Lab Exercise: Encapsulation

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

Encapsulation

Encapsulation is a two-way street. Proper encapsulation prevents objects from having data that is illegal. Proper encapsulation also hides many of the implementation details in a class from users of that class. This allows the class developer to make more changes with little or no impact on users of the class.

In this lab, you see the importance of encapsulation from the class writer's and the class user's perspective.

Specifically, in this lab you will:

- Encapsulate the fields of your application classes
- See how an IDE can help to encapsulate data
- Understand the importance of encapsulation and how it can help isolate later code changes

Scenario

Although the Acme Order System seems to be rolling along, a code review at Acme has enlightened everyone to the fact that proper data encapsulation of MyDate and Order objects is not occurring. Developers can create instances of MyDate with dates like 2/31/2008, and Order instances can be created with an order amount of -1.00 or quantities of -20. In this lab, you properly encapsulate and protect your object data.

Step 1: Encapsulate MyDate

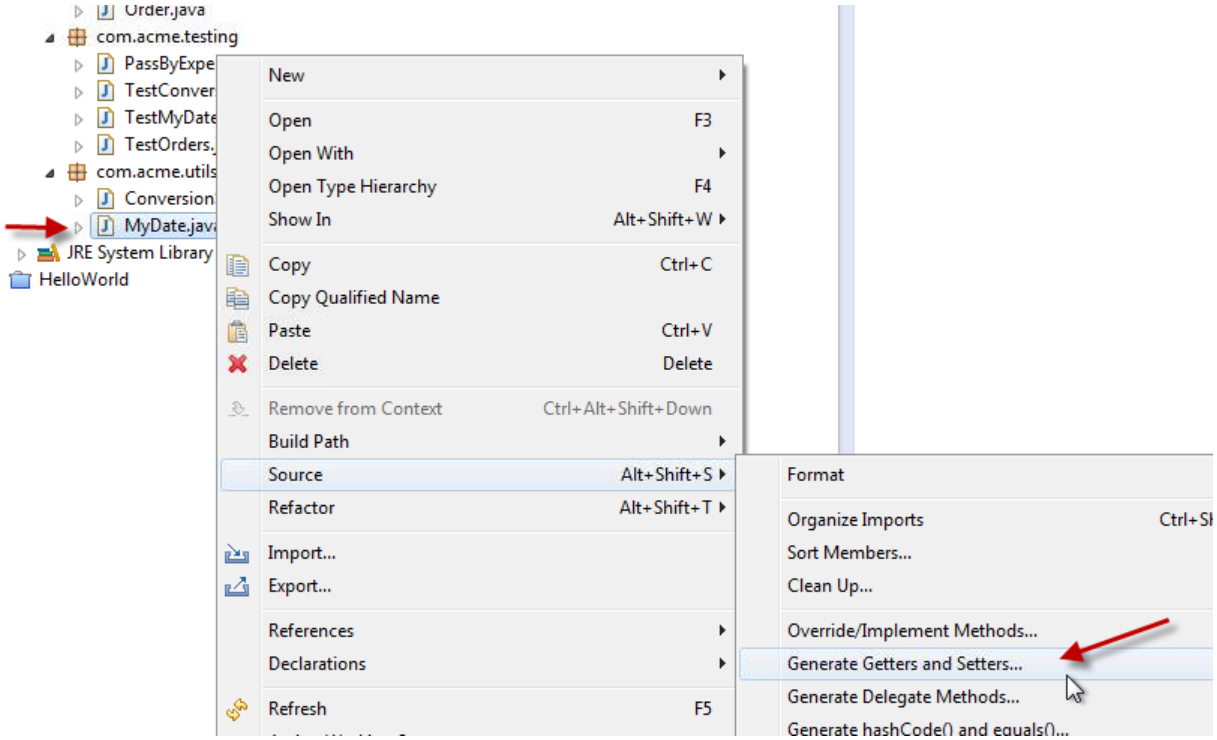
1.1 Encapsulate the fields of MyDate. In the Package Explorer view, double-click on the MyDate.java file in the com.acme.utils package to open the file in a Java editor. Encapsulate the month, day, and year fields of MyDate by providing appropriate getter/setter methods and privatizing the fields themselves.

