Lab Exercise: Abstract and Interfaces

INTERTECH

Instructors Who Consult. | Consultants Who Teach.

About Intertech

Thank you for choosing Intertech for your training. The next page of this document will get you started with your lab if you'd like to skip ahead. Below is a brief overview of Intertech as well as a promo code for you for future live Intertech trainings.

Intertech (www.intertech.com) is the largest combined software developer **training** and **consulting** firm in Minnesota. Our unique blend of training, consulting, and mentoring has empowered technology teams in small businesses, Fortune 500 corporations and government agencies since 1991.

Our training organization offers live in-classroom and online deliveries, private on-site deliveries, and on-demand options. We cover a broad spectrum of .NET, Java, Agile/Scrum, Web Development, and Mobile technologies. See more information on our training and search for courses by **clicking here**.

We appreciate you choosing Intertech!

Lab Exercise

Abstract and Interfaces

Abstract classes and interfaces extend the power of polymorphism in Java. They also prevent the construction of nonsensical objects and make sure classes adhere to expected functionality. In this lab, you continue to explore polymorphism in Java.

Specifically, in this lab you will:

- Modify the Good class to be abstract
- Create an interface that generically describes a good or service
- Use polymorphism with interfaces to allow Order objects to be associated with either a good or a service
- Optionally use static imports to simplify code

Scenario

Development of the Acme Order System continues to go well, but as typical in business, it has been determined that it must be expanded to handle orders for services, as well as products. Before adding services and allowing orders for services, you decide to fix a small issue with regard to the Good class to prevent an instance of Good from ever being created.

Step 1: Abstract Good

Currently, the Good class allows instances of a Good to be created. In actuality, this class serves as a template for the subclasses (Solid and Liquid), so instances of a Good should not be created. For instance, consider the volume method on Good. What is the volume of such a nebulous object?

1.1 Make Good abstract. In the Package Explorer view, double-click on the Good.java file in the com.acme.domain package to open the file in a Java editor. Use the "abstract" key word to make this class abstract.

```
public abstract class Good {
...
}
```

1.2 Make the volume( ) method abstract in Good. Remove the current implementation of the volume( ) method in Good, and replace it with an abstract definition (see below). Making this method abstract forces the Solid and Liquid classes to implement and override volume( ).

```
  public abstract double volume();
```

1.3 Save Good.java and run the TestOrders.java file to ensure your application still works the same way.

1.4 Try to create an instance of Good.

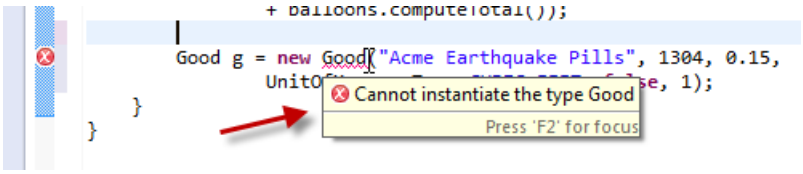
1.4.1 In the main method of TestOrders.java, try to create an instance of the now abstract Good.

```
Good g = new Good("Acme Earthquake Pills", 1304, 0.15,
UnitOfMeasureType.CUBIC_FEET, false, 1);
```

