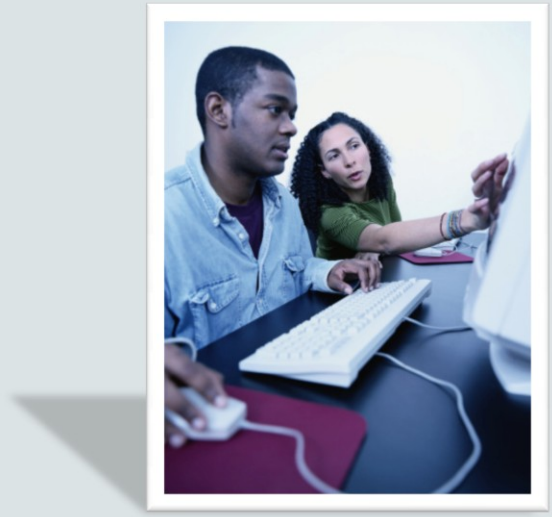
Release Date: August, 2015

Updates:

Java Programming

2-1
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 are the foundation of your programs. Without a solid foundation then you will increase the difficulty in producing good quality and maintainable software.

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:
  - Class variables
  - Define and implement methods.
  - Implement methods from implemented interfaces.

When we create a class we aim to have it model one purpose. This makes changes in the future a lot easier to incorporate.

Objects

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

An object is when a class comes to life by giving it state. State is when we start to add values to its fields.

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.

Most classes will have a public constructor, but there are some cases where you can have a private constructor. Perhaps your class only contains constants, or it contains only static helper methods.

The Singleton design pattern is when only one instance of your class can ever exist. This can be done by moving the initialization to a method.

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 variables.
  - Make all fields final and private

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 overwritten. We would also have to use other strategies such as eliminating any methods that change instance variables.

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 public boolean someSecureAuthentication(username,password);
         return someSecureAuthentication(username,password);
         }
}
```


Using final is the simplest way to make a class immutable.

Immutable by Limiting Instantiation to the Class Constructor

- By removing any method that changes instance variables and limiting their setting to the constructor, the class variables will be made immutable.
- Example: When we create an instance of the ImmutableClass, the immutableInt variable cannot be changed.

```
public final class ImmutableClass {
  private final int immutableInt;
    public ImmutableClass (int mutableIntIn) {
        immutableInt = mutableIntIn;
    }
    private int getImutableInt() {
    return immutableInt;
  }
}
```


Once we create a new instance of our class, then the value can never be overwritten.

```
ImmutableClass c = new ImmutableClass(5);
```

We have no method to set the private class variable immutableInt.

Interface

- An interface is a Java construct that helps define the roles that an object can assume.
- It is implemented by a class or extended by another interface.
- An interface looks like a class with abstract methods (no implementation), but we cannot create an instance of it.
- Interfaces define collections of related methods without implementations.
- All methods in a Java interface are abstract.

It may not be clear at this point why you would like to create an interface. We will explore this feature throughout the course.

Why Use Interfaces

- When implementing a class from an interface we force it to implement all of the abstract methods.
- The interface forces separation of what a class can do, to how it actually does it.
- So 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 will be able to add default methods that do not have to be implemented on classes that implement 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.

It is important to remember the points above. They are key to help you understand what an interface can offer.

The instanceof operator compares an object to a specific type. We will expand on this later.

What An Interface Can Do

- While a class can only inherit a single superclass, a class can implement more than one interface.

```
public class className implements interfaceName{
...class implementation...}
```

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

- Example of SavingsAccount implementing two interfaces – Account and Transactionlog

```
public class SavingsAccount implements Account,Transactionlog{
...class implementation...}
```


Using multiple interfaces gives us greater flexibility in our design. During the course you will see some of the places that java uses interfaces to expand its functionality.

Interface Method

- An interface method:
  - Each method is public even when you forget to declare it as public.
  - Is implicitly abstract but you can also use the abstract keyword.


