CS 61B Data Structures and Programming Methodology

July 1 2008

David Sun

Today

- **Inheritance**
- **Polymorphism**
- **Dynamic Binding**

Inheritance

- **Fundamental to Object-oriented Programming**
  - Allows one to create new classes that are built on existing classes.

Example

- **Suppose you are asked to create a payroll system for a company.**
- **The company consists of Employees and Managers:**
  - Employees and Managers share many aspects in *common,* e.g., both receive a salary.
  - However, unlike employees, who must get finish their work to receive salary, managers receive a bonus if they complete their work.
- **How should we design our classes?**

Example

```
class Employee {
   private String name;
   private int salary;

   public String getName() { ...}
   public String getSalary() { ... }
   public void raiseSalary(double percent) { ...}
}

class Manager {
   private String name; //same as Employee
   private int salary; //same as Employee
   private int bonus; //new method

   public String getName() { ...} //same as Employee
   public String getSalary() { ... } //same as Employee
   public void raiseSalary(double percent) { ...}//same as Employee
   public void setBonus(double b) { ...} //new method

}
```

- We've already defined Employee.
- Managers a very similar to employees but with a little more *functionality* and *state*.
- Why repeat code? We can reuse the definition of Employee through **Inheritance**!

Example

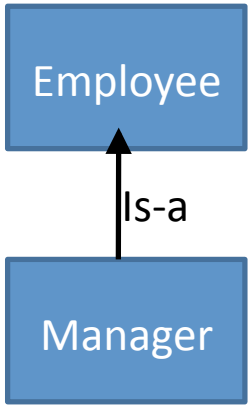
```
class Employee {
   private String name;
   private int salary;

   public String getName() { ...}
   public String getSalary() { ... }
   public void raiseSalary(double percent) { ...}
}

class Manager extends Employee{
        private double bonus;

        public void setBonus(double b) { ...}
}

Manager m = new Manager(. . .);
m.getName(); //OK, .getName() is inherited
m.setBonus();
```

Comments

- `Manager` **is *derived* from an existing class** `Employee`**.**
- `Employee` **is called the *superclass* or parent class,**
- `Manager` **is called the *subclass* or child class.**
- **Within the subclass, one can**
  - Add fields
  - Add methods
  - Override existing methods
- **Super doesn't mean there is more functionality. Subclasses have more *functionality* and more *data* than the superclass.**

Is-a Relationship

- **One good way to see if inheritance is appropriate is to ask:**
  - Is there an “is-a” relationship between the two classes under consideration.



What's been inherited?

- **All the fields of superclass.**
- **All the instance methods of superclass.**
- **When defining a new subclass, you define**
  - The *difference* between the subclass and the superclass
  - Factor out the most general methods into the superclass and put more specialized methods in the subclass.

Overriding Methods

```
class Employee {
   private String name;
   private int salary;

   public String getName() { ...}
   public double getSalary() { return salary; }
   public void raiseSalary(double percent) { ...}
}

class Manager extends Employees{
   private double bonus;
   public void setBonus(double b) { ...}
   public double getSalary() { ... }
}

Manager bigBoss = new Manager("TheBoss", "10000000");
bigBoss.setBonus(10000);
System.out.println(bigBoss.getSalary());
```

The `getSalary()` is redefined or ***overriden***

Should return the sum of the base salary and the bonus for a manager

```
class Employee {
   private String name;
   private int salary;

   public String getName() { ...}
   public String getSalary() { ... }
   public void raiseSalary(double percent) { ...}
}

class Manager extends Employees{
   private double bonus;
   public void setBonus(double b) { ...}
   public double getSalary() { //Compile time error
        return salary + bonus;
   }
}
```

Access Rights of the Subclass

- `Manager` class has no direct access to the `salary` field, even though every `Manager` object has a field called `salary`.
- This is consistent with the rule that "the private fields of a class are accessible only by the (non-static) methods of that class".
- Only the methods of the `Employee` class have access to the private field `salary`.
- If `Manager` class wants access to those private fields it must use the *public* interface of `Employee`.

Calling Public Methods of the Superclass

```
public double getSalary() {
      return getSalary() + bonus; //Infinite loop
}

public double getSalary() {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
}
```

- **You shouldn't think of `super` as the analog of `this`.**
  - `super` **is not a reference to an object.**
  - **You can assign `this` to a variable, but you can't do this with `super`.**
  - `super` **is a special keyword that directs the compiler to invoke the super class method.**

Rules for Overriding

- **You can only override `public` methods.**
- **When overriding a method, the subclass method must have the same visibility as the superclass method.**
  - If the superclass method is `public` then the subclass method must also be `public`. Otherwise the compiler will complain.
- **The return types only need to be compatible:**

```
public Employee getBuddy( ... );
public Manager getBuddy( ... );
```

