5 Understanding Constructors; Equality

5.1 Standard Java and the tester library

Goals

Starting with this lab we will use the standard Java language. Of course, we only know a small part of the language. We will learn new features when they are needed to support our program design process.

Moving to standard Java: File organization

Standard Java *Project* differs very little from the projects we have built so far. The main difference is that standard Java expects you to define every `class` and every `interface` in a separate file whose name is the name of the `class` or `interface`, followed by *.java*. So, if our project contains classes `Book`, the class `Author`, and the class `ExamplesBooks`, we will need to define these classes in files *Book.java*, *Author.java*, and *ExamplesBooks.java*. Typically, each *Project* contains all files that are used to solve one problem.

Moving to standard Java: Visibility modifiers

The first new feature of the standard Java we need to introduce is the use of *visibility modifiers*. In Java every class, interface, field, method declaration, and method definition in Java typically starts with one of the words `public`, `private`, or `protected`. The fields and methods declared to be `public` can be accessed and are visible to all other classes — the way we have been using the fields and methods in *FunJava*. Fields or methods declared to be `private` can only be accessed within the class in which they are defined. So, for example, if we need a helper method that is not relevant for anyone using our class, we would make this a `private` method. We will have example of the use of the `private` visibility modifiers over the next couple of weeks.

If the visibility modifier is omitted, as we have done, the methods and fields can be used by any other classes within the same *package*. In our projects, all classes are defined in the *default* package, and so we only need to add the *visibility modifiers* when it serves a specific purpose:

- When a class implements an interface which includes method declarations, every method definition in the class that implements a method declared in the interface must be annotated with the `public` visibility modifier. This is because defining a `private` method in an interface would be meaningless.

- If a class (possibly `abstract`) defines a method, the class that `extends` it *cannot reduce the visibility* of this method. If the `super` class defines the method as `public`, the subclass must also define it as `public`.

We will worry about the `protected` visibility modifiers later.

Moving to standard Java: Setting up a Project

- Create a new *Project* in Eclipse, name it *Date*.
- Right click on the *src* block under *Date* in the *Pacakage Explorer* pane. Select *New* then *File* in the *File* menu name your file *Date.java*.
- Copy the following data definition into your **Date.java** file and save the file:

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

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

- Create a new file *ExamplesDates.java* while the *default package* block (under the *src* block)is highlighted. This is where you will define the examples and tests for the `Date` class.
- Define the *default constructor* for the class `ExamplesDates`:

```
ExamplesDates(){}
```

- Define in the `ExamplesDates` class three examples of valid dates.
- Import *tester.jar* as *External Jar*, as we have done before.

Moving to standard Java: Setting up the Run Configuration

- Highlight *Date* project in the *Package Explorer* pane.
- In the *Run* menu select *Run Configurations....*
- In the top left corner of the inner pane click on the leftmost item. When you mouse over it should show *New launch configuration*.
- Select the name for this configuration - usually the same as the name of your project.
- In the *Main class:* click on *Search....*
- Among *Matching items* select *Main - tester* and hit *OK*.
- Select the *Arguments* tab and type in the name of your *Examples* class in double quotes. For this example it would be `"ExamplesDates"`. Notice, this is the name of the class, not the name of the file.
- At the bottom of the *Run Configurations* select *Apply* then *Run*.
- Next time you want to run the same project, make sure *Date.java* is shown in the main pane, then hit the green circle with the white triangle on the top left side of the main menu.

Moving to standard Java: Zipping up the Project

You can create an archive of your project by highlighting the project, then choose **Export** then select **Archive File**. Eclipse will ask you for a folder where to place the zip file and will let you choose the name for the zip file.

Your project will remain in the Eclipse workspace, but now you have saved a copy that will not change as you keep working.

This is also the file that you will be submitting as your homework.

5.2 Understanding Constructors: Data Integrity; Signaling Errors

Goals

In this part of this lab you will practice the use of constructors in assuring data integrity and providing a better interface for the user.

Designing constructors to assure integrity of data.

The data definitions at times do not capture the meaning of data and the restrictions on what values can be used to initialize different fields. For example, if we have a class that represents a date in the calendar using three integers for the day, month, and year, we know that our program is interested only in some years (maybe between the years 1500 and 2500), the month must be between 1 and 12, and the day must be between 1 and 31 (though there are additional restrictions on the day, depending on the month and whether we are in a leap year).

Suppose we use the `Date` class to check for overdue books.

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

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

and a simple set of examples:

```
class ExamplesDates {
 ExamplesDates() {}

  // good dates
  Date d20060928 = new Date(2010, 2, 28);     // February 28, 2010
  Date d20071012 = new Date(2009, 10, 12);    // Oct 12, 2009

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

Look at the third example of a date.

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 1500 and 2500.

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.

Design at least one of these methods - you can finish the others at home. For the purposes of being able to test at least the part of the program that is completed, have the other methods produce `true` for the time being. (We call such temporary method definitions *stubs*.)

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

```
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

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

in the code causes the program to terminate and print the specified error message.

We want to make sure that this constructor will indeed accept only the valid dates.

The *tester* library **version 1.3.5 released on 5 February 2010** (please, download the new version) allows us to test this constructor.

It provides two test cases:

```
t.checkConstructorException(String testName,
  Exception e, String className,
  Arg1Type arg1, Arg2Type arg2, ...);

t.checkConstructorException(
  Exception e, String className,
  Arg1Type arg1, Arg2Type arg2, ...);
```

The following test case verifies that the constructor throws the correct exception with the expected message, if the supplied year is 3000:

```
t.checkConstructorException(
  new IllegalArgumentException("Invalid year in Date."),
  "Date", 3000, 12, 30);
```

Run the program with this test. Now change the test by providing an incorrect message, incorrect exception (e.g. `NoSuchElementException`), or by supplying data that do not cause the constructor to throw an exception. Observe the messages that come with the failed tests.