1.4.2 This will also require you import the Good class.

```
import com.acme.domain.Good;
```

1.4.3 You should get a compiler error that prevents an instance of Good from being created.

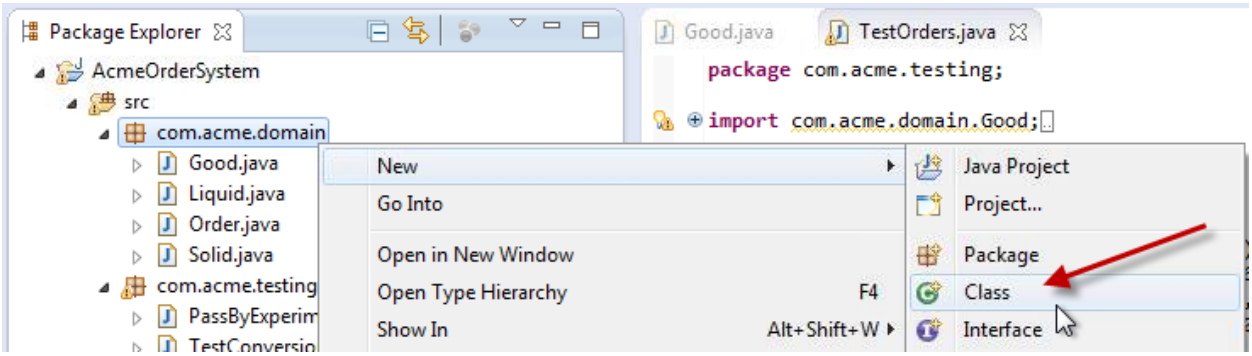


1.4.4 Remove these two lines of code from TestOrders.java.

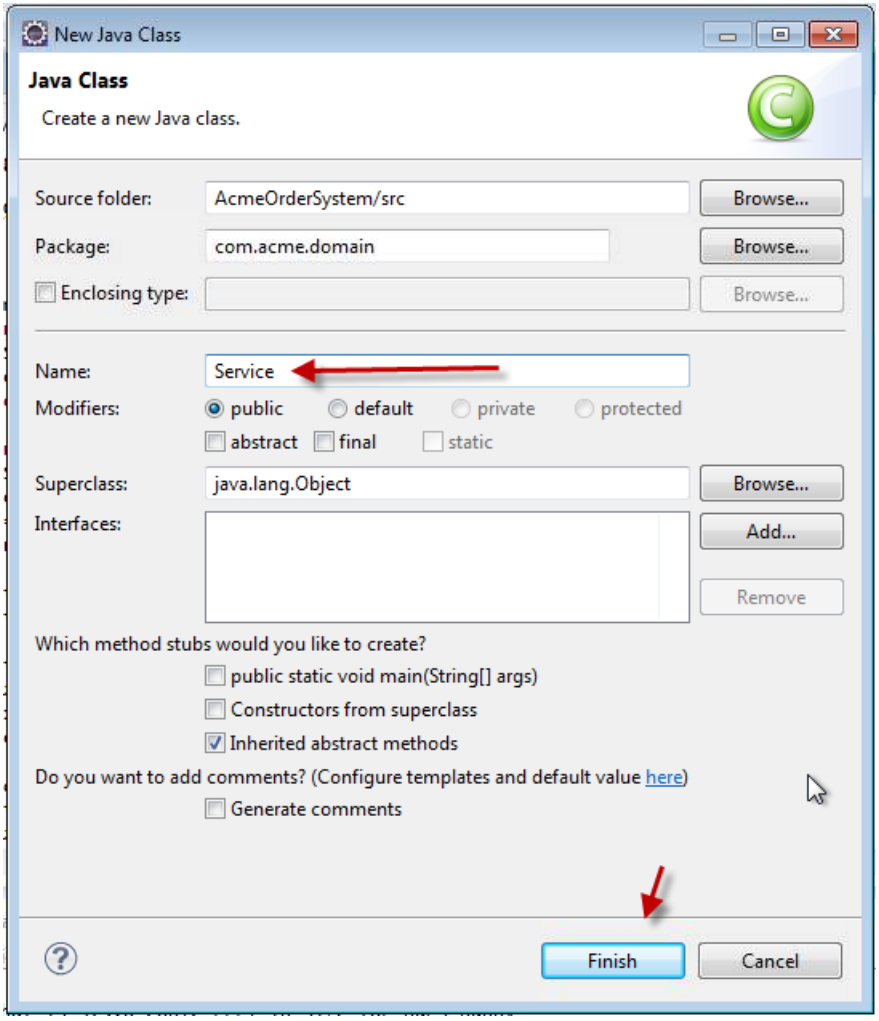
Step 2: Creating Services

In addition to its goods, like any large conglomerate, Acme also offers a number of services, such as Road Runner Eradication and Rocket Sled Assembly. An order can then be placed for a good or a service. In this step, you create a new Service class and an interface that both Goods and Services implement.

2.1 Create the Service Class. In the Package Explorer view, right-click on the com.acme.domain package in the AcmeOrderSystem project, and select ***New > Class***.



2.2 In the New Java Class window, enter Service as the new class name, and click the ***Finish*** button.



The Service class should not extend any class other than java.lang.Object.

