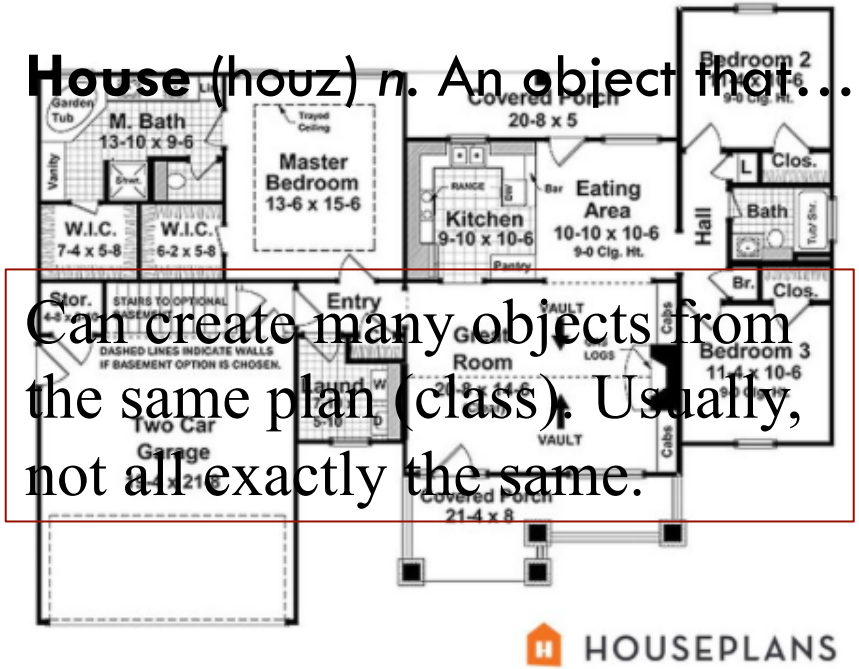
CS/ENGRD 2110 FALL 2017

Lecture 3: Fields, getters and setters, constructors, testing

http://courses.cs.cornell.edu/cs2110

Object-Oriented Programming

Classes



House (houz) *n.* An object that…

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

A blueprint, plan, **a definition** A

Objects

class House

Object is an instance of a house. Contains bdrs (number of bedrooms) and baths (number of bathrooms)

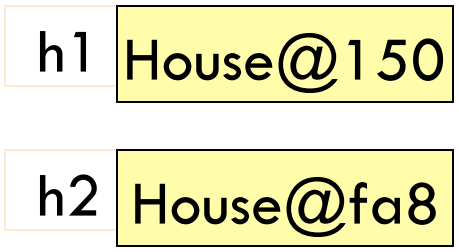
Methods in object refer to fields in object.

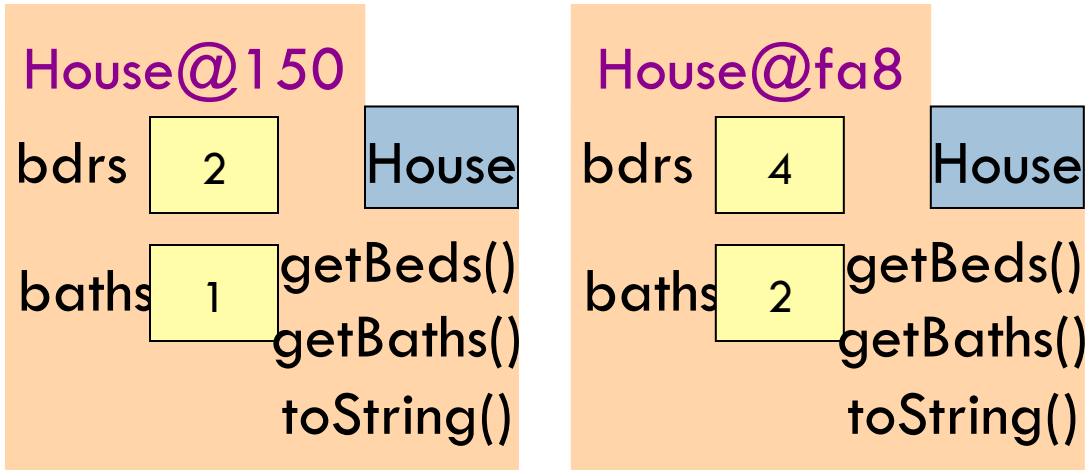
Could have an array of such objects to list the apartments in a building.

