Practices for Lesson 11

Practices Overview

In these practices, you design and create a class hierarchy for the Employee Tracking System of the Marketing department of Duke's Choice Company. You also create an interface and implement it in the classes you created.

Practice 11-1: Creating and Using Superclasses and Subclasses

Overview

In this practice, you design and then create a class hierarchy that forms the basis for an Employee Tracking System in the Marketing department in the Duke's Choice company. This practice comprises two sections. In the first section, you create a simple design model for the class hierarchy. In the second section, you create the actual classes and test them.

Assumptions

This practice assumes that the following file appears in the practice folder for this lesson, `Lesson11`:

- EmployeeTest.java

Design the Class Hierarchy

In this section, you design subclasses and superclasses using the information in the following paragraphs.

The Marketing department of the Duke's Choice company has employees in several different positions. Some of these positions are: Technical Writers, Graphic Illustrators, Managers, and Editors.

Marketing wants you to create a program for tracking information about each of its workers. This information consists of: the worker's name, job title, employee ID, and level (1, 2, or 3). Additionally:

- Managers must have a list of employees that they manage.
- Technical Writers, Graphic Illustrators, and Editors must have a list of skills that they possess.
- Editors must have a value indicating whether they prefer to do electronic editing or paper-based editing.
- There must be a means by which to display all the information for a given employee type.

1. Create a class hierarchy of superclass and subclass relationships for the employees of the Marketing department. Draw the diagram on a piece of paper. Or, if you prefer, you may use the UMLet tool on your desktop.