Java provides the class `RuntimeException` with a number of subclasses that can be used to signal different types of errors.

We will learn how to design a new subclass of the `RuntimeException` class that is designed to deal with errors specific to our program at some later date.

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 `2010`. 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 (`2010` 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:

```
Date(int month, int day){
  this(2010, month, day);
}
```

Add examples that use only the month and day to see that the constructor works properly. Include tests with invalid month or year as well.

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

The user may want to enter the date in the form "Oct 20 2010". To make this possible, we can add another constructor:

```
Date(String month, int day){ ...
}
```

Our first task is to convert the `String` that represents the month into a number. We can do it in a helper method `getMonthNo`:

```
// convert a three letter month code into the numeric month value
// return 13 if the month code is not valid
int getMonthNo(String month){
if (month.equals("Jan")){ return 1;}
else {if (month.equals("Feb")){ return 2;}
else {if (month.equals("Mar")){ return 3;}
else {if (month.equals("Apr")){ return 4;}
        ...
else {return 13;}}}}}}}}}}}}
}
```

Our constructor can then invoke this method as follows:

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

  if (this.validMonth(this.getMonthNo(month)))
    this.month = this.getMonthNo(month);
  else
    throw new IllegalArgumentException("Invalid month in Date.");

  if (this.validDay(day))
    this.day = day;
  else
    throw new IllegalArgumentException("Invalid day 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 it would be designed differently.

5.3 Understanding Equality

Note: This material is covered in pages 321 - 330 in the textbook. Read it carefully.

1. Download the file *Lab5a.zip*. Create a Java Project and add following files to it's source directory.

   - *Account.java*
   - *Checking.java*
   - *Savings.java*
   - *Credit.java*
   - *ExamplesBankAccts.java*

   We now want to define a method that will determine whether an account is the same as the given account. We may need such method to find the desired account in a list of accounts.

   Of course, now that we have the abstract class it would be easy to compare just account number and the name on the account. But, maybe, we want to make sure that the customer's data match the data we have on file exactly - including the balances, the interest rates, and the minimum balances - as applicable.

   The design of the method `same` is similar to the technique described in the textbook. The relevant classes and examples that were handed out in the class can be found in the file *Coffee.java*. You may want to look at the code there as you work through this problem.

2. Begin by designing the method `same` for the abstract class `Account`.

3. Make examples that compare all kinds of accounts - both of the same kind and of the different kinds. For the accounts of the same kind you need both the expected `true` answer and the expected `false` answer. Comparing any checking account with another savings account must produce `false`.

4. Now that you have sufficient examples, follow with the design of the `same` method in one of the concrete account classes (for example the `Checking` class). Write the template and think of what data and methods are available to us.

5. You will need a helper method that determines whether the given account is a Checking account. So, design the method `isChecking` that determines whether this account is a checking account. You need to design this method for the whole class hierarchy - the abstract class `Account` and all subclasses. Do the same to define the methods `isSavings` and `isCredit`.

6. We are not done. This helps with the first part of the `same` method. We need another helper method that tells Java that our account is of the specific type. Here is the method header and purpose for the checking account case:

   ```
   // produce a checking account from this account
   Checking toChecking();
   ```

   In the class `Checking` the body will be just

   ```
   // produce a checking account from this account
   Checking toChecking(){
     return this; }
   ```

   Of course, we cannot convert other accounts into checking account, and so the method should throw a `RuntimeException` with the appropriate message. We need the same kind of method for every class that extends the `Account` class.

7. Finally, we can define the body of the `same` method in the class `Checking`:

   ```
   // produce a checking account from this account
   boolean same(Account that){
     if (that.isChecking()){
       return that.toChecking().sameChecking(this);
     } else {
       return false;
     }
   }
   ```

   That means, we still need the method `sameChecking` but this only needs to be defined within the `Checking` class and can be defined with a `private` visibility.

   Finish this - with appropriate test cases.

8. Finish designing the `same` method for the other two account classes.

Alternative approaches - bad and good

Note 1 - Incorrect alternative:

The method above can be written with two Java language *features*, the `instanceof` operator and *casting* as follows:

```
// produce a checking account from this account
boolean same(Account that){
  if (that instanceof Checking){
    return ((Checking)that).sameChecking(this);
  } else {
    return false;
  }
}
```

However, this version is problematic and not safe.

If the class `PremiumChecking` extends `Checking`, then any object constructed with a `PremiumChecking` constructor will be an instance of `Checking` and the trouble that can result is illustrated in the example *TestSame.java*. You can make a simple project and run the examples, but we include the output from the *tester* for illustration.

Note 2 - A correct alternative:

In the lecture we have introduced another version that also works correctly. It requires us to add a new method for each `class` that `implements` the common interface.

Lecture Notes for the lecture on equality for unions of classes show this technique for the classes that represent geometric shapes (`IShape`, `Circle`, `Rect`, and `Combo`).

Here the methods were:

```
// is this shape the same as the given shape?
boolean sameShape(IShape that);

// is this shape the same as the given circle?
boolean sameCircle(Circle that);

// is this shape the same as the given rectangle?
boolean sameRect(Rect that);

// is this shape the same as the given circle?
boolean sameCombo(Combo that);
```