

1

CS/ENGRD 2110 SPRING 2014

Lecture 3: Fields, getters and setters, constructors, testing
<http://courses.cs.cornell.edu/cs2110>

Assignment A1 is on the CMS and Piazza

2

Write a simple class to maintain information about bees.
 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.

Homework

3

1. Course website contains 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

4

- 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

5

Declaration of fields: `B.5-B.6` slide 12
 Getter/setter methods: `B.6` slide 13, 14
 Constructors: `B.17-B.18` slide 15
 Class String: `A.67-A.73`
 Junit Testing Class: none slide 74-80
 Overloading method names: `B-21` slide 22

class Time

6

Object contains the time of day in hours and minutes.
 Methods in object refer to field 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
`t2.getHour()` is 9
`t2.toString()` is "09:05"

```

t1 Time@150
   hr 8
   min 0
   getHour()
   getMin()
   toString()

t2 Time@fa8
   hr 9
   min 5
   getHour()
   getMin()
   toString()
    
```

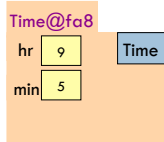
A class Time

```

7
/** 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



Class invariant

```

8
/** 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:
collection of defs of variables and constraints on them (green stuff)

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.

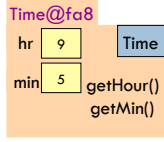
Frequent reference to class invariant while programming can prevent mistakes.

Getter methods (functions)

```

9
/** 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;
    }
}
    
```

Spec goes before method. It's a Javadoc comment —starts with /**



A little about type (class) String

```

10
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 "" + i;
        return "0" + i;
    }
}
    
```

Java: double quotes for String literals

Java: + is String catenation

Catenate with empty String to change any value to a String

"helper" function is private, so it can't be seen outside class

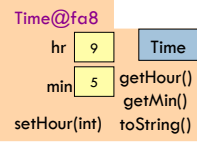
Setter methods (procedures)

```

11
/** 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"

setHour(int) is now in the object



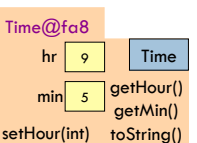
Setter methods (procedures)

```

12
/** An instance maintains a time of day */
public class Time {
    private int hr; //hour of day, in 0..23
    private int min; // minute of 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 implementation 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 using a Junit testing class

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

Write and save a suite of "test cases" in **TimeTester**, to test that all methods in **Time** are correct

Store new **Time** object in **t1**.

Give green light if expected value equals computed value, red light if not: **assertEquals(expected value, computed value);**

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

```
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:

```
C var = new C();
var.setA(2);
var.setB(20);
var.setC(35);
var.setD(-1.5);
var.setE(1.50);
```

C = new C(2, 20, 35, -1.5, 1.50);

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

Constructors —new kind of method

```
/** 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

Memorize!

No return type or void

Name of constructor is the class name

Revisit the new-expression

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

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

Evaluation of new-expression:

- Create a new object of class, with default values in fields
- Execute the constructor-call
- Give as value of the expression the name of the new object

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

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

How to test a constructor

19

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

```

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

20

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

```

/** 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
        m minutes.
        Precondition: m in 0..23*60 +59 */
    public Time(int m) {
        ??? What do we put here ???
    }
    ...
    new Time(9, 5)
    new Time(125)
    
```

Time@fa8
hr 9 min 5 Time
getHour() getMin()
toString() setHour(int)
Time(int, int) Time (int)

Method specs should not mention fields

21

```

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

→ Decide to change implementation

```

public class Time {
    // min, in 0..23*60+59
    private int min;
    /** return hour of day*/
    public void getHour() {
        return min / 60;
    }
}
    
```

Time@fa8
min 5 Time
getHour() getMin()
toString() setHour(int)

Time@fa8
hr 9 Time
min 5 getHour() getMin()
setHour(int) toString()

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