With variables h1 and h2 below,

t1.getBeds() is 2

t2.getBeds() is 4





Class invariants

Class invariant: collection of defs of variables and constraints on them (green stuff)

```
/** An instance maintains info for a house */
public class House {
    private int bdrs;    // number of bedrooms, in 0...10
    private int baths;   // number of bathrooms, in 1...5
}
```

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.

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!

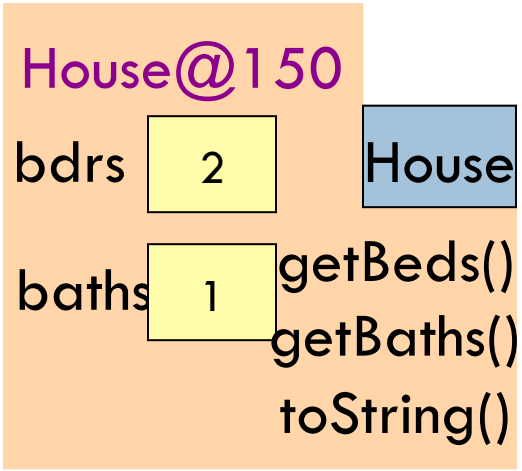
- That is how the API specs were created.

Class House


```
/** An instance maintains info for a House */
public class House {
    private int bdrs;    // number of bedrooms, in 0...10
    private int baths;   // number of bathrooms, in 1...5

}
```

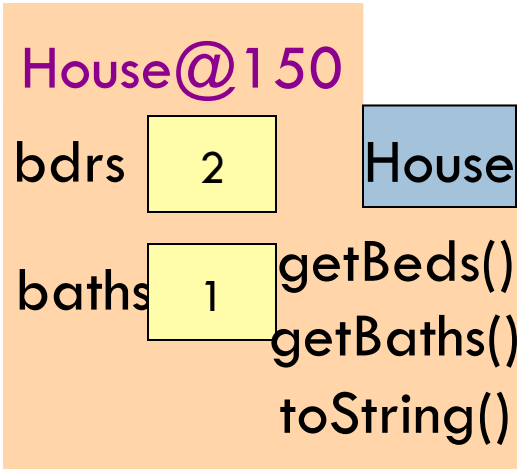
Access modifier **private**:
can't see field from outside class
Software engineering principle:
make fields private, unless there is a real reason to make public



Getter methods (functions)


```
/** An instance maintains info for a house */
public class House{
   private int bdrs;    // number of bedrooms, in 0..10
   private int baths; // number of bathrooms, in 1..5
   /** Return number of bedrooms */
   public int getBeds() {
      return bdrs;
   }
   /** Return number of bathrooms */
   public int getBaths() {
      return baths;
   }
}
```

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



Setter methods (procedures)


```
/** An instance maintains info for a house */
public class House{
   private int bdrs;    // number of bedrooms, i
   private int baths; // number of bathrooms, i

   …
   /** Change number of bathrooms to b */
   public void setBeds(int b) {
         bdrs=  b;
   }

}
```

setBeds(int) is now in the object

Do not say

"set field bdrs to b"

User does not know there is a field. All user knows is that House maintains bedrooms and bathrooms. Later, we show an imple-mentation that doesn't have field b but "behavior" is the same

