6

Interfaces and Lambda Expressions

Objectives

After completing this lesson, you should be able to do the following:

- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a lambda expression

Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- Are similar to abstract classes containing only public abstract methods
- Outline methods that must be implemented by a class
  - Methods must not have an implementation {braces}.
- Can contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns

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

A Problem Solved by Interfaces

Given: A company sells an assortment of products, very different from each other, and needs a way to access financial data in a similar manner.

- Products include:
  - Crushed Rock
    - Measured in pounds
  - Red Paint
    - Measured in gallons
  - Widgets
    - Measured by Quantity
- Need to calculate per item
  - Sales price
  - Cost
  - Profit

How can interfaces solve this problem? All these products are sold by the same company but are measured in different ways.

`CrushedRock` Class

- The `CrushedRock` class before interfaces

```
public class CrushedRock {
  private String name;
  private double salesPrice = 0;
  private double cost = 0;
  private double weight = 0; // In pounds

  public CrushedRock(double salesPrice, double cost,
double weight){
    this.salesPrice = salesPrice;
    this.cost = cost;
    this.weight = weight;
  }
}
```


Notice this class has a `weight` field. However, the `RedPaint` class has a `gallons` field and the `Widget` class has a `quantity` field. Calculating the required data for the three classes is similar but different for each.

The `SalesCalcs` Interface

- The `SalesCalcs` interface specifies the types of calculations required for our products.
  - Public, top-level interfaces are declared in their own `.java` file.

```
public interface SalesCalcs {
  public String getName();
  public double calcSalesPrice();
  public double calcCost();
  public double calcProfit();
}
```


The `SalesCalcs` interface specifies what methods much be implemented by a class. The method signature specifies what is passed in and what is returned.

Rules for Interfaces

Access Modifiers

All methods in an interface are `public`, even if you forget to declare them as `public`. You may not declare methods as `private` or `protected` in an interface.

Abstract Modifier

Because all methods are implicitly `abstract`, it is redundant (but allowed) to declare a method as `abstract`. Because all interface methods are abstract, you may not provide any method implementation, not even an empty set of braces.

Implement Multiple Interfaces

A class can implement more than one interface in a comma-separated list at the end of the class declaration.

Adding an Interface

- The updated `CrushedRock` class implements `SalesCalcs`.

```
public class CrushedRock implements SalesCalcs{
  private String name = "Crushed Rock";
...  // a number of lines not shown
  @Override
  public double calcCost(){
    return this.cost * this.weight;
  }

  @Override
  public double calcProfit(){
    return this.calcSalesPrice() - this.calcCost();
  }
}
```


On the class declaration line, the implements keyword specifies the `SalesCalcs` interface for this class. Each of the methods specified by `SalesCalcs` must be implemented. However, how the methods are implemented may differ from class to class. The only requirement is that the method signature matches. This allows the cost or sales price calculations to differ between classes.

Interface References

- Any class that implements an interface can be referenced by using that interface.
- Notice how the `calcSalesPrice` method can be referenced by the `CrushedRock` class or the `SalesCalcs` interface.

```
CrushedRock rock1 = new CrushedRock(12, 10, 50);
SalesCalcs rock2 = new CrushedRock(12, 10, 50);
System.out.println("Sales Price: " +
rock1.calcSalesPrice());
System.out.println("Sales Price: " +
rock2.calcSalesPrice());
```

- Output

```
Sales Price: 600.0
Sales Price: 600.0
```


Because `CrushedRock` implements `SalesCalcs`, a `SalesCalcs` reference can be used to access the data of a `CrushedRock` object.

Interface Reference Usefulness

- Any class implementing an interface can be referenced by using that interface. For example:

```
SalesCalcs[] itemList = new SalesCalcs[5];
ItemReport report = new ItemReport();

itemList[0] = new CrushedRock(12.0, 10.0, 50.0);
itemList[1] = new CrushedRock(8.0, 6.0, 10.0);
itemList[2] = new RedPaint(10.0, 8.0, 25.0);
itemList[3] = new Widget(6.0, 5.0, 10);
itemList[4] = new Widget(14.0, 12.0, 20);

System.out.println("==Sales Report==");
for(SalesCalcs item:itemList){
  report.printItemData(item);
}
```


Because all three classes share a common interface, a list of different classes like the above can be created and processed in the same way.

