Principles of Computer Science II

Prof. Nadeem Abdul Hamid

CSC 121A – Spring 2005

Lecture Slides 3 -
Inheritance and Polymorphism

Constructors in Subclasses

- The first task of any subclass constructor is to call its direct superclass's constructor explicitly or implicitly
  - Ensures that fields inherited from the superclass are properly initialized
  - If superclass's constructor is not explicitly invoked, Java tries to invoke the default (no-argument) constructor
  - Subclass can explicitly invoke superclass constructor using keyword `super` followed by constructor arguments
- Even if constructor does not assign a value to a field, Java initializes it to a default value (0 for primitive numeric types, `false` for `boolean`, `null` for references)

Pet Example

```
public class PetTest {
    public static void main( String args[] ) {
        Cat c = new Cat( "Tiger", 3, true );
    }
}

$ java PetTest
Invoking Pet constructor: Tiger 3
Invoking Cat constructor: Tiger 3 true
```

```
public class Pet {
    String name;
    int age;

    public Pet( String name, int age ) {
        this.name = name;
        this.age = age;
        System.out.println( "Invoking Pet constructor: " + name + " " + age );
    }
}

public class Cat extends Pet {

    boolean likesMilk;

    /*
    public Cat( String name, int age, boolean milk ) {
    // This gives an error:
    // Pet.java: Pet(String,int) in Pet cannot be applied to ()
        this.name = name;
        this.age = age;
        this.likesMilk = milk;
    }
    */

    public Cat( String name, int age, boolean milk ) {
        super( name, age );  // must be first statement in constructor
        likesMilk = milk;
        System.out.println( "Invoking Cat constructor: "
                            + name + " " + age + " " + milk );
    }
}
```

The `Object` class

- All classes in Java inherit directly or indirectly from `Object` (in `java.lang` package)
- `Object` contains 11 methods which are inherited by all other classes
  - `clone`: (`protected` method) takes no arguments and returns an `Object` reference
    - Makes a copy of the object on which it is called
    - Default implementation makes a shallow copy; overridden versions usually make a deep copy
  - `equals`: compares two objects for equality and returns a `boolean`; takes single `Object` as argument

The `Object` class (cont.)

- `getClass`: Returns object of class `Class` containing information about the object's type, such as class name
- `toString`: returns `String` representation of an object
  - Default implementation returns package name and class name of the object, followed by hexadecimal representation of object's address in memory
- Other methods: `finalize`, `hashCode`, `notify`, `notifyAll`, `wait`

Polymorphism

- Allows us to write code that processes objects sharing the same superclass as if they are all objects of that superclass

```
public class PolymorphismTest {
    public static void main( String args[] ) {
        CommissionEmployee a
            = new CommissionEmployee( "Jim", 10000, 0.1 );
        CommissionEmployee b
            = new BasePlusCommissionEmployee3( "Jane", 7000, .07, 1200 );

        System.out.println( a.getName() + "'s earnings: " + a.earnings() );
        System.out.println( b.getName() + "'s earnings: " + b.earnings() );

        CommissionEmployee emps[] = new CommissionEmployee[2];
        emps[0] = a;
        emps[1] = b;
        for ( int i = 0; i < emps.length; i++ ) {
            System.out.println( emps[i].getName() + "'s earnings: " +
                                emps[i].earnings() );
        }
    } // end method main
} // end class PolymorphismTest
```

Abstract Classes and Methods

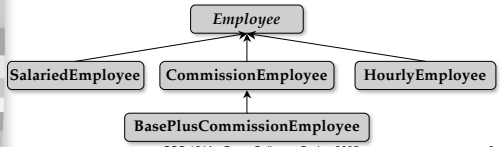
- So far, for any class we have defined, we can instantiate objects of that class
- Sometimes, it is useful to declare a class which will never be used to instantiate objects
  - Such classes called *abstract classes*
  - Used only as superclasses in inheritance hierarchies
  - Abstract classes usually contain at least one *abstract method* - a method without a body
  - For example, `Shape` class in the shapes hierarchy and the `draw` method
- Sample declarations:
  - Abstract class: `public abstract class Employee { …`
  - Abstract method: `public abstract double earnings();`

