Inheritance and Abstract Classes

Java™ How to Program, 10/e

OBJECTIVES

In this chapter you'll:

- Learn the concept of polymorphism.
- Use overridden methods to effect polymorphism.
- Distinguish between abstract and concrete classes.
- Declare abstract methods to create abstract classes.
- Learn how polymorphism makes systems extensible and maintainable.
- Determine an object's type at execution time.
- Declare and implement interfaces, and become familiar with the Java SE 8 interface enhancements.

10.1 Introduction
10.2 Polymorphism Examples
10.3 Demonstrating Polymorphic Behavior
10.4 Abstract Classes and Methods
10.5 Case Study: Payroll System Using Polymorphism
- 10.5.1 Abstract Superclass `Employee`
- 10.5.2 Concrete Subclass `SalariedEmployee`
- 10.5.3 Concrete Subclass `HourlyEmployee`
- 10.5.4 Concrete Subclass `CommissionEmployee`
- 10.5.5 Indirect Concrete Subclass `BasePlusCommissionEmployee`
- 10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting

10.6 Allowed Assignments Between Superclass and Subclass Variables
10.7 `final` Methods and Classes
10.8 A Deeper Explanation of Issues with Calling Methods from Constructors

10.9 Creating and Using Interfaces

- 10.9.1 Developing a **Payable** Hierarchy
- 10.9.2 Interface **Payable**
- 10.9.3 Class **Invoice**
- 10.9.4 Modifying Class **Employee** to Implement Interface **Payable**
- 10.9.5 Modifying Class **SalariedEmployee** for Use in the **Payable** Hierarchy
- 10.9.6 Using Interface **Payable** to Process **Invoices** and **Employees** Polymorphically
- 10.9.7 Some Common Interfaces of the Java API

10.10 Java SE 8 Interface Enhancements

- 10.10.1 **default** Interface Methods
- 10.10.2 **static** Interface Methods
- 10.10.3 Functional Interfaces

10.11 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism

10.12 Wrap-Up

10.1 Introduction

- Polymorphism
  - Enables you to "program in the *general*" rather than "program in the *specific*."
  - Polymorphism enables you to write programs that process objects that share the same superclass as if they were all objects of the superclass; this can simplify programming.

10.1 Introduction (Cont.)

- Example: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation.
  - Each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x*-*y* coordinates. Each subclass implements method `move`.
  - A program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per second—namely, `move`.

10.1 Introduction (Cont.)

- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same message (i.e., `move`) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

10.1 Introduction (Cont.)

- With polymorphism, we can design and implement systems that are easily *extensible*
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
  - The new classes simply "plug right in."
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

10.1 Introduction (Cont.)

- Once a class implements an interface, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface.
- This is true of all subclasses of that class as well.
- Interfaces are particularly useful for assigning common functionality to possibly unrelated classes.
  - Allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls.

10.1 Introduction (Cont.)

- The chapter continues with an introduction to Java *interfaces*, which are particularly useful for assigning *common* functionality to possibly unrelated classes.
- This allows objects of these classes to be processed polymorphically—objects of classes that implement the *same* interface can respond to all of the interface method calls.

10.2 Polymorphism Examples

- Example: Quadrilaterals
  - If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
  - Any operation that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`.
  - These operations can also be performed on other `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.
  - Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

10.2 Polymorphism Examples (Cont.)

- Example: Space Objects in a Video Game
  - A video game manipulates objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Each inherits from `SpaceObject` and overrides its `draw` method.
  - A screen manager maintains a collection of references to objects of the various classes and periodically sends each object the same message—namely, `draw`.
  - Each object responds in a unique way.
    - A `Martian` object might draw itself in red with green eyes and the appropriate number of antennae.
    - A `SpaceShip` object might draw itself as a bright silver flying saucer.
    - A `LaserBeam` object might draw itself as a bright red beam across the screen.
  - The same message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

10.2 Polymorphism Examples (Cont.)

- A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code.
- To add new objects to our video game:
  - Build a class that extends `SpaceObject` and provides its own `draw` method implementation.
  - When objects of that class appear in the `SpaceObject` collection, the screen-manager code *invokes method `draw`, exactly as it does for every other object in the collection, regardless of its type.*
  - So the new objects simply "plug right in" without any modification of the screen manager code by the programmer.

