Handout 7

Defining Classes – part 1.
Instance variables and instance methods.

- In Object Oriented programming, applications are comprised from Objects that interact with each other.
- Complex data items (Objects) cannot be represented with a single primitive value, but use a combination of values, also called attributes, (or, instance variables) in Java.
- Class types define
  - what each object of that class consists of (attributes, a.k.a. instance variables),
  - and
  - what each object of that class can do (behaviors, a.k.a. instance methods).

Each Object is an instance of its Class.

Example: Date class – represents a calendar date
  DateDemo – contains a main method that uses Date.

```java
/**
 *  File: Date.java
 *  The simplest definition of class Date
 *  Contains only instance variables.
 *  Note that instance variables are declared public
 */

public class Date
{
    // declare instance variables
    public String month; //always 3 letters long, as in Jan, Feb, etc.
    public int day;
    public int year; //a four digit positive number.
}
```

Important points:
- A class definition as shown above defines the structure of any object of class Date.
  Using UML notation, this structure is depicted as follows:

```
CLASS Date
    String month
    int day
    int year

Instance variables

Instance methods
```

- It does not create any objects.
- It cannot be executed by itself, because it does not contain method main.
- When method main is present, the class can be executed.
/** File: DateDemo.java  
* Contains main method that creates objects of class Date*/

public class DateDemo {
    public static void main(String[] args) {
        Date date1, date2;  // declare two variables of class Date
        date1 = new Date();  // create a new Date object, assign to date1
        date2 = new Date();  // create a new Date object, assign to date2

        // assign values of instance variables
        // only possible, because instance vars declared public
        date1.month = "Dec";
        date1.day = 31;
        date1.year = 2006;
        System.out.println("date1: defines " + date1.day + " "+
                           date1.month + ", " + date1.year);
        date2.month = "Oct";
        date2.day = 27;
        date2.year = date1.year - 1;
        System.out.println("date2: defines " + date2.day + " "+
                           date2.month + ", " + date2.year);

        Date date3 = date2;  // declare and assign date3
    }
}

Important points:
- Class DateDemo contains a main method, which creates objects of Date class. When program reaches statement new Date(); Java locates the definition of class Date (Date.java/.class) and creates an object of class Date in memory, with all of the instance variables of the class Date.

<table>
<thead>
<tr>
<th>CLASS Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>String month</td>
</tr>
<tr>
<td>int day</td>
</tr>
<tr>
<td>int year</td>
</tr>
</tbody>
</table>

- Class type variables are reference variables. They store a reference (pointer) to the location of the actual object.

Each Object is an instance of its Class.
Has specific values for all attributes
• Class DateDemo should be compiled together with Date class.
• The same main method could be placed inside the Date class and then the Date class could be executed. Placement of the main method is a program designer’s choice.

**Instantiation:** Creation of an object from a class.

Use the keyword `new` followed by the name of the class and (), as in

```java
Date date1 = new Date();
```

**Instance variables:**
Each object contains its own copies of instance variables.
Can be accessed by any instance method of the same class.

**Public versus private** access modifier:

• **Public** instance variables/methods can be accessed from any method of any class.
• **Private** instance variables/methods can only be accessed from instance methods of their own class

**Proper OO design** uses information hiding, a.k.a. encapsulation
Define instance variables as **private**. This is also the default.

• Only methods of that class can access private instance variables

Define **operations** on that class as **public methods**

• Available to users of the class

**Proper definition of class Date:**
Instance variables are declared as **private**, i.e. cannot be set as

```java
date1.year = 2006,
```

or used as in

```java
System.out.println("date1:" + date1.day + " " + date1.month + " " + date1.year);
```

Therefore, need to provide **instance methods** to modify and return the value of the instance variables.

To provide information about the Object’s private data to other Objects – must define **accessor** methods. **Accessor (get)** method for each attribute returns the value of the attribute (instance variable).

To provide a way of assigning values of an Object’s private attributes – **mutator (set)** methods are provided. Mutator methods can also perform a check on the validity of the new value, as done in method `setDate()` (see further definition of class Date).
Methods

Methods are used to implement a functionally independent task. For an example, consider methods of class `String`: `length()`, `charAt()`, `Substring`, etc..

All methods have
- name
- zero or more input parameters (input) and
- one or no return value (output).

Instance methods also receive a reference to the **calling object** as an implicit (i.e. unspecified) parameter and therefore can access instance variables of the calling object.

- The calling object is referenced by keyword **this**.