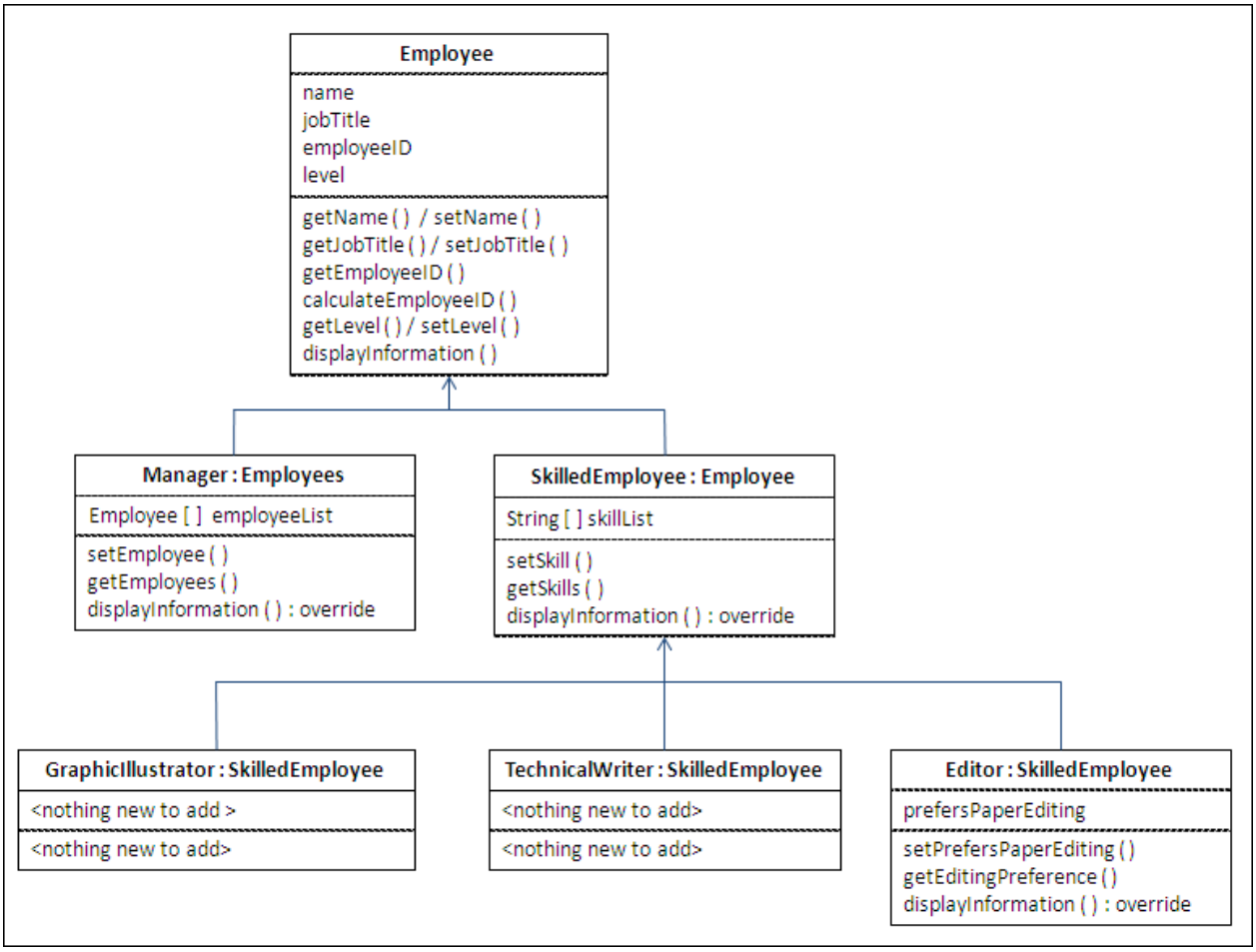
   Hints

   - **Use the "is a" phrase:** Ask yourself if all or many of the job types have some of the same attributes (fields) and operations (methods). For instance, all of the different job types mentioned above can also be called Employees (in the general sense). They share certain fields and operations. Therefore, a Manager "is a(n)" Employee. An Editor has an "is a(n)" Employee.
   - **Consider an interim superclass:** If you find that certain employee types share common fields and/or operations that are *not* shared by other employee types (for instance a list of skills), yet they are all "Employees", consider creating a common superclass for these employees: inherited from the top level superclass: Employee.
   - **Displaying information:** Remember that many of the fields that would be displayed are shared in common by all these employees (for instance: name, job title, employeeID). You might be able to display this common information from the top level superclass. In the subclass, simply "add to" what was displayed by the superclass, showing the fields that are unique to this particular employee type.

- **Note:** This is done by overriding the method from the superclass and calling the overridden method from within the subclass method, which then adds more code to display additional fields.

- **Encapsulation:** Demonstrate encapsulation for each of the classes in your design by including `get` and `set` methods for each private field, according to the type of access required.
- **Modeling:** Model the class hierarchy using class diagrams similar to those you saw in this lesson.

