
ORACLE
Academy

Java Programming

2-2

Java Class Design - Interfaces

ORACLE
Academy

Overview

- This lesson covers the following topics:
  - Model business problems using Java classes
  - Make classes immutable
  - Use Interfaces

Classes

- A Java class is a template/blueprint that defines the features of an object
- A class can be thought of as a category used to define groups of things
- Classes:
  - Declare fields
  - Define and implement methods
  - Implement methods from implemented interfaces

```
public class Dog {
   //instance field declarations
   private String name;
   private String breed;
   private String barkNoise = "Woof";
   private double weight;

}//end class Dog
```

Objects

- An object is an instance of a class
- A program may have many objects
- An object stores data in the fields to give it state
- This state will differentiate it from other objects of the same class

```
 public Dog(String name, String breed,
  String noise, double weight, String colour){
   super(breed, colour);
   this.name = name;
   barkNoise = noise;
   this.weight = weight;
}//end constructor method
```

Name	Bailey
Breed	Boerboel
Bark noise	arf-arf
weight	80.2

What Classes Can and Cannot Do

- Classes can be instantiated by:
  - A public or protected constructor
  - A public or protected static method or nested class

- Classes cannot:
  - Override inherited methods when the method is final

Immutable Objects

- Immutable objects have a number of advantages in certain circumstances
  - As they are immutable then we know their state cannot be changed which means they are always consistent
  - Making a class final does not on its own make it immutable, but it does stop it being subclassed and its methods overridden
  - Eliminating any methods that change instance variables would be required to make an object truly immutable

When Classes Can be Subclassed or Made Immutable

- A class can be subclassed when:
  - The class is not declared final
  - The methods are public or protected
- Strategy for making a class immutable:
  - Make it final
  - Limit instantiation to the class constructors
  - Eliminate any methods that change instance fields
  - Make all fields final and private

Immutable Using Final

- Declaring a class as final means that it cannot be extended
- Example: You may have a class that has a method to allow users to login by using some secure call
  - You would not want someone to later extend it and remove the security

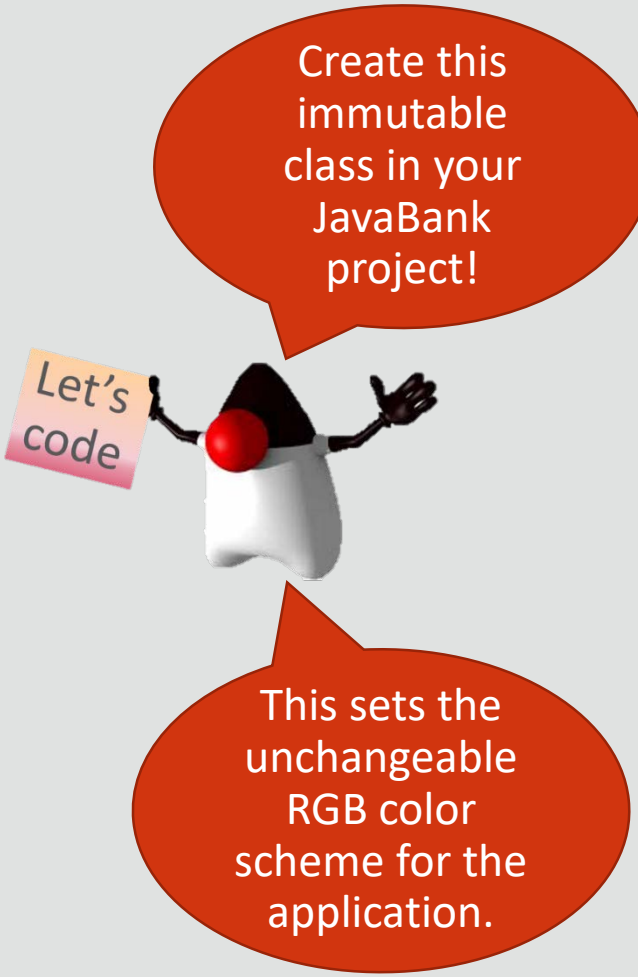
```
public final class ImmutableClass {
  public static boolean logOn(String username, String password) {
    //call to private boolean method
    return someSecureAuthentication(username,password);
  }//end method logOn
    .
    .
}//end class ImmutableClass
```

