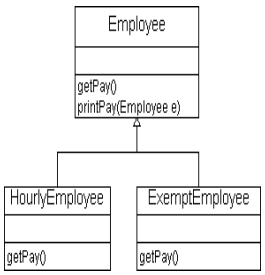
CSE 143 Java

More About Inheritance
Access Modifiers, super and this, methods of Object

Topics for Today

- Abstract classes
- 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

A Bit Unsatisfying...



1. In reality, there are no "Employee" employees in the company.
2. The Employee.getPay() is useless

Abstract Methods and Classes

- Solution: don't implement *getPay* in the base class
  - Just declare it without giving code
- Called an ***abstract method***
- Must be tagged with the keyword abstract

```
/** 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 { … }
```

- A class can be declared abstract even if it contains no abstract methods

Using Abstract Classes

- Key fact: *Instances of abstract classes cannot be created*
  - Because the class is missing implementations of one or more methods
- An abstract class is intended to be extended, rather than used directly
  - A class that is not abstract is often called a ***concrete class***
- Remind you of interfaces? It's a similar idea
  - In an interface, all methods are automatically abstract

Extending Abstract Classes

- 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)

Member (Field, I.V.) Access

- *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
- Note: these are issues of *scope*

Member Access in Subclasses

- *private*: Derived classes do NOT have access

- A new mode: ***protected***:
  - accessible inside the defining class and all its subclasses
- Use protected for "internal" things that subclasses also may need to access
  - Consider this carefully – often better to keep private data private and provide appropriate (protected) set/get methods

Using Protected

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

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

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

Super

- If a subclass constructor wants to call 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;
}
```

- Good practice to always have a super(...) at the start of a subclass's constructor

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 do this
- Rule 2: If you do not write super as the first line of a constructor, the compiler will assume the extended class 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

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;
}
```

  - Question: what if we had written "this.getPay()" instead

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

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

Overloading vs Overriding

- Overloading is *static*
  - The compiler decides which of several possible methods should be called
  - The decision is based on the arguments

- Overriding is *dynamic*
  - The JVM, at run time, decides which of several possible methods should be called
  - The decision is based on the object type

Overloaded Constructors and *this*

- Constructors in a class are often related
  - 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
  - Must be the first line in the constructor
  - Can have other statements in the constructor following the "this" call

Example: HourlyEmployee Constructors

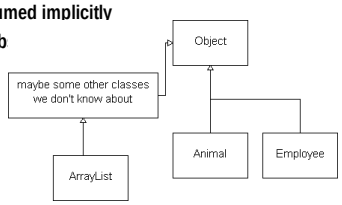
```
/** Construct an hourly employee with name, id, and pay rate */
public HourlyEmployee(String name, int id, double pay) {
   super(name, id);
   payRate = pay;
   hoursWorked = 0.0;
 }

 // 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);
 }
```

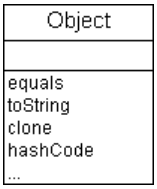

Class Object

- ***Object*** **is at the root of the Java class hierarchy**
- **Every class extends *Object*, directly or indirectly**
  - **If extends does not appear in a class declaration, "*extends Object*" is assumed implicitly**
  - ***Object*** **is not ab**



Methods of Class Object

- ***Object*** **defines a small number of methods appropriate for all objects**
- **These methods are inherited by all classes, but can be overridden – often necessary or at least a good idea**



toString Method

- **A method with this exact signature:**

  public String toString( );

```
/** Return a string representation of this employee */
public String toString( ) {
  return "Employee(name = " + name + ", id = " + id +
    ", pay = " + pay + ")";
}
```

- **Java treats toString in a special way**
  - **In many cases, will automatically call toString when a String value is needed:**

    System.out.println(**anObject**); //same as:

    System.out.println(**anObject.toString()**);

toString and You

- Recommended practice: Define a *toString* for all classes
  - Well, almost all
  - In CSE143, when in doubt -- do it!
- Good for debugging and testing
  - Anywhere you want to see an object's value, just write

  System.out.println( );

- *All* Objects in Java have a built-in, default toString method
- So why define your own??

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 undesirable
- More normal 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

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 OtherObject and return appropriate answer ...
  } else {
     return false;
  }
}
```

- Overuse of instanceof is often a sign of bad design that doesn't use inheritance and overriding well

Comparing The Order of Objects

- Many objects have a natural linear or total order
  - For any two values, one is always <= 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 kind of signature:

  int compareTo(Object otherObject)
  - return negative, 0, or positive value in a conventional way
- The Comparable interface requires exactly this method to exist
  - Any class that provides compareTo should implement this interface
  - A "marker" interface

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 just makes 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 "Clonable"

Main Ideas of Inheritance

- Main idea: use *inheritance* to reuse existing 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

Inheritance: Summary

- Lots of new concepts and terms
- Lots of new programming and modeling power
- Some of the Big Ideas
  - Composition ("has a") vs. specialization ("is a") and other relationships
  - Inheritance
  - Method overriding
  - Polymorphism, static vs. dynamic types
  - Method lookup, dynamic dispatch
  - Abstract methods, abstract classes
  - Class *Object* and its methods