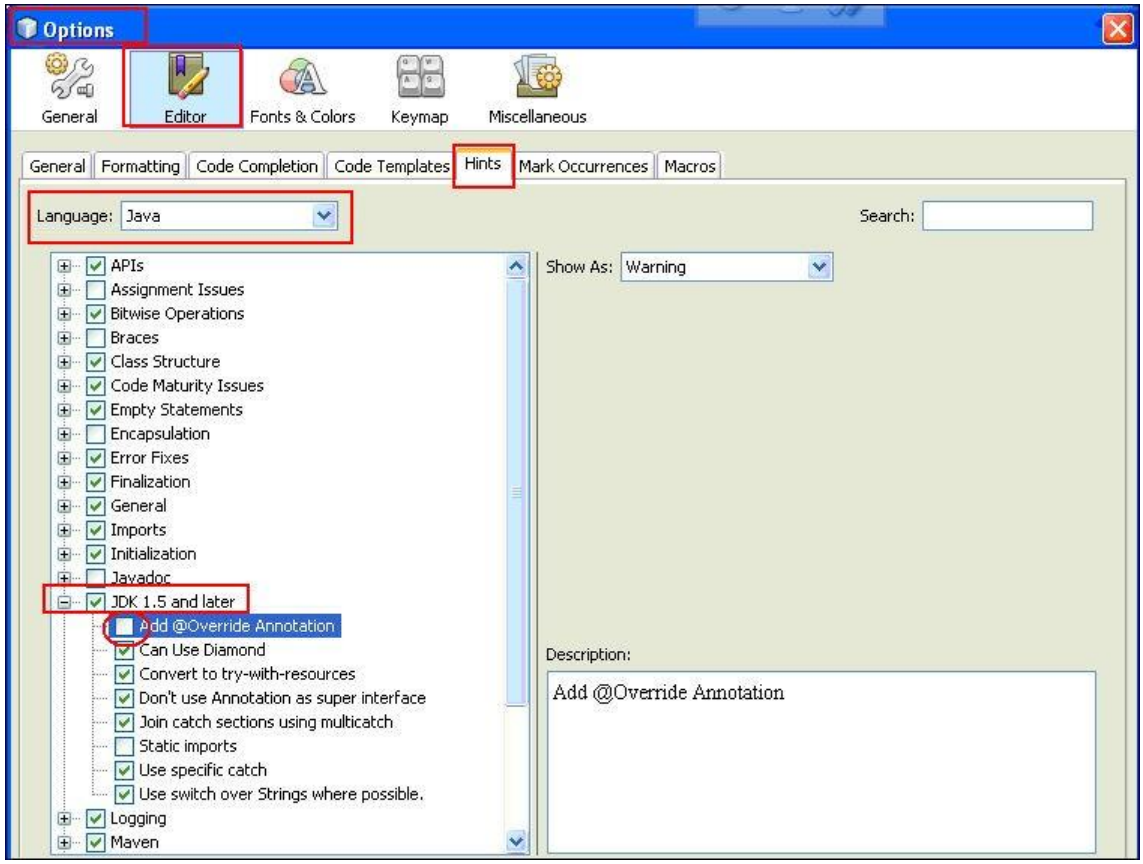
Solution:



Create the Classes

2. In NetBeans, create a new project from existing sources called Practice12. Set the Source Package Folder to point to `D:\labs\les12`. Remember to set the Binary Source Format property of the project. If you need further details, refer to Practice 2-2, Steps 3 and 4.
3. Before you begin creating the classes, change a property of the NetBeans IDE. The **Add @Override Annotation** property of the editor is useful when you are creating javadocs for your application. This property is applied when you override a method in the superclass. Since we are not creating javadocs in this course, you turn off this property as it is merely distracting for our purposes. Follow the steps below to make this change:
   a. Select Tools > Options from the main menu.

b. In the Options window, click the **Editor** toolbar button and then click the **Hints** tab.
c. Change the **Language** to **Java**. The hints in the left column change accordingly.
d. Expand the **JDK 1.5 and later** node. Beneath this node, deselect **Add @Override Annotation**.
e. Click **OK** to save the change and close the Options window.



4. Create the Employee class shown in the diagram above. The following steps provide more details.
   a. All of the fields shown in the diagram should be private. Be sure to follow the same naming pattern that you have been using (camelCase).
   b. Use the Refactor feature of NetBeans to encapsulate these fields (create `get` methods for each field and `set` methods for each field). Change the access modifier for the `setEmployeeID` method to `private`.
      - **Note:** Employee IDs are *calculated* to ensure uniqueness, and you must restrict public *write* access to this field so that the IDs are always unique. ID values are only set by the `calculateEmployeeID` method.
   c. Add another field, not shown in the diagram, called `employeeIDCounter`. Make it a `protected static int` field and initialize it to 0 (zero).

Note

A `static` field is a "class" field. There is only one value for this field that is shared by all instances of this class. The `static` field is used here to store an integer value that is incremented from within the `calculateEmployeeID` method to generate the next ID value. The `employeeIDCounter` is accessed and incremented by all instances of the

`Employee` and its subclasses, thus ensuring that no duplicate employee IDs are generated.

In a real business application, this technique would not be robust enough to guarantee unique IDs. Instead, a database would probably generate the IDs. However, this technique suffices for our simple application.

d. Create the `calculateEmployeeID` method. It takes no arguments and does not return a value. In the body of this method, increment the `employeeIDCounter` and then set the new value in the `employeeID` field (use the `set` method of the field).

e. Create the `displayInformation` method. It takes no arguments and does not return a value. In this method, print out the value of each field of the class with a suitable label.