Immutable by Limiting Instantiation to the Class Constructor

- By removing any method that changes instance fields and limiting their setting to the constructor, the class fields will automatically be made immutable
- Example: When an instance of the ImmutableClass is created, the immutableInt field cannot be changed

```
public final class ImmutableClass {
  private final int immutableInt;

  public ImmutableClass (int mutableIntIn) {
     immutableInt = mutableIntIn;
  }//end constructor method
  private int getImutableInt() {
     return immutableInt;
  }//end method getImutableInt

}//end class ImmutableClass
```

Immutable Objects Task




```
public final class CompanyColor {

   private final int R = 238;
   private final int G = 130;
   private final int B = 238;

   //this class uses the default
   //constructor provided by Java

   public int getR() {
      return R;
   }//end method getR

   public int getG() {
      return G;
   }//end method getG

   public int getB() {
      return B;
   }//end method getB

}//end class CompanyColor
```

ORACLE Academy

Immutable Objects Task

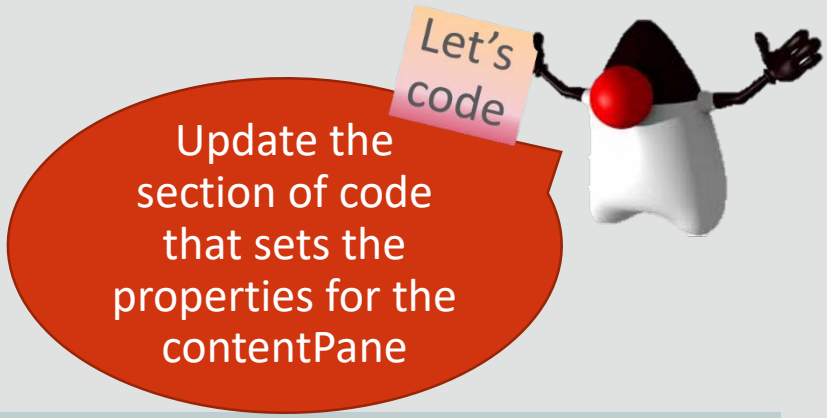
Update the Java Bank class as follows:

```
public class JavaBank extends JFrame {

   private static final long serialVersionUID = 1L;
      // Make these variables publicly available
      public String name;
      public int accountNum;
      public int balance;
      CompanyColor companyColor = new CompanyColor();
      private Color myColor = new
        Color(companyColor.getR(),companyColor.getG(),companyColor.getB());

      // JPanel for user inputs
      private JPanel inputDetailJPanel;
         .
         .
         .
         .
}//end class JavaBank
```

Create a new object based on the immutable class

Create a new Color object and set the RGB values based on the immutable class fields.

Immutable Objects Task




```
// create and position GUI components; register event handlers
private void createUserInterface() {
   // get content pane for attaching GUI components
   Container contentPane = getContentPane();
   contentPane.setBackground(myColor);
```

Sets the color of the contentPane based on the RGB values provided.

Immutable Objects Task

Update the section of code that sets the properties for the inputDetailJPanel

```
// create and position GUI components;
// register event handlers
private void createUserInterface() {
   // get content pane for attaching GUI components
   Container contentPane = getContentPane();
   contentPane.setBackground(myColor);
   // enable explicit positioning of GUI components
   contentPane.setLayout(null);

   // set up inputDetailJPanel
   inputDetailJPanel = new JPanel();
   inputDetailJPanel.setBounds(16, 16, 208, 250);
   inputDetailJPanel.setBorder(new TitledBorder("Input Details"));
   inputDetailJPanel.setLayout(null);

   inputDetailJPanel.setBackground(myColor);

   contentPane.add(inputDetailJPanel);
```

Sets the color of the panel based on the RGB values provided.

