7 Understanding Constructors, Equality, Testing

Goals

In the first part of this lab you will practice the use of constructors in assuring data integrity and providing a better interface for the user. The exercises follow those posted in lecture notes for week 5.

In the second part of the lab you will learn to use a test harness in designing the test suite, and will practice the design of tests for methods with effects.

7.1 Data Integrity: Constructor Overloading, Encapsulation, Error Handling

Allow 20 minutes for this part. Finish the work at home and save it as a part of your *Etudes* portfolio.

7.1.1 Designing constructors to assure integrity of data.

We start with a simple *Date* class:

```
// to represent a calendar date
public class Date {
  public int year;
  public int month;
  public int day;

  public Date(int year, int month, int day){
    this.year = year;
    this.month = month;
    this.day = day;
  }

  // represent the information in this class as a String
  public String toString(){
    return "new Date(year = " + String.valueOf(year) + ",\n" +
           "         month = " + String.valueOf(month) + ",\n" +
           "         day = " + String.valueOf(day) + ")\n";
  }
}
```

and a simple set of examples:

```
public class Examples {
  Examples() {}

  // good dates
```

```
  Date d20060928 = new Date(2006, 9, 28);     // Sept 28, 2006
  Date d20051012 = new Date(2005, 10, 12);  // Oct 12, 2005

  // bad dates
  Date b34453323 = new Date(3445, 33, 23);

  public static void main(String argv[]){

    Examples e = new Examples();
    System.out.println(e.d20060928.toString());
    System.out.println(e.d20051012.toString());
    System.out.println(e.b34453323.toString());
  }
}
```

Create a project *Date* in the Eclipse and add two files: the file *Data.java* with the definition of the class *Date* and the file *Examples.java* with the definition of the *Examples* class. (You may copy and paste from the web page.) Now run the project.

Of course, the third example is pure nonsense. Only the year is possibly valid - still not really an expected value. To validate the date completely (taking into account all the special cases for different months, as well as leap years, and the change of the calendar at several times in the history) is a project on its own. For the purposes of learning about the use of constructors, we will only make sure that the month is between 1 and 12, the day is between 1 and 31, and the year is between 1000 and 2200.

Did you notice the repetition in the description of the valid parts of the date? This suggests, we start with the following methods:

- method *validNumber* that consumes a number and the low and high bound and returns true if the number is within the bounds (inclusive).
- methods *validDay*, *validMonth*, and *validYear* designed in a similar manner.

Once you have done so, change the constructor for the class *Date* as follows:

```
  public Date(int year, int month, int day){
    if (this.validYear(year))
      this.year = year;
    else
      throw new IllegalArgumentException("Invalid year in Date.");

    if (this.validMonth(month))
```

```
    this.month = month;
  else
    throw new IllegalArgumentException("Invalid month in Date.");

  if (this.validDay(day))
    this.day = day;
  else
    throw new IllegalArgumentException("Invalid day in Date.");
}
```

This example show you how you can signal errors in Java. The class *IllegalArgumentException* is a subclass of the *RuntimeException*. Including the clause

```
throws new ...Exception("message");
```

in the code causes the program to terminate and print the specified error message. Later we will learn how we can customize the error reporting and also how to respond to errors without terminating the program execution.

Make additional examples with invalid day, invalid month, and invalid year. Run the program, then comment out one invalid example at a time, to see that all checks work correctly.

7.1.2 Overloading constructors to provide flexibility for the user: providing defaults.

When entering dates in the current year it is tedious to always have to enter 2006. We can make avoid the need to type in the year by providing an additional constructor that requires the user to give only the day and month and assumes that the year is the current year (2006 in our case).

Remembering the *single point of control* rule, we make sure that the new **overloaded** constructor defers all of the work to the primary **full** constructor:

```
public Date(int month, int day){
  this(2006, month, day);
}
```

Add examples that use only the month and day to see that the constructor works properly. Include examples with invalid month or year as well. (Of course, you will have to comment them out.)

7.1.3 Overloading constructors to provide flexibility for the user: expanding the options.

The user may want to enter the date in the form "Oct 20 2006". To make this possible, we can add another constructor:

```
public Date(String month, int day){
  this(1, day);        // make an instance with a wrong month
  if (month.equals("Jan"))
    this.month = 1;
  else if ...

  else
    throw new IllegalArgumentException("Invalid month in Date.");
}
```

To check that it works, allow the user to enter only the first three months ("Jan", "Feb", and "Mar"). The rest is tedious, and in a real program would be designed differently.