Solution:

```
public class Employee {
    protected static int employeeIDCounter = 0;
    private int employeeID;
    private String name;
    private String jobTitle;
    private int level;

    public void calculateEmployeeID() {
          employeeIDCounter++; // inc so employeeID is unique
          setEmployeeID(employeeIDCounter);
    }

    public void displayInformation() {
          System.out.println("Name: " + getName());
          System.out.println("Job Title:" + getJobTitle());
          System.out.println("Employee ID: " +
             getEmployeeID());
          System.out.println("Level: " + getLevel());
    }

    // The set and get methods are not shown here
}
```

f. Click Save to compile the class.

5. Create the Manager class from the diagram. The steps below provide more details.
   a. After creating the new Java Class file, add the following phrase (shown in bold below) to the class declaration to indicate that it is a subclass of Employee:

      ```
      public class Manager extends Employee {
      ```

   b. Declare and instantiate the `employeeList` field as a private ArrayList (instead of the array of type `Employee` that is indicated in the diagram). This is simpler to work with than an array.

      ```
      private ArrayList employeeList = new ArrayList();
      ```

   c. Add the necessary `import` statement to import the `java.util.ArrayList` class.

Hint: Click on the error icon in the left margin and let NetBeans add the import statement for you.

   d. Add a public `setEmployee` method to add a single employee to the `employeeList`. The method takes an argument of type `Employee`. Use the `add` method of the `ArrayList` to add the `Employee` object to the `employeeList` object.

      ```
      public void setEmployee(Employee emp){
          employeeList.add(emp);
      }
      ```

      Question: What validation might you need to do in this method in a real-world application?

   e. Add a public `getEmployees` method that simply returns the `employeeList`.

      ```
      public ArrayList getEmployees(){
          return employeeList;
      }
      ```

   f. Add a `displayInformation` method to override the method in the Employee class. In this method, you invoke the `displayInformation` method in the superclass and then display additional information specific to the `Manager` class.
      i. Declare the method with the exact same signature as in the superclass method (returning void and accepting no arguments). NetBeans displays a green circle icon in the margin as you have finished typing the method declaration. This indicates that this method overrides the superclass method. Clicking the green circle opens the Employee class in the editor to show you the ancestor method. This can be helpful sometimes.
      ii. In the method block, invoke the superclass method using the `super` keyword as a reference to the Employee class.
      iii. Display the following message: "Manager has the following employees: "
      iv. Now iterate through the `employeeList` using an enhanced `for` loop. Remember that the `employeeList` is an `ArrayList` that holds `Objects`. The compiler does not know that these `Objects` happen to be Employee objects. Therefore, to get the `name` field from each object to display it, you have to cast the `Object` to an `Employee` (an Employee "is a(n)" Object). Declare a local variable at the top of this method of type `Employee`. This holds the cast value. The code for this method is provided for you here:

```
public void displayInformation(){
     Employee emp;
     // Invoke the ancestor method
     super.displayInformation();
     System.out.println
       ("The manager has the following employees: ");

     for(Object obj : employeeList){
          // Cast the object as an Employee
          emp = (Employee)obj;
          // print the name, indented by a tab
          System.out.println("\t" + emp.getName());
     }
}
```

g. Save and compile your program.

6. Create the SkilledEmployee class from the diagram. This class should also extend Employee.
   a. Use an `ArrayList` instead of a `String` array when you declare the `skillList` field. Instantiate the field to an empty `ArrayList`.

      ```
      private ArrayList skillList = new ArrayList();
      ```

   b. Add the necessary `import` statement to import for the `ArrayList` class.
   c. Add a public `setSkill` method to add a single skill to the `skillList`. The method takes an argument of type `String`. Use the `add` method of the `ArrayList` to add the `String` to the `skillList` object.
   d. Add a public `getSkills` method that returns the `skillList`.
   e. Override the `displayInformation` method as you did in the Manager class. After invoking the superclass method, display the following message: "Employee has the following skills: ". Iterate through the `skillList` using an enhanced `for` loop, displaying each skill, indented by a tab as you did in the `Manager` class.
      - **Note:** The `skillList` object contains `String` objects. In this case, you can directly print the `Object` reference from the `ArrayList` without casting it to a `String`. The reason for this is that every `Object` has a `toString` method and the `println` method invokes this for you, resulting in the display of the `String` value (i.e. the skill).

f. Click Save to compile the program.
   - **Note:** Consult the solution file for the SkilledEmployee class if you need help.