1.1.1 Change the access modifier on month, day, and year to be private rather than public.

```
private int day;
private int month;
private int year;
```

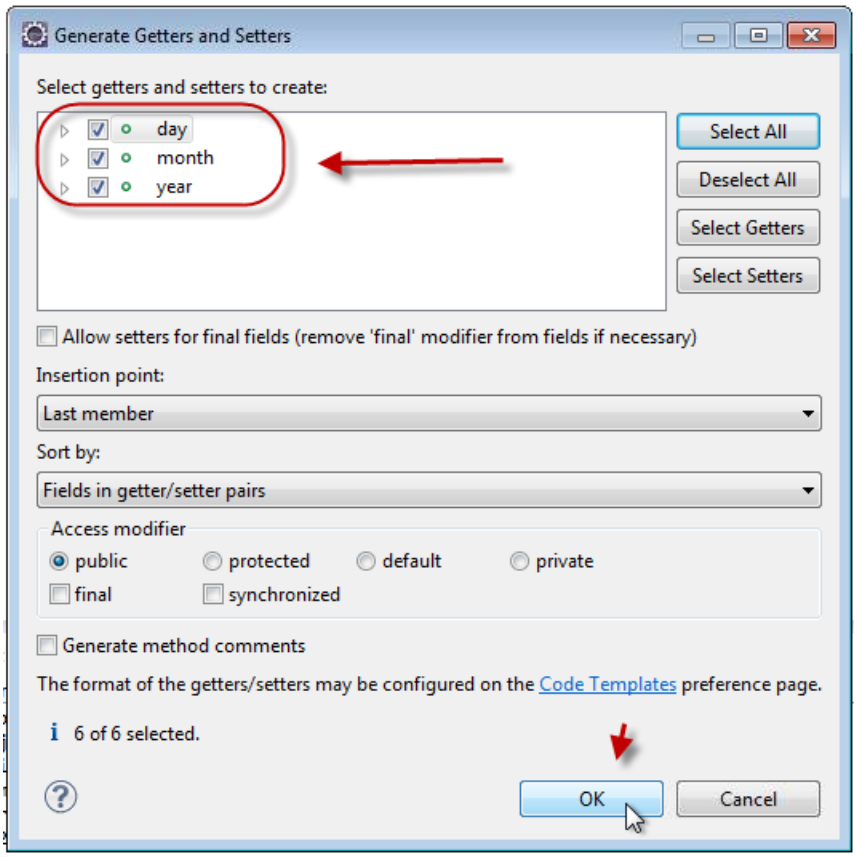
Note: This will cause several compiler errors to appear throughout the AcmeOrderSystem code. You fix the errors later.

1.1.2 Add field accessor and mutator methods. This can be done by hand by typing all the method code in the MyDate class or by using one of the IDE's wizards as shown here. In the Package Explorer view, right-click on the MyDate class, and select *Source > Generate Getters and Setters….*



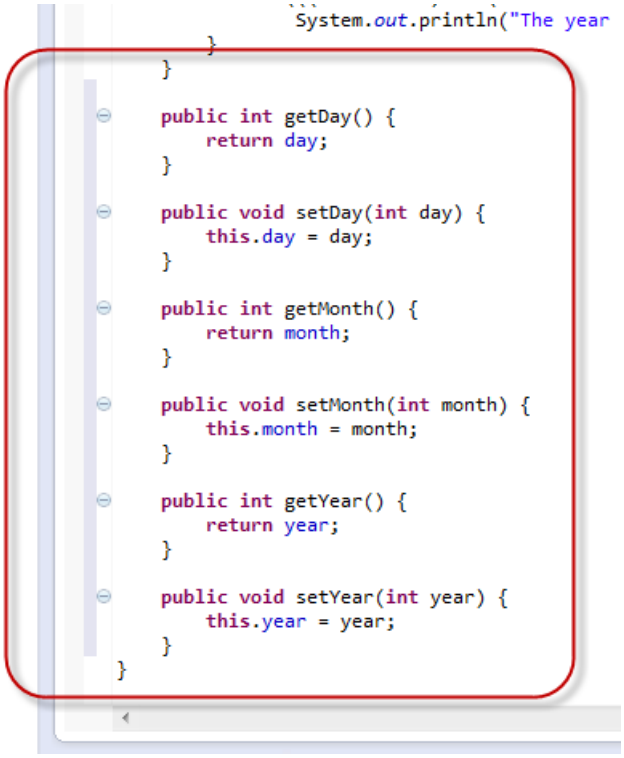
Note: This operation will generate the accessors (a.k.a. getter) and mutator (a.k.a. setter) methods according to Java conventional naming standards.