Software Engineering Observation 10.1

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics. You can tell objects to behave in manners appropriate to those objects, without knowing their specific types, as long as they belong to the same inheritance hierarchy.

Software Engineering Observation 10.2

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

10.3 Demonstrating Polymorphic Behavior

- In the next example, we aim a superclass reference at a subclass object.
  - Invoking a method on a subclass object via a superclass reference invokes the subclass functionality
  - The type of the referenced object, not the type of the variable, determines which method is called
- This example demonstrates that an object of a subclass can be treated as an object of its superclass, enabling various interesting manipulations.
- A program can create an array of superclass variables that refer to objects of many subclass types.
  - Allowed because each subclass object *is an* object of its superclass.

10.3 Demonstrating Polymorphic Behavior (Cont.)

- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly *cast* the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.

Software Engineering Observation 10.3

Although it's allowed, you should generally avoid downcasting.

```
1  // Fig. 10.1: PolymorphismTest.java
2  // Assigning superclass and subclass references to superclass and
3  // subclass variables.
4
5  public class PolymorphismTest
6  {
7     public static void main(String[] args)
8     {
9        // assign superclass reference to superclass variable
10       CommissionEmployee commissionEmployee = new CommissionEmployee(
11          "Sue", "Jones", "222-22-2222", 10000, .06);
12
13       // assign subclass reference to subclass variable
14       BasePlusCommissionEmployee basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
17
18       // invoke toString on superclass object using superclass variable
19       System.out.printf("%s %s:%n%n%s%n%n",
20          "Call CommissionEmployee's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString());
22
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 4.)

```
23          // invoke toString on subclass object using subclass variable
24          System.out.printf("%s %s:%n%n%s%n%n",
25             "Call BasePlusCommissionEmployee's toString with subclass",
26             "reference to subclass object",
27             basePlusCommissionEmployee.toString());
28
29          // invoke toString on subclass object using superclass variable
30          CommissionEmployee commissionEmployee2 =
31             basePlusCommissionEmployee;
32          System.out.printf("%s %s:%n%n%s%n",
33             "Call BasePlusCommissionEmployee's toString with superclass",
34             "reference to subclass object", commissionEmployee2.toString());
35       } // end main
36    } // end class PolymorphismTest
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 4.)

```
Call CommissionEmployee's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 3 of 4.)

```
Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 4 of 4.)

10.3 Demonstrating Polymorphic Behavior (Cont.)

- *When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.*
  - The Java compiler allows this "crossover" because an object of a subclass *is an* object of its superclass (but *not* vice versa).
- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type.
  - If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use.
  - This process is called dynamic binding.

10.4 Abstract Classes and Methods

- Abstract classes
  - Sometimes it's useful to declare classes for which you never intend to create objects.
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called abstract superclasses.
  - Cannot be used to instantiate objects—abstract classes are *incomplete*.
  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

10.4 Abstract Classes and Methods (Cont.)

- Classes that can be used to instantiate objects are called concrete classes.
- Such classes provide implementations of every method they declare (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.

10.4 Abstract Classes and Methods (Cont.)

- Programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of subclass types.
  - You can write a method with a parameter of an abstract superclass type.
  - When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.
- Abstract classes sometimes constitute several levels of a hierarchy.

10.4 Abstract Classes and Methods (Cont.)

- You make a class abstract by declaring it with keyword abstract.
- An abstract class normally contains one or more abstract methods.
  - An abstract method is an instance method with keyword `abstract` in its declaration, as in

```
public abstract void draw(); // abstract method
```

- Abstract methods do not provide implementations.
- A class that contains abstract methods must be an abstract class even if that class contains some concrete (nonabstract) methods.
- Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods.
- **Constructors and `static` methods cannot be declared `abstract`.**

Software Engineering Observation 10.4

An abstract class declares common attributes and behaviors (both abstract and concrete) of the various classes in a class hierarchy. An abstract class typically contains one or more abstract methods that subclasses must override if they are to be concrete. The instance variables and concrete methods of an abstract class are subject to the normal rules of inheritance.

Common Programming Error 10.1

Attempting to instantiate an object of an abstract class is a compilation error.