7. Create the Editor class as a subclass of SkilledEmployee.
   a. Declare the `prefersPaperEditing` field as a `private boolean`. (It is initialized to a default value of false.)
   b. Add a `setPrefersPaperEditing` method that takes a `boolean` argument and returns `void`. Assign the argument to the private field.
   c. Add a `getEditingPreference` method that returns a `String` value. Use an `if/else` construct to check the value of `prefersPaperEditing` and set the return value to either "Paper" or "Electronic".
   d. Override the `displayInformation` method as you did in the Manager class, invoking the superclass method first and then displaying the return value of `this.getEditingPreference()` with a suitable label.
   e. Click Save to compile the program.
      - **Note:** Consult the solution file for the Editor class if you need help.
8. Create the remaining two classes from the diagram: GraphicIllustrator and TechnicalWriter. Both of these classes extend the SkilledEmployee class. It is not necessary to add any additional fields or methods, nor is it necessary to override the `displayInformation` method.
9. Save and compile the program.
10. Open the EmployeeTest class in the editor and examine the code.

    Note

    If there are any error indicators, check to make sure that you have spelled all of your method names the same way they are spelled in this class. If there are still error indicators after making any changes, try clicking the Save button again and/or try just clicking on a line in EmployeeTest that indicates an error. This reminds the syntax checker in NetBeans to try resolving the references once more.
11. Run the EmployeeTest class to test your program. You should see an output similar to the following screenshot:

```
Name: Fred Hanson
Job Title:Editor
Employee ID: 1
Level: 1
Employee has the following skills:
        technical editing
        typing
Editing preference: Paper
**** *****
Name: Frank Moses
Job Title:Graphic Illustrator
Employee ID: 2
Level: 3
Employee has the following skills:
        technical illustration
        video production
        media authoring
**** *****
Name: James Ralph
Job Title:Technical Writer
Employee ID: 3
Level: 1
Employee has the following skills:
        technical writing
**** *****
Name: Susan Smith
Job Title:Manager
Employee ID: 4
Level: 2
Manager has the following employees:
        Fred Hanson
        Frank Moses
        James Ralph
