

Announcements

2

- A1 Due Friday
- A2 Out Today



Where am I? Big ideas so far.

3

- Java variables have *types* (L1)
 - A type is a set of values and operations on them (**int**: +, -, *, /, %, etc.)
- *Classes* define new types (L2) and define the contents of each object of the class.
 - *Methods* are the operations on objects of that class.
 - *Fields* allow objects to contain data (L3)

Class House

4

```
public class House {  
    private int nBed; // number of bedrooms, >= 0.  
    private int nBath; // number of bathrooms, in 1..5
```

```
/** Constructor: bed is number of bedrooms,  
 *           bath is number of bathrooms  
 *   Prec: bed >= 0, 0 < bath <= 5 */
```

```
public House(int bed, int bath) {  
    nBed= bed; nBath= bath;  
}
```

```
/** Return no. of bedrooms */
```

```
public int getNumBed() {  
    return nBed;  
}
```

Contains other methods!

...

House@af8

nBed 3

nBath 1

House(...) getNumBed()

getNumBath() setNumBed(...)

setNumBath(...)

toString()

equals(Object) hashCode()

House

Class Object

5

Class Object

java.lang.Object

```
public class Object
```

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

See Also:

Class

Constructor Summary

Constructors

Constructor and Description

`Object()`

Method Summary

All Methods

Instance Methods

Concrete Methods

Modifier and Type

Method and Description

protected `Object`

`clone()`

Creates and returns a copy of this object.

boolean

`equals(Object obj)`

Indicates whether some other object is "equal to" this one.

protected void

`finalize()`

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

`Class<?>`

`getClass()`

Returns the runtime class of this Object.

int

`hashCode()`

Returns a hash code value for the object.

Class Object: the superest class of all

6

```
public class House extends Object {  
    private int nBed; // number of bedrooms, >= 0.  
    private int nBath; // number of bathrooms, in 1..5  
  
    /** Constructor: bed is number of bedrooms,  
     *                bath is number of bathrooms  
     * Prec: bed >= 0, 0 < bath <= 5 */  
    public House(int bed, int bath) {  
        nBed= bed; nBath= bath;  
    }  
  
    /** Return no. of bedrooms */  
    public int getNumBed() {  
        return nBed;  
    }  
  
    ...  
}
```

Java: Every class that does not extend another class extends class Object.

House@af8

nBed 3

nBath 1

House(...) getNumBed()

getNumBath() setNumBed(...)

House

Class Object: the superest class of all

7

```
public class House extends Object {  
    private int nBed; // number of bedrooms, >= 0.  
    private int nBath; // number of bathrooms, in 1..5
```

```
    /** Constructor: bed is number of bedrooms,  
     *                bath is number of bathrooms  
     * Prec: bed >= 0, 0 < bath <= 5 */
```

```
    public House(int bed, int bath) {  
        nBed= bed; nBath= bath;  
    }  
}
```

```
    /** Return no. of bedrooms */
```

We often omit the Object partition to reduce clutter; we know that it is always there.

Java: Every class that does not extend another class extends class Object.

House@af8

nBed 3

nBath 1

House(...) getNumBed()

getNumBath() setNumBed(...)

House

```
    ...  
}
```

Class Object: the superest class of all

8

```
public class House extends Object {  
    private int nBed; // number of bedrooms, >= 0.  
    private int nBath; // number of bathrooms, in 1..5
```

```
    /** Constructor: bed is number of bedrooms,  
     *                bath is number of bathrooms  
     * Prec: bed >= 0, 0 < bath <= 5 */
```

```
    public House(int bed, int bath) {  
        nBed= bed; nBath= bath;  
    }  
}
```

```
    /** Return no. of bedrooms */
```

We often omit the Object partition to reduce clutter; we know that it is always there.

Java: Every class that does not extend another class extends class Object.

House@af8

toString()
equals(Object) hashCode()

Object

nBed

3

nBath

1

House

House(...) getNumBed()
getNumBath() setNumBed(...)

```
    ...  
}
```

Classes can extend other classes

We saw this in L2!

