CSE 143 Notes 5/22/06

Inheritance
Fine Points & Loose Ends

Topics for Today

- Inheritance recap – what does it mean
- Using super in methods
- (aside: protected)
- Constructors, defaults, and super
- equals and compareTo
- Abstract classes & Interfaces

A good reference for this (and much else about Java) is *Effective Java* by Joshua Bloch (A-W, 2001)

Review: Inheritance Facts

- A subclass *inherits* all instance variables and methods of the inherited class
  - All instance variables and methods of the superclass are *automatically* part of the subclass
  - Constructors are a special case (later)
- Subclass can *add* additional methods and instance variables
- Subclass can provide *different versions* of inherited methods

Example (review): Generic Employees

```
/** Representation of a generic employee. */
public class Employee {
 // instance variables
 private String name;      // employee name
 private int    id;        // employee id number
 /** Construct a new employee with the give name and id number... */
 public Employee(String name, int id) {
  this.name = name;
  this.id   = id;
 }
 /** Return the name of this employee */
 public String getName( ) { return name; }
 ...
 /** Return the pay earned by this employee */
 public double getPay( ) { return -17.42; }   // ???
  ...
}
```


Example (review): Specific Kinds of Employees

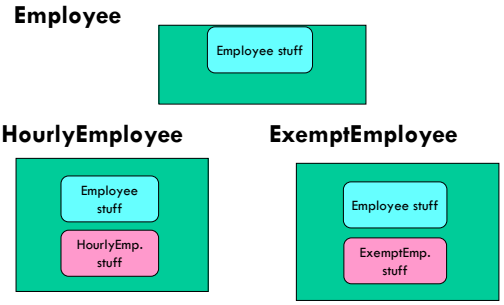
- Hourly Employee

```
public class HourlyEmployee
              extends Employee {
 // additional instance variables
 private double hours;  // hours worked
 private double hourlyPay;  // pay rate

 /** Return pay earned  */
 public double getPay( ) {
   return hours * hourlyPay;
 }
 ...
}
```

- Exempt Employee

```
public class ExemptEmployee
              extends Employee {
 // additional instance variable
 private double salary;  // weekly pay

 /** Return pay earned  */
 public double getPay( ) {
   return salary;
 }
 ...
}
```


In Pictures



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
- Issue: Normally need/want to use superclass constructors and methods to initialize and manipulate private data in superclass
- Solution: super

Super in Method Calls (Review)

- One use for super: in any subclass, super.m(args) can be used to call the version of the method in the superclass, even if it has been overridden

```
/** Return the pay of this manager.  Managers receive a 20% bonus */
public double getPay() {
   double basePay = super.getPay();
   return basePay * 1.2;
 }
```

- Typical usage pattern: "wrapper" methods – a method defined in subclass that does some computation before and/or after calling corresponding superclass method
  - A "custom" version of the method suitable for the subclass

Constructors

- Constructors in subclasses have no more access to private superclass data than any other methods
- Constructors are not inherited
- Yet we want to use constructors to guarantee proper initial state of all parts of an object
- Solution: superclass constructors always are executed one way or another when an object is created
  - How? ...

Explicit Super in Constructors

- A subclass constructor can explicitly indicate which superclass constructor should be used for the superclass part of the object. 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;
 }
```


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(...)

A Minimal Class

- If you write `class Empty { }`

  it is equivalent to

```
class Empty extends Object {
  public Empty() {
    super();
  }
}
```

- In other words:
  - All classes extend Object (either explicitly or implicitly – possibly through a chain of superclasses)
  - When an object is created, a constructor for *every* class in the inheritance chain *will* be called, either explicitly or implicitly
  - A class with no constructors or with constructors that don't explicitly use super(...) can fail to compile if its superclass does not contain a zero-argument constructor

Comparing Objects

- The built-in operators == and != answer the question "are these two things the *same* object?"
  - Sometimes appropriate; often not what we really want
- Class Object contains (roughly) the following method

```
public boolean equals(Object other) {
  return this == other;
}
```

  - All classes inherit this method if they do nothing further
- Often the meaning of equality depends on the data in the object
- Implementation: methods equals(...) and compareTo(...)
- How do we define these ourselves?

Defining Equals

- Suppose we want to define equals for Employee with the following meaning: two Employee objects are equal if they contain the same name and employee number
- First attempt:

```
public boolean equals(Employee other) {
  return this.name.equals(other.name) && this.id == other.id;
}
```

- Critique:
  - Does it capture our notion of "equals"?
  - Does it work?

Defining Equals – 2nd Try

- Problem: the method

```
public boolean equals(Employee other) { ... }
```

  overloads equals(Object), it doesn't *override* it
- OK, then what about

```
public boolean equals(Object other) {
  return this.name.equals(other.name) && this.id == other.id;
}
```

- Trouble:
  - What if "other" isn't an Employee object?
  - What if it is?

instanceof

- We can't just cast the parameter to Employee – that might fail, but we can check with

```
<object> instanceof <classOrInterface>
```

  which is true if the object is an instance of the given class or interface (or any subclass or subinterface of the one given)
- Overuse (or even use?) of instanceof is often a sign of bad design that doesn't use inheritance and overriding appropriately
  - But it is what we need to get equals right

Defining Equals – Last Try

- This time for sure....

```
public boolean equals(Object other) {
  if (other instanceof Employee) {
    Employee e = (Employee) other;
    return this.name.equals(e.name) && this.id == e.id;
  } else {
    return false;
  }
}
```


Comparisons

- Method compareTo is not defined in Object
  - There are classes for which compareTo makes no sense, so we don't want to inherit it everywhere
    (unlike equals – it always makes sense to ask if one object equals another)
- Instead, classes for which ordering makes sense should implement interface Comparable
  - This interface contains one method: compareTo
  - It actually is a generic interface (Comparable<T>), but we'll ignore that for now

Example: compareTo for Employee

- Let's say that two Employees are compared using their name and, if that is the same, use the id as a tie breaker

```
public int compareTo(Object other) {          // good enough for an example, but
    Employee eo = (Employee) other;           // could probably be more concise
    int comp = this.name.compareTo(eo.name);
    if (comp != 0) {
      return comp;
    } else if (this.id == eo.id) {            // or   return this.id – eo.id;
      return 0;
    } else if (this.id > eo.id) {
      return 1;
    } else {
      return -1;
    }
  }
```

- Unlike equals, compareTo should throw an exception if the other object is not an appropriate type.

Abstract Methods and Classes

- Recall that the Employee class contained a getPay() method
- Have to include it there so polymorphic code can use it (why?)

```
public double getPay(Employee e) {
    ...
}
```

- But no implementation really makes sense
  - Class Employee doesn't contain "pay" instance variables
  - So including an implementation of this in Employee is really bogus

```
/** Return the pay earned by this employee */
public double getPay( ) {
    return 0.0;    // ???
}
```


Abstract Methods and Classees

- An *abstract method* is one that is declared but not implemented in a class

```
/** Return the pay earned by this employee */
public abstract double getPay( ) ;
```

- A class that contains any abstract method(s) must itself be declared abstract

```
public abstract class Employee { … }
```

- Instances of abstract classes cannot be created
  - Usually because they are missing implementations of one or more methods

Using Abstract Classes

- An abstract class is intended to be extended
- Extending classes can override abstract methods they inherit to provide actual implementations

```
class HourlyEmployee extends Employee {
    ...
    /** Return the pay of this Hourly Employee */
    public double getPay( ) { return hoursWorked * payRate; }
}
```

  - Instances of these extended classes can be created
- A class that extends an abstract class without overriding all inherited abstract methods is itself abstract (and can be further extended)
- A class that is not abstract is often called a *concrete class*

Interfaces vs Abstract Classes

- Both of these specify a type
- Interface
  - Pure specification
  - No method implementation (code), no instance variables, no constructors
  - Classes can implement as many interfaces as they want
- Abstract class
  - Method specification plus, optionally:
    - Partial or full default method implementation
    - Instance variables
    - Constructors (called from subclasses using super)
- Which to use?

A Design Strategy

- These rules seem to provide a nice balance for designing software that can evolve over time, particularly in large systems:
  - Any major type should be defined in an interface
  - If it makes sense, provide a class that gives a default implementation (either partial or complete)
  - Client code can choose to either extend the default implementation, overriding methods that need to be changed, or implement the interface directly (the later is required if the client class explicitly extends a different class)
- This pattern occurs frequently in the standard Java libraries (see, e.g., List, AbstractList, ArrayList, et al)