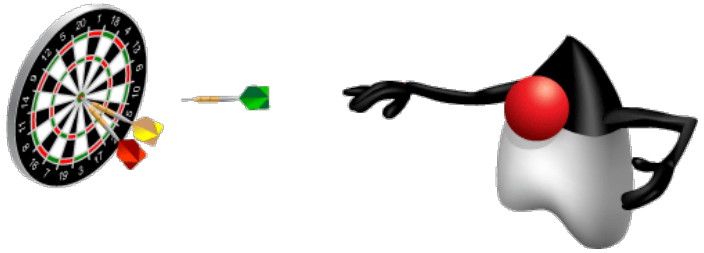
Interfaces and Lambda Expressions

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

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

`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;
  }
}
```

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();
}
```

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();
  }
}
```

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
```

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);
}
```

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());
  }
}
```

`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

`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();
}
```

`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());
      }
}
```

Constant Fields

Interfaces can have constant fields.

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

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;
  }
}
```

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`

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.

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
```

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 }
```

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 }
```

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   }
```

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
```

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   }
```

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    }
```

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   }
```

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")
```

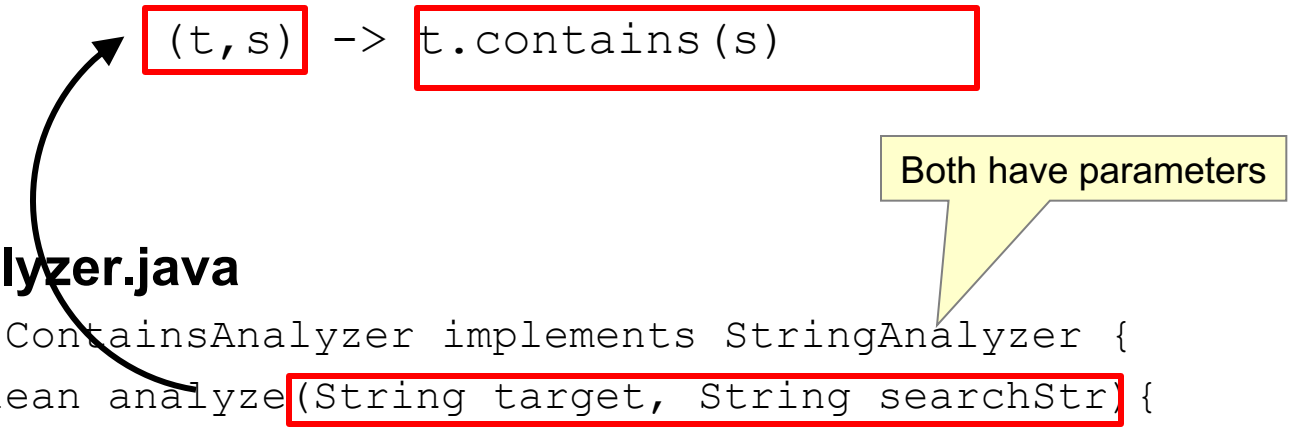
What Is a Lambda Expression?

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

ContainsAnalyzer.java

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

What Is a Lambda Expression?



ContainsAnalyzer.java

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

What Is a Lambda Expression?

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

Both have parameters

ContainsAnalyzer.java

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

Both have a body with one or more statements

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)
```

Lambda Expressions as Variables

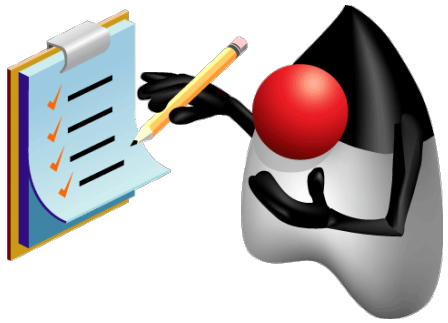
- Lambda expressions can be treated like variables.
- They can be assigned, passed around, and reused.

```
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:

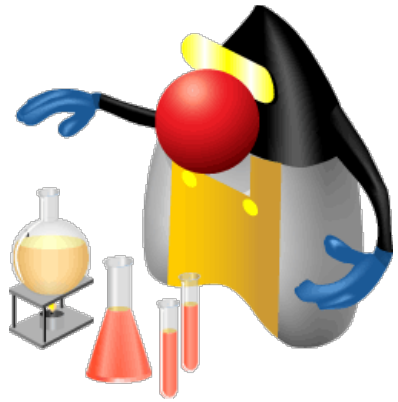
- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a Lambda Expression



Practice 6-1: Implementing an Interface

This practice covers the following topics:

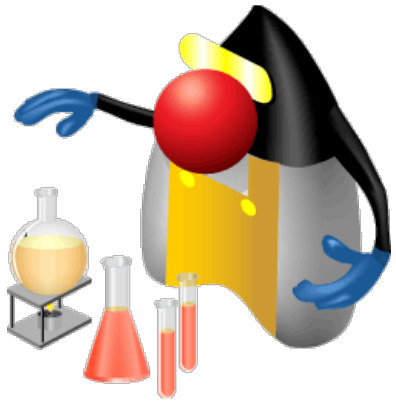
- Writing an interface
- Implementing an interface
- Creating references of an interface type
- Casting to interface types



Practice 6-2: Using Java Interfaces

This practice covers the following topics:

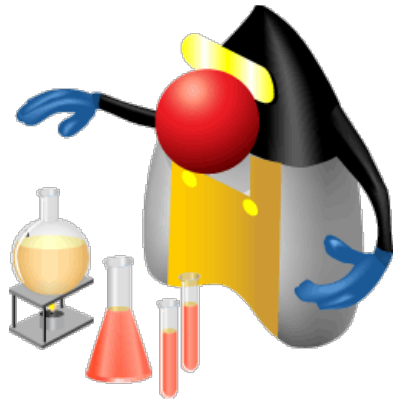
- Updating the banking application to use an interface
- Using interfaces to implement accounts



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.