Common Programming Error 10.2

Failure to implement a superclass's abstract methods in a subclass is a compilation error unless the subclass is also declared `abstract`.

10.4 Abstract Classes and Methods (Cont.)

- Cannot instantiate objects of abstract superclasses, but you can use abstract superclasses to declare variables
  - These can hold references to objects of *any* concrete class *derived from* those abstract superclasses.
  - We'll use such variables to manipulate subclass objects *polymorphically*.
- Can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

10.4 Abstract Classes and Methods (Cont.)

- Polymorphism is particularly effective for implementing so-called *layered software systems*.
- Example: Operating systems and device drivers.
  - Commands to read or write data from and to devices may have a certain uniformity.
  - Device drivers control all communication between the operating system and the devices.
  - A write message sent to a device-driver object is interpreted in the context of that driver and how it manipulates devices of a specific type.
  - The write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device.

10.4 Abstract Classes and Methods (Cont.)

- An object-oriented operating system might use an abstract superclass to provide an "interface" appropriate for all device drivers.
  - Subclasses are formed that all behave similarly.
  - The device-driver methods are declared as abstract methods in the abstract superclass.
  - The implementations of these abstract methods are provided in the subclasses that correspond to the specific types of device drivers.
- New devices are always being developed.
  - When you buy a new device, it comes with a device driver provided by the device vendor and is immediately operational after you connect it and install the driver.
- This is another elegant example of how polymorphism makes systems extensible.

10.5 Case Study: Payroll System Using Polymorphism

- Use an abstract method and polymorphism to perform payroll calculations based on the type of inheritance hierarchy headed by an employee.
- Enhanced employee inheritance hierarchy requirements:
  - A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants you to write a Java application that performs its payroll calculations polymorphically.

10.5 Case Study: Payroll System Using Polymorphism (Cont.)

- `abstract` class `Employee` represents the general concept of an employee.
- Subclasses: `SalariedEmployee`, `CommissionEmployee`, `HourlyEmployee` and `BasePlusCommissionEmployee` (an indirect subclass)
- Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application.
- Abstract class names are italicized in the UML.

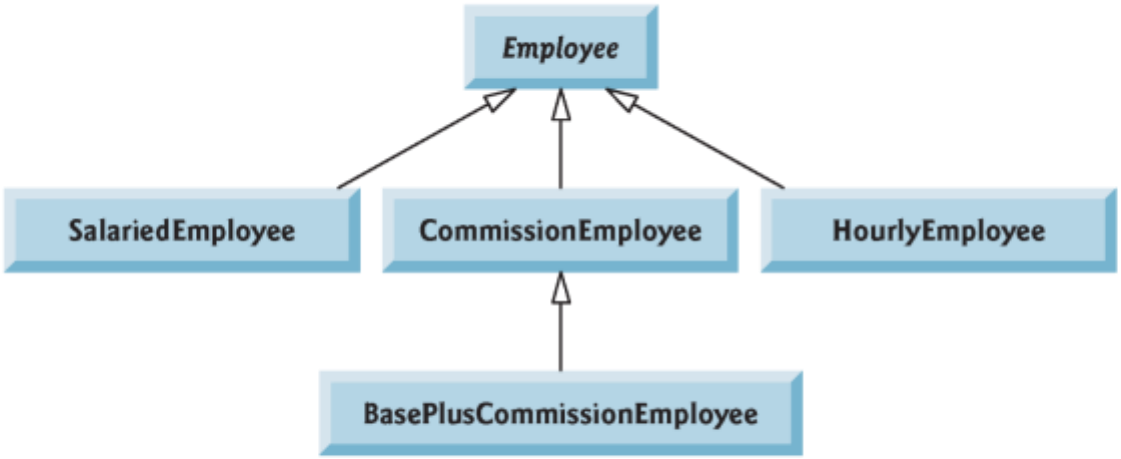

Fig. 10.2 | `Employee` hierarchy UML class diagram.

10.5 Case Study: Payroll System Using Polymorphism (Cont.)

- Abstract superclass `Employee` declares the common front to the hierarchy
- Each employee has a first name, a last name and a social security number defined in abstract superclass `Employee`.

