Java Programming

Inheritance in Java

Review of Lecture 4

- **Controlling access to members:**
  - **Instance variables – private, not accessible outside class.**
  - **Instance methods – public**
  - **Use Exception Handling to provide validation**
    - **try/catch/finally blocks**
    - **throw statement**
- **Debugging**
  - **Breakpoints, Eclipse debug window**
- **this** keyword – an object references itself
- Use 'this' in constructors to refer to the object's instance variables and other methods.
- **Overloaded** constructors - enable objects of a class to be initialized in different ways.
- **set** methods – **mutator**
- **get** methods - **accessor**
- **predicate** methods – return true or false

Review of Lecture 4

- **Composition –** a class declares variables that are objects of other classes
  - **Employee** class declares **Date** objects
- **Enum types** – contain
  - A comma separated list of enum constants (static & final)
  - Optionally contains constructors, fields, methods
- **Garbage collection** – performed by JVM in a separate thread, to claim memory from unused objects.
- Use try-with-resources to close objects automatically.
- ***static*** *fields – describe class-wide information, have class scope.*
- *Use* ***ClassName.staticFieldName*** *to access a public static field.*
- *static import – to import the static members of a class or interface,* so you can **access them via their unqualified names in your class**

Lesson 5 Objectives

- Understand the concept of **inheritance** and how to use it to **create new classes based on existing classes**.
- Understand the concept of **superclasses** and subclasses.
- Use protected keyword in superclass to give subclass methods access to superclass members.
- Understand how **constructors** are used in inheritance hierarchies.
- Develop a Java application that implements an **inheritance hierarchy**.

Introduction to inheritance

- A **new class is created by acquiring an existing class's members** and possibly adding new or modified capabilities.
- Can **save time during program development** by basing new classes on existing proven and debugged high-quality software.
- Increases the likelihood that a system will be implemented and maintained **effectively**.

Introduction to inheritance (Cont.)

- When creating a class, rather than declaring completely new members, you can **designate that the new class should inherit the members of an existing class**.
- Existing class is the **superclass**
- New class is the **subclass**
- A **subclass can be a superclass** of future subclasses.
- A **subclass can add its own fields and methods**.
- A **subclass is more specific** than its superclass and represents a more specialized group of objects.
- The subclass **exhibits the behaviors of its superclass** and can add behaviors that are specific to the subclass.
- This is why inheritance is sometimes referred to as **specialization**.

Introduction to inheritance (Cont.)

- The **direct superclass** is the superclass from which the subclass explicitly inherits.
- An **indirect superclass** is any class above the direct superclass in the class hierarchy.
- The Java class hierarchy begins with **class Object** (in package java.lang)
- Every class in Java **directly or indirectly extends** (or “inherits from”) Object.
- **Java supports only single inheritance**, in which **each class is derived from exactly one direct superclass**.

Introduction to inheritance (Cont.)

- We distinguish between the **is-a relationship** and the **has-a relationship**
- **Is-a** represents inheritance
- In an **is-a** relationship, an object of a subclass can also be treated as an object of its superclass
- **Has-a** represents composition
- In a **has-a** relationship, an object contains as members references to other objects

Superclasses and Subclasses

- Figure 9.1 lists several simple examples of superclasses and subclasses
- Superclasses tend to be “more general” and subclasses “more specific.”
- Because every subclass object is an object of its superclass, and one superclass can have many subclasses, the **set of objects represented by a superclass is typically larger than the set of objects represented by any of its subclasses**.

Superclasses and Subclasses (Cont.)

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Fig. 9.1 | Inheritance examples.

Superclasses and Subclasses (Cont.)

- A superclass exists in a **hierarchical relationship** with its subclasses.
- Fig. 9.2 shows a sample university community class hierarchy
- Also called an **inheritance hierarchy**.
- Each arrow in the hierarchy represents an **is-a** relationship.
- Follow the arrows upward in the class hierarchy
  - an Employee **is a** CommunityMember”
  - “a Teacher **is a** Faculty member.”
  - CommunityMember is the direct superclass of Employee, Student and Alumnus and is an indirect superclass of all the other classes in the diagram.
- Starting from the bottom, you can follow the arrows and apply the **is-a** relationship up to the topmost superclass.

