

CSE 143 Java

More About Inheritance

9/29/2004

(c) 2001-3, University of Washington

04-1

Topics for Today

- Protected members of classes
- Super in constructors and other methods
- Using "this" to run other constructors
- Overloading, constructors and "this"
- Overriding some common methods declared in Object – equals, compareTo, clone
- instanceof operator


9/29/2004

(c) 2001-3, University of Washington

04-2

Member Access in Subclasses

- *public*: accessible anywhere the class can be accessed
- *private*: accessible only inside the same class
 - Does *not* include subclasses – derived classes have no special permissions

- A new mode: *protected*  accessible inside the defining class and all its subclasses

- Use protected for "internal" things that subclasses also are intended to access
- Consider this carefully – often better to keep private data private and provide appropriate (protected) set/get methods

9/29/2004

(c) 2001-3, University of Washington

04-3

Using Protected

- If we had declared the Employee instance variables protected, instead of private, then this constructor would be legal

```
public HourlyEmployee(String name, int id, double pay) {  
    // initialize inherited fields  
    this.name = name;  
    this.id = id;  
    // initialize local fields  
    this.payRate = pay;  
    this.hoursWorked = 0.0;  
}
```

- But it's still poor code [why?]

9/29/2004

(c) 2001-3, University of Washington

04-4

Super

- If a subclass constructor wants to run a superclass constructor, it can do that using the syntax

`super(<-possibly empty list of argument expressions->)`

as the first thing in the subclass constructor's body

- Example:

```
public HourlyEmployee(String name, int id, double pay) {  
    super(name, id);  
    payRate = pay;  
    hoursWorked = 0.0;  
}
```

9/29/2004

(c) 2001-3, University of Washington

04-5

Constructor Rules

- Rule 1: If you do not write any constructor in a class, Java assumes there is a zero-argument, empty one
`ClassName() {}`
 - If you write any constructor, Java does not make this assumption
- Rule 2: If you do not write `super(...)` as the first line of a constructor, the compiler will assume the constructor starts with `super();`
- Rule 3: When an extended class object is constructed, there must be a constructor in the parent class whose parameter list matches the explicit or implicit call to `super(...)`
- Corollary: a constructor is always called at each level of the inheritance chain when an object is created

9/29/2004

(c) 2001-3, University of Washington

04-6

Super

- Another use for super: in any subclass, `super.msg(args)` can be used to call the version of the method in the superclass, even if it has been overridden
 - Can be done anywhere in the code – does not need to be at the beginning of the calling method, as for constructors
 - Often used to create “wrapper” methods
- ```
/** Return the pay of this manager. Managers receive a 20% bonus */
public double getPay() {
 double basePay = super.getPay();
 return basePay * 1.2;
}
```



9/29/2004

(c) 2001-3, University of Washington

04-7

## Overriding and Overloading (Review)

- In spite of the similar names, these are very different
  - **Overriding**: replacing an inherited method in a subclass
- ```
class One {
    public int method(String arg1, double arg2) { ... }
}
class Two extends One {
    public int method(String arg1, double arg2) { ... }
}
```
- Argument lists and results must match *exactly* (number and types)
 - Method called depends on actual (dynamic) type of the receiver

9/29/2004

(c) 2001-3, University of Washington

04-8

Overloading

- **Overloading**: a class may contain multiple definitions for constructors or methods with the same name, but different argument lists
- ```
class Many {
 public Many() { ... }
 public Many(int x) { ... }
 public Many(double x, String s) { ... }
 public void another(Many m, String s) { ... }
 public int another(String[] names) { ... }
}
```
- Parameter lists must differ in number and/or type of parameters  
Result types can differ, or not
  - Method calls are resolved automatically depending on number and (static) types of arguments – must be a unique best match



9/29/2004

(c) 2001-3, University of Washington

04-9

## Overloaded Constructors and this

- Classes often have several related Constructors
  - Common pattern: some provide explicit parameters while others assume default values
- “this” can be used at the beginning of a constructor to execute another constructor in the same class
  - Syntax similar to super
  - Can have other statements in the constructor following the “this” call
  - Good practice – can provide a single implementation of code common to both constructors

9/29/2004

(c) 2001-3, University of Washington

04-10

## Example: HourlyEmployee Constructors

```
/** Construct an hourly employee with name, id, and pay rate */
public HourlyEmployee(String name, int id, double pay) {
 super(name, id);
 payRate = pay;
 hoursWorked = 0.0;
}

// default pay for new hires
private static double defaultPay = 17.42;

/** Construct an hourly employee with name, id, and default pay rate */
public HourlyEmployee(String name, int id) {
 this(name, id, defaultPay);
}
```

9/29/2004

(c) 2001-3, University of Washington

04-11

## Comparing Objects

- **Object** defines a boolean function *equals* to test whether two objects are the same
- **Object**'s implementation just compares objects for identity, using `==`
  - This behavior is often not what you want
- Probably more appropriate concept of equality:
  - `obj1.equals(obj2)` should return true if *obj1* and *obj2* represent the “same value”
  - A class that wants this behavior must override `equals()`  
Somewhat tricky to do right – see Bloch, “Effective Java” (A-W, 2001) for a discussion

9/29/2004

(c) 2001-3, University of Washington

04-12

## instanceof

- The expression `<object> instanceof <classOrInterface>` is true if the object is an instance of the given class or interface (or any subclass of the one given)

- One common use: checking types of generic objects before casting

```
/* Compare this Blob to another Blob and return true if equal, otherwise false */
public boolean equals(Object otherObject) {
 if (otherObject instanceof Blob) {
 Blob bob = (Blob) otherObject;
 compare this to bob and return appropriate answer ...
 } else {
 return false;
 }
}
```

- Overuse (or even use?) of instanceof is often a sign of bad design that doesn't use inheritance and overriding appropriately

9/29/2004

(c) 2001-3, University of Washington

04-13

## Comparing The Order of Objects

- Many objects have a natural linear or total order
  - For any two values, one is always  $\leq$  the other
- A boolean comparison doesn't tell about relative order
- Type *Object* does not have a method for this kind of comparison (why not?)
- The most commonly used order comparison method has this signature:
  - `int compareTo(Object otherObject)`
  - return negative, 0, or positive value to indicate  $<$ ,  $=$ ,  $>$
- The Comparable interface specifies this method
  - Any class that provides `compareTo` should implement this interface
  - A "marker" interface – things like sort methods require Comparable objects

9/29/2004

(c) 2001-3, University of Washington

04-14

## Copying Object and clone()

- Review: what does  $a = b$ ; mean? (Hint: draw the picture)
- This behavior is not always desirable
- In Java, the `=` operator cannot be overridden
- Instead, a method to copy can be written
- `obj.clone()` should return a copy of `obj` with the "same" value
  - Object's implementation returns a new instance of the same class whose instance variables have the same values as `obj`
  - Object's implementation is protected
  - If a subclass needs to do something different, e.g. clone some of the instance variables too, then it should override `clone()`
- clone cannot be used at will...
  - Class must be marked as "Cloneable"

9/29/2004

(c) 2001-3, University of Washington

04-15

## Main Ideas of Inheritance

- Main idea: use *inheritance* to relate similar classes
  - Better modeling
  - Supports writing polymorphic code
  - Avoids code duplication
- Other ideas:
  - Use *protected* rather than *private* for things that will be needed by subclasses
  - Use *overriding* to make changes to superclass methods
  - Use *super* in constructors and methods to invoke superclass operations



9/29/2004

(c) 2001-3, University of Washington

04-16