1.1.3 In the resulting Generate Getters and Setters window, check the day, month, and year boxes (properties that you want getter and setter methods created for), and click the *OK* button.



Note: the Insertion point drop down in the window above allows you to determine where to put the generate code (the getters and setters). You may want to pick to insert the new methods after the "Last member" (after the last instance variable) or after any of the methods in the class.

1.1.4 This should result in a number of get and set methods being generated and added to the MyDate code.



```
                System.out.println("The year
            }
        }

        public int getDay() {
            return day;
        }

        public void setDay(int day) {
            this.day = day;
        }

        public int getMonth() {
            return month;
        }

        public void setMonth(int month) {
            this.month = month;
        }

        public int getYear() {
            return year;
        }

        public void setYear(int year) {
            this.year = year;
        }
    }
```

1.1.5 Save the MyDate.java file. While there may be compiler errors in other files, the MyData.java file should not have errors.

1.2 Add validation to MyDate. While a good start, the encapsulation of the fields with getter/setter methods does not yet prevent someone from creating a MyDate object with a date of 2/31/2008 or something worse, for example. Validation of the data is needed whenever a MyDate is created or modified.

1.2.1 Create a new method in MyDate called valid( ) that takes the proposed month, day, and year as parameters and returns a true or false boolean indicating whether the proposed values make for a valid date. A simple validation method is provided below.

```
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;
}
```

1.2.2 Modify the setter methods, the setDate( ) method, and the MyDate constructors to use the valid( ) method to make sure all dates created are valid. For example, in the setDay( ) method, check the validity of the new day (in relation to the month and year) before allowing the new value to be used.

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

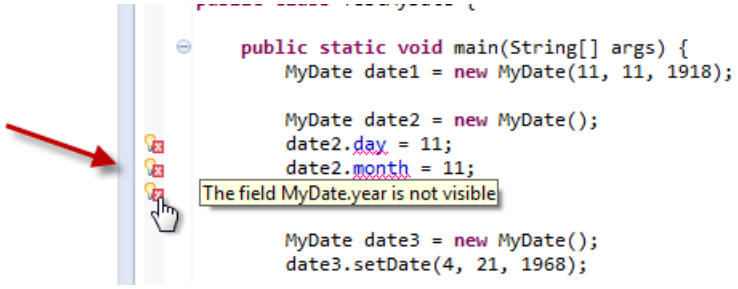
1.2.3 The default constructor, if used as it stands, could result in a date being created with 0 as the day, month, or year. Make sure the day, month, and year are initialized so that a valid date exists when this constructor is called. This can be accomplished with an initialization block, by chaining constructors, or even by

providing default initialization values for the fields themselves. The example below uses chained constructors to accomplish the work.

```
public MyDate() {
   this(1,1,1900);
}
```

1.3 Modify using classes. Now data in MyDate is properly encapsulated, but using classes such as TestMyDate are broken. You must fix these to use the appropriate getter/setter methods, as opposed to using the fields directly.

1.3.1 In TestMyDate, notice how after creating an instance of MyDate, the class attempts to set the day, month, and year fields directly. What error is the compiler giving you?