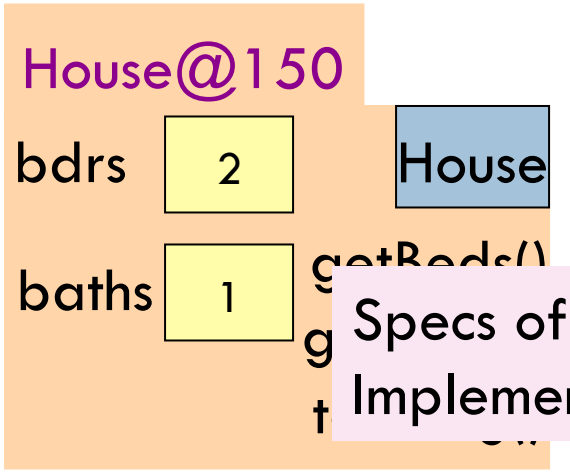
Method specs should not mention fields

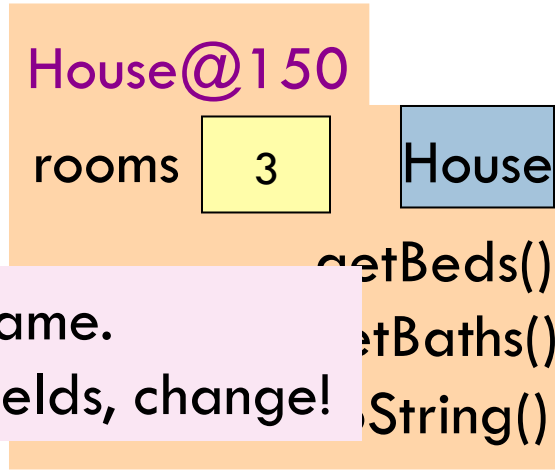

```
public class House{
  private int bdrs;    //in 0..10
  private int baths; //in 1..5
  /** return number of rooms*/
  public int getRooms() {
     return bdrs+baths;
  }
```

Decide to change implemen-tation

```
public class House{

  private int rooms; // rooms, in
                                1..15
/** return number of rooms*/
public int getRooms() {
      return rooms;
}
```





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

A little about type (class) String


```
public class House{
   private int bdrs;    // number of bedrooms, in 0..10
   private int baths; // number of bathrooms, in 1..5
   /** Return a representation of this house */
   public String toString() {
     return plural(bdrs)  +  "," + baths;

   /** Return i with preceding 0, if
       necessary, to make two chars. */
   private String plural(int i) {
     if (i ==1) return "" + i + "bedroom";
     return "" + i + "bedrooms";
   }
   …
```

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

Test using a JUnit testing class

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

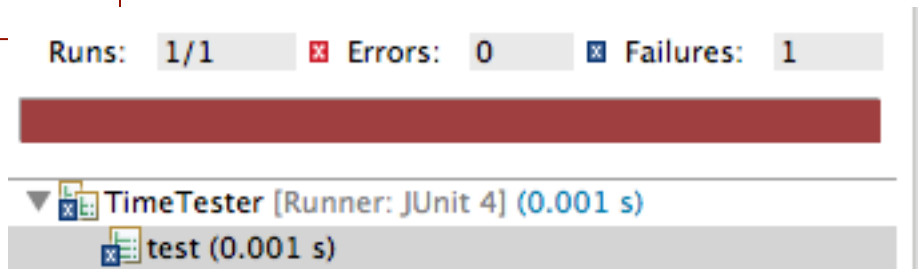
```
import static org.junit.Assert.*;
import org.junit.Test;

public class HouseTester {
    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```

Select HouseTester 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 HouseTester {

    …


    @Test
    public void testSetters() {
        House h= new House();
        h.setBeds(2);
        assertEquals(2, h.getBeds());
    }
}
```

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

Store new House object in h.

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

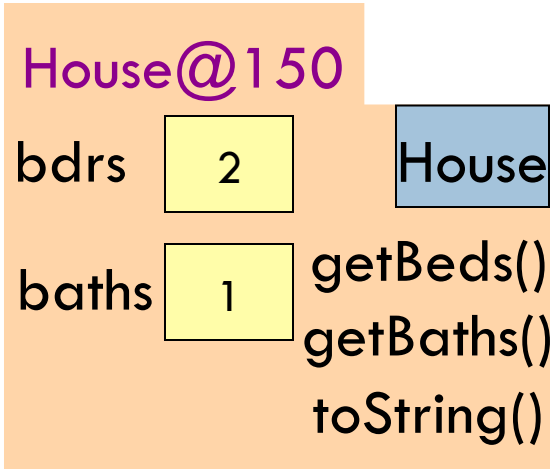

```
public class HouseTester {

    …


    @Test
    public void testSetters() {
        House h= new House();
        h.setBeds(2);
        assertEquals(2, h.getBeds());
    }
}
```