Interface Code Flexibility

- A utility class that references the interface can process any implementing class.

```
public class ItemReport {
  public void printItemData(SalesCalcs item){
    System.out.println("--" + item.getName() + " Report-
    -");
    System.out.println("Sales Price: " +
    item.calcSalesPrice());
    System.out.println("Cost: " + item.calcCost());
    System.out.println("Profit: " + item.calcProfit());
  }
}
```


Instead of having to write a method to print the data from each class, the interface reference allows you to retrieve the data from all three classes.

`default` Methods in Interfaces

Java 8 has added `default` methods as a new feature:

```
public interface SalesCalcs {
... // A number of lines omitted
  public default void printItemReport(){
    System.out.println("--" + this.getName() + " Report--");
    System.out.println("Sales Price: " + this.calcSalesPrice());
    System.out.println("Cost: " + this.calcCost());
    System.out.println("Profit: " + this.calcProfit());
  }
}
```

`default` methods:

- Are declared by using the keyword `default`
- Are fully implemented methods within an interface
- Provide useful inheritance mechanics

Java 8 adds `default` methods as a new feature. Using the `default` keyword allows you to provide fully implemented methods to all implementing classes. The above example shows how the item report, which was implemented as a separate class earlier, can be fully implemented as a default method. Now all three classes automatically get a fully implemented `printItemReport` method.

The feature was added to simplify the development of APIs that rely heavily on interfaces. Before, simply adding a new method breaks all implementing and extended classes. Now, default methods can be added or changed without harming API hierarchies.

`default` Method: Example

Here is an updated version of the item report using `default` methods.

```
SalesCalcs[] itemList = new SalesCalcs[5];

itemList[0] = new CrushedRock(12, 10, 50);
itemList[1] = new CrushedRock(8, 6, 10);
itemList[2] = new RedPaint(10, 8, 25);
itemList[3] = new Widget(6, 5, 10);
itemList[4] = new Widget(14, 12, 20);

System.out.println("==Sales Report==");
for(SalesCalcs item:itemList){
  item.printItemReport();
}
```


Printing the report now involves simply calling the `printItemReport` method.

`static` Methods in Interfaces

Java 8 allows `static` methods in an interface. So it is possible to create helper methods like the following.

```
public interface SalesCalcs {
... // A number of lines omitted
  public static void printItemArray(SalesCalcs[] items){
    System.out.println(reportTitle);
      for(SalesCalcs item:items){
        System.out.println("--" + item.getName() + " Report--");
        System.out.println("Sales Price: " +
item.calcSalesPrice());
        System.out.println("Cost: " + item.calcCost());
        System.out.println("Profit: " + item.calcProfit());
      }
}
```


This is a convenience feature. Now you can include a helper method, like the above, in an interface instead of in a separate class.

Here is an example of calling the method:

```
SalesCalcs.printItemArray(itemList);
```

Constant Fields

Interfaces can have constant fields.

```
public interface SalesCalcs {
   public static final String reportTitle="\n==Static
    List Report==";
... // A number of lines omitted
```


Constant fields are permitted in an interface. When you declare a field in an interface, it is implicitly `public`, `static`, and `final`. You may redundantly specify these modifiers.

Extending Interfaces

- Interfaces can extend interfaces:

```
public interface WidgetSalesCalcs extends SalesCalcs{
  public String getWidgetType();
}
```

- So now any class implementing `WidgetSalesCalc` must implement all the methods of `SalesCalcs` in addition to the new method specified here.

Implementing and Extending

- Classes can extend a parent class and implement an interface:

```
public class WidgetPro extends Widget implements
WidgetSalesCalcs{
  private String type;

  public WidgetPro(double salesPrice, double cost, long
quantity, String type){
    super(salesPrice, cost, quantity);
    this.type = type;
  }

  public String getWidgetType(){
    return type;
  }
}
```


Extends First

If you use both `extends` and `implements`, `extends` must come first.

