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:
1 Sep. Spend 20 minutes reading the assignment.
3 Sep. Write and test the Group A methods. This includes writing the Junit test procedure for the group.
5 Sep. Write and test the Group B methods AND the Group C methods.
6 Sep. Write and test the Group D methods.
8 Sep. 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.
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;
    }
}

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.
}

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

    public String toString() {
        return prepend(hr)  +  "\" +  prepend(min);
    }
}

Time@fa8
hr 9
min 5

t1 @150
hr 8
getMinute()
t2 @fa8
hr 9
getHour()
Concatenate or catenate?

I never concatenate strings; I just catenate 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;
    }
}
```

Test using a JUnit testing class

```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());
    }
}
```
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.

```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
/** 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: h in 0..23, m in 0..59
     */
    public Time(int h, int m) {
        hr = h;
        min = m;
    }
}
```

**Purpose of constructor:** Initialize field of a new object so that its class invariant is true.

**No return type or void**

**Name of constructor** is the class name

**Memorize!**

Precondition: h in 0..23, m in 0..59

```java
new Time(int h, int m)
```

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> () { }
```

**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 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

**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

| public class Time {
| private int hr;  //in 0..23
| private int min; //in 0..59
| /** return hour of day*/
| public int getHour() {
|     return hr;
| }

private int hr;  //in 0..23
private int min; //in 0..59
/** return hour of day*/
public int getHour() {
    return min / 60;
}

Time hr 9
min 5
getHour() getMin() toString() setHour(int)

Decide to change implementation

Decide to change implementation

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