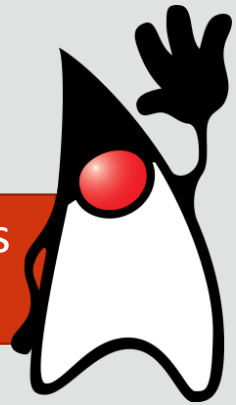
Interface

- An interface is a Java construct that helps define the roles that an object must assume
- You create an interface using the interface keyword instead of the class keyword
- An interface looks like a class with abstract methods (no implementation code for the method), but we cannot create an instance of it
- An interface does not have a constructor method

Interface

- It is implemented by a class (using the keyword implements) or extended by another interface
- Interfaces define collections of related methods without providing the implementation
- All methods in a Java interface are abstract

Implements is a keyword in Java that is used when a class inherits an interface.



Why Use Interfaces

- When implementing a class from an interface we force it to implement all the abstract methods
  - The interface forces separation of what a class can do, to how it actually does it
  - A programmer can change how something is done at any point, without changing the function of the class
  - This facilitates the idea of polymorphism as the methods described in the interface will be implemented by all classes that implement the interface

From Java 8 onwards you have been able to add default methods that do not have to be implemented by classes implementing the interface.

What An Interface Can Do

- An interface:
  - Can declare public constants
  - Define methods without implementation
  - Can only refer to its constants and defined methods
  - Can be used with the instanceof operator

The points above are key to help you understand what an interface can offer.
The instanceof operator compares an object to a specific type, this will be expanded on later.

What An Interface Can Do

- While a class can only inherit from a single superclass

```
public class ClassName extends Superclass {
   //class implementation
}//end class ClassName
```

- A class can implement from one interface

```
public class ClassName implements InterfaceName {
   //class implementation
}//end class ClassName
```

- A class can implement from more than one interface

```
public class ClassName implements InterfaceName, InterfaceName2 {
   //class implementation
}//end class ClassName
```

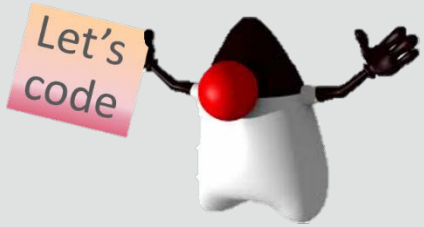
Interface Method

- An interface method:
- Each method is public even when you forget to declare it as public

```
void getName();
```
is equivalent to
```
public void getName();
```
in an interface.

- Is implicitly abstract but you can also use the abstract keyword

Declaring an Interface Example



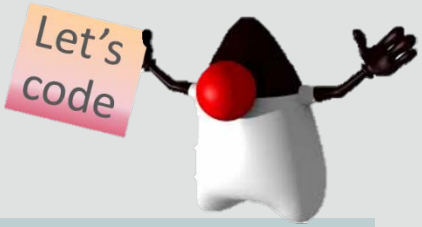
1. To declare a class as an interface you must replace the keyword class with the keyword interface
2. This will declare your interface and force all methods to be abstract and make the default access modifier public

Replace class with interface.

```
abstract interface InterfaceBankAccount
{
   public final String BANK= "JavaBank";

   public void deposit(int amt);
   public void withdraw(int amt);
   public int getBalance();
   public String getBankName();
}//end interface InterfaceBankAccount
```

Create this interface in your JavaBank project!

Implementing an Interface Example




```
public class Account implements InterfaceBankAccount{
   public Account() {
   }//end constructor method
   public void deposit(int amt)
   { /* deposit code */ }
   public void withdraw(int amt)
   {/* withdraw code */ }
   public int getBalance()
   { /* getBalance code */ }
   public String getBankName() {
      return InterfaceBankAccount.BANK;
   }//end method getBankName
}//end class Account
```

Implement the interface in your Account class!

