Lab Exercise: Polymorphism

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

Polymorphism

Polymorphism allows an object reference of a generic type to point to an object of a more specific type. For example, a Good reference can point to a Solid or Liquid product object. In this lab, you use polymorphism to assign any type of good to an order's product field. You also explore the equals( ) method to compare MyDate objects.

Specifically, in this lab you will:

- Update the Order type to use polymorphism and the good type hierarchy (Good, Liquid, Solid) for products, rather than using a string
- Update MyDate to override the polymorphic equals( ) method from java.lang.Object.
- Optionally explore the cast operation to cast a general reference back to a more specific type of object.

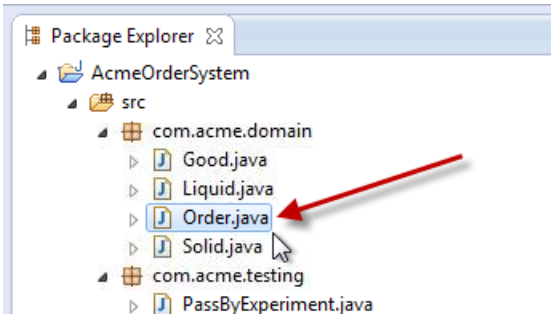
Scenario

Given the product hierarchy (Good, Solid, Liquid), you now have a fine set of "product" objects to associate with an order (replacing the current String for product). Of course, an order could be for a Solid or a Liquid type, so what type should be used on the product field of Order? Here is where polymorphism can help out.

Step 1: Refactor Order

Currently, the product field on Order is of type String. Now it is time to associate the Good type objects to orders.

1.1 Change the product field's type. In the Package Explorer view, double-click on the Order.java file in the com.acme.domain package to open the file in a Java editor.



In Order.java, change the type of product from String to Good.

```
private Good product;
```

Note: This will cause a compiler error on the getter and setter for product, as well as the Order's constructor.

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

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

1.3 Update the Order constructor to create a new order with a Good object, instead of a String.

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

Note: This will cause compile errors in TestOrders.

1.4 Update TestOrders.java to create and use an instance of a Good for the product field of an Order, rather than a String. Replace the first few lines of the main( ) method shown below…

```
MyDate date1 = new MyDate(1, 20, 2008);
Order anvil = new Order(date1, 2000.00, "Wile E Coyote",
"Anvil", 10);

MyDate date2 = new MyDate(4, 10, 2008);
Order balloons = new Order(date2, 1000.00, "Bugs Bunny",
"Balloon", 125);
```

…with what is shown here.

```
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);
// ... rest of the main method as before
```

Note: This is polymorphism at work. Notice how on Order the type specified was a Good. Here you create a Solid for use in the product field! You could have also created a Liquid object and passed it in as the product for the order. Polymorphism allows any specific object (Solid or Liquid) to be used in a general reference (Good in this case).

1.5 You now need to import UnifOfMeasureType and Solid in TestOrders.java.

1.6 Test the new Order class with associated Good products by running TestOrders. The output should change a little. Why does it? How does the display of product information differ, and what is occurring to make this happen?

```
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 2: Implement Equals on MyDate

Given two MyDate objects, do they represent the same date? In other words, are they equal? Given the current MyDate definition, you might be surprised by the result. In this step, you observe the default implementation of the equals( ) method and then override the java.lang.Object equals( ) method to provide a more appropriate response among MyDate objects.

2.1 Test if two MyDate are equal.

2.1.1 Open TestMyDate in an editor view. In the main( ) method of the TestMyDate class, create two MyDate objects that have the same day, month, and year field values.

```
MyDate newYear = new MyDate(1,1,2009);
MyDate fiscalStart = new MyDate(1,1,2009);
```

2.1.2 Following the creation of these objects, use the equals( ) method to determine whether the two are equal.

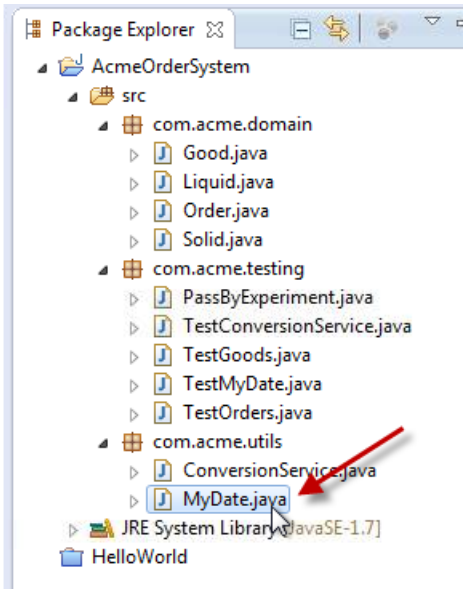
```
if (newYear.equals(fiscalStart))
  System.out.println("These two dates are equal");
else
  System.out.println("These two dates are not equal");
```

2.1.3 Save TestMyDate and run the TestMyDate.java file. The results for this operation should indicate the two objects are not equal! What does this tell you about the equals( ) method?

```
...
These two dates are not equal
```

2.2 Add an equals( ) method to MyDate.

2.2.1 In the com.acme.utils package, find and open MyDate.java in an editor view.



Add a new equals( ) method that overrides the java.lang.Object equals( ) method.

```
public boolean equals(Object o) {
  if (o instanceof MyDate) {
    MyDate d = (MyDate) o;
    if ((d.day == day) && (d.month == month) && (d.year ==
year)) {
      return true;
    }
  }
  return false;
}
```

Note: Did you see the use of the instanceof operator in this method? Why should you use this operator in the equal method?

2.2.2 Save the MyDate.java file, and rerun the TestMyDate.java file. Now the two dates should be equal.