Abstract Classes and Methods

- An abstract class declares common attributes and behaviors of various classes in an inheritance hierarchy
- Typically contain one or more abstract methods that subclasses must override if the subclass is to be *concrete*
- Instance variables and concrete methods of the abstract class are inherited in the usual way
- Trying to instantiate an object from an abstract class is a compilation error
- Not implementing a superclass's abstract method(s) in a subclass is a compilation error unless the subclass is also declared abstract

Case Study: Payroll System

A company pays its employees on a weekly basis. The employees are of four types: (1) Salaried employees paid a fixed weekly salary regardless of hours, (2) hourly employees paid by the hour and receiving overtime for excess of 40 hours, (3) commission employees paid a percentage of their sales, and (4) salaried-commission employees receiving a base salary plus percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by addition 10% to their base salaries. The company wants to implement a Java application that performs its payroll calculations (polymorphically).



Payroll System Classes

	fields	*method* `earnings`	*method* `toString`
`Employee`	`fullName` `socialSecNumber`	`abstract`	*fullName* `social security number:` *SSN*
`Salaried-Employee`	+ `weeklySalary`	`weeklySalary`	`salaried employee:` *fullName* `social security number:` *SSN* `weekly salary:` *weeklySalary*
`Hourly-Employee`	+ `hourlyRate` `hoursWorked`	*If hours <= 40* `wage * hours` *If hours > 40* `40 * wage + (hours-40) * wage * 1.5`	`hourly employee:` *fullName* `social security number:` *SSN* `hourly wage:` *wage* `hours worked:` *hours*
`Commission-Employee`	+ `grossSales` `commissionRate`	`grossSales * commissionRate`	`commission employee:` *fullName* `social security number:` *SSN* `gross sales:` *grossSales* `commission rate:` *commissionRate*
`BasePlus-Commission-Employee`	+ `grossSales` `commissionRate` `baseSalary`	`grossSales * commissionRate + baseSalary`	`base plus commission employee:` *fullName* `social security number:` *SSN* `gross sales:` *grossSales* `commission rate:` *commissionRate* `base salary:` *baseSalary*

Files

lec03/Employee.java
lec03/SalariedEmployee.java
lec03/HourlyEmployee.java
lec03/CommissionEmployee.java
lec03/BasePlusCommissionEmployee.java

Polymorphism and Binding

- Binding: connecting a method call to a method body
- Dynamic binding/late binding:
  - Binding occurs at run-time based on the actual class of an object
- Behavior inside constructors? …
  - lec03/PolyConstructor.java

Initialization Order of Objects

- Storage allocated for the object initialized to binary zero before anything else
- Base-class constructors called as described previously
  - At this point, the overridden draw( ) method is called (before the Circle constructor is called), which discovers a radius value of zero
- Member initializers are called in the order of declaration
- Body of the derived-class constructor is called

`final` Methods and Classes

- Variables declared `final` cannot be modified after initial declaration
  - i.e. they represent constant values
- Methods declared `final` cannot be overridden in subclasses
  - `private` and `static` methods are implicitly `final`
  - Methods calls resolved at compile time (static/early binding)
- Classes declared `final` cannot be derived from
  - `String` class is `final`
  - Prevents programmers from creating subclasses that might bypass security restrictions

Extending the Payroll Application

- Company wishes to extend payroll system
  - Handle several accounting operations in one accounts payable application
  - Calculate payment due on invoices as well as employee earnings
- Java offers capability to specify a requirement that several unrelated classes (Invoice, Employee) implement a set of common methods (calculating a payment amount)…

Interfaces

- Only specify *what* operations are available- not *how* they are performed
- Use the keyword `interface` (instead of class)
  - All `interface` methods are implicitly `public abstract`
  - All `interface` fields are implicitly `public`, `static`, `final`
- To use an `interface`, a class
  - Specifies that it `implement`s the interface
  - Declares and implements methods exactly as specified in the interface declaration
  - If class does not implement all the methods of the interface, the class must be declared abstract
- Interface used when unrelated classes need to share common methods and constants
  - Used in place of an abstract class when no default implementations are needed

Interface Case Study

- lec03/Payable.java
- lec03/Invoice.java
- lec03/Employee2.java
- lec03/SalariedEmployee2.java
- lec03/PayableTest.java

Interfaces and Multiple Inheritance

- A class can only extend a single superclass (single inheritance)
- However, a class can implement multiple interfaces
  - Simply provide a comma-separated list of interface names after keyword `implements` in the class declaration