Classes that extend an interface have to provide working methods for the methods defined in the interface.

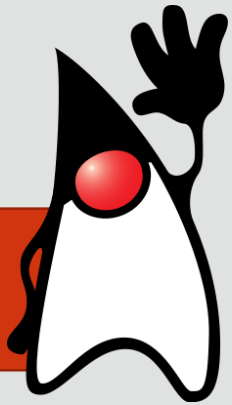
Luckily you only have to add the getBankName() method here!

Bank Example

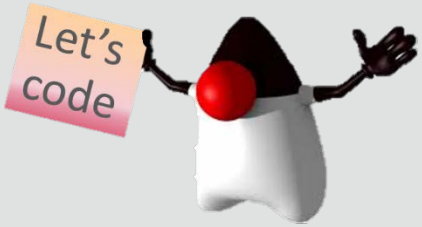
- The keyword final means that the field **BANK** is a constant in the interface
- Only constants and method stubs can be defined in the interface

```
public final String BANK= "JavaBank";
```

Classes implementing an interface can access the interface constants.



Bank Example



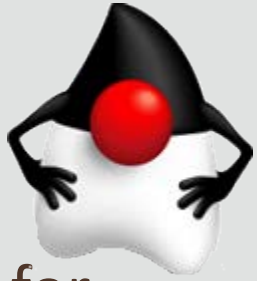
3. Accessing a constant from an interface uses the same dot notation syntax as accessing a static field:
   - InterfaceName.fieldName

```
public String getBankName() {
   return InterfaceBankAccount.BANK;
}//end method getBankName
```

```
public void print()
{
   System.out.println(getBankName()
                      + " " + accountName
                      + " " + accountNum
                      + " " + balance);
}//end method print;
```

Update the print statement in Account to include the bank name from the interface!

Bank Example Task



a) Update the print statement to include identifiers for each field so that the output looks like this:

```
Bank Name           : JavaBank
Account Holder      : Sanjay Gupta
Account Number      : 11556
Account balance     : 300
```

b) Do the same in the JavaBank class so that the text Area in the application produces the same output

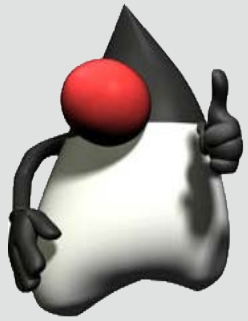
c) Create a private method called displayAccountDetails that displays the values to the Text Area

d) Use this method to display the individual account details

Bank Example Task Suggested Solution

a) Update the print statement to include identifiers for each field so that the output looks like this:

```
Bank Name           : JavaBank
Account Holder      : Sanjay Gupta
Account Number      : 11556
Account balance     : 300
```



```
public void print()
{
   System.out.println("\nBank Name      : " + getBankName() +
                      "\nAccount Holder : " + accountName +
                      "\nAccount Number : " + accountNum +
                      "\nAccount balance: " + balance);
}//end method print
```

Bank Example Task Suggested Solution

b) Do the same in the JavaBank class so that the text Area in the application produces the same output

```
public class JavaBank extends JFrame {
        . //additional class code
        .
   for (int i=0; i<noAccounts; i++) {
      displayJTextArea.setText(
            "Bank Name      : " + myAccounts[i].getBankName()
         + "\nAccount Holder : " + myAccounts[i].getAccountName()
         + "\nAccount Number : " + myAccounts[i].getAccountNum()
         + "\nAccount balance: " + myAccounts[i].getBalance());
          .
          . //additional class code
          .
}//end class JavaBank
```

Bank Example Task Suggested Solution

c) Create a private method displayAccountDetails that displays the values to the Text Area, use this method to display the individual account details

```
private void displayAccountDetails(String bName, String aName,
                                   int aNum, int aBal){
   displayJTextArea.setText( "Bank Name      : " + bName
                           + "\nAccount Holder : " + aName
                           + "\nAccount Number : " + aNum
                           + "\nAccount balance: " + aBal);
    }//end method displayAccountDetails
```

d) Method Call

```
displayAccountDetails(myAccounts[i].getBankName(),
                      myAccounts[i].getAccountName(),
                      myAccounts[i].getAccountNum(),
                      myAccounts[i].getBalance());
```

Why use interfaces with Bank Example?