2.3 In the Service.java editor view, add name, estimatedTaskDuration, and timeAndMaterial fields.

```
private String name;
private int estimatedTaskDuration;
private boolean timeAndMaterials;
```

2.4 Properly encapsulate these properties by generating public getter and setter methods for each.

2.5 Add a constructor to create a Service.

```
public Service(String n, int dur, boolean tAndM) {
  this.estimatedTaskDuration = dur;
  this.name = n;
  this.timeAndMaterials = tAndM;
}
```

2.6 Add a toString( ) method to Service. The toString( ) method for a Service should display the name and estimated duration for a Service.

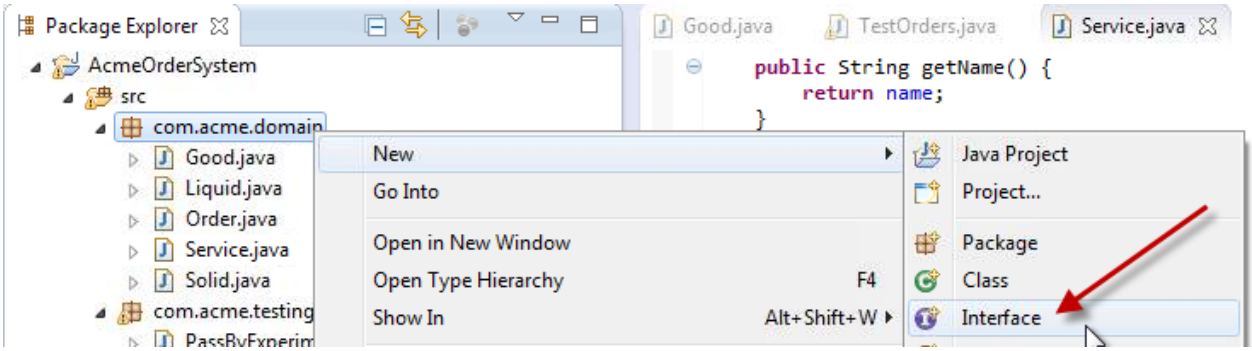
```
public String toString() {
  return name + "(a " + estimatedTaskDuration + " day service)";
}
```

2.7 Save the Service.java file once you have finished coding, and ensure there are no compiler errors.

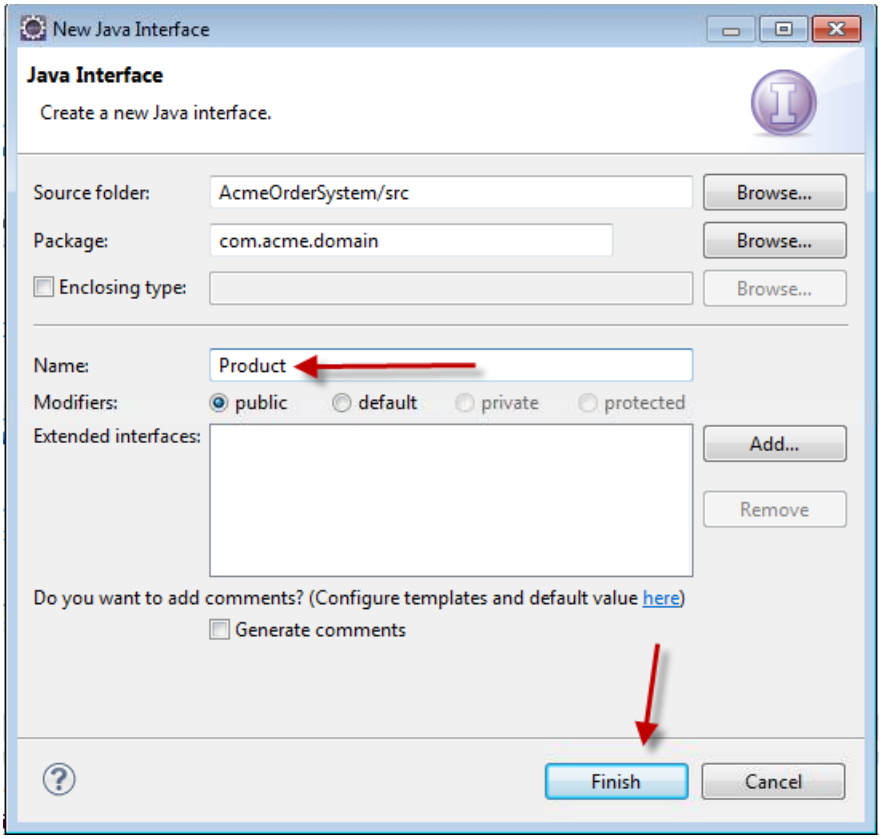
Step 3: Creating Product

Unfortunately, a Service is not a descendant of the abstract Good class. Therefore, an instance cannot be used as the "product" on an Order. To solve this, a new interface, Product, is created, which both Goods and Services can implement.