9

```
/** An instance is a subclass of JFrame */  
public class C extends javax.swing.JFrame {  
  
}
```

C: **subclass** of JFrame

JFrame: **superclass** of C

C **inherits** all methods
that are in a JFrame

object has 3 partitions:
for Object components,
for JFrame components,
for C components

C@6667f34e

equals() toString() ... Object

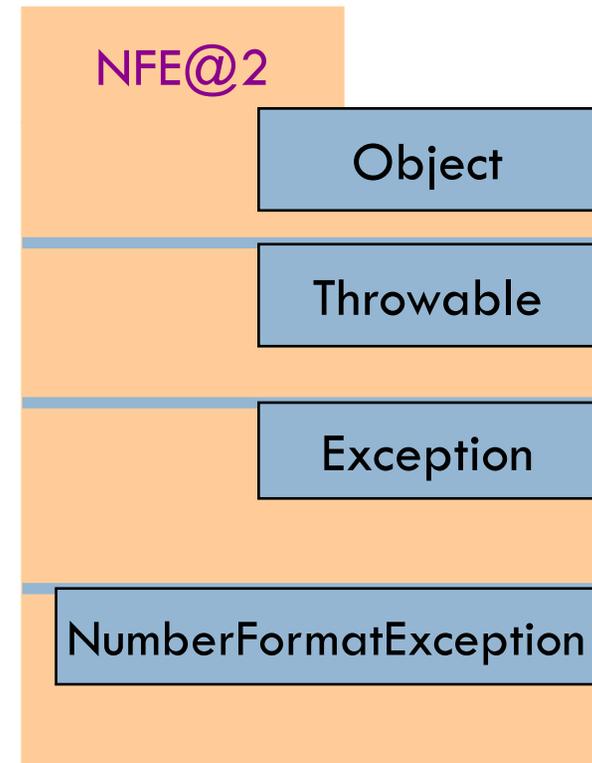
hide() show() JFrame
setTitle(String) getTitle()
getWidth() getHeight() ...
getX() getY() setLocation(int, int)

C

Classes can extend other classes

10

- You also saw this in the tutorial for this week's recitation
- There are subclasses of Exception for different types of exceptions



Accessing superclass things

11

Subclasses are different classes

- ▣ Public fields and methods can be accessed
- ▣ Private fields and methods cannot be accessed
- ▣ Protected fields can be access by subclasses

Keywords: **this**

12

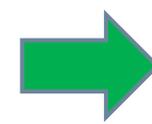
```
public class House {  
    private int nBed; // number of bedrooms, >= 0.  
    private int nBath; // number of bathrooms, in 1..5
```

```
/* * Constructor: */
```

```
public House(int nBed, int nBath) {  
    nBed= nBed; // has no effect!  
    nBath= nBath;
```

```
}  
}
```

Inside-out rule shows that field nBed is inaccessible! ☹️



```
this.nBed= nBed;  
this.nBath= nBath;
```

this avoids overshadowed field names

- **this** evaluates to the name of the object in which it occurs
- Makes it possible for an object to access its own name (or pointer)
- Example: Referencing a shadowed class field

A Subclass Example

13

```
public class House {  
    private int nBed; // num bedrooms, >= 0  
    private int nBath; // num bathrooms, in 1..5  
  
    /** Constructor: bed is number of bedrooms  
     * bath is number of bathrooms  
     * Prec: bed >= 0, 0 < bath <= 5 */  
    public House(int bed, int bath) {  
        nBed= bed; nBath= bath;  
    }  
  
    public int getNumBed() {  
        return nBed;  
    }  
    ...  
}
```

```
public class Apt extends House {  
    private int floor;  
    private Apt downstairsApt;  
  
    public Apt(int floor, Apt downstairs) {  
        this.floor= floor;  
        downstairsApt= downstairs;  
    }  
}
```

