Inheritance

- A form of software reuse in which a new class is created by absorbing an existing class’s members and embellishing them with 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.

Inheritance

- 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

- 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.

- A subclass is more specific than its superclass and represents a more specialized group of objects.
Inheritance

- 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.

Advantages of inheritance

- When a class inherits from another class, there are three benefits:
  1. You can **reuse** the methods and data of the existing class
  2. You can **extend** the existing class by adding new data and new methods
  3. You can **modify** the existing class by overloading its methods with your own implementations

Relationships between classes

- 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

<table>
<thead>
<tr>
<th>Superclass</th>
<th>Subclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>GraduateStudent, UndergraduateStudent</td>
</tr>
<tr>
<td>Shape</td>
<td>Circle, Triangle, Rectangle, Sphere, Cube</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan, HomeImprovementLoan, MortgageLoan</td>
</tr>
<tr>
<td>Employee</td>
<td>Faculty, Staff</td>
</tr>
<tr>
<td>BankAccount</td>
<td>CheckingAccount, SavingsAccount</td>
</tr>
</tbody>
</table>

**Fig. 9.1** | Inheritance examples.

- Superclasses tend to be “more general” and subclasses “more specific.”
• 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.”

---

**Fig. 9.2** | Inheritance hierarchy for university CommunityMembers.

---

**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.)**

• Below is Shape inheritance hierarchy.
  • Follow the arrows from the bottom of the diagram to the topmost superclass to identify several **is-a** relationships.
  ▪ A Triangle **is a** TwoDimensionalShape and **is a** Shape
  ▪ A Sphere **is a** ThreeDimensionalShape and **is a** Shape.

---

**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.
protected Members (Cont.)

- A superclass's private members are hidden in 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 method can be accessed from the subclass by preceding the superclass method name with keyword `super` and a dot (.) separator.

Case Study: Commission Employees

- 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

```
1 // Fig. 9.4: CommissionEmployee.java
2 // CommissionEmployee class represents an employee paid a
3 // percentage of gross sales.
4 public class CommissionEmployee extends Object
5 {
6    private String firstName;
7    private String lastName;
8    private String socialSecurityNumber;
9    private double grossSales; // gross weekly sales
10   private double commissionRate; // commission percentage
11   // five-argument constructor
12   public CommissionEmployee(String first, String last, String ssn,
13     double sales, double rate)
14   { // explicit call to Object constructor occurs here
15     firstName = first;
16     lastName = last;
17     socialSecurityNumber = ssn;
18     setGrossSales(sales); // validate and store gross sales
19     setCommissionRate(rate); // validate and store commission rate
20   } // end five-argument CommissionEmployee constructor
21
22   // set firstName
23   public void setFirstName(String first)
24   { first = first; // should validate
25   } // end method setFirstName
26
27   // return firstName
28   public String getFirstName()
29   { return firstName;
30   } // end method getFirstName
31
32   // set lastName
33   public void setLastName(String last)
34   { lastName = last; // should validate
35   } // end method setLastName
36
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 1 of 6.)

- CommissionEmployee inherits Object's methods.
- If you don't explicitly specify which class a new class extends, the class extends Object implicitly.

```
37   // set last name
38   public void setLastName(String last)
39   { lastName = last; // should validate
40   } // end method setLastName
41
42   // set social security number
43   public void setSocialSecurityNumber(String ssn)
44   { socialSecurityNumber = ssn; // validate and store SSN
45   } // end method setSocialSecurityNumber
46
```
public String getCommissionRate() {
    double commissionRate = rate; // Get the rate
    if (rate > 0.0 && rate < 1.0) {
        throw new IllegalArgumentException("Commission rate must be greater than 0.0 and less than 1.0");
    }
    return commissionRate;
}

public double getCommissionRate() {
    return commissionRate;
}

// Calculate earnings
public double earnings() {
    return commissionRate * grossSales;
}

// Set the gross sales amount
public void setGrossSales(double sales) {
    if (sales <= 0) {
        grossSales = sales;
    } else {
        throw new IllegalArgumentException("Gross sales must be greater than 0.0");
    }
}

// Return the gross sales amount
public double getGrossSales() {
    return grossSales;
}
Creating and Using a CommissionEmployee Class (Cont.)

- 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.

- `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.

Common Programming Error 9.1
Using an incorrect method signature when attempting to override a superclass method causes an unintentional method overload that can lead to subtle logic errors.

Error-Prevention Tip 9.1
Declare overridden methods with the `@Override` annotation to ensure at compilation time that you defined their signatures correctly. It’s always better to find errors at compile time rather than at runtime.
Case Study Part 2: Creating and Using a BasePlus-CommissionEmployee Class

Class BasePlusCommissionEmployee 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.