3.1 Create the Product interface. In the Package Explorer view, right-click on the com.acme.domain package in the AcmeOrderSystem project, and select ***New > Interface***.



3.2 In the New Java Interface window, enter Product as the new interface name, and click the ***Finish*** button.



3.3 Add the abstract methods. In the Product.java editor view, add the three common methods shared by all types of products.

3.3.1 All products (either goods or services) have a product name. Therefore, create getName( ) and setName( ) abstract methods in the new interface.

```
public abstract String getName();
public abstract void setName(String n);
```

3.3.2 All products also have a toString( ) method. Therefore, create the toString( ) abstract method in the new interface.

```
public abstract String toString();
```

3.4 Save the Product.java interface, and make sure there are no compiler errors.

3.5 Modify Good and Service to implement the new Product interface.

3.5.1 Open the Good.java and Service.java files in editor views.

3.5.2 Use the "implements" key word on both of these classes to signal to the compiler that all Good and Service must provide the getName( ), setName( ), and toString( ) methods. The change to Good is shown below. Do the same work in Service.

```
public abstract class Good implements Product {
...
}
```

Note: Why do Solid and Liquid not have to use the "implements" keyword in their class definitions? These classes also must implement the Product interface, as they extend Good, which implements Product.

3.5.3 Since these classes already provide the three abstract methods of Product, no further work is required. Save Product, Good, and Service, and make sure there are no compiler errors.

Step 4: Refactor Order (again)

Through the power of polymorphism, orders currently allow any type of Good to be created and associated with an order through the product field. However, an order may be for a service as well. As a Service is not a type of Good, the Order class must be modified to allow for any Product type.

4.1 Change the product field's type. In Order.java, change the type of product from Good to Product.

```
private Product product;
```

Note: This will cause compiler errors in Order.

4.2 Update the getter/setter. Modify the getter and setter method for product in the Order class to get and set a Product, rather than a Good. The code for the new getter method is shown below.

```
public Product getProduct() {
  return product;
}
```

4.3 Change the Order constructor to use a Product, rather than a Good, when creating an Order.

```
   public Order(MyDate d, double amt, String c, Product p, int q)
{
         orderDate = d;
         orderAmount = amt;
         customer = c;
         product = p;
         quantity = q;
   }
```

4.4 Save your changes to Order, and then test the new Order class that uses the Product interface. Run TestOrders and check the output. The output should not change.

```
Attempting to set the quantity to a value less than or equal to
zero
10 ea. Acme Anvil-1668 that is 0.0225 CUBIC_METER in size for
Wile E Coyote
125 ea. Acme Balloon-1401 that is 375.0 CUBIC_FEET in size for
Bugs Bunny
The tax Rate is currently: 0.05
The tax for 3000.0 is: 150.0
The tax for this order is: 100.0
The tax for this order is: 50.0
The tax Rate is currently: 0.06
The tax for 3000.0 is: 180.0
The tax for this order is: 120.0
The tax for this order is: 60.0
The total bill for: 10 ea. Acme Anvil-1668 that is 0.0225
CUBIC_METER in size for Wile E Coyote is 2000.0
The tax for this order is: 60.0
The total bill for: 125 ea. Acme Balloon-1401 that is 375.0
CUBIC_FEET in size for Bugs Bunny is 1040.0
```

Step 5: Create Orders with Service

Because Services share the same interface (Product) as Goods, you should be able to create Orders with Service objects as the "product" without requiring other changes.