10.5.1 Abstract Superclass Employee

- Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString`, in addition to the *get* and *set* methods that manipulate `Employee`'s instance variables.
- An `earnings` method applies to all employees, but each earnings calculation depends on the employee's class.
  - An `abstract` method—there is not enough information to determine what amount `earnings` should return.
  - Each subclass overrides `earnings` with an appropriate implementation.
- Iterate through the array of `Employee`s and call method `earnings` for each `Employee` subclass object.
  - Method calls processed polymorphically.

10.5.1 Abstract Superclass `Employee` (Cont.)

- The diagram in Fig. 10.3 shows each of the five classes in the hierarchy down the left side and methods `earnings` and `toString` across the top.
- For each class, the diagram shows the desired results of each method.
- Declaring the `earnings` method `abstract` indicates that each concrete subclass must provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*

Hourly-Employee	if (hours <= 40)
	  wage * hours
	else if (hours > 40)
	{
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee
5  {
6     private final String firstName;
7     private final String lastName;
8     private final String socialSecurityNumber;
9
10    // constructor
11    public Employee(String firstName, String lastName,
12       String socialSecurityNumber)
13    {
14       this.firstName = firstName;
15       this.lastName = lastName;
16       this.socialSecurityNumber = socialSecurityNumber;
17    }
18
19    // return first name
20    public String getFirstName()
21    {
22       return firstName;
23    }
24
```

Fig. 10.4 | Employee abstract superclass. (Part 1 of 2.)

```
25      // return last name
26      public String getLastName()
27      {
28         return lastName;
29      }
30
31      // return social security number
32      public String getSocialSecurityNumber()
33      {
34         return socialSecurityNumber;
35      }
36
37      // return String representation of Employee object
38      @Override
39      public String toString()
40      {
41         return String.format("%s %s%nsocial security number: %s",
42            getFirstName(), getLastName(), getSocialSecurityNumber());
43      }
44
45      // abstract method must be overridden by concrete subclasses
46      public abstract double earnings(); // no implementation here
47   } // end abstract class Employee
```

Fig. 10.4 | Employee abstract superclass. (Part 2 of 2.)

10.5.2 Concrete Subclass SalariedEmployee

```
1  // Fig. 10.5: SalariedEmployee.java
2  // SalariedEmployee concrete class extends abstract class Employee.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // constructor
9     public SalariedEmployee(String firstName, String lastName,
10       String socialSecurityNumber, double weeklySalary)
11    {
12       super(firstName, lastName, socialSecurityNumber);
13
14       if (weeklySalary < 0.0)
15          throw new IllegalArgumentException(
16             "Weekly salary must be >= 0.0");
17
18       this.weeklySalary = weeklySalary;
19    }
20
```

Fig. 10.5 | SalariedEmployee concrete class extends abstract class Employee. (Part 1 of 3.)

```
21      // set salary
22      public void setWeeklySalary(double weeklySalary)
23      {
24         if (weeklySalary < 0.0)
25            throw new IllegalArgumentException(
26               "Weekly salary must be >= 0.0");
27
28         this.weeklySalary = weeklySalary;
29      }
30
31      // return salary
32      public double getWeeklySalary()
33      {
34         return weeklySalary;
35      }
36
37      // calculate earnings; override abstract method earnings in Employee
38      @Override
39      public double earnings()
40      {
41         return getWeeklySalary();
42      }
43
```

Fig. 10.5 | SalariedEmployee concrete class extends abstract class Employee. (Part 2 of 3.)

```
44      // return String representation of SalariedEmployee object
45      @Override
46      public String toString()
47      {
48         return String.format("salaried employee: %s%n%s: $%,.2f",
49            super.toString(), "weekly salary", getWeeklySalary());
50      }
51   } // end class SalariedEmployee
```

Fig. 10.5 | `SalariedEmployee` concrete class extends `abstract` class `Employee`. (Part 3 of 3.)

10.5.3 Concrete Subclass HourlyEmployee

```
1  // Fig. 10.6: HourlyEmployee.java
2  // HourlyEmployee class extends Employee.
3
4  public class HourlyEmployee extends Employee
5  {
6     private double wage; // wage per hour
7     private double hours; // hours worked for week
8
9     // constructor
10    public HourlyEmployee(String firstName, String lastName,
11       String socialSecurityNumber, double wage, double hours)
12    {
13       super(firstName, lastName, socialSecurityNumber);
14
15       if (wage < 0.0) // validate wage
16          throw new IllegalArgumentException(
17             "Hourly wage must be >= 0.0");
18
19       if ((hours < 0.0) || (hours > 168.0)) // validate hours
20          throw new IllegalArgumentException(
21             "Hours worked must be >= 0.0 and <= 168.0");
22
```

Fig. 10.6 | HourlyEmployee class extends Employee. (Part 1 of 4.)