Superclasses and Subclasses (Cont.)

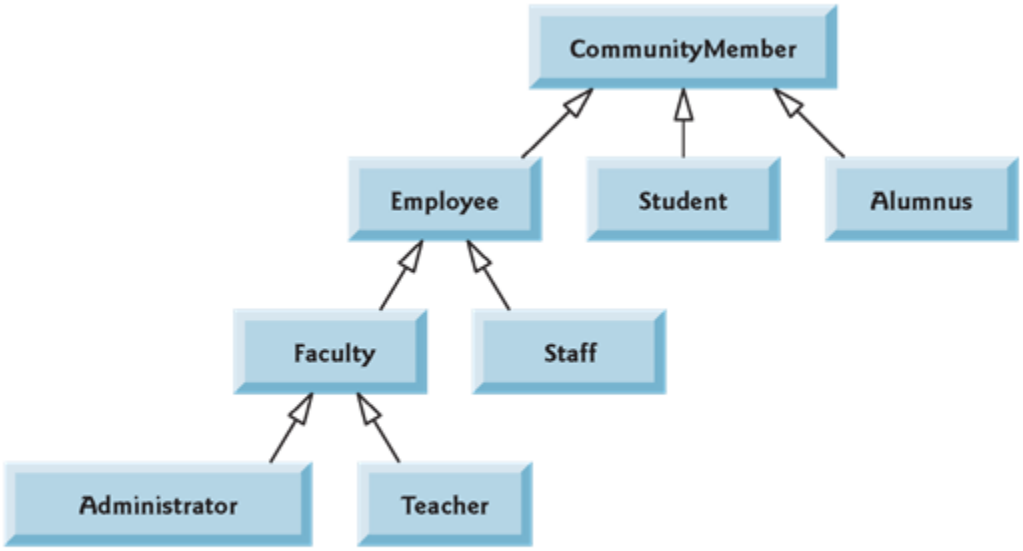

Fig. 9.2 | Inheritance hierarchy UML class diagram for university `CommunityMembers`.

Superclasses and Subclasses (Cont.)

- Fig. 9.3 shows a Shape inheritance hierarchy.
- Van follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several is-a relationships.
- A Triangle is a TwoDimensionalShape and is a Shape
- A Sphere is a ThreeDimensionalShape and is a Shape.

Superclasses and Subclasses (Cont.)