7.1.4 Encapsulation of constructors to assure data integrity.

Look at the following code:

```
public class Rat{
  int life;

  // A Rat with the given number of days to live
  public Rat(int life){
    this.life = life;
  }

  // a day goes by, the rat starves
  public Rat starve(){
    return new Rat(this.life - 1);
  }

  // rat found some food
  public Rat eat(int foodsize){
    return new Rat(this.life + foodsize);
  }

  // is the rat dead?
  public boolean isDead(){
    return this.life <= 0;
  }
```

Create project *RatLife* and include the files *Rat.java* and *Examples.java* where you define your examples of rats. Add the method *toString* to the class *Rat*.

When this class is used in the *Rat Race* game a new rat starting the game always gets five years to live. After that, every new instance of a *Rat* is produced by the methods *starve* or *eat* (and possibly *findPoison*). To prevent the game programmer from constructing rats with an arbitrary life span, we can equip the class *Rat* with two constructors, one *public* and one *private*. The *public* constructor is used by the *World* to start the game, or to add a new rat, if the old one dies. These instances have always the lifespan of five years. New instances of the *Rat* needed as the game goes on are produced when the methods *starve* or *eat* (and possibly *findPoison*) are invoked. The code is as follows:

```
// Rat always starts its life with five ticks to go
public Rat(){
  this.life = 5;
}

// Rat can change its life expectancy only by eating or starving
private Rat(int life){
  this.life = life;
}
```

Add examples to the *Examples* class and test the behavior of these constructors. (See the errors generated when you attempt to use the *private* constructor.)

7.2 Equality Tests: The Test Harness

You noticed that instead of using one file to keep all of our work we now have several different files. Java requires that each (*public*) class or interface is saved in a separate file and the name of that file must be the same as the name of the class or interface, with the extension *.java*. That means, you will always need several files for each problem you are working on.

We will now learn how to use our test harness for Java programs in the context of a simple class. Start by implementing a method *nextYear* method in the *Date* class. (This is the method for which we design the tests.) Design the examples as we did in the last lab, and run the project.

Now add to your project the library *TestHarness.jar* and a library *jpt.jar*.

These libraries automatically provide two things of interest for this assignment. The first is the *ISame* interface:

```
public interface ISame {
  // is this object the same as the given object?
  public boolean same(Object obj);
}
```

The second is the class *TestHarness*. The *TestHarness* class defines methods that run test cases, keep track of the test success and failure, and print out detailed reports. To be able to use these methods and get the reports, the *Examples* class must create an instance of the *TestHarness*.

7.2.1 Implementing the ISame interface

We already know what it means for a class to implement an interface. Now make the class *Date* implement the *ISame* interface. Our equality tests need to compare whether two *Date* objects represent the same date: same year, same month, and the same day. However, the argument for the method *same* is an arbitrary instance of the class *Object*.

We solve the dilemma by implementing two methods. The method *sameDate* determines whether two instances of the class *Date* represent the same date:

```
// is this date the same as the given date?
public boolean sameDate(Date that){
  return this.year == that.year &&
         this.month == that.month &&
         this.day == that.day;
}
```

In the method *same* we start by making sure the argument represents an instance of the class *Date*, and delegate the rest of the equality testing to the method *sameDate* if the answer is positive:

```
// is this date the same as the given object?
public boolean same(Object that){
  if (that instanceof Date)
    return ((Date)that).sameDate(this);
  else
    return false;
}
```

This is a simplified test that allows an instance of a subclass of the class *Date* to be considered the same as an instance of the class *Date*, as long as they both give the same *year*, *month*, and *day*.For examples the class *WeekDate* includes the day of the week information:

```
class WeekDate extends Date{
  String weekday;

  WeekDate(int year, int month, int day, String weekday){
    super(year, month, day);
    this.weekday = weekday;
  }
}
```

The following two dates would be considered the same:

```
Date d20061020 = new Date(2006, 10, 20);
WeekDate wd20061020Fr = new WeekDate(2006, 10, 20, "Friday");

d20061020.same(wd20061020Fr) --> true
```

However, if in the class *WeekDate* we have also overriden the method *same* by invoking the method *sameWeekDate*, the comparison

```
wd20061020Fr.same(d20061020)
```

would produce *false*. Try it.

For all other classes we implement the *same* method in a similar fashion. We first design the method that compares two instances of the same class, just as we have learned in the previous labs, then implement the method *same* that tests whether the argument is an instance of the same class and invoke our method if the answer is true.

7.2.2 The test harness: Introduction

The class *TestHarness* included in the *TestHarness.jar* library defines the following methods:

```
// test that compares two boolean-s using == operator
test(String testname, boolean testvalue, boolean expected)

// test that compares two char-s using == operator
test(String testname, char testvalue, char expected)

// test that compares two integers using == operator
test(String testname, int testvalue, int expected)

// test that compares two double-s using == operator
test(String testname, double testvalue, double expected, double within)

// test that compares two objects using same method
test(String testname, ISame testvalue, ISame expected)

// test that compares two objects using Java (or overridden) equals
test(String testname, Object testvalue, Object expected)

// test that only reports success or failure
void test(String testname, boolean result)

// report on the number and nature of failed tests
void testReport()

// produce test names and values compared for all tests
void fullTestReport()
```

(There are methods for the primitive types *short*, *long*, and *float* as well.)