Anonymous Inner Classes

- Define a class in place instead of in a separate file
- Why would you do this?
  - Logically group code in one place
  - Increase encapsulation
  - Make code more readable
- `StringAnalyzer` interface

```
public interface StringAnalyzer {
  public boolean analyze(String target, String
    searchStr);
}
```

- A single method interface
  - **Functional Interface**
- Takes two strings and returns a `boolean`

An interface like this with a single method is called a **Functional Interface**.

Anonymous Inner Class: Example

- Example method call with concrete class

```
20     // Call concrete class that implments StringAnalyzer
21     ContainsAnalyzer contains = new ContainsAnalyzer();
22
23     System.out.println("===Contains===");
24     Z03Analyzer.searchArr(strList01, searchStr, contains);
```

- Anonymous inner class example

```
22     Z04Analyzer.searchArr(strList01, searchStr,
23       new StringAnalyzer(){
24         @Override
25         public boolean analyze(String target, String
    searchStr){
26           return target.contains(searchStr);
27         }
28 });
```

- The class is created in place.

The example shows how an anonymous inner class can be substituted for an object.

Here is the source code for `ContainsAnalyzer`:

```
public class ContainsAnalyzer implements StringAnalyzer {
  public boolean analyze(String target, String searchStr){
    return target.contains(searchStr);
  }
}
```

Note that the anonymous inner class specifies no name but implements almost exactly the same code. The syntax is a little complicated as the class is defined where a parameter variable would normally be.

The slides that follow explain some of the advantages of this approach.

String Analysis Regular Class

- Class analyzes an array of strings given a search string
  - Print strings that contain the search string
  - Other methods could be written to perform similar string test
- Regular Class Example method

```
1 package com.example;
2
3 public class AnalyzerTool {
4   public boolean arrContains(String sourceStr, String
    searchStr){
5     return sourceStr.contains(searchStr);
6   }
7 }
8
```


The method takes the source string and searches for text matching the text in the search string. If there is a match, `true` is returned. If no match is found, `false` is returned.

Additional methods could be written to compare lengths or determine if the string starts with the search string. Only one method is implemented for simplicity.

String Analysis Regular Test Class

- Here is the code to test the class, `Z01Analyzer`

```
 4   public static void main(String[] args) {
 5     String[] strList =
 6     {"tomorrow","toto","to","timbukto","the","hello","heat"};
 7     String searchStr = "to";
 8     System.out.println("Searching for: " + searchStr);
 9
10     // Create regular class
11     AnalyzerTool analyzeTool = new AnalyzerTool();
12
13     System.out.println("===Contains===");
14     for(String currentStr:strList){
15       if  (analyzeTool.arrContains(currentStr, searchStr)){
16         System.out.println("Match: " + currentStr);
17       }
18     }
19 }
```


This first class is pretty standard. The test array is passed to `foreach` loop where the method is used to print out matching words. Here is the output:

```
==Contains==
Match: tomorrow
Match: toto
Match: to
Match: timbukto
```

String Analysis Interface: Example

- What about using an interface?

```
3 public interface StringAnalyzer {
4   public boolean analyze(String sourceStr, String
    searchStr);
5 }
```

- `StringAnalyzer` is a single method functional interface.
- Replacing the previous example and implementing the interface looks like this:

```
3 public class ContainsAnalyzer implements StringAnalyzer {
4   @Override
5   public boolean analyze(String target, String searchStr){
6     return target.contains(searchStr);
7   }
8 }
```

ORACLE

In the example, a switch is made to use an interface instead of just a plain class. Notice that `StringAnalyzer` is a functional interface because it has only one method. Other than the addition of the `implements` clause, the class is unchanged.

String Analyzer Interface Test Class

```
 4   public static void main(String[] args) {
 5     String[] strList =
 6     {"tomorrow","toto","to","timbukto","the","hello","heat"};
 7     String searchStr = "to";
 8     System.out.println("Searching for: " + searchStr);
 9
10     // Call concrete class that implments StringAnalyzer
11     ContainsAnalyzer contains = new ContainsAnalyzer();
12
13     System.out.println("===Contains===");
14     for(String currentStr:strList){
15       if (contains.analyze(currentStr, searchStr)){
16         System.out.println("Match: " + currentStr);
17       }
18     }
19   }
```


The change to an interface does not change this test class much. The only difference is that a different class is used to perform the string testing. Also, if additional tests need to be performed, this would require additional `foreach` loops and a separate class for each test condition. Arguably, this might be a step back.