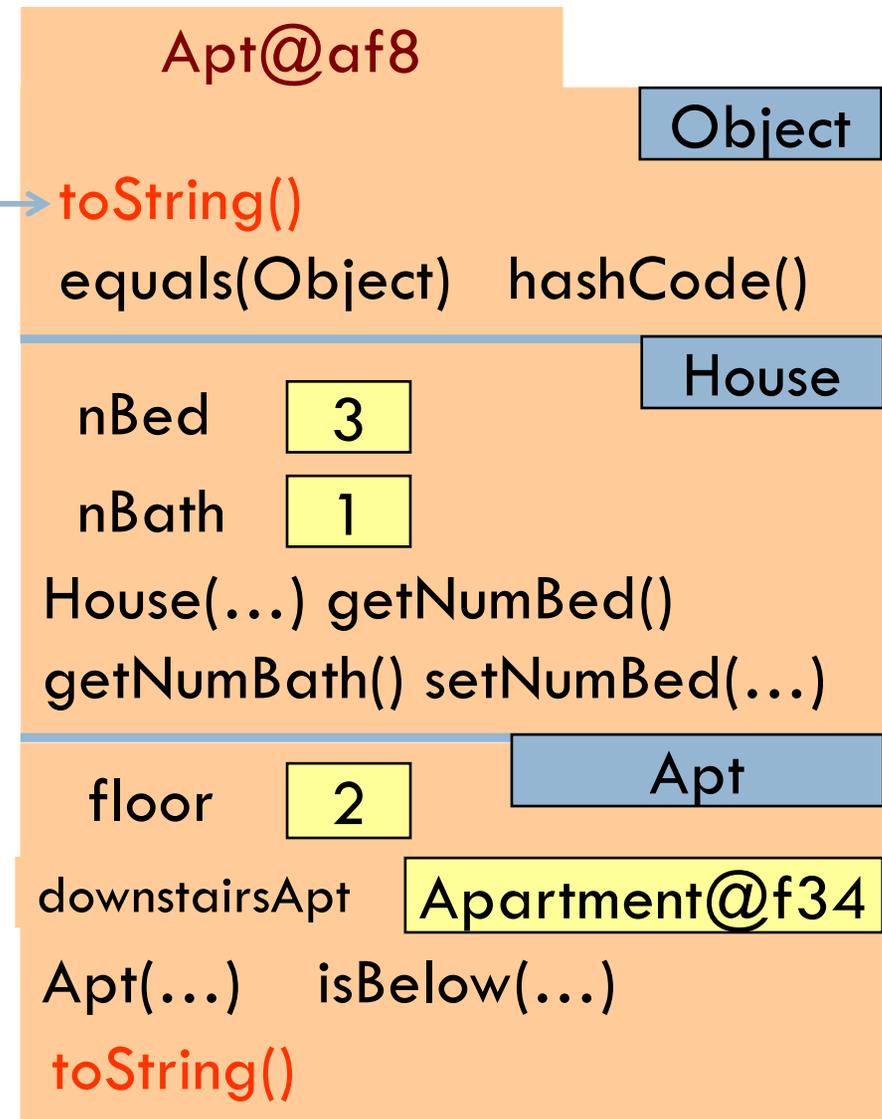
Overriding methods

14

Object defines a method `toString()` that returns the name of the object
`Apt@af8`

Java Convention: Define `toString()` in any class to return a representation of an object, giving info about the values in its fields.

New definitions of `toString()` **override** the definition in `Object.toString()`