- You may be wondering why you would want to create a class that has no implementation
- In the bank example we would know that all classes that implement the interface **InterfaceBankAccount** must have methods for deposit, withdraw, getBalance and getBankName
- Classes can only have one superclass, but can implement multiple interfaces
- We know if the ParentClass implements an interface then all the methods from the interface are defined at all levels

A Store Example

- A store owner wants to create a website that displays all items in the store
- We know:
  - Each item has a name
  - Each item has a price
  - Each item is organized by department
- It would be in the store owner's best interest to create an interface for what defines an item
- This will serve as the blueprints for all items in the store, requiring all items to at least have the above defined qualities

Adding a New Item to the Store Example

- The owner adds a new item to his store named cookie:
  - Each cookie costs between 1 and 3 US dollars
  - Cookies can be found in the Bakery department
  - Each cookie is identified by a type
- The owner may create a Cookie class that implements the Item interface such as shown on the next slide, adding methods or fields that are specific to cookies

ORACLE Academy

Item Interface

- Possible Item interface

```
public interface Item {
  public String getItemName();
  public double getPrice();
  public void setPrice(double price);
  public String getDepartment();

}//end interface Item
```

- We now force any class or interface that implements the Item interface to implement the methods defined within it

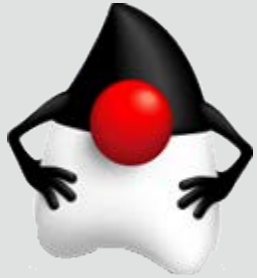
Remember the public keyword could be left out as the accessibility from an interface defaults to this.

Create Cookie Class

- The owner may create a Cookie class that implements the Item interface, adding methods specific to cookie items

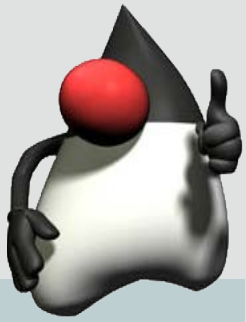
```
public class Cookie implements Item{
  public String cookieType;
  private double price;

  public Cookie(String type, double price){
     cookieType = type;
     this.price = price;
  }//end constructor method

  public String getItemName() { return "Cookie";}
  public double getPrice() {return price;}
  public void setPrice(double price){this.price = price;}
  public String getDepartment() {return "Bakery";}
  public String getType() {return cookieType;}
}//end class Cookie
```

Cookie Shop Example Task



a) Create a bakery project
b) Create the Item interface as shown in the slide
c) Create the Cookie class that implements the Item interface as shown in the slide. Add the getType() method as well as the methods defined in the interface
d) Create a driver class called BakeryDriver that will create multiple Cookie objects and test the Cookie methods by displaying values to the console

Bank Example Task Suggested Solution



```
package bakery;

public class BakeryDriver {
   public static void main(String[] args) {
      Cookie cookie = new Cookie("Choc Chip", 1);

      System.out.println(cookie.getDepartment());
      System.out.println(cookie.getItemName());
      System.out.println(cookie.getType());
      System.out.println(cookie.getPrice());
      cookie.setPrice(1.5);
      System.out.println(cookie.getPrice());

   }//end method main
}//end class BakeryDriver
```

Terminology

- Key terms used in this lesson included:
  - Class
  - Object
  - Immutable
  - Interface
  - implements

Summary

- In this lesson, you should have learned how to:
  - Model business problems using Java classes
  - Make classes immutable
  - Use Interfaces


ORACLE
Academy