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

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(2, 20, 35, -15, 150);
```

But first, must write a new method called a constructor

Constructors —new kind of method


```
/** An object maintains info about a house */
public class House{
   private int bdrs;    // number of bedrooms
   private int baths; // number of bathrooms.
   /** Constructor: an instance with
       bd bedrooms and bth bathrooms.
       Precondition: bd in 0..10, bth in 1..5*/
   public House(int bd, int bth) {
      bdrs= bd;
      baths= bth;
   }
```

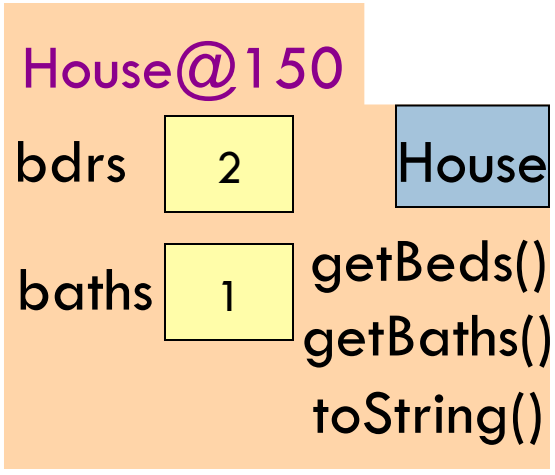
Purpose of constructor: Initialize fields of a new object so that its class invariant is true

Memorize!

Need precondition

No return type or void

Name of constructor is the class name



Revisit the new-expression

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

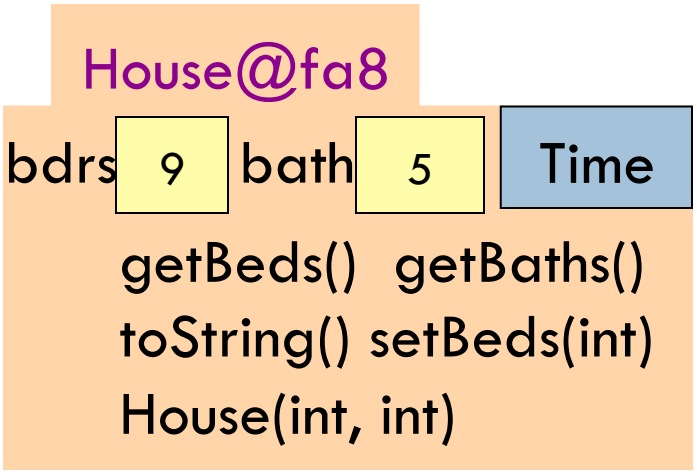
Example: **new** House(9, 5)

House@fa8

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:
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