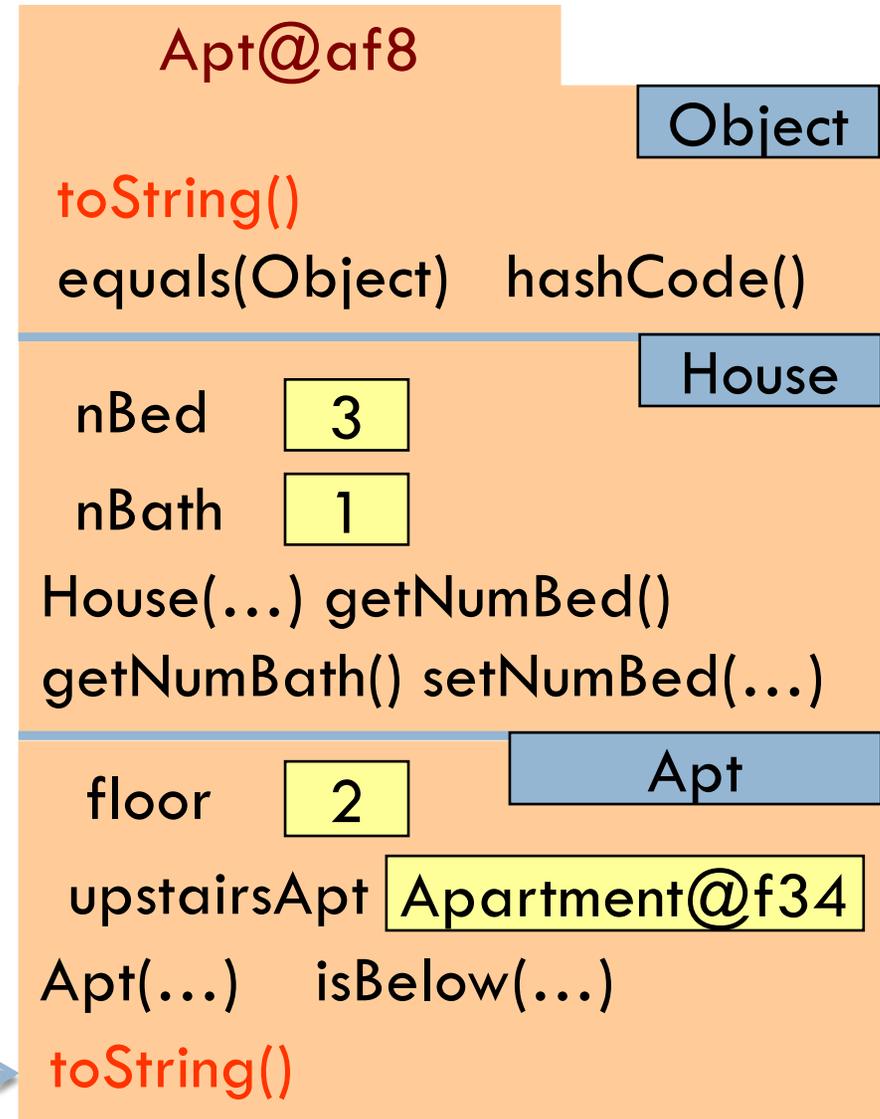


Overriding methods

15

```
public class Apt{  
    ...  
    /** Return a representation of an  
        Apartment*/  
    @Override  
    public String toString() {  
        return "" +  
            (getNumBed() + getNumBath()) +  
            " room apartment on " +  
            floor + "th floor";  
    }  
}
```

a.toString() calls this method



When should you make a subclass?

16

- The inheritance hierarchy should reflect **modeling semantics**, not implementation shortcuts
- **A** should extend **B** if and only if **A “is a” B**
 - An elephant is an animal, so **Elephant extends Animal**
 - A car is a vehicle, so **Car extends Vehicle**
 - An instance of any class is an object, so **AnyClass extends java.lang.Object**
- Don't use **extends** just to get access to protected fields!

When should you make a subclass?

17

Which of the following seem like reasonable designs?

- A. Triangle extends Shape { ... }
- B. PHDTester extends PHD { ... }
- C. BankAccount extends CheckingAccount { ... }

When should you make a subclass?

18

- Which of the following seem like reasonable designs?
 - A. Triangle extends Shape { ... }
 - Yes! A triangle is a kind of shape.
 - ~~B. PHDTester extends PHD { ... }~~
 - No! A PHDTester “tests a” PHD, but itself is not a PHD.
 - ~~C. BankAccount extends CheckingAccount { ... }~~
 - No! A checking account is a kind of bank account; we likely would prefer:
CheckingAccount extends BankAccount { ... }

Static Methods

19

- Most methods are **instance methods**: every instance of the class has a copy of the method
- There is only one copy of a **static method**.
There is not a copy in each object.

Make a method static if the body does not refer to any field or method in the object.