However, there are advantages of this approach.

Encapsulate the for Loop

- An improvement to the code is to encapsulate the `forloop`:

```
 3 public class Z03Analyzer {
 4
 5   public static void searchArr(String[] strList, String
   searchStr, StringAnalyzer analyzer){
 6     for(String currentStr:strList){
 7       if (analyzer.analyze(currentStr, searchStr)){
 8         System.out.println("Match: " + currentStr);
 9       }
10     }
11   }
// A number of lines omitted
```


By encapsulating the `forloop` into a static helper method, only one loop is needed to process any sort of string test using the `StringAnalyzer` interface. The `searchArr` method remains unchanged in all the examples that follow.

Note the parameters for the method:

1. The string array
2. The search string
3. A class that implements the `StringAnalyzer` interface

String Analysis Test Class with Helper Method

- With the helper method, the main method shrinks to this:

```
13   public static void main(String[] args) {
14     String[] strList01 =
15     {"tomorrow","toto","to","timbukto","the","hello","heat"};
16     String searchStr = "to";
17     System.out.println("Searching for: " + searchStr);
18
19     // Call concrete class that implments StringAnalyzer
20     ContainsAnalyzer contains = new ContainsAnalyzer();
21
22     System.out.println("===Contains===");
23     Z03Analyzer.searchArr(strList01, searchStr, contains);
24   }
```


Now the array can be searched and the results displayed with the single call on line 23.

String Analysis Anonymous Inner Class

- Create anonymous inner class for third argument.

```
19      // Implement anonymous inner class
20      System.out.println("===Contains===");
21      Z04Analyzer.searchArr(strList01, searchStr,
22        new StringAnalyzer(){
23          @Override
24          public boolean analyze(String target, String
    searchStr){
25            return target.contains(searchStr);
26          }
27        });
28    }
```


In this example, the third argument to the method call is an anonymous inner class. Notice that the class structure is the same as that of the `ContainsAnalyzer` in the previous example. Using the approach, the code is stored in the calling class. In addition, the logic for the `analyze` method can easily be changed depending on the circumstances. However, there are a few drawbacks.

1. The syntax is a little complicated. The entire class definition is included between the parentheses of the argument list.
2. Because there is no class name, when the code is compiled, a class file will be generated and a number assigned for the class. This is not a problem if there is only one anonymous inner class. However, when there is more than one in multiple classes, it is difficult to figure out which class file goes with which source file.

String Analysis Lambda Expression

- Use lambda expression for the third argument.

```
13   public static void main(String[] args) {
14     String[] strList =
15     {"tomorrow","toto","to","timbukto","the","hello","heat"};
16     String searchStr = "to";
17     System.out.println("Searching for: " + searchStr);
18
19     // Lambda Expression replaces anonymous inner class
20     System.out.println("==Contains==");
21     Z05Analyzer.searchArr(strList, searchStr,
22       (String target, String search) -> target.contains(search));
23   }
```


With Java 8, a lambda expression can be substituted for an anonymous inner class. Notice some key facts.

- **The lambda expression has two arguments:** Just like in the two previous examples, the lambda expression uses the same arguments as the analyze method.
- **The lambda expression returns a `boolean`:** Just like in the two previous examples, a `boolean` is returned just like the analyze method.

In fact, the lambda expression, anonymous inner class, and concrete class are all essentially equivalent. A lambda expression is a new way to express code logic by using a functional interface as the base.

Lambda Expression Defined

Argument List	Arrow Token	Body
(int x, int y)	->	x + y

Basic Lambda examples

```
(int x, int y) -> x + y
(x, y) -> x + y

(x, y) -> { system.out.println(x + y);}

(String s) -> s.contains("word")
s -> s.contains("word")
```


The lambda expression is an argument list, the arrow token, and then a block or expression. When a code block is used, multiple statements could be included in the block. The parameter types can be specified or inferred.

What Is a Lambda Expression?