5.1 Add an import statement. In TestOrders, add an import statement so that the class can use the new Service objects.

```
import com.acme.domain.Service;
```

5.2 Create an Order with a service. In TestOrders, add the following lines of code at the end of the main( ) method.

```
MyDate date3 = new MyDate(4,10,2008);
Service s3 = new Service("Road Runner Eradication", 14, false);
Order birdEradication = new Order(date3, 20000, "Daffy Duck",
s3, 1);
System.out.println("The total bill for: " + birdEradication + "
is "
                    + birdEradication.computeTotal());
```

5.3 Save the file, and make sure there are no compiler errors in your code. Run TestOrders, and make sure that an Order associated with a Service works the same as an Order associated with a Good.

```
Attempting to set the quantity to a value less than or equal to
zero
10 ea. Acme Anvil-1668 that is 0.0225 CUBIC_METER in size for
Wile E Coyote
125 ea. Acme Balloon-1401 that is 375.0 CUBIC_FEET in size for
Bugs Bunny
The tax Rate is currently: 0.05
The tax for 3000.0 is: 150.0
The tax for this order is: 100.0
The tax for this order is: 50.0
The tax Rate is currently: 0.06
The tax for 3000.0 is: 180.0
The tax for this order is: 120.0
The tax for this order is: 60.0
The total bill for: 10 ea. Acme Anvil-1668 that is 0.0225
CUBIC_METER in size for Wile E Coyote is 2000.0
The tax for this order is: 60.0
```

```
The total bill for: 125 ea. Acme Balloon-1401 that is 375.0
CUBIC_FEET in size for Bugs Bunny is 1040.0
The total bill for: 1 ea. Road Runner Eradication(a 14 day
service) for Daffy Duck is 20000.0
```

Because both Service and Good objects share the same interface, they can be used interchangeably (to some extent) with Orders. This is polymorphism through interfaces!

Lab Solutions

Service.java

```
package com.acme.domain;

public class Service implements Product {
  private String name;
  private int estimatedTaskDuration;
  private boolean timeAndMaterials;

  public Service(String n, int dur, boolean tAndM) {
    this.estimatedTaskDuration = dur;
    this.name = n;
    this.timeAndMaterials = tAndM;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getEstimatedTaskDuration() {
    return estimatedTaskDuration;
  }

  public void setEstimatedTaskDuration(int
estimatedTaskDuration) {
    this.estimatedTaskDuration = estimatedTaskDuration;
  }

  public boolean isTimeAndMaterials() {
    return timeAndMaterials;
  }

  public void setTimeAndMaterials(boolean timeAndMaterials) {
    this.timeAndMaterials = timeAndMaterials;
  }

  public String toString() {
    return name + "(a " + estimatedTaskDuration + " day
service)";
  }

}
```

Product.java

```
package com.acme.domain;

public interface Product {

  public abstract String getName();
  public abstract void setName(String n);

  public abstract String toString();

}
```

Good.java

```
package com.acme.domain;

public abstract class Good implements Product {
  public enum UnitOfMeasureType {
    LITER, GALLON, CUBIC_METER, CUBIC_FEET
  }

  private String name;
  private int modelNumber;
  private double height;
  private UnitOfMeasureType unitOfMeasure;
  private boolean flammable = true;
  private double weightPerUofM;

  public Good(String name, int modelNumber, double height,
      UnitOfMeasureType uoM,
    boolean flammable, double wgtPerUoM) {
    this.name = name;
    this.modelNumber = modelNumber;
    this.height = height;
    this.unitOfMeasure = uoM;
    this.flammable = flammable;
    this.weightPerUofM = wgtPerUoM;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getModelNumber() {
    return modelNumber;
  }

  public void setModelNumber(int modelNumber) {
    this.modelNumber = modelNumber;
  }

  public double getHeight() {
    return height;
  }

  public void setHeight(double height) {
    this.height = height;
```

```
  }

  public UnitOfMeasureType getUnitOfMeasure() {
    return unitOfMeasure;
  }

  public void setUnitOfMeasure(UnitOfMeasureType unitOfMeasure)
{
    this.unitOfMeasure = unitOfMeasure;
  }

  public boolean isFlammable() {
    return flammable;
  }

  public void setFlammable(boolean flammable) {
    this.flammable = flammable;
  }

  public double getWeightPerUofM() {
    return weightPerUofM;
  }

  public void setWeightPerUofM(double weightPerUofM) {
    this.weightPerUofM = weightPerUofM;
  }

  public String toString() {
    return name + "-" + modelNumber;
  }

  public abstract double volume();

  public double weight() {
    return volume() * weightPerUofM;
  }
}
```