- Shape Hierarchy

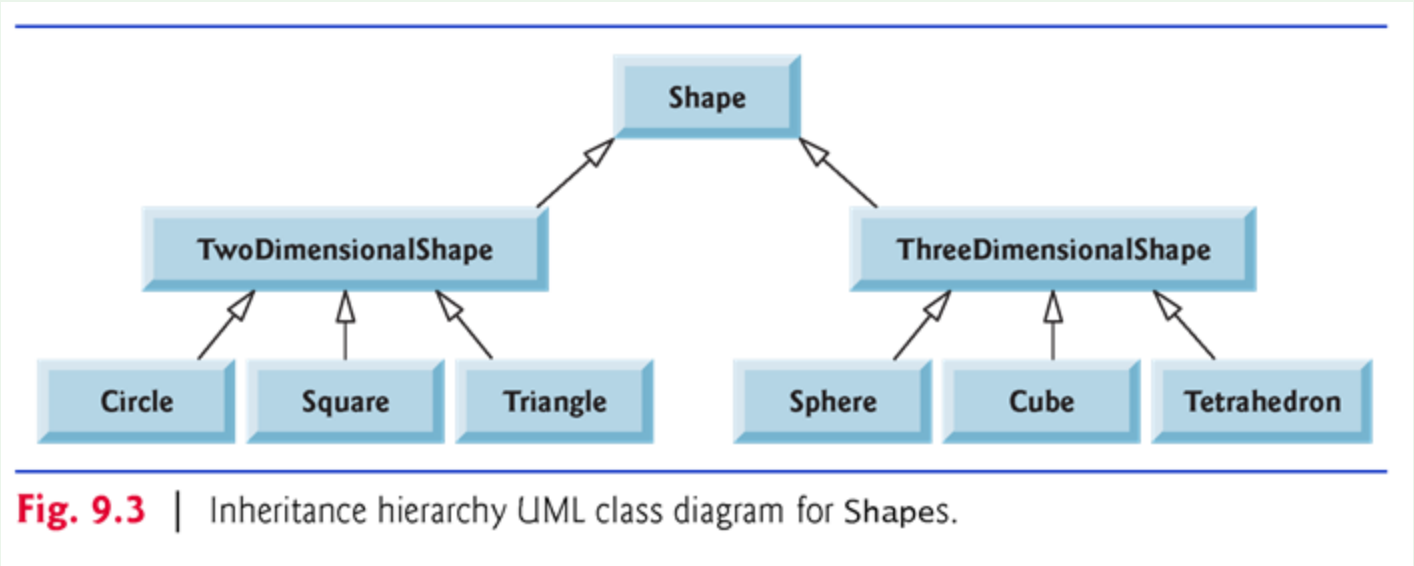

Fig. 9.3 | Inheritance hierarchy UML class diagram for Shapes.

Superclasses and Subclasses (Cont.)

- Not every class relationship is an inheritance relationship.
- Has-a relationship
- Create classes by **composition of existing classes**.
- Example: Given the classes Employee, BirthDate and TelephoneNumber, it's improper to say that an Employee is a BirthDate or that an Employee is a TelephoneNumber.
- However, an Employee has a BirthDate, and an Employee has a TelephoneNumber.

Superclasses and Subclasses (Cont.)

- Objects of all classes that extend a common superclass can be treated as objects of that superclass.
- Commonality expressed in the members of the superclass.
- Inheritance issue
  - A subclass can **inherit methods that it does not need** or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often **needs a customized version of the method**.
- The subclass can **override (redefine) the superclass method** with an appropriate implementation.

protected Members

- A class's public members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A class's private members are accessible only within the class itself.
- protected access is an intermediate level of access between public and private.
- A superclass's protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package
- protected members also have **package access**.
- All public and protected superclass members retain their original access modifier when they become members of the subclass.

protected Members (Cont.)

- A superclass's private members are hidden from its subclasses
- They can be accessed only through the public or protected methods inherited from the superclass
- Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names.
- When a subclass method **overrides** an inherited superclass method, the **superclass** version of the method can be accessed from the **subclass** by preceding the superclass method name with keyword super and a dot (.) separator.

Relationship Between Superclasses and Subclasses

- Inheritance hierarchy containing types of employees in a company's payroll application
- **Commission employees** are paid a percentage of their sales
- **Base-salaried commission employees** receive a base salary plus a percentage of their sales.

Creating and Using a CommissionEmployee Class

- Class *CommissionEmployee* (Fig. 9.4) extends class Object (from package java.lang).
- *CommissionEmployee* inherits *Object*'s methods.
- If you don't explicitly specify which class a new class extends, the class extends *Object* implicitly.

Creating and Using a CommissionEmployee Class (Cont.)

- **Inheritance Rules:**
- Constructors are **not inherited**.
- The **first task of a subclass constructor is to call its direct superclass's constructor explicitly or implicitly**
- Ensures that the instance variables inherited from the superclass are **initialized properly**.
- If the code does not include an **explicit call to the superclass constructor**, Java implicitly calls the superclass's default or no-argument constructor.
- A **class's default constructor** calls the superclass's default or no-argument constructor.

Creating and Using a CommissionEmployee Class (Cont.)

- **toString** is one of the methods that every class inherits directly or indirectly from class **Object**.
- Returns a String representing an object.
- Called implicitly whenever an object must be converted to a String representation.
- Class **Object**'s toString method returns a **String** that includes the name of the object's class.
- This is primarily a placeholder that can be **overridden** by a subclass to specify an appropriate **String** representation.

Creating and Using a CommissionEmployee Class (Cont.)

- To override a superclass method, a subclass must declare a method with the same signature as the superclass method
- @Override annotation
- Indicates that a method should override a superclass method with the same signature.
- If it does not, a compilation error occurs.