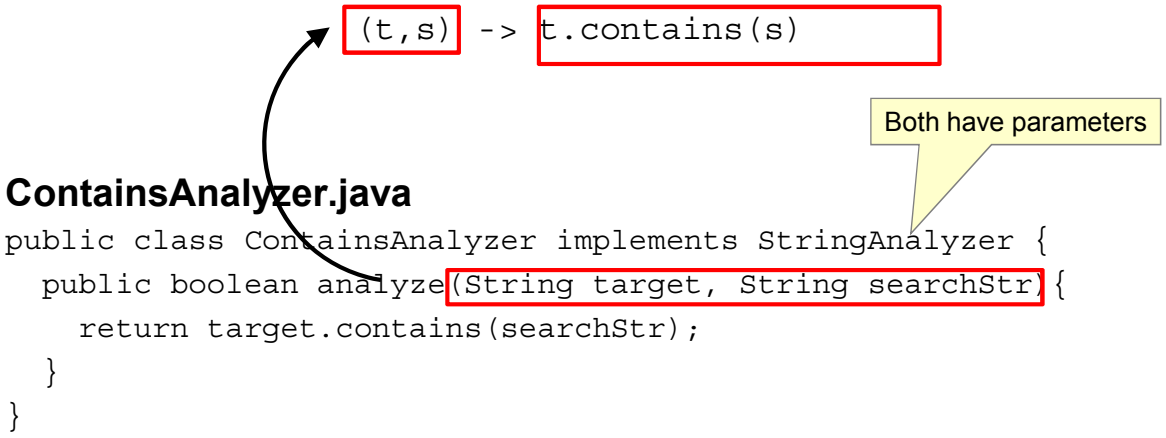
```
(t,s) -> t.contains(s)
```

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {
  public boolean analyze(String target, String searchStr){
    return target.contains(searchStr);
  }
}
```

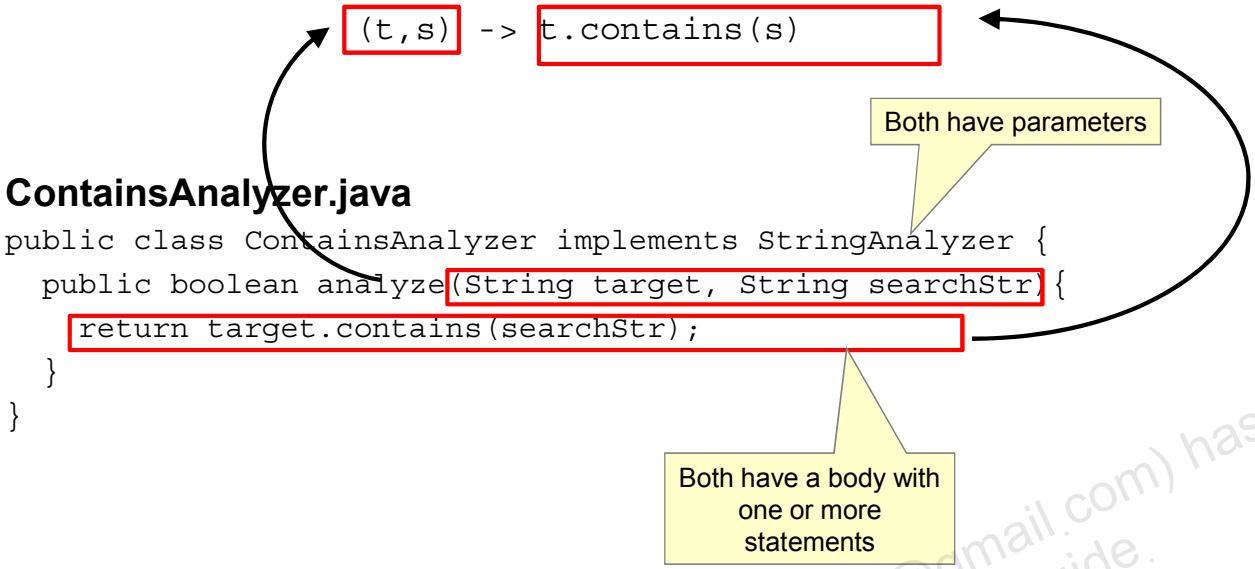

Compare a lambda expression to an implementation of the `StringAnalyzer` interface.

What Is a Lambda Expression?



In the slide, a lambda expression is compared to an implementation of the `StringAnalyzer` interface. In essence, they are equivalent because the lambda expression can be substituted for an anonymous inner class or the implementing class. All three cases rely on the `StringAnalyzer` interface as the underlying plumbing.

What Is a Lambda Expression?



In the slide, a lambda expression is compared to an implementation of the `StringAnalyzer` interface. In essence, they are equivalent because the lambda expression can be substituted for an anonymous inner class or the implementing class. All three cases rely on the `StringAnalyzer` interface as the underlying plumbing.

Lambda Expression Shorthand

- Lambda expressions using shortened syntax