```java
// Fig. 9.5: CommissionEmployeeTest.java
// CommissionEmployee class test program.
public class CommissionEmployeeTest
{
    public static void main(String[] args)
    {
        // Instantiate CommissionEmployee object
        CommissionEmployee employee = new CommissionEmployee("Sue", "Jones", 222-22-2222, 10000.00);
        // get commission employee data
        System.out.printf("Employee information obtained by get methods: 
", employee);
        System.out.printf("First name is ", employee.getFirstName());
        System.out.printf("Last name is ", employee.getLastName());
        System.out.printf("Social security number is ", employee.getSocialSecurityNumber());
        System.out.printf("Gross sales is ", employee.getGrossSales());
        System.out.printf("Commission rate is ", employee.getCommissionRate());
    }
}
```

```
Fig. 9.5 | CommissionEmployee class test program (Part 1 of 2)
```

```
// Fig. 9.6: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class represents an employee who receives
// a base salary in addition to commission.
public class BasePlusCommissionEmployee
{
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;
    private double grossSales; // gross weekly sales
    private double commissionRate; // commission percentage
    private double baseSalary; // base salary per week

    // six-argument constructor
    public BasePlusCommissionEmployee(String firstName, String lastName,
                                        double grossSales, double commissionRate,
                                        double baseSalary)
    {
        // implicit call to Object constructor occurs here
        firstName = firstName;
        lastName = lastName;
        socialSecurityNumber = ssn;
        setGrossSales(sales); // validate and store gross sales
    }
}
```

```
Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 1 of 7)
```
Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 2 of 7.)

```java
23       setCommissionRate( rate ); // validate and store commission rate
24       setBaseSalary( salary ); // validate and store base salary
25     ) // end six-argument BasePlusCommissionEmployee constructor
26     
27     // set first name
28     public void setFirstName( String first )
29     {
30       firstName = first; // should validate
31     } // end method setFirstName
32     
33     // return first name
34     public String getFirstName()
35     {
36       return firstName;
37     } // end method getFirstName
38     
39     // set last name
40     public void setLastName( String last )
41     {
42       lastName = last; // should validate
43     } // end method setLastName
44

Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 3 of 7.)

```
Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 6 of 7.)

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 1 of 2.)

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 2 of 2.)
Case Study Part 2: 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 base-salary.
- The toString methods are nearly identical
  - BasePlusCommissionEmployee's toString also outputs instance variable baseSalary.

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 nightmare.

Software Engineering Observation 9.3

With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are made for these common features in the superclass—subclasses then inherit the changes. Without inheritance, changes would need to be made to all the source-code files that contain a copy of the code in question.

Case Study Part 3: 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 the superclass's public and protected members are directly accessible in the subclass.
Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 1 of 5.)

```java
1 // Fig. 9.8: BasePlusCommissionEmployee.java
2 // private superclass members cannot be accessed in a subclass.
3 public class BasePlusCommissionEmployee extends CommissionEmployee
4 {
5     private double baseSalary; // base salary per week
6     // six-argument constructor
7     public BasePlusCommissionEmployee(String firstName, String lastName,
8             String ssn, double sales, double rate, double salary)
9             throws IllegalArgumentException
10     {
11         // explicit call to superclass CommissionEmployee constructor
12         super( firstName, lastName, 
13             ssn, sales, rate );
14
15         setBaseSalary( salary ); // validate and store base salary
16     } // end six-argument BasePlusCommissionEmployee constructor
17 
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 2 of 5.)

```java
18     // set base salary
19     public void setBaseSalary( double salary )
20     {
21         if ( salary <= 0.0 )
22             baseSalary = salary;
23     } else
24         throw new IllegalArgumentException(
25             "Base salary must be >= 0.0."
26         );
27     } // end method setBaseSalary
28 
29     // return base salary
30     public double getBaseSalary()
31     {
32         return baseSalary;
33     } // end method getBaseSalary
34 
35     // calculate earnings
36     @Override // indicates that this method overrides a superclass method
37     public double earnings()
38     {
39         // not allowed; commissionRate and grossSales private in superclass
40         return baseSalary + ( commissionRate * grossSales );
41     } // end method earnings
42 
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 3 of 5.)

```java
43 // return String representation of BasePlusCommissionEmployee
44 public String toString()
45 {
46     // not allowed; attempts to access private superclass members
47     return String.format(
48         "No. 12 Value: Sales: %.2fYDS: %.2fYDS: %.2f",
49         "base-salaried commission employee", firstName, lastName,
50         "social security number", socialSecurityNumber,
51         "gross sales", grossSales, "commission rate", commissionRate,
52         "base salary", baseSalary );
53 } // end method toString
54 } // end class BasePlusCommissionEmployee
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 4 of 5.)
Case Study Part 3: Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)

- Each subclass constructor must implicitly or explicitly call its superclass constructor 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 subclass constructor’s body.
- If the subclass constructor did not invoke the superclass’s constructor explicitly, Java would attempt to invoke the superclass’s no-argument or default 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.

