracle and/or its affiliates. All rights reserved.

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

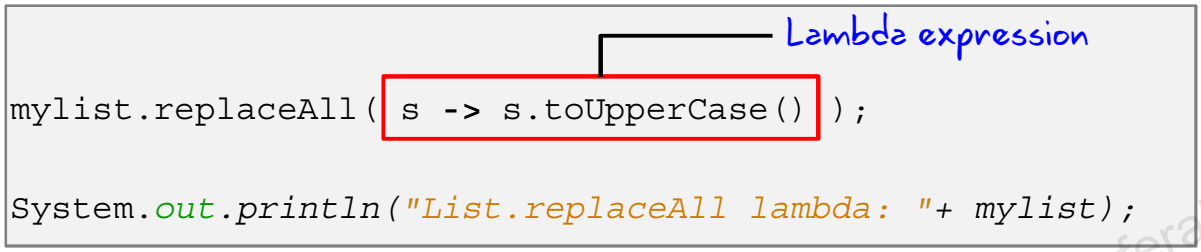
Remember that `String` objects are immutable. You cannot change them in place. All you can do is create a new `String` with the desired changes and then reassign the reference to point to the new `String`. You could do that here, but it would not be trivial.

A lambda expression makes this much easier!

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Using a Lambda Expression with `replaceAll`

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



Output:

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

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

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

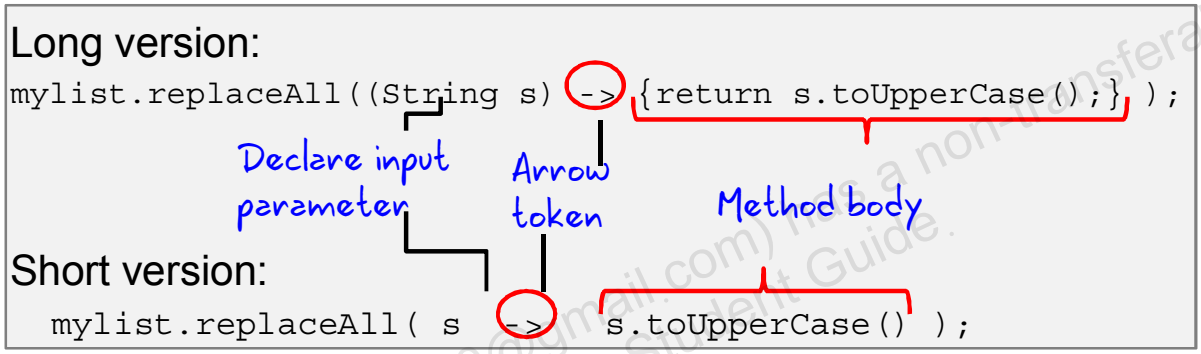
The output of this code shows that the actual elements of the list were modified.

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Lambda Expressions

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

- Input parameters
- A method body
- A return value



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

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

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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
Dario Hurtado (dayos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

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

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

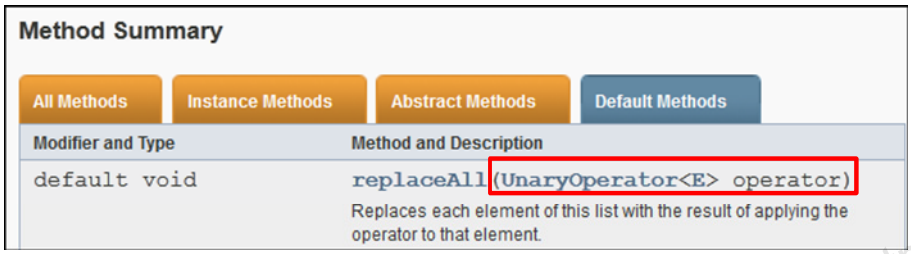
For a much more comprehensive treatment of lambda expressions, take the *Java SE 8 Programming* course.

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
Dario Hurtado (dariohs8900@gmail.com) has a non-transferable license to use this Student Guide.

Lambda Types

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

- `replaceAll` takes a `UnaryOperator` type expression.



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

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

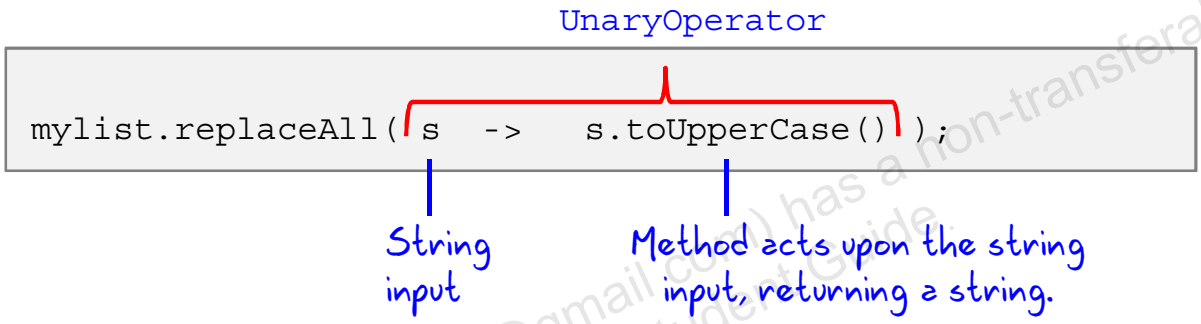
The lambda types can be viewed by looking at the `java.util.functions` package in the JDK API documentation. There are a great many of these and they are actually interfaces. They specify the interface of the expression. Much like a method signature, they indicate the inputs, statements, and outputs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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

The `UnaryOperator` Lambda Type

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

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



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

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

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

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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.
Dario Hurtado (dariosa900@gmail.com) has a non-transferable license to use this Student Guide.

The `Predicate` Lambda Type

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

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

removeIf

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

- Examples:

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

ORACLE

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

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

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

- In the first example, the `Predicate` uses the `equals` method of the `String` argument to compare its value with the string "Rick". If it is equal, the `Predicate` returns `true`. The long version of the `Predicate` expression would look like this:

```
mylist.removeIf ((String s) -> {return s.equals("Rick"); } )
```

- In the second example, the `Predicate` uses the `length()` method of the `String` argument, returning `true` if the string has less than 5 characters. The long version of this `Predicate` expression would look like this:

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Exercise 13-2: Using a `Predicate` Lambda Expression

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

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

ORACLE

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

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

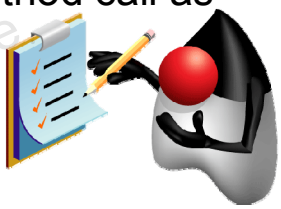
Unauthorized reproduction or distribution prohibited. Copyright© 2017, Oracle and/or its affiliates.

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Summary

In this lesson, you should have learned the following:

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



ORACLE

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

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

Practice 13-1 Overview: Overriding the `toString` Method

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

ORACLE

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

Practice 13-2 Overview: Implementing an Interface

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

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

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

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

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

David Hurtado (davos8900@gmail.com) has a non-transferable license to use this Student Guide.

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

David Hurtado (davos8900@gmail.com) has a non-transferable
license to use this Student Guide.