Change the code to make use of the new MyDate setters, as opposed to using the field directly. For example, as opposed to calling "date2.day = 11", use the code below.

```
date2.setDay(11);
```

1.3.2 In PassByExperiment, similar errors exist. Use the new MyDate get and set methods to replace direct field access.

1.3.3 Save changes in your code, and fix any remaining compiler errors.

1.4 Test MyDate with TestMyDate and PassByExperiment classes.

1.4.1 After encapsulating MyDate and repairing the test classes, run both TestMyDate and PassByExperiment. The functionality of your application should not change.

1.4.2 In TestMyDate, attempt to set one of the MyDate objects to a nonsensical date as shown in the example below.

```
date3.setDate(13, 40, -1);
```

This should result in a message in the Console view indicating the update to the MyDate object was not successful.

```
Attempting to create a non-valid date 13/40/-1
11/11/1918
11/11/1918
1/1/1900
```

Step 2: Encapsulate Order

Now that you have seen how to properly encapsulate the data of a class, do the same for Order and its properties.

2.1 Make the fields of Order private.

2.2 Generate (or write) getter and setter methods for Order fields.

2.3 Improve the setOrderAmount( ) and setQuantity( ) methods to ensure only positive numbers can be used to update these fields.

2.4 Test the encapsulated Order class by running TestOrders.

2.5 Attempt to change the quantity or orderAmount of an Order object created in the main( ) method of TestOrders to a negative number in TestOrders to ensure the encapsulation is working.

```
balloons.setQuantity(-200);
```

This should result in output that does not differ, except for an indication that the setting of the quantity did not work.

```
Attempting to set the quantity to a value less than or equal to
zero
10 ea. Anvil for Wile E Coyote
125 ea. Balloon for Bugs Bunny
...
```

Step 3: Demonstrate the power and protection of encapsulation

The database team from Acme has arrived to take a look at your Acme Order System code. They are planning for the eventuality that Order data will have to be stored in a database. In their code review, they make a suggestion: "Why not change the MyDate day and month fields to bytes and the year to a short?" This will save on table space. Ah, nuts! Now you have a lot of rework, and users of Order are not going to be happy! Wait a minute! You remember that with proper encapsulation, things may not be so bad.

3.1 Change the types on MyDate to use the smaller of primitive types.

```
private byte day;
private byte month;
private short year;
```

Note: Again, this will cause compiler errors in the code. However, notice right away that the errors are limited to the MyDate class! You'll fix these errors shortly.

3.2 Modify the setter methods to allow users to work with and pass the larger data types (ints) while casting to the appropriate type under the covers. Below is an example of the change required with the setDay( ) method.

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

Note: You can also make similar changes to get methods (upcasting to ints from bytes or shorts), but this is not explicitly required.

3.3 Rerun TestMyDate to ensure the change of types in the implementation of MyDate had no effect on other MyDate-using code.

Lab Solutions

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;
  }
}
```

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();
  }
}
```

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 String 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 String getProduct() {
    return product;
  }

  public void setProduct(String 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, String 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.utils.MyDate;

public class TestOrders {

  public static void main(String[] args) {
    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);
    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());
  }
}
```

PassByExperiment.java

```
package com.acme.testing;

import com.acme.utils.MyDate;

public class PassByExperiment {

  public static void main(String[] args) {
    MyDate date = new MyDate(1, 20, 2008);
    System.out.println("Before passing an object " + date);
    passObject(date);
    System.out.println("After passing an object " + date);

    System.out.println("Before passing a primitive  " +
       date.getYear());
    passPrimitive(date.getYear());
    System.out.println("After passing a primitive  " +
       date.getYear());

    String x = date.toString();
    System.out.println("Before passing a String  " + x);
    passString(x);
    System.out.println("After passing a String  " + x);
  }

  public static void passObject(MyDate d) {
    d.setYear(2009);
  }

  public static void passPrimitive(int i) {
    i = 2010;
  }

  public static void passString(String s) {
    int yearSlash = s.lastIndexOf('/');
    s = s.substring(0, yearSlash + 1);
    s += "2010";
    System.out.println("New date string: " + s);
  }
}
```

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