```
These two dates are equal
```

Try creating two MyDate objects that do not have the same day, month, and year field values, and then check to see if they are equal.

Lab Solutions

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 Good 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 Good getProduct() {
    return product;
```

```
  }

  public void setProduct(Good 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, Good 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.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());

  }
}
```

TestMyDate.java

```
package com.acme.testing;

import com.acme.utils.MyDate;

public class TestMyDate {

  public static void main(String[] args) {
    MyDate date1 = new MyDate(11, 11, 1918);

    MyDate date2 = new MyDate();
    date2.setDay(11);
    date2.setMonth(11);
    date2.setYear(1918);

    MyDate date3 = new MyDate();
    date3.setDate(13, 40, -1);

    String str1 = date1.toString();
    String str2 = date2.toString();
    String str3 = date3.toString();

    System.out.println(str1);
    System.out.println(str2);
    System.out.println(str3);

    MyDate.leapYears();

    MyDate newYear = new MyDate(1, 1, 2009);
    MyDate fiscalStart = new MyDate(1, 1, 2009);
    if (newYear.equals(fiscalStart))
      System.out.println("These two dates are equal");
    else
      System.out.println("These two dates are not equal");
    MyDate endOfYear = new MyDate(12,31,2009);
    if (newYear.equals(endOfYear))
      System.out.println("These two dates are equal");
    else
      System.out.println("These two dates are not equal");
  }
}
```

MyDate.java

```
package com.acme.utils;

public class MyDate {
  // Member/instance variables (a.k.a.
fields/properties/attributes)
  private byte day;
  private byte month;
  private short year;

  // Constructors:
  // 1. Same name as the class
  // 2. No return type

  // The no-args constructor
  public MyDate() {
    this(1, 1, 1900);
  }

  // Constructor that takes 3 arguments
  public MyDate(int m, int d, int y) {
    setDate(m, d, y);
  }

  // Methods
  public String toString() {
    return month + "/" + day + "/" + year;
  }

  public void setDate(int m, int d, int y) {
    if (valid(d, m, y)) {
      day = (byte) d;
      year = (short) y;
      month = (byte) m;
    }
  }

  public static void leapYears() {
    for (int i = 1752; i <= 2020; i = i + 4) {
      if (((i % 4 == 0) && (i % 100 != 0)) || (i % 400 == 0))
        System.out.println("The year " + i + " is a leap year");
    }
  }

  public int getDay() {
    return day;
  }
```

```
  public void setDay(int day) {
    if (valid(day, month, year)) {
      this.day = (byte) day;
    }
  }

  public int getMonth() {
    return month;
  }

  public void setMonth(int month) {
    if (valid(day, month, year)) {
      this.month = (byte) month;
    }
  }

  public int getYear() {
    return year;
  }

  public void setYear(int year) {
    if (valid(day, month, year)) {
      this.year = (short) year;
    }
  }

  private boolean valid(int day, int month, int year) {
    if (day > 31 || day < 1 || month > 12 || month < 1) {
      System.out.println("Attempting to create a non-valid date
" +
        month + "/" + day + "/" + year);
      return false;
    }
    switch (month) {
    case 4:
    case 6:
    case 9:
    case 11:
      return (day <= 30);
    case 2:
      return day <= 28 || (day == 29 && year % 4 == 0);
    }
    return true;
  }

  public boolean equals(Object o) {
    if (o instanceof MyDate) {
      MyDate d = (MyDate) o;
      if ((d.day == day) && (d.month == month) && (d.year ==
year)) {
```

```
            return true;
          }
        }
        return false;
      }
    }
```

Bonus Lab

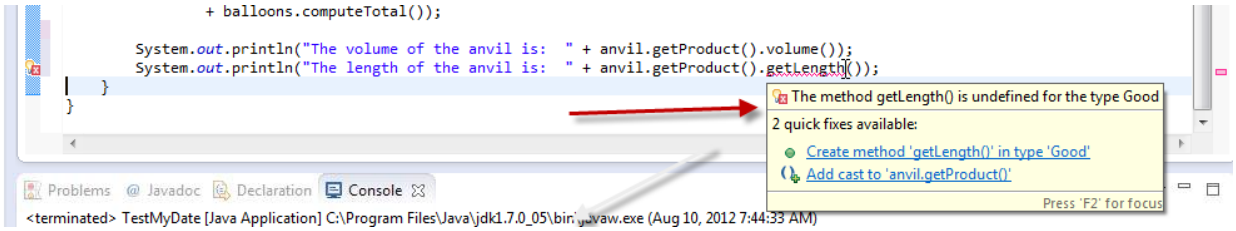
Step 3: Understand casting

Although polymorphism is a powerful mechanism that makes code more flexible, it can be a bit of a double-edged sword when trying to deal with the more specific properties or capabilities of an object through its generic reference. For example, attempt to get the Good object from an order, and then call on its volume( ) method. Using the same reference, try to get the radius of the Good.

3.1 At the bottom of the main( ) method in TestOrders.java, attempt to get and print out the volume and length of the Good associated with the anvil order.

```
System.out.println("The volume of the anvil is:  " +
anvil.getProduct().volume());
System.out.println("The length of the anvil is:  " +
anvil.getProduct().getLength());
```

Calling on the volume method on the Order's product works, but you get a compiler error when trying to get the length.



Can you explain this?

3.2 Fix this by using casting to cast the product reference back to its real Solid type and then calling the getLength( ) method.

3.3 What would happen, however, if the product was really a Liquid and not a Solid?

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