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practice 11-2: Using a Java Interface

Overview

In this practice, you create an interface called Printable and implement it within the class hierarchy that you built in Practice 12-1. You also examine and run another small application that uses the same Printable interface to better understand the benefits of using interfaces.

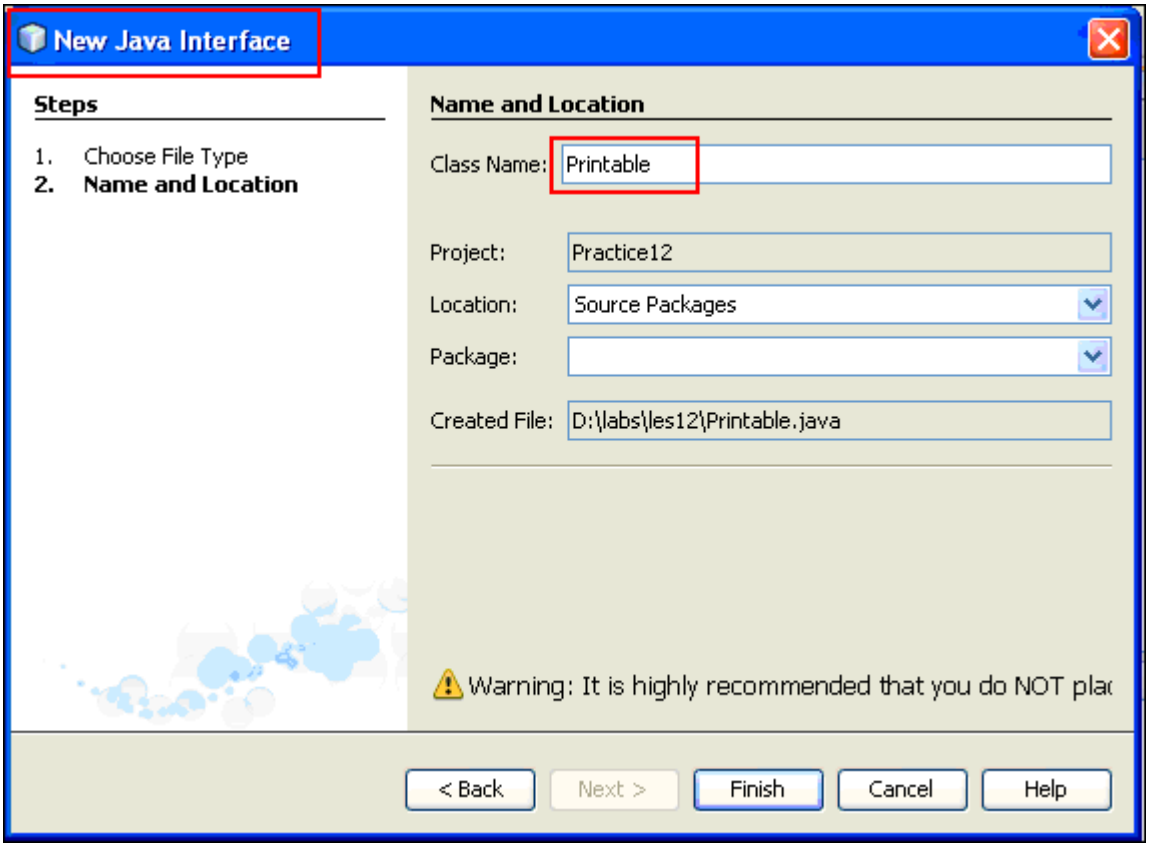
Assumptions

This practice assumes that the following files appear in the practice folder for this lesson, `Lesson11`:

- Printer.java
- Country.java
- Region.java
- Company.java
- CompanyTest.java

Tasks

1. Create a new Java Interface using the NetBeans **File** wizard.
   a. Right click Practice12 in the Projects window.
   b. Select **New > Java Interface** from the popup menu.
   c. Enter `Printable` in the Class Name field as shown below.



   d. Click **Finish**.

2. In the Printable interface, declare a `public abstract` method called `print`. It should return `void` and accept zero arguments.

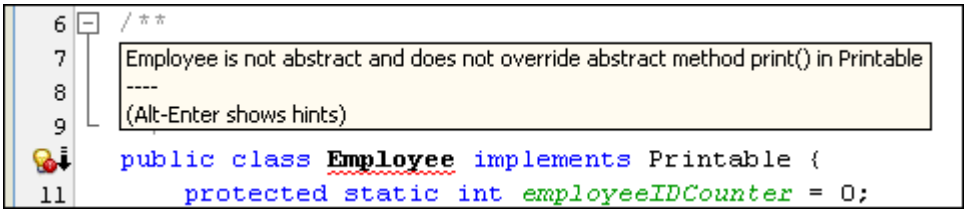
```
public abstract void print();
```

3. Click Save.
4. Implement the Printable interface in the Employee class.

   Note: Remember that all of the other classes in this hierarchy are subclasses of Employee, therefore, they also now implement Printable through inheritance.

```
public class Employee implements Printable {
```

5. The syntax checker now shows an error icon in the margin of this line. Move your cursor over the error icon to see the potential compilation error that it recognizes.



   Explanation: Any non-abstract classes that implement an interface must also implement all of the abstract methods of the interface. In this case, the only abstract method in Printable is `print`.

6. Change name of the `displayInformation` method to `print`.
7. Make this same change (`displayInformation` to `print`) in each of the following classes to ensure that they also implement the `print` method. You also need to change the name of the superclass method called in the first line of the new `print` method. (It is no longer called `displayInformation`.)
   - Manager
   - SkilledEmployee
   - Editor
8. Open the Printer class in the editor and examine its only method: `printToScreen`. Notice that this method takes an argument of type `Printable`. Any class that implements Printable would be accepted as an argument. This method invokes the `print` method of the Printable object.