```
public void getName();
```

Is equivalent to

```
void getName();
```

in an interface.

Declaring an Interface

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

```
public interface InterfaceBankAccount
{
   public final String bank= "JavaBank";
   public void deposit(int amt);
   public void withdraw(int amt);
   public int getbalance();
}
```

Replace class with interface.

We have called our Interface InterfaceBankAccount. Normally we wouldn't use the interface word within the name, but simplification of identifying the class and its type in this example we have.

In JavaBank we could modify the Account class to use the interface InterfaceBankAccount. This would force it to implement the methods deposit, withdraw and getbalance.

Every other class that was based on this interface would also be forced to implement these methods. Thus creating a contract between the interface and the class.

Bank Example

```
public interface InterfaceBankAccount
{
  public final String bank= "JavaBank";
  public void deposit(int amt);
  public void withdraw(int amt);
  public int getbalance();
}
```

Classes that extend InterfaceBankAccount will have to provide working methods for methods defined here.

- Class implementing the BankAccount interface:

```
public class Account implements InterfaceBankAccount{
  private String bankname;
  public Account() {
    this.bankname = InterfaceBankAccount.bank; }
  public void deposit(int amt) { /* deposit code */ }
  public void withdraw(int amt) {/* withdraw code */ }
  public int getbalance() { /* getBalance code */ }
```

Classes implementing an interface can access the interface constants.

Class fields have been omitted. The example would contain other class fields.

Bank Example Explained

- The keyword final means that the variable bank is a constant in the interface because you can only define constants and method stubs here.

```
this.bankname = InterfaceBankAccount.bank
```

- The assignment of the constant from an interface uses the same syntax that you use when assigning a static variable to another variable.

In our example the bankname defaults to the interface constant InterfaceBankAccount.bank string. We could set up another abstract method called setBankName() to change this at any point.

Bank Example Explained 2

- The interface defined 3 methods – deposit, withdraw and getbalance.
- These must be implemented in our class.

```
public class Account implements InterfaceBankAccount{
  private String bankname;
  int balance;
  public Account() {
      this.bankname = InterfaceBankAccount.bank; }
  public void deposit(int amt) {
      balance = balance + amt;
  }
  public void withdraw(int amt) {
      balance = balance – amt
  }
  public int getbalance() {
      return balance
  }
}
```


Class fields have been omitted. The example would contain other class variables.

There would also have to be checks on withdraws etc

```
if (balance > amt) {
  balance = balance – amt;
}
```

Why use interfaces with Bank Example?

- You may be wondering why you would want to create a class that has no implementation.
- One reason could be to force all classes that implement the interface to have specific qualities.
- In our bank example we would know that all classes that implement the interface InterfaceBankAccount must have methods for deposit, withdraw and getbalance.
- Classes can only have one superclass, but can implement multiple interfaces.

Knowing that classes that implement our interface must have defined the methods within it then we can make use of this in our code, especially in polymorphism.

```
ParentClass p = new ParentSubClass();
```

We know if the ParentClass implements an interface then all the methods in this are defined at all levels.

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 defined qualities above.

In the interface we should only define the common methods.

Adding a New Item to Store Example

- The owner adds a new item to his store named cookie:
  - Each costs 1 US dollar.
  - 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.

Item Interface

- Possible Item interface

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

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

Remember that we could leave the public keyword out, as its accessibility defaults to this.

Create Cookie Class

- The owner may create a Cookie class that implements the Item interface, such as shown below, adding a method or two specific to cookie items.

```
public class Cookie implements Item{
  public String cookieType;
  private int price;
  public Cookie(String type){
    cookieType = type;
    price = 1;
  }
  public String getItemName() { return "Cookie";}
  public int getPrice() {return price;}
  public void setPrice(int price){this.price = price;}
  public String getDepartment() {return "Bakery";}
  public String getType() {return cookieType;}
}
```


Notice we have added class fields like cookieType and an accessor getType() to return this value. Failure to implement any of the methods in the interface will mean the class will not compile.

Terminology

Key terms used in this lesson included:

- Immutable
- Interface

Can you define these terms?

Summary

In this lesson, you should have learned how to:

- Model business problems using Java classes
- Make classes immutable
- Use Interfaces

These are the objectives that you should have learned in this lesson.


ORACLE
ACADEMY