Method Overloading

- ***Overloading*** **a method means that another method with the same name but different arguments is defined.**

```
private boolean setBonus(double b) {
   if (b < MIN || b > MAX) return false;
   this.bonus = b;
   return true;
}
public void setBonus(String sb) {
   double b = Double.parseDouble(sb);
   if (setBonus(b))
        System.out.println("Setting bonus succeeded !")
   else
         System.out.println("Setting bonus failed!")
}
```

- The argument list MUST be different
- The return types need not to be compatible
- The access level can be different

- **It has nothing to do with method *overriding or inheritance.***

Constructor

```
public Employee(String name, double salary) {
   this.name = name;
   this.salary = salary;
}

public Manager(String name, double salary, double
   bonus) {
       super(name, salary);
       this.bonus = bonus;
}
```

- `super(name, salary)` **means "Call the constructor of the Employee superclass with name, salary as parameters".**
- **Since** `Manager` **class has no direct access to the private fields** `name` **and** `salary`**, it can only initialize them through the superclass constructor.**
- `super` **must be the first statement in the constructor.**

Constructors

- **If you don't explicitly call a superclass constructor, Java automatically calls one for you: the empty constructor.**

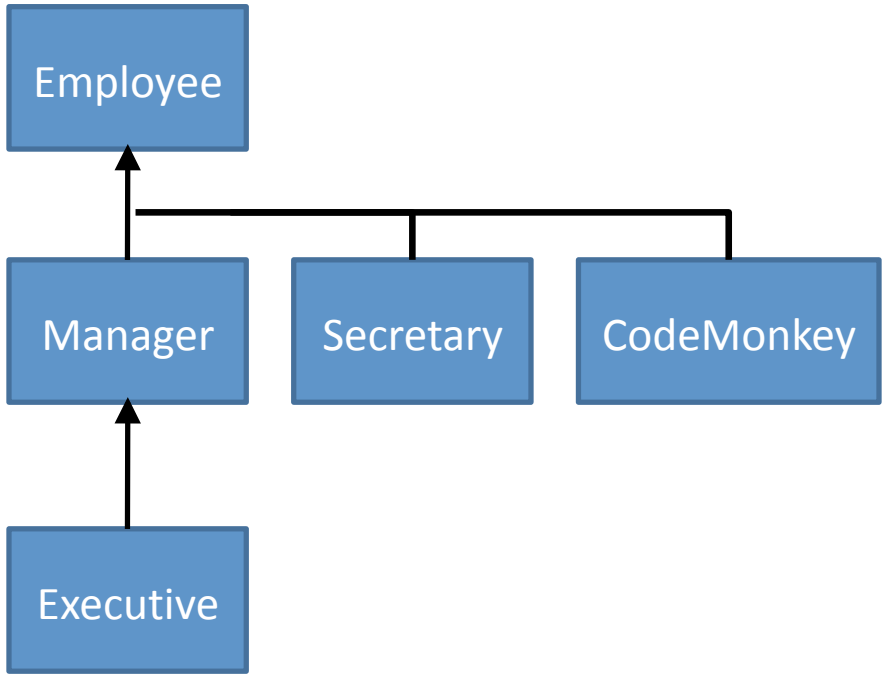
```
public Manager(double bonus) {
       //Employee() constructor called automatically.
       this.bonus = bonus;
}
```

- **If the superclass has no default (zero parameter) constructor, Java will produce a compile time error.**
- **There is no way to tell Java to not call a constructor; you only have control to choose which one is called.**
- **When used to invoke constructors, the `this` and `super` keywords are closely related.**

Class Hierarchy

- **Inheritance need not to stop at a single level: subclass can have subclasses.**
- **Subclassing is transitive.**

The collection of all classes extending from a common subclass is called an inheritance hierarchy



There can be more than one chain of descendents from a super class

Protect Access

- **Any class feature declared to be private is not accessible from other classes, including subclasses.**
- **Sometimes we want to restrict a method to subclasses only or, less commonly, to allow subclass methods to access a superclass field via** `protected`**.**
- `protected` **is a level of protection somewhere between "public" and "private".**
  - A "protected" field or method is visible to the declaring class and all its subclasses, but not to other classes.