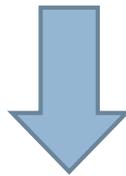
An Example

20

```
/** returns true if this object is below Apt a".
```

```
Pre: a is not null. */
```

```
public Boolean isBelow(Apt a){  
    return this == a.downstairsApt;  
}
```



```
/** returns true if Apt b is below Apt a
```

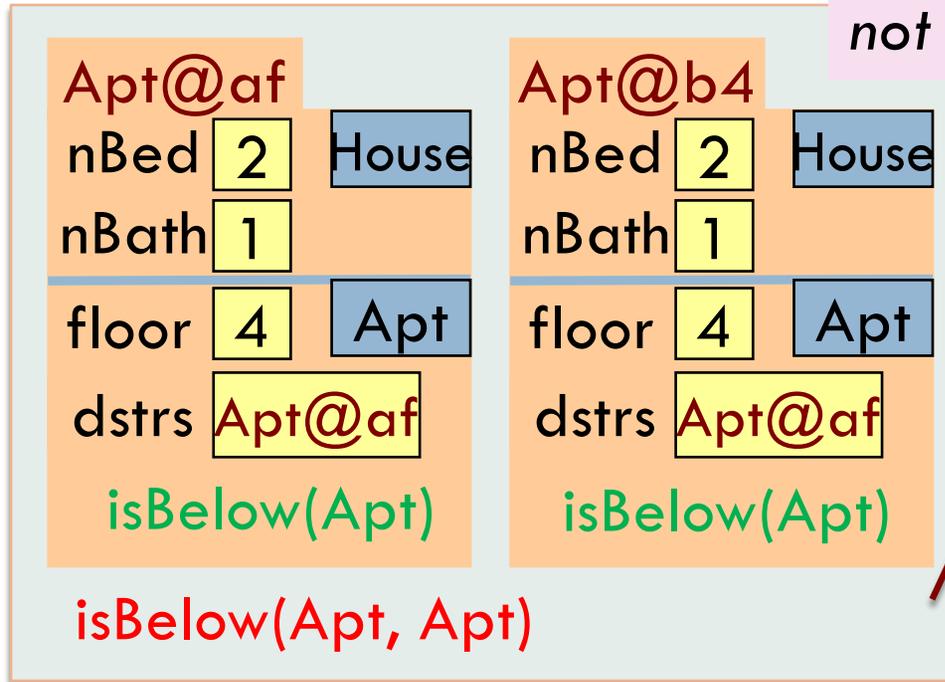
```
Pre: b and c are not null. */
```

```
public static boolean isBelow(Apt b, Apt a){  
    return b == a.downstairsApt;  
}
```

Referencing a static method

21

static: there is only **one** copy of the method. It is **not** in each object



Container for Apartment

contains: objects, **static** components

```
{  
  a= new Apt(...);  
  b= new Apt(...);  
  if (a.isBelow(b)) ...  
  if (Apt.isBelow(a, b)) ...  
}
```

Good example of static methods

22

java.lang.Math

<http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

Or find it by googling

Java 8 Math

Static Fields

23

- There is only one copy of a **static method**.
There is not a copy in each object.
- There is only one copy of a **static field**.
There is not a copy in each object.