```
public class HouseTester {
    @Test
    public void testConstructor() {
        House h= new House(9, 5);
         assertEquals(9, h.getBeds());
         assertEquals(5, h.getBaths();
    }
    …
}
```

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

But, main purpose: check constructor

Recap

- An object is defined by a class. An object can contain variables (fields) as well as methods (functions/procedures).
- Use comments and javadoc to document invariants and specify behavior
- 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.

CS2110 FAQs

- **Lecture Videos:** they're available http://cornell.videonote.com/channels/1027/videos
- **Grading Options:** S/U is fine by us. Check with your advisor/major.
- **Prelim conflicts:** Please don't email us about prelim conflicts! We'll tell you at the appropriate time how we handle them.
- **Other Questions:** check course Piazza regularly for announcements.

Recitation This Week

You must read/watch the tutorial BEFORE the recitation:

www.cs.cornell.edu/courses/cs2110/2017fa/online/exceptions/EX1.html

Get to it from the Tutorials page of the course website.
NOTE THAT THERE ARE SIX WEB PAGES!

Bring your laptop to class, ready to answer questions, solve problems.

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.

Homework questions are on the course website. You will have until a week after the recitation (on a Wednesday night) to submit answers on the CMS.

Assignments

- A0 out: Due this Thursday (8/31)
- A1 out: Due next Wednesday (9/6)
- A2 out: Due the following week (9/13)

Assignment A1

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

Objectives in brief:

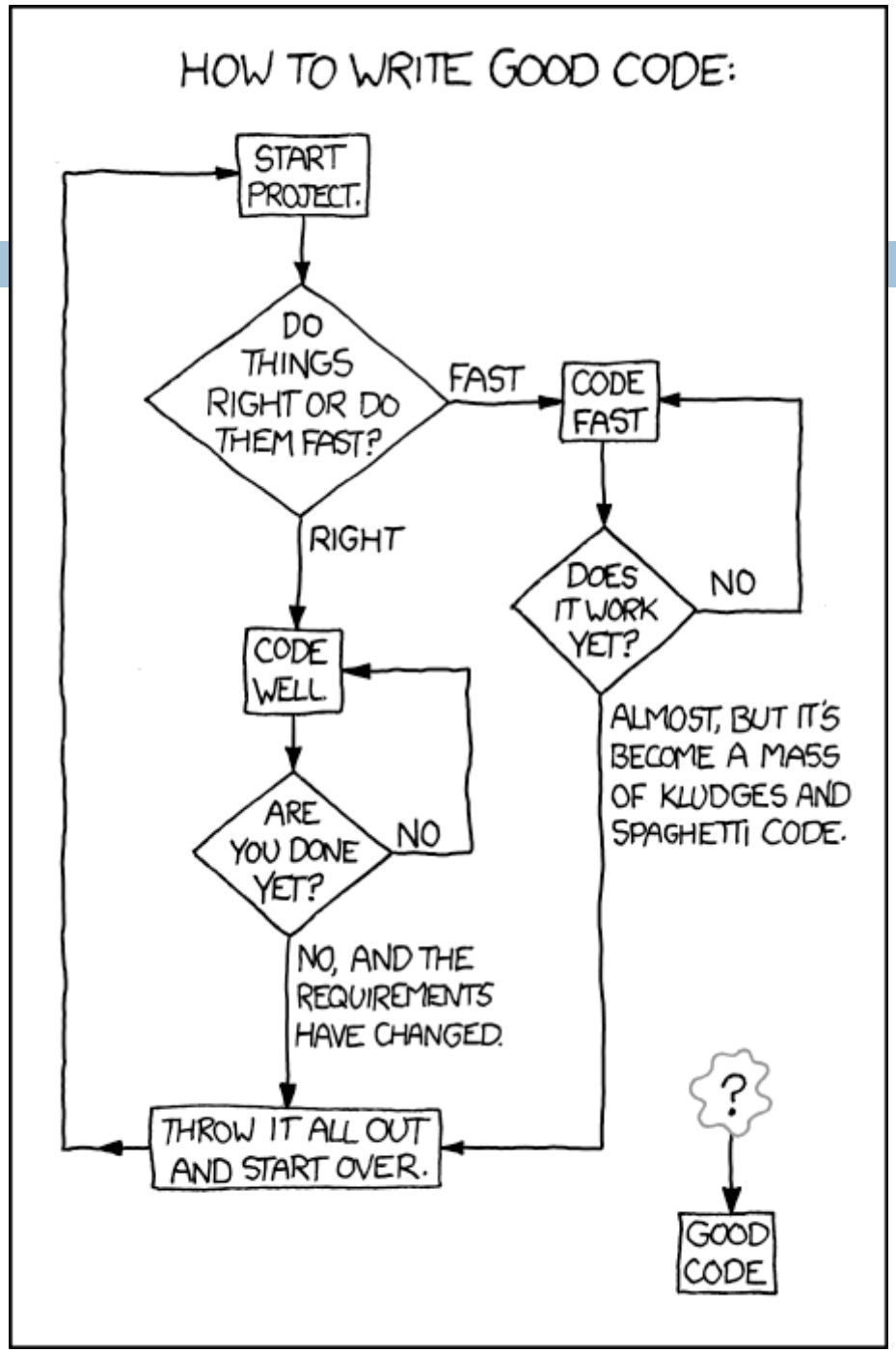
- Get used to Eclipse and writing a simple Java class
- Learn conventions for Javadoc specs, formatting code (e.g. indentation), class invariants, method preconditions
- Learn about and use JUnit testing

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

Assignment A1

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

CHECK the pinned A1 note on the Piazza every day.


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ARE YOU DONE YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE