Assignment A1

Write a class to maintain information about PhDs --- e.g. their advisor(s) and date of PhD. Pay attention today, you will do exactly what I do in creating and testing a class.

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 8, which reviews what the assignment is graded on.

Groups. You can do A1 with 1 other person. FORM YOUR GROUP EARLY! Use Piazza Note @5 to search for partner!

Homework (not to be handed in)

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

Difference between class and object

A blueprint, design, plan
A class
Can create many objects from the same plan (class). Usually, not all exactly the same.

A house built from the blueprint
An object

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 in JavaHyperText entries

Look at these JavaHyperText entries:

- Declaration of fields: field
- Getter/setter methods: getter setter
- Constructors: constructor
- Class String: toString
- JUnit Testing Class: Junit
- Overloading method names: overload
- Overriding method names: override

---

class Time

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 $t_1$ and $t_2$ below,

$t_1$.getHour() is 8  
$t_2$.getHour() is 9  
$t_2$.toString() is "09:05"

---

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
}

---

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
  \n  \n  **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.
}

---

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;
  }

  // ...
Concatenate or concatenate?

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)

/** 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;
  }
}

Do not say "set field hr to h"
User does not know there is a field. All
user knows is that Time maintains hours
and minutes. Later, we show an imple-
mentation that doesn't have field h but "behavior" is
the same

Test using a JUnit testing class

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

import static org.junit.Assert.*;
import org.junit.Test;
public class TimeTester {
  @Test
  public void test() {
    fail("Not yet implemented");
  }
}

Select TimeTester in Package
Explorer.
Use menu item Run → Run.
Procedure test is called, and the call fail(...) causes
execution to fail.

Test setter method in JUnit testing class

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

**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 fields of a new object so that its class invariant is true.

No return type or void
Name of constructor is the class name

Memorize!

**Constructors** — new kind of method

```
new <constructor-call>
```

Example:

```java
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.

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;
    }
}
```

Next week's section: Exception Handling

Suppose we are supposed to read an integer from the keyboard and do something with it. If the user types something other than an integer, we want to ask the user again to type an integer.

```java
String st= the integer from the keyboard;
int k= Integer.parseInt(st);  // return the int that is in st
```

User typed “x13”, it was discovered here

`parseInt` doesn’t know what to do with the error

So it creates and throws a `NumberFormatException` to the caller. `parseInt` is then terminated. It’s done.

You will learn how the caller can catch the exception and ask user again to type an int.

Next week’s section: Exception Handling

You must read/watch the tutorial BEFORE the recitation:

Look at the pinned Piazza note Recitation/Homework.
Bring your laptop to class, ready to answer questions, solve problems. The questions will be on the course website the night before section (Monday evening).

During the section, you can talk to neighbors, discuss things, answer questions together. The TA will walk around and help. The TA will give a short presentation on some issue if needed.

You will have until Friday after the recitation to submit answers on the CMS.