```
public void printToScreen(Printable p){
    p.print();
}
```

9. In the `main` method of the EmployeeTest class, make the following changes:
   - Declare and create an instance of the Printer class.
   - For *every* invocation of the `displayInformation` method, comment out the line and instead, invoke the `printToScreen` method of the `Printer` object. Pass in a reference to the Printable object as shown below:

```
//myManager.displayInformation();
myPrinter.printToScreen(myManager);
```

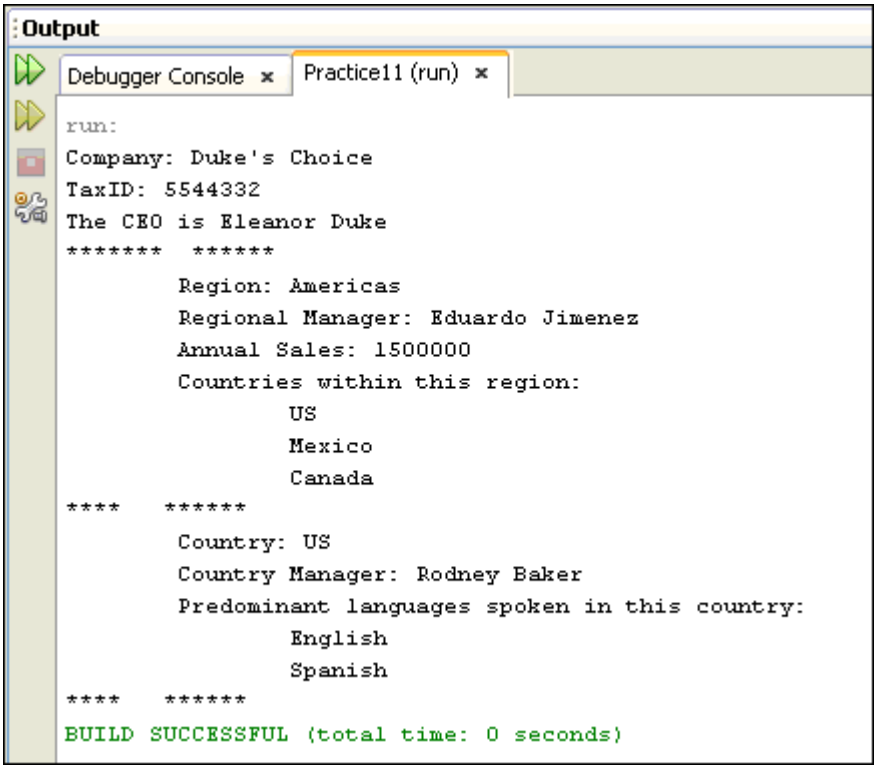
10. Save and compile your program.
11. Run the EmployeeTest class and examine the output. It should be identical to the output you saw before implementing the interface.

Discussion

One of the benefits of using interfaces is that you can abstract functionality that is used in different applications and different class hierarchies. This functionality is moved into the interface and can then be used anywhere that the functionality is required. For example, in this practice, the ability to display class fields with labels and formatting has been moved into the Printable interface.

Now you test the cross-application benefit by running a different application that also implements Printable. The Company class hierarchy displays information about Duke's Choice top level management, as well as that of its regional and divisional management. The code is very similar to what you saw in the Employee hierarchy.

12. Close all of the classes you have been working on and open the following classes in the editor:
    - Company
    - Region
    - Country
    - CompanyTest
13. Examine the Company class first. This is the superclass of Region and Country. Notice that it implements the same Printable interface that you used in the Employee hierarchy.
14. Examine the Region, Country and CompanyTest classes as well.
15. Run the CompanyTest class to view the output of this application.



16. Close the Practice12 project in NetBeans.

You have now had an introductory exposure to Java Interfaces, one of the most valuable tools of the Java language. This topic is covered in much more detail in the *Java SE7 Programming* class.