Example

```
public class List {
  private int size;
  . . .
}

public class ArrayList extends List{
   . . .
  public void InsertFront(Object o) {
      . . .
      size++; //error
  }
}
```

Example

```
public class List {
  protected int size;
  . . .
}

public class ArrayList extends List{
   . . .
  public void InsertFront(Object o) {
      . . .
      size++; //OK
  }
}
```

Polymorphism

- ***The "is-a" Rule*: every object of the subclass is an object of the superclass, e.g., every Manager is-a Employee.**
- ***Substitution Principle:* you can use a subclass object whenever a program expects a superclass object.**

```
Employee e; //can hold a normal employee,a  manager,
            //or a codemonkey
e = new Employee(. . .); //Employee object
e = new Manager(. . .); //OK
```

- **In the Java, object variables are polymorphic.**

Polymorphism

- **Polymorphism works in "one-direction"**
  - A variable of a subclass cannot directly refer to an object of a superclass.

```
Employee e;
e = new Manager(. . .); //OK.
Manager b;
b = new Employee(. . .); //Compile time error!
```

Subtleties

```
Manager m = new Manager( ...);
Employee e = m; //OK
m.setBonus(10000);//OK
e.setBonus(20000);//COMPILE TIME ERROR!
```

- e is declared to be an `Employee`, and `Employee` class does not have the `setBonus()` method.

```
m = e; //ERROR!
```

- If this were to succeed, then someone could sneak in an `Employee` as a `Manager` and claim bonuses!

```
e = new Employee(. . .);
m = e;
m.setBonus( . . .); //Runtime error would occur
                    //because the employee
  object has                //no bonus field
```

More Subtleties

```
Manager[] managers = new Managers[10];
Employee[] employees = managers; //OK
employees[0] = new Employee("Hacker", …);
manager[0].setBonus(...);
```

Compiler allows this.

- **It seems that we've smuggled in a lowly employee into management level. But…**
- **Java ensures that no such corruption can occur.**
  - All arrays remember the **type** of the elements when it is created.
  - Arrays monitor that only compatible references are stored into them (e.g., Managers and Executives)
  - The **run time** throws an **exception** if you tried the above.

Dynamic Binding

```
Manager m = new Manager( ...);
Employee e = m; //OK
e.getSalary();
```

- **Which** `getSalary()` **method gets called? The one defined in the** `Employee` **class or the one in the** `Manager` **class?**
- **The** `getSalary()` **of** `Manager` **is called. Why?**

Dynamic Binding

- **Static type: the type of the variable.**
- **Dynamic type: the actual type of the object the variable references.**

```
Employee e = new Manager(); // Employee is the static type
                            // Manager is the dynamic type
```

- **Rule: When a method is called on an object, Java *binds* the method call to the object's dynamic type, regardless of the static type.**
- **This binding happens at runtime, hence the name *dynamic*.**

Casting

- **Polymorphism works in one direction only:**

```
Manager b = new Manager( . . . );
Employee e;
b = e; //Compile time error!
```

- **We can force a conversion from one *primitive* type to another via *casting.* e.g., double to int.**

```
double  x = 9.997;
int ix = (int)x // ix is 9.
```

- **We can also use the same syntax to force conversion from one *class* to another.**

Casting

- **Why?**
  - To use an object in its full capacity after its actual type has been temporarily forgotten.

```
Employee e = new Manager( … );//OK
e.setBonus(); //ERROR!
Manager m = e;
m.setBonus(); //ERROR!
Manager m = (Manager) e; //surround the target class name
                         //with parenthesis and put it in
                         // front of the reference
m.setBonus(); //OK!
```

Casting

- **No need to explicitly cast if you are assigning a subclass reference to a superclass variable.**
  - Remember superclass has less functionality compared to subclass. So casting up means you are promising less and the compiler will let you do it.

```
Employee e = new Manager( ... ); //casting up
```

- **Must explicitly cast if you are assigning a superclass reference to a subclass variable.**
  - Subclass has more functionality so casting down means you are promising more. You must do a cast explicitly so this promise can be checked at runtime.

```
Manager m = (Manager) e; //casting down
```

Casting Exception

- **What if you try to lie with casting?**

```
Employee e = new Employee( … );
Manager m = (Manager) e;
```

- **When the program runs, Java runtime notices the broken promise and issues a `ClassCastException`. If the exception is not *caught*, the program will terminate.**
- **We'll talk about exceptions in 2 lectures.**

`instanceof`

- **You can check whether a cast will succeed before attempting it:**

```
Employee e = new Manager( … );
Manager m;
if (e instanceof Manager)
{
  m = (Manager)e;
}
```

- **Always a good idea to use `instanceof` before casting from a superclass to a subclass**

Compile Time Exception

- **The compiler will complain if there is no chance for the cast to succeed.**

```
Manager m = new Manager( … );
CodeMonkey e = (CodeMonkey) m;  //compile time error.
```

- **The compiler ensures that you can only cast within an inheritance hierarchy.**

Comments

- **In general it's not a good idea to be doing casting.**
  - It only takes one uncaught `ClassCastException` to terminate your program.
  - The only reason for casting is if an object looses its actual type and you want to restore its full functionality.
- **If you ever find yourself doing this, ask yourself if this is a design flaw? Maybe it makes sense to add the method to the superclass?**

Next Lecture

- **Readings**
  - Chapter 7
- **Due:**
  - Tomorrow: Lab assignment “day2”