Creating and Using a BasePlus-CommissionEmployee Class

- Class *BasePlusCommissionEmployee* (Fig. 9.6) contains a *first name*, *last name*, *social security number*, *gross sales amount*, *commission rate* and *base salary*.
- All but the base salary are in common with class *CommissionEmployee*.
- Class *BasePlusCommissionEmployee*'s public services include a constructor, and methods *earnings*, *toString* and *get* and *set* for each instance variable
- Most of these are in common with class *CommissionEmployee*.

Creating and Using a BasePlus-CommissionEmployee Class (Cont.)

- Class *BasePlusCommissionEmployee* does not specify “extends Object”
- Implicitly extends *Object*.
- *BasePlusCommissionEmployee*’s constructor invokes class *Object*’s default constructor implicitly.

Creating and Using a BasePlus-CommissionEmployee Class (Cont.)

- Much of *BasePlusCommissionEmployee*'s code is similar, or identical, to that of *CommissionEmployee*.
- **private instance variables** *firstName* and *lastName* and methods *setFirstName*, *getFirstName*, *setLastName* and *getLastName* are identical.
- Both classes also contain corresponding **get** and **set** methods.
- The **constructors** are almost identical
- *BasePlusCommissionEmployee*'s constructor also sets the *baseSalary*.
- The *toString* methods are almost identical
- *BasePlusCommissionEmployee*'s *toString* also outputs instance variable *baseSalary*

Creating and Using a BasePlus-CommissionEmployee Class (Cont.)

- We literally *copied CommissionEmployee*'s code, pasted it into *BasePlusCommissionEmployee*, then modified the new class to include a base salary and methods that manipulate the base salary.
- This "*copy-and-paste*" approach is often error prone and time consuming.
- It spreads copies of the same code throughout a system, creating a code-maintenance problems - changes to the code would need to be made in multiple classes.

Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy

- Class *BasePlusCommissionEmployee* class extends class *CommissionEmployee*
- A *BasePlusCommissionEmployee* object is a *CommissionEmployee*
- Inheritance passes on class *CommissionEmployee*'s capabilities.
- Class *BasePlusCommissionEmployee* also has instance variable *baseSalary*.
- Subclass *BasePlusCommissionEmployee* inherits CommissionEmployee's instance variables and methods
- Only *CommissionEmployee*'s public and protected members are directly accessible in the subclass.

Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)

- Each subclass **constructor must implicitly or explicitly call one of its superclass's constructors** to initialize the instance variables inherited from the superclass.
- **Superclass constructor call syntax** - keyword super, followed by a set of parentheses containing the superclass constructor arguments - must be the first statement in the constructor's body.
- If the subclass constructor did not invoke the superclass's constructor explicitly, the compiler would attempt to insert a call to the superclass's default or no-argument constructor.
- Class *CommissionEmployee* does not have such a constructor, so the compiler would issue an error.
- You can explicitly use *super()* to call the superclass's no-argument or default constructor, but this is rarely done.

Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)

- Compilation errors occur when the subclass attempts to access the superclass's private instance variables.
- These lines could have used appropriate ***get*** methods to retrieve the values of the superclass's instance variables.

BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables

- To enable a subclass to directly access superclass instance variables, we can declare those members as *protected* in the superclass.
- New CommissionEmployee class modified only lines 6–10 of Fig. 9.4 as follows:

```
protected final String firstName;
protected final String lastName;
protected final String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

- With *protected* instance variables, the subclass gets access to the instance variables, but classes that are not subclasses and classes that are not in the same package cannot access these variables directly.

CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Class *BasePlusCommissionEmployee* (Fig. 9.9) extends the new version of class *CommissionEmployee* with protected instance variables.
- These variables are now protected members of *BasePlusCommissionEmployee*.
- If another class extends this version of class *BasePlusCommissionEmployee*, the new subclass also can access the protected members.
- The source code in Fig. 9.9 (59 lines) is considerably shorter than that in Fig. 9.6 (127 lines)
- Most of the functionality is now inherited from *CommissionEmployee*
- There is now only one copy of the functionality.
- Code is easier to maintain, modify and debug - the code related to a *CommissionEmployee* exists only in that class.

CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Inheriting *protected* instance variables **enables direct access to the variables by subclasses**.
- In most cases, it's better to use *private* instance variables to encourage proper software engineering.
- Code will be easier to maintain, modify and debug.

CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Using protected instance variables **creates several potential problems**.
  - The subclass object can set an inherited variable's value directly without using a set method.
  - A subclass object can **assign an invalid value** to the variable
  - Subclass methods are more likely to be written so that they depend on the superclass's data implementation.
  - Subclasses should depend only on the superclass services and not on the superclass data implementation.

CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- With protected instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes.
- Such a class is said to be **fragile** or **brittle**, because a small change in the superclass can "break" subclass implementation.
- You should be able to change the superclass implementation while still providing the same services to the subclasses.
- If the superclass services change, we must reimplement our subclasses.
- A class's protected members are visible to all classes in the same package as the class containing the protected members - this is not always desirable.

CommissionEmployee Inheritance Hierarchy Using private Instance Variables

- Class *CommissionEmployee* declares instance variables *firstName*, *lastName*, *socialSecurityNumber*, *grossSales* and *commissionRate* as private and provides **public methods for manipulating these values**.

CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- *CommissionEmployee* methods earnings and *toString* use the class's get methods to obtain the values of its instance variables.
- If we decide to change the internal representation of the data (e.g., variable names) only the bodies of the get and set methods that directly manipulate the instance variables will need to change.
- These changes occur solely within the superclass-—no changes to the subclass are needed.
- Localizing the effects of changes like this is a good software engineering practice.
- Subclass *BasePlusCommissionEmployee* inherits *Commission-Employee*'s non-private methods and can access the private superclass members via those methods.

CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- Class *BasePlusCommissionEmployee* (Fig. 9.11) has several changes that distinguish it from Fig. 9.9.
- Methods earnings and *toString* each invoke their superclass versions and do not access instance variables directly.

CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- Method *earnings* overrides class the superclass's earnings method.
- The new version calls *CommissionEmployee*'s earnings method with *super.earnings()*.
- Obtains the *earnings* based on commission alone
- Placing the keyword super and a dot (.) separator before the superclass method name invokes the superclass version of an overridden method.
- Good software engineering practice
- If a method performs all or some of the actions needed by another method, **call that method rather than duplicate its code**.

CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- *BasePlusCommissionEmployee*'s *toString* method overrides class *CommissionEmployee*'s toString method.
- The new version creates part of the String representation by calling *CommissionEmployee*'s *toString* method with the expression *super.toString()*.

Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
- The subclass constructor, before performing its own tasks, explicitly **uses super to call one of the constructors in its direct superclass** or **implicitly calls the superclass's default or no-argument constructor**
- If the superclass is derived from another class, the superclass constructor **invokes the constructor of the next class up the hierarchy**, and so on.
- The last constructor called in the chain is always **Object**'s constructor.
- Original subclass constructor's body **finishes executing last**.
- Each superclass's constructor **manipulates the superclass instance variables that the subclass object in**herits.

Class Object

- All classes in Java **inherit directly or indirectly from class Object**, so its 11 methods are inherited by all other classes.
- Figure 9.12 summarizes Object's methods.
- Every array has an overridden *clone* method that copies the array.
- If the array stores references to objects, the objects are not copied - a **shallow copy** is performed.
  - A shallow copy can be made by **simply copying the reference**
  - A deep copy means actually creating a new array and copying over the values.

http://www.cs.utexas.edu/~scottm/cs307/handouts/deepCopying.htm

(Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels

- Labels are a convenient way of identifying GUI components on the screen and keeping the user informed about the current state of the program.
- A JLabel (from package javax.swing) can display text, an image or both.
- The example in Fig. 9.13 demonstrates several JLabel features, including a plain text label, an image label and a label with both text and an image.

References

- Textbook
- Java documentation
  - https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
  - https://docs.oracle.com/javase/tutorial/uiswing/components/index.html
- https://en.wikipedia.org/wiki/Object_copying