```
20      // Use short form Lambda
21      System.out.println("==Contains==");
22      Z06Analyzer.searchArr(strList01, searchStr,
23        (t, s) -> t.contains(s));
24
25      // Changing logic becomes easy
26      System.out.println("==Starts With==");
27      Z06Analyzer.searchArr(strList01, searchStr,
28        (t, s) -> t.startsWith(s));
```

- The `searchArr` method arguments are:

```
public static void searchArr(String[] strList, String
  searchStr, StringAnalyzer analyzer)
```


The first example in the slide is equivalent to the previous code example. The only difference is that lambda expression shorthand has been used. The type of the arguments is inferred from the context the lambda expression is used in. So the compiler knows that the signature for the `StringAnalyzer.analyze` is:

```
public boolean analyze(String sourceStr, String searchStr);
```

Thus, two strings are passed in and a `boolean` is returned.

Notice the second example on line 28. Now it becomes trivial to change the logic for a functional interface.

Lambda Expressions as Variables

- Lambda expressions can be treated like variables.
- They can be assigned, passed around, and reused.

```
19      // Lambda expressions can be treated like variables
20      StringAnalyzer contains = (t, s) -> t.contains(s);
21      StringAnalyzer startsWith = (t, s) -> t.startsWith(s);
22
23      System.out.println("==Contains==");
24      Z07Analyzer.searchArr(strList, searchStr,
25        contains);
26
27      System.out.println("==Starts With==");
28      Z07Analyzer.searchArr(strList, searchStr,
29        startsWith);
```

Summary

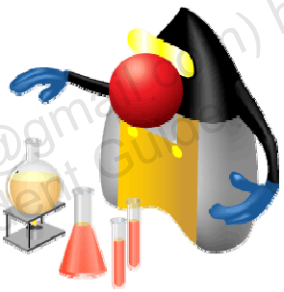
In this lesson, you should have learned how to:

- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a Lambda Expression

Practice 6-1: Implementing an Interface

This practice covers the following topics:

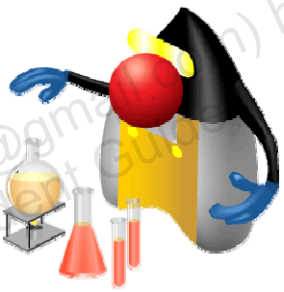
- Writing an interface
- Implementing an interface
- Creating references of an interface type
- Casting to interface types



Practice 6-2: Using Java Interfaces

This practice covers the following topics:

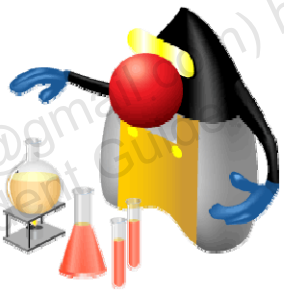
- Updating the banking application to use an interface
- Using interfaces to implement accounts



Practice 6-3: Creating Lambda Expression

This practice covers the following topics:

- Performing string analysis using lambda expressions
- Practicing writing lambda expressions for the `StringAnalyzer` interface



Quiz

All methods in an interface are:

a. `final`
b. `abstract`
c. `private`
d. `volatile`

Quiz

When a developer creates an anonymous inner class, the new class is typically based on which one of the following?

a. `enums`
b. Executors
c. Functional interfaces
d. Static variables

Quiz

Which is true about the parameters passed into the following lambda expression?

```
(t,s) -> t.contains(s)
```

a. Their type is inferred from the context.
b. Their type is executed.
c. Their type must be explicitly defined.
d. Their type is undetermined.