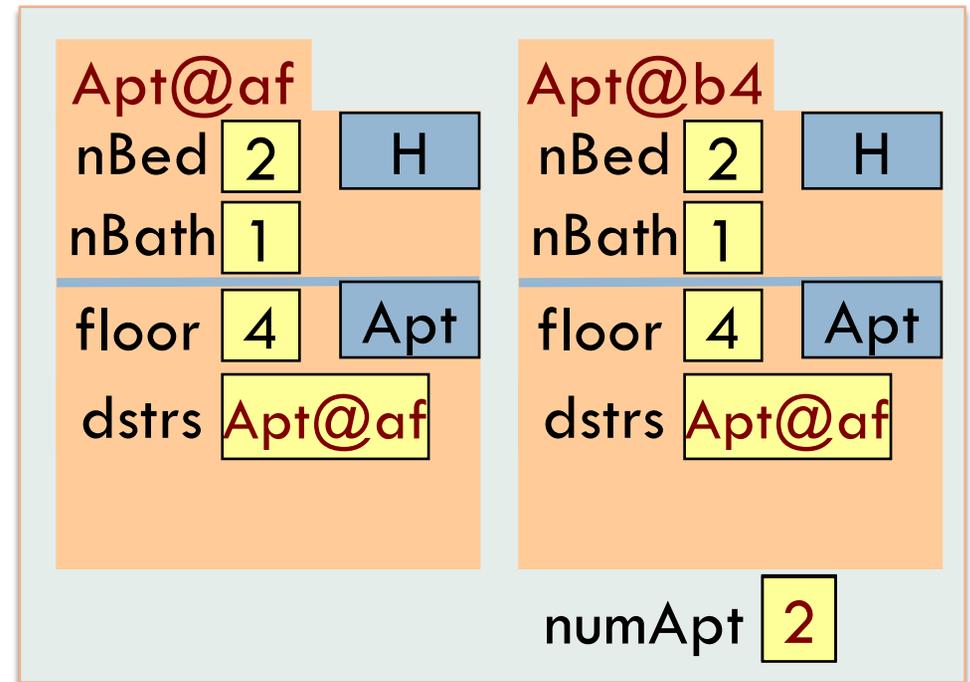
What are static fields good for?

Use of static variables: Maintain info about created objects

24

```
public class Apt extends House {  
    public static int numApt; // number of Apartments created  
  
    /** Constructor: */  
    public Apt(...) {  
        ...  
        numApt= numApt + 1;  
    }  
}
```

To have `numApt` contain the number of objects of class `Apartment` that have been created, simply increment it in constructors.



`numApt` stored in the Container for Apartment
To access: `Apartment.numApt`

Class `java.awt.Color` uses static variables

25

An instance of class `Color` describes a color in the RGB (Red-Green-Blue) color space. The class contains about 20 static variables, each of which is (i.e. contains a pointer to) a non-changeable `Color` object for a given color:

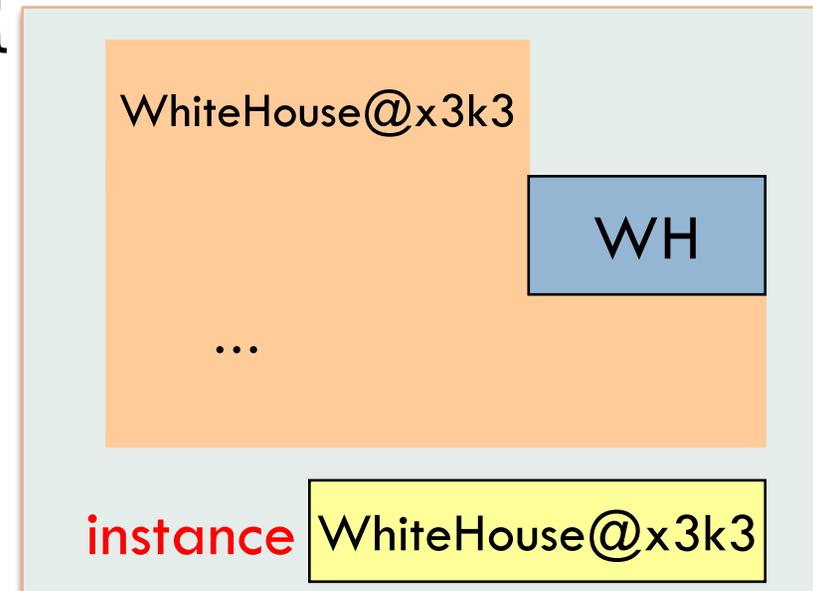
```
public static final Color black= ...;
public static final Color blue= ...;
public static final Color cyan= new Color(0, 255, 255);
public static final Color darkGray= ...;
public static final Color gray= ...;
public static final Color green= ...;
...
```

Uses of static variables: Implement the singleton pattern

26

Only one WhiteHouse can ever exist.

```
public class WhiteHouse extends House{  
    private static final WhiteHouse instance = new WhiteHouse();  
  
    private WhiteHouse() { } // ... constructor  
  
    public static WhiteHouse getInstance() {  
        return instance;  
    }  
  
    // ... methods  
}
```



Container for WhiteHouse