Assignment A1

Write a class to maintain information about PhDs—their advisor(s) and date of PhD.

Objectives in brief:
- Get used to Eclipse and writing a simple Java class
- Learn conventions for Javadoc specs, formatting code (e.g., indentation), class invariants, method preconditions
- Learn about and use JUnit testing

Important: READ CAREFULLY, including Step 7, which reviews what the assignment is graded on.

Groups. You can do the assignment with 1 other person. FORM YOUR GROUP EARLY! Use Piazza Note @5 to search for partner!

Recommended time-table for doing A1

Start A1 the day before it is due? You may be frustrated, upset, rushed because you can’t get the help you need. With 500 students, too many will be trying to get help at the last minute. Not a good educational experience. Instead, use following schedule, which gives you a day or two after each part to get help if you need it:

4 Feb. Spend 20 minutes reading the assignment.
6 Feb. Write and test the Group A methods. This includes writing the Junit test procedure for the group.
7 Feb. Write and test the Group B methods AND the Group C methods.
8 Feb. Write and test the GroupD methods.
11 Feb. Do point 7 of the handout: Review the learning objectives and check each of the items given in point 7. Submit on the CMS.

CHECK the pinned A1 note on the Piazza every day.

Homework

1. Course website will contain classes Time and TimeTester. The body of the one-parameter constructor is not written. Write it. The one-parameter constructor is not tested in TimeTester. Write a procedure to test it.
2. Visit course website, click on Resources and then on Code Style Guidelines. Study
   1. Naming conventions
   3.3 Class invariant
   4. Code organization
   4.1 Placement of field declarations
   5. Public/private access modifiers
3. Look at slides for next lecture; bring them to next lecture.

Overview

- An object can contain variables as well as methods. Variable in an object is called a field.
- Declare fields in the class definition. Generally, make fields private so they can’t be seen from outside the class.
- May add getter methods (functions) and setter methods (procedures) to allow access to some or all fields.
- Use a new kind of method, the constructor, to initialize fields of a new object during evaluation of a new-expression.
- Create a JUnit Testing Class to save a suite of test cases.
**References to text and JavaSummary.pptx**

- Declaration of fields: 8.5-8.6, slide 12
- Getter/setter methods: 8.6, slide 13, 14
- Constructors: 8.17-8.18, slide 15
- Class String: A.67-A.73
- JUnit Testing Class: none, slide 74-80
- Overloading method names: 8.21, slide 22

---

**Class Time**

/\** An instance maintains a time of day */
public class Time {
    private int hr; // hour of the day, in 0..23
    private int min; // minute of the hour, in 0..59

    Access modifier private:
    can't see field from outside class
Software engineering principle:
make fields private, unless there
is a real reason to make public
}

**Getter methods (functions)**

/\** An instance maintains a time of day */
public class Time {
    private int hr; // hour of the day, in 0..23
    private int min; // minute of the hour, in 0..59

    /** Return hour of the day */
    public int getHour() {
        return hr;
    }
    /** Return minute of the hour */
    public int getMin() {
        return min;
    }
}

---

**A little about type (class) String**

public class Time {
    private int hr; // hour of the day, in 0..23
    private int min; // minute of the hour, in 0..59

    /** Return a representation of this time, e.g. 09:05 */
    public String toString() {
        return prepend(hr) + " : " + prepend(min);
    }
    /** Return i with preceding 0, if necessary, to make two chars. */
    private String prepend(int i) {
        if (i > 9 || i < 0)
            return "0" + i;
        return i;
    }
    ...

---

**Class invariant**

/\** An instance maintains a time of day */
public class Time {
    private int hr; // hour of the day, in 0..23
    private int min; // minute of the hour, in 0..59

    Software engineering principle: Always write a clear,
precise class invariant, which describes all fields.
Call of every method starts with class invariant true
and should end with class invariant true.
Frequent reference to class invariant while
programming can prevent mistakes.

---

**Object contains the time of day in hours and minutes.**
Methods in object refer to fields in object.
Could have an array of such objects to list the times at which
classes start at Cornell.

With variables t1 and t2 below,
t1.getHour() is 8

```
t1.getHour() is 8
```
t2.getHour() is 9

```
t2.getHour() is 9
```
t2.toString() is "09:05"

```
t2.toString() is "09:05"
```

---

**Class Time**

```
public class Time {
    private int hr; // hour of the day, in 0..23
    private int min; // minute of the hour, in 0..59

    /** Return hour of the day */
    public int getHour() {
        return hr;
    }
    /** Return minute of the hour */
    public int getMin() {
        return min;
    }
    /** Return i with preceding 0, if necessary, to make two chars. */
    private String prepend(int i) {
        if (i > 9 || i < 0)
            return "0" + i;
        return i;
    }
    ...
}
```
I never concatenate strings; I just concatenate those little things. Of syllables few, I'm a man through and through. Shorter words? My heart joyfully sings!

Setter methods (procedures)

```java
/** An instance maintains a time of day */
public class Time {
    private int hr; //hour of the day, in 0..23
    private int min; // minute of the hour, in 0..59

    /** Change this object's hour to h */
    public void setHour(int h) {
        hr = h;
    }
}
```

No way to store value in a field! We can add a "setter method"

Test using a JUnit testing class

In Eclipse, use menu item File → New → JUnit Test Case to create a class that looks like this:

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class TimeTester {
    @Test
    public void testConstructor() {
        Time t1 = new Time();
        assertEquals(0, t1.getHour());
        assertEquals(0, t1.getMin());
        assertEquals("00:00", t1.toString());
    }
}
```

Test setter method in JUnit testing class

```java
public class TimeTester {
    @Test
    public void testSetters() {
        Time t1 = new Time();
        t1.setHour(21);
        assertEquals(21, t1.getHour());
    }
}
```

TimeTester can have several test methods, each preceded by @Test. All are called when menu item Run → Run is selected.
**Constructors — new kind of method**

```java
public class C {
    private int a;
    private int b;
    private int c;
    private int d;
    private int e;
}
```

C has lots of fields. Initializing an object can be a pain — assuming there are suitable setter methods.

Easier way to initialize the fields, in the new-expression itself. Use:

```java
C var = new C();
var.setA(2);
var.setB(20);
var.setC(35);
var.setD(-15);
var.setE(150);
```

But first, must write a new method called a constructor.

```java
C var = new C(2, 20, 35, -15, 150);
```

Easier way to initialize the fields, in the new-expression itself. Use:

```java
public class C {
    private int a;
    private int b;
    private int c;
    private int d;
    private int e;

    public C(int a, int b, int c, int d, int e) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
        this.e = e;
    }
}
```

### Revisit the new-expression

**Syntax of new-expression:**  
`new <constructor-call>`

**Example:**  
`new Time(9, 5)`

**Evaluation of new-expression:**  
1. Create a new object of class, with default values in fields
2. Execute the constructor-call
3. Give as value of the expression the name of the new object

If you do not declare a constructor, Java puts in this one:

```java
public <class-name> () {
}
```

### How to test a constructor

Create an object using the constructor. Then check that all fields are properly initialized — even those that are not given values in the constructor call.

```java
public class TimeTester {
    @Test
    public void testConstructor1() {
        Time t1 = new Time(9, 5);
        assertEquals(9, t1.getHour());
        assertEquals(5, t1.getMin());
    }
    ...
}
```

Note: This also checks the getter methods! No need to check them separately.

But, main purpose: check constructor.

### A second constructor

```java
/** An object maintains a time of day */
public class Time {
    private int hr; // hour of day, 0..23
    private int min; // minute of hour, 0..59

    /** Constructor: an instance with h hours and m minutes. 
     * Precondition: m in 0..(23*60 + 59)
     */
    public Time(int m) {
        hr = m/60; min = m%60;
    }
    ...
}
```

**Time is overloaded: 2 constructors! Have different parameter types. Constructor call determines which one is called**

```java
Time t1 = new Time(9, 5);
assertEquals(9, t1.getHour());
assertEquals(5, t1.getMin());
```

### Generate javadoc

- With project selected in Package explorer, use menu item Project -> Generate javadoc
- In Package Explorer, click on the project -> doc -> index.html
- You get a pane with an API like specification of class Time, in which javadoc comments (start with /****) have been extracted!
- That is how the API specs were created.
Method specs should not mention fields

```java
public class Time {
    private int hr;  // in 0..23
    private int min; // in 0..59

    /** return hour of day*/
    public int getHour() {
        return hr;
    }
}
```

Decide to change implementation

Specs of methods stay the same.
Implementations, including fields, change!