```
23          this.wage = wage;
24          this.hours = hours;
25       }
26
27       // set wage
28       public void setWage(double wage)
29       {
30          if (wage < 0.0) // validate wage
31             throw new IllegalArgumentException(
32                "Hourly wage must be >= 0.0");
33
34          this.wage = wage;
35       }
36
37       // return wage
38       public double getWage()
39       {
40          return wage;
41       }
42
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 2 of 4.)

```
43      // set hours worked
44      public void setHours(double hours)
45      {
46         if ((hours < 0.0) || (hours > 168.0)) // validate hours
47            throw new IllegalArgumentException(
48               "Hours worked must be >= 0.0 and <= 168.0");
49
50         this.hours = hours;
51      }
52
53      // return hours worked
54      public double getHours()
55      {
56         return hours;
57      }
58
59      // calculate earnings; override abstract method earnings in Employee
60      @Override
61      public double earnings()
62      {
63         if (getHours() <= 40) // no overtime
64            return getWage() * getHours();
65         else
66            return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
67      }
```

Fig. 10.6 | HourlyEmployee class extends Employee. (Part 3 of 4.)

```
68
69      // return String representation of HourlyEmployee object
70      @Override
71      public String toString()
72      {
73         return String.format("hourly employee: %s%n%s: $%,.2f; %s: %,.2f",
74            super.toString(), "hourly wage", getWage(),
75            "hours worked", getHours());
76      }
77   } // end class HourlyEmployee
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 4 of 4.)

10.5.4 Concrete Subclass CommissionEmployee

```
1  // Fig. 10.7: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3
4  public class CommissionEmployee extends Employee
5  {
6     private double grossSales; // gross weekly sales
7     private double commissionRate; // commission percentage
8
9     // constructor
10    public CommissionEmployee(String firstName, String lastName,
11       String socialSecurityNumber, double grossSales,
12       double commissionRate)
13    {
14       super(firstName, lastName, socialSecurityNumber);
15
16       if (commissionRate <= 0.0 || commissionRate >= 1.0) // validate
17          throw new IllegalArgumentException(
18             "Commission rate must be > 0.0 and < 1.0");
19
20       if (grossSales < 0.0) // validate
21          throw new IllegalArgumentException("Gross sales must be >= 0.0");
22
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 1 of 4.)

```
23          this.grossSales = grossSales;
24          this.commissionRate = commissionRate;
25       }
26
27       // set gross sales amount
28       public void setGrossSales(double grossSales)
29       {
30          if (grossSales < 0.0) // validate
31             throw new IllegalArgumentException("Gross sales must be >= 0.0");
32
33          this.grossSales = grossSales;
34       }
35
36       // return gross sales amount
37       public double getGrossSales()
38       {
39          return grossSales;
40       }
41
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 2 of 4.)

```
42      // set commission rate
43      public void setCommissionRate(double commissionRate)
44      {
45         if (commissionRate <= 0.0 || commissionRate >= 1.0) // validate
46            throw new IllegalArgumentException(
47               "Commission rate must be > 0.0 and < 1.0");
48
49         this.commissionRate = commissionRate;
50      }
51
52      // return commission rate
53      public double getCommissionRate()
54      {
55         return commissionRate;
56      }
57
58      // calculate earnings; override abstract method earnings in Employee
59      @Override
60      public double earnings()
61      {
62         return getCommissionRate() * getGrossSales();
63      }
64
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 3 of 4.)

```
65      // return String representation of CommissionEmployee object
66      @Override
67      public String toString()
68      {
69         return String.format("%s: %s%n%s: $%,.2f; %s: %.2f",
70            "commission employee", super.toString(),
71            "gross sales", getGrossSales(),
72            "commission rate", getCommissionRate());
73      }
74   } // end class CommissionEmployee
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 4 of 4.)

10.5.5 Indirect Concrete Subclass BasePlusCommissionEmployee