---

Case Study Part 4: CommissionEmployee–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:

```java
protected String firstName;
protected String lastName;
protected 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.

---

```java
public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary; // base salary per week
    // six-argument constructor
    public BasePlusCommissionEmployee(String firstName, String lastName,
        String ssn, double sales, double rate, double salary )
    {
        super(firstName, lastName, ssn, sales, rate ); // call superclass constructor
        setBaseSalary( salary ); // validate and store base salary
    } // end six-argument BasePlusCommissionEmployee constructor

    Fig. 9.9 | BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee (Part 1 of 3)
```
Case Study Part 4: CommissionEmployee–BasePlusCommissionEmployee 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 (51 lines) is considerably shorter than that in Fig. 9.6 (128 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 commission employee exists only in class CommissionEmployee.

Fig. 9.9  |  BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee. (Part 2 of 3.)
Case Study Part 4: CommissionEmployee—BasePlus-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.
- We may need to modify all the subclasses of the superclass if the superclass implementation changes.
  - You should be able to change the superclass implementation while still providing the same services to the subclasses.

Case Study Part 5: CommissionEmployee—BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables => BEST DESIGN

```java
1 // Fig. 9.10: CommissionEmployee.java
2 // CommissionEmployee class uses methods to manipulate its
3 // private instance variables.
4 public class CommissionEmployee
5 {
6   private String firstName;
7   private String lastName;
8   private String socialSecurityNumber;
9   private double grossSales; // gross weekly sales
10   private double commissionRate; // commission percentage
11
12   // five-argument constructor
13   public CommissionEmployee(String first, String last, String ssn,
14                             double sales, double rate)
15   {
16     // implicit call to Object constructor occurs here
17     firstName = first;
18     lastName = last;
19     socialSecurityNumber = ssn;
20     setGrossSales(sales); // validate and store gross sales
21     setCommissionRate(rate); // validate and store commission rate
22   } // end five-argument CommissionEmployee constructor

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its private instance variables. (Part 1 of 6.)
```

Error-Prevention Tip 9.2
When possible, do not include protected instance variables in a superclass. Instead, include non-private methods that access private instance variables. This will help ensure that objects of the class maintain consistent states.
// return last name
public String getLastName()
{
    return lastName;
}// end method getLastName

// set social security number
public void setSocialSecurityNumber(String ssn)
{
    socialSecurityNumber = ssn; // should validate
}// end method setSocialSecurityNumber

// return social security number
public String getSocialSecurityNumber()
{
    return socialSecurityNumber;
}// end method getSocialSecurityNumber

// set commission rate
public void setCommissionRate(double rate)
{
    if (rate > 0.0 && rate < 1.0)
    {
        commissionRate = rate;
    } else
    throw new IllegalArgumentException("Commission rate must be > 0.0 and < 1.0");
}// end method setCommissionRate

// return commission rate
public double getCommissionRate()
{
    return commissionRate;
}// end method getCommissionRate

// calculate earnings
public double earnings()
{
    return getCommissionRate() * getGrossSales();
}// end method earnings

// set gross sales amount
public void setGrossSales(double sales)
{
    if (sales == 0.0)
    {
        grossSales = sales;
    } else
    throw new IllegalArgumentException("Gross sales must be > 0.0");
}// end method setGrossSales

// return gross sales amount
public double getGrossSales()
{
    return grossSales;
}// end method getGrossSales

// return String representation of CommissionEmployee object
@Override // indicates that this method overrides a superclass method
public String toString()
{
    return String.format("%s %s %s: %.2f, 
    "commission employee", getName(), getlastName(),
    "social security number", getSocialSecurityNumber(),
    "gross sales", getGrossSales(),
    "commission rate", getCommissionRate());
}// end class CommissionEmployee

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its private instance variables. (Part 3 of 6.)
Case Study Part 5: CommissionEmployee–BasePlusCommissionEmployee 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 `CommissionEmployee`’s non-private methods and can access the private superclass members via those methods.

```java
public void setBaseSalary(double salary)
{
    if (salary > 0.0)
        baseSalary = salary;
    else
        throw new IllegalArgumentException("Base salary must be >= 0.0");
}
```

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.

```java
public class BasePlusCommissionEmployee extends CommissionEmployee
{
    private double baseSalary; // base salary per week

    public BasePlusCommissionEmployee(String first, String last, String ssn, double sales, double rate, double salary)
    {
        super(first, last, ssn, sales, rate, salary);
        setBaseSalary(salary); // validate and store base salary
    }
}
```

```java
@override // indicates that this method overrides a superclass method
public String toString()
{
    return String.format("%s %s, base salary = ", firstName, lastName);
    super.toString(), "base salary", getBaseSalary());
}
```

Method `earnings` overrides class the superclass’s `earnings` method.

Method `toString` overrides class `CommissionEmployee`’s `toString` method with `super.toString()`.

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, invokes its direct superclass's 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 class Object's constructor.
- Original subclass constructor's body finishes executing last.
- Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits.

UML Inheritance Diagrams

- Some details of UML class hierarchy from previous figure

```
Person
- name: String
  + setName(String newName): void
  + getName(): String
  + writeOutput(): void
  + hasSameName(Person otherPerson): boolean

Student
- studentNumber: int
  + reset(String newName, int newStudentNumber): void
  + getStudentNumber(): int
  + setStudentNumber(int newStudentNumber): void
  + writeOutput(): void
  + equals(Student otherStudent): boolean
```

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012