Notice that invoking the *test* method is very similar to our use of *check* construct in *ProfessorJ*.

Convert the *Examples* class that tests the *Date* constructors to use the *TestHarness* library as follows:

```
public class Examples {
  Examples() {}

  // sample dates
  Date d20060928 = new Date(2006, 9, 28);     // Sept 28, 2006
  Date d20070928 = new Date(2007, 9, 28);     // Sept 28, 2007
  Date d20051012 = new Date(2005, 10, 12);    // Oct 12, 2005

  TestHarness th = new TestHarness("Test same Method ");


  // Run the test suite for the nextYear method
  public void testNextYearMethod() {
    th.test("nextYear: OK", d20060928.nextYear(), d20070928);
    th.test("nextYear: NO", d20060928.nextYear(), d20051012);
  }

  // Run the test suite for the same method
  public void testSameMethod() {
    th.test("same: OK", d20060928.same(d20060928), true);
    th.test("same: NO", d20060928.same(d20051012), false);
  }

  public static void main(String argv[]){

    Examples e = new Examples();
    e.testSameMethod();

    e.th.testReport();
    e.th.fullTestReport();
  }
}
```

We expect the second test for the method *nextYear* to fail. Both the short test report (reporting only failures) and the full test report (reporting all test results) provide the expected feedback. Each test case prints out the test name, and then indicates success or failure. It also prints out the expected and actual values.

Important: If your expected and actual values are objects, then you must remember to implement the toString() method in order to see a readable printout! You'll know that you've forgotten to do so if you see text like Date@7fde3 in the results.

7.3 Using the test harness

In **Lab 5** you defined the *same* method for the classes that represent a list of stars in the *World*. We will use that example to practice the use of the test harness.

- Create a project *Stars* by creating one file per class or interface.
- Comment out the test cases in the *Examples* class. Do not delete them — as they will be converted to the new test format later.
- In **lab 5** we had the following examples/tests:

```
// Sample data
Star s1 = new Star(new CartPt(20, 40), 10);
Star s2 = new Star(new CartPt(30, 40), 5);
Star s3 = new Star(new CartPt(10, 30), 8);
Star s4 = new Star(new CartPt(10, 50), 10);

LoStars mtstars = new MTLoStars();
LoStars list1 = new ConsLoStars(s1, mtstars);
LoStars list2 = new ConsLoStars(s2, list1);
LoStars list3 = new ConsLoStars(s3, list2);
LoStars list4 = new ConsLoStars(s4, list3);

// dealing with the empty list
mtstars.same(new MTLoStars()) --> true
mtstars.same(list1) --> false
list1.same(mtstars) --> false
list4.same(mtstars) --> false

// dealing with a list with one item
list1.same(new ConsLoStars(s1, mtstars)) --> true
list1.same(list2)) --> false
list1.same(new ConsLoStars(s2,
           new ConsLoStars(s3, mtstars))) --> false

// dealing with a list with more than one item
list4.same(new ConsLoStars(s1, mtstars)) --> false
list4.same(new ConsLoStars(s4,
           new ConsLoStars(s3, list2)) --> true
list4.same(new ConsLoStars(s4,
           new ConsLoStars(s2, mtstars)) --> false
```

  Convert these tests to the tests that use the *TestHarness*.

- Now replace those where the expected value is *true* as follows:

```
mtstars.same(new MTLoStars()) --> true
list1.same(new ConsLoStars(s1, mtstars)) --> true
```

become statements

```
th.test("test same for empty class",
        mtstars,
        new MTLoStars());
th.test("test same for nonempty class",
        list1,
        new ConsLoStars(s1, mtstars));
```

within the test methods in the *Examples* class.

Notice which tests fail, even though they were successful before. The test harness is using the Java method *equals* that only checks whether two objects are the same instances. We need to define our own *measure of equality*.

- Make the classes *CartPt*, *Star*, and *ALoStars* implement the *ISame* interface. For the classes *CartPt* and *Star* this is straightforward. In the classes that extend *ALoStars* modify the implementation of the *same* method so that it starts with the *instanceof* test.
- Make sure you add tests that compare an instance of *MTLoStars* and *ConsLoStars* with an object that is not an instance of the *ALoStars* class hierarchy.
- Add the method *toString*() to all classes in this project.

You must complete the problems in this lab and include the solutions in your *Etudes* **portfolio.**