Order.java

```
package com.acme.domain;

import com.acme.utils.MyDate;

public class Order {
  private MyDate orderDate;
  private double orderAmount = 0.00;
  private String customer;
  private Product product;
  private int quantity;

  public MyDate getOrderDate() {
    return orderDate;
  }

  public void setOrderDate(MyDate orderDate) {
    this.orderDate = orderDate;
  }

  public double getOrderAmount() {
    return orderAmount;
  }

  public void setOrderAmount(double orderAmount) {
    if (orderAmount > 0) {
      this.orderAmount = orderAmount;
    } else {
      System.out
          .println("Attempting to set the orderAmount to a value
less
             than or equal to zero");
    }
  }

  public String getCustomer() {
    return customer;
  }

  public void setCustomer(String customer) {
    this.customer = customer;
  }

  public Product getProduct() {
    return product;
  }

  public void setProduct(Product product) {
```

```
    this.product = product;
  }

  public int getQuantity() {
    return quantity;
  }

  public void setQuantity(int quantity) {
    if (quantity > 0) {
      this.quantity = quantity;
    } else {
      System.out
          .println("Attempting to set the quantity to a value
less
             than or equal to zero");
    }
  }

  public static double getTaxRate() {
    return taxRate;
  }

  public static double taxRate = 0.05;

  public static void setTaxRate(double newRate) {
    taxRate = newRate;
  }

  public static void computeTaxOn(double anAmount) {
    System.out.println("The tax for " + anAmount + " is: " +
anAmount
        * Order.taxRate);
  }

  public Order(MyDate d, double amt, String c, Product p, int q)
{
    orderDate = d;
    orderAmount = amt;
    customer = c;
    product = p;
    quantity = q;
  }

  public String toString() {
    return quantity + " ea. " + product + " for " + customer;
  }

  public double computeTax() {
    System.out.println("The tax for this order is: " +
orderAmount
```

```
        * Order.taxRate);
    return orderAmount * Order.taxRate;
  }

  public char jobSize() {
    if (quantity <= 25) {
      return 'S';
    } else if (quantity <= 75) {
      return 'M';
    } else if (quantity <= 150) {
      return 'L';
    }
    return 'X';
  }

  public double computeTotal() {
    double total = orderAmount;
    switch (jobSize()) {
    case 'M':
      total = total - (orderAmount * 0.01);
      break;
    case 'L':
      total = total - (orderAmount * 0.02);
      break;
    case 'X':
      total = total - (orderAmount * 0.03);
      break;
    }
    if (orderAmount <= 1500) {
      total = total + computeTax();
    }
    return total;
  }
}
```

TestOrders.java

```
package com.acme.testing;

import com.acme.domain.Order;
import com.acme.domain.Service;
import com.acme.domain.Solid;
import com.acme.domain.Good.UnitOfMeasureType;
import com.acme.utils.MyDate;

public class TestOrders {

  public static void main(String[] args) {
    MyDate date1 = new MyDate(1, 20, 2008);
    Solid s1 = new Solid("Acme Anvil", 1668, 0.3,
      UnitOfMeasureType.CUBIC_METER, false, 500, 0.25, 0.3);
    Order anvil = new Order(date1, 2000.00, "Wile E Coyote", s1,
10);

    MyDate date2 = new MyDate(4, 10, 2008);
    Solid s2 = new Solid("Acme Balloon", 1401, 15,
      UnitOfMeasureType.CUBIC_FEET, false, 10, 5, 5);
    Order balloons = new Order(date2, 1000.00, "Bugs Bunny", s2,
125);
    balloons.setQuantity(-200);

    System.out.println(anvil);
    System.out.println(balloons);

    System.out.println("The tax Rate is currently: " +
Order.taxRate);
    Order.computeTaxOn(3000.00);
    anvil.computeTax();
    balloons.computeTax();

    Order.setTaxRate(0.06);
    System.out.println("The tax Rate is currently: " +
Order.taxRate);
    Order.computeTaxOn(3000.00);
    anvil.computeTax();
    balloons.computeTax();
    System.out.println("The total bill for: " + anvil + " is "
        + anvil.computeTotal());
    System.out.println("The total bill for: " + balloons + " is
"
        + balloons.computeTotal());

    MyDate date3 = new MyDate(4,10,2008);
```

```
    Service s3 = new Service("Road Runner Eradication", 14,
false);
    Order birdEradication = new Order(date3, 20000, "Daffy
Duck", s3, 1);
    System.out.println("The total bill for: " + birdEradication
+ " is " + birdEradication.computeTotal());
  }
}
```

Give Intertech a call at **1.800.866.9884** or visit Intertech's website.