Methods may also have
- Local variables and constants

Method declaration

- A **method declaration** specifies the code that will be executed when the method is **invoked** (or **called**)

  A method declaration begins with a method **header**:

  
  ```java
  public boolean setDate (String aMonth, int aDay, int aYear)
  ```

  return type, i.e. type of return value method name type and name of formal parameter (a.k.a. argument)

- Followed by method **body** in `{ ... }

Method invocation

When a method is invoked,

- arguments are evaluated and passed as values of parameters
- the flow of control jumps to the first line of the method’s body and executes the code inside method body.
- return statement (or the end of method) terminates method execution. Then the control flow returns to the place where the method was called

- The invocation may or may not return a value, depending on how the method is defined. A method that does not return a value has a **void** return type.
Illustration of Parameter passing:

Method setDate() is invoked in DateDemo.java

```java
date1.setDate("Dec", 31, 2006);
```

What happens:

a. Parameters in the call are evaluated and assigned to the method’s arguments, i.e.

```
"Dec"  31  2006
```

```java
public void setDate (String aMonth, int aDay, int aYear)
```

b. The control is passed to the first statement of the method.

c. Statements are executed in order until a return statement is encountered, or the end of method is reached.

d. Return statement terminates the execution of the method and passes control back to the point from where the method was invoked. The returned value is substituted in place of the call.

Notes on the return statement:

- the type of value returned must match the method’s return type, specified in the method header.
- a method may have any number of return statements, but it is a rule of good programming style to have a unique exit point from a method.
- return statement terminates the method execution.
- a method that does not return a value has a void return type.

Scoping rules

- Instance methods of class can use instance variables of the class. These variables are referenced by name (e.g. month), or using the keyword this (e.g. this.month)
- Public instance variables/methods can be accessed from any method of any class.
- Private instance variables/methods can only be accessed from instance methods of their own class.
- Local variables can be declared inside a method.
- Local variables and formal parameters are accessible only from within the method definition (or more precisely, within the enclosing {}’s) and are not accessible from outside of the method.
- The formal parameters of a method create automatic local variables when the method is invoked.
- When the method execution is complete, all local variables are destroyed, including the formal parameters.
public class Date
{
    // declare instance variables
    private String month;  // always 3 letters long, as in Jan, Feb, etc.
    private int day;
    private int year;  // a four digit number.

    // public boolean setDate (String aMonth, int aDay, int aYear )
    // mutator method - assigns values to instance vars
    // performs simple error checking to test validity of date params
    // and returns true if date was successfully created
    // returns false, otherwise.
    public boolean setDate (String aMonth, int aDay, int aYear ){
        boolean valid;  // local variable designating if date is valid

        if (aYear < 0 || aDay <= 0 || aDay > 31) {
            System.out.println("Error: invalid date parameters.");
            valid = false;
        } else {
            // set instance variables of calling obj. to values of params
            this.month = aMonth;
            this.day = aDay;
            this.year = aYear;
            valid = true;
        }
        return valid;  // terminates the method, passing value of valid
        // back to where setDate method was called
    }

    // accessor methods
    public int getYear() {
        return this.year;  // return value of instance variable year
    }
    public int getDay(){
        return this.day;  // return value of instance variable day
    }
    public String getMonth(){
        return this.month;  // return value of instance variable month
    }
}
/**
 * File: DateDemo.java
 * Contains main method that creates objects of class Date
 */

public class DateDemo
{
    public static void main(String[] args)
    {
        Date date1;          // declare a variable of class Date
        date1 = new Date (); // create a new Date object, assign to date1
        /* The following code no longer works, due to instance variables declared as private
         * date1.month = "Dec";
         * date1.day = 31;
         * date1.year = 2006;
         * System.out.println("date1:" + date1.day + " " + date1.month +
         *          " " + date1.year);
         * Instead, must use accessor (getter) and mutator (setter) methods as shown below */
        date1.setDate("Dec", 31, 2006);

        System.out.println("date1:"+date1.getDay()+ " "+
                date1.getMonth() + " " + date1.getYear());
    }
}

Practice problems

1. Modify the main method of the DateDemo class to prompt the user to enter a value of month, day and year that you will use to create two separate Date objects. Then, create two Date objects and call the mutator method setDate() with the data received from the user to assign instance variables of the created Date objects. Use the value returned by setDate() to verify if the date was set successfully.
2. Add instance method `century()` to the Date class. This method should have no parameters and should return an integer specifying the century corresponding to the year of the calling object.

In the main method of the `DateDemo` class, invoke `century()` to print out the value of the century for the create `Date` object.

3. Add instance method `isSameYear()` to the Date class. This method should have one argument – an object of class `Date`, and should return a boolean value, which is true if and only if the value of the year instance variable of the calling object is the same as the year instance variable of the argument.

In the main method of the `DateDemo` class, invoke `isSameYear()` to print out if two dates are within the same year.