```
1  // Fig. 10.8: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class extends CommissionEmployee.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // constructor
9     public BasePlusCommissionEmployee(String firstName, String lastName,
10       String socialSecurityNumber, double grossSales,
11       double commissionRate, double baseSalary)
12    {
13       super(firstName, lastName, socialSecurityNumber,
14          grossSales, commissionRate);
15
16       if (baseSalary < 0.0) // validate baseSalary
17          throw new IllegalArgumentException("Base salary must be >= 0.0");
18
19       this.baseSalary = baseSalary;
20    }
21
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee.
(Part 1 of 3.)

```
22      // set base salary
23      public void setBaseSalary(double baseSalary)
24      {
25         if (baseSalary < 0.0) // validate baseSalary
26            throw new IllegalArgumentException("Base salary must be >= 0.0");
27
28         this.baseSalary = baseSalary;
29      }
30
31      // return base salary
32      public double getBaseSalary()
33      {
34         return baseSalary;
35      }
36
37      // calculate earnings; override method earnings in CommissionEmployee
38      @Override
39      public double earnings()
40      {
41         return getBaseSalary() + super.earnings();
42      }
43
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee. (Part 2 of 3.)

```
44      // return String representation of BasePlusCommissionEmployee object
45      @Override
46      public String toString()
47      {
48         return String.format("%s %s; %s: $%,.2f",
49            "base-salaried", super.toString(),
50            "base salary", getBaseSalary());
51      }
52   } // end class BasePlusCommissionEmployee
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee. (Part 3 of 3.)

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting

- Fig. 10.9 creates an object of each of the four concrete.
  - Manipulates these objects *nonpolymorphically*, via variables of each object's own type, then *polymorphically*, using an array of `Employee` variables.
- While processing the objects polymorphically, the program increases the base salary of each `BasePlusCommissionEmployee` by 10%
  - Requires *determining the object's type at execution time*.
- Finally, the program polymorphically determines and outputs the *type* of each object in the `Employee` array.

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest
5  {
6     public static void main(String[] args)
7     {
8        // create subclass objects
9        SalariedEmployee salariedEmployee =
10          new SalariedEmployee("John", "Smith", "111-11-1111", 800.00);
11       HourlyEmployee hourlyEmployee =
12          new HourlyEmployee("Karen", "Price", "222-22-2222", 16.75, 40);
13       CommissionEmployee commissionEmployee =
14          new CommissionEmployee(
15          "Sue", "Jones", "333-33-3333", 10000, .06);
16       BasePlusCommissionEmployee basePlusCommissionEmployee =
17          new BasePlusCommissionEmployee(
18          "Bob", "Lewis", "444-44-4444", 5000, .04, 300);
19
20       System.out.println("Employees processed individually:");
21
```

Fig. 10.9 | Employee hierarchy test program. (Part 1 of 6.)

```
22          System.out.printf("%n%s%n%s: $%,.2f%n%n",
23             salariedEmployee, "earned", salariedEmployee.earnings());
24          System.out.printf("%s%n%s: $%,.2f%n%n",
25             hourlyEmployee, "earned", hourlyEmployee.earnings());
26          System.out.printf("%s%n%s: $%,.2f%n%n",
27             commissionEmployee, "earned", commissionEmployee.earnings());
28          System.out.printf("%s%n%s: $%,.2f%n%n",
29             basePlusCommissionEmployee,
30             "earned", basePlusCommissionEmployee.earnings());
31
32          // create four-element Employee array
33          Employee[] employees = new Employee[4];
34
35          // initialize array with Employees
36          employees[0] = salariedEmployee;
37          employees[1] = hourlyEmployee;
38          employees[2] = commissionEmployee;
39          employees[3] = basePlusCommissionEmployee;
40
41          System.out.printf("Employees processed polymorphically:%n%n");
42
```

Fig. 10.9 | Employee hierarchy test program. (Part 2 of 6.)

```
43          // generically process each element in array employees
44          for (Employee currentEmployee : employees)
45          {
46             System.out.println(currentEmployee); // invokes toString
47
48             // determine whether element is a BasePlusCommissionEmployee
49             if currentEmployee instanceof BasePlusCommissionEmployee()
50             {
51                // downcast Employee reference to
52                // BasePlusCommissionEmployee reference
53                BasePlusCommissionEmployee employee =
54                   (BasePlusCommissionEmployee) currentEmployee  ;
55
56                employee.setBaseSalary(1.10 * employee.getBaseSalary());
57
58                System.out.printf(
59                   "new base salary with 10%% increase is: $%,.2f%n",
60                   employee.getBaseSalary());
61             } // end if
62
63             System.out.printf(
64                "earned $%,.2f%n%n", currentEmployee.earnings());
65          } // end for
66
```

Fig. 10.9 | Employee hierarchy test program. (Part 3 of 6.)

```
67          // get type name of each object in employees array
68          for (int j = 0; j < employees.length; j++)
69             System.out.printf("Employee %d is a %s%n", j,
70                employees[j].getClass().getName());
71       } // end main
72    } // end class PayrollSystemTest
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 4 of 6.)

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00

Employees processed polymorphically:
```

Fig. 10.9 | Employee hierarchy test program. (Part 5 of 6.)

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Fig. 10.9 | Employee hierarchy test program. (Part 6 of 6.)

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting (Cont.)

- All calls to method `toString` and `earnings` are resolved at execution time, based on the *type* of the object to which `currentEmployee` refers.
  - Known as dynamic binding or late binding.
  - Java decides which class's `toString` method to call at execution time rather than at compile time
- A superclass reference can be used to invoke only methods of the *superclass*—the *subclass* method implementations are invoked *polymorphically*.
- Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.

Common Programming Error 10.3

Assigning a superclass variable to a subclass variable is a compilation error.

Common Programming Error 10.4

When downcasting a reference, a `ClassCastException` *occurs if the referenced object at execution time does not have an* is-a *relationship with the type specified in the cast operator.*

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting (Cont.)

- Every object *knows its own class* and can access this information through the getClass method, which all classes inherit from class `Object`.
  - The `getClass` method returns an object of type Class (from package `java.lang`), which contains information about the object's type, including its class name.
  - The result of the `getClass` call is used to invoke getName to get the object's class name.

Software Engineering Observation 10.5

Although the actual method that's called depends on the runtime type of the object to which a variable refers, a variable can be used to invoke only those methods that are members of that variable's type, which the compiler verifies.

10.6 Summary of the Allowed Assignments Between Superclass and Subclass Variables

- There are three proper ways to assign superclass and subclass references to variables of superclass and subclass types.
- Assigning a superclass reference to a superclass variable is straightforward.
- Assigning a subclass reference to a subclass variable is straightfor-ward.
- Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an object of its superclass*.
  - The superclass variable can be used to refer only to superclass members.
  - If this code refers to subclass-only mem-bers through the superclass variable, the compiler reports errors.

Why abstract classes?

Consider the following three classes

```
class Database {
    public String[] getTableNames() { return null; }
}
class SqlDatabase extends Database { } //TODO: override getTableNames
class OracleDatabase extends Database { }  //TODO: override getTableNames
```

- You don't have to make the Database class abstract, even though there is an obvious problem with its implementation: When you are writing this program, you could type **new Database()** which will result in logical error.

- Abstract classes improve the situation by preventing a developer from instantiating the base class, because a developer has marked it as having missing functionality.

- It also provides compile-time safety so that you can ensure that any classes that extend your abstract class provide the bare minimum functionality to work, and you don't need to worry about putting stub methods (like the one above) that inheritors somehow have to override.

10.7 `final` Methods and Classes

- A final method in a superclass cannot be overridden in a subclass.
  - Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass.
  - Methods that are declared `static` are implicitly `final`.
  - A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as static binding.

10.7 `final` Methods and Classes (Cont.)

- A final class cannot be extended to create a subclass.
  - All methods in a `final` class are implicitly `final`.
- Class `String` is an example of a `final` class.
  - If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected.
  - Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions.

Common Programming Error 10.5

Attempting to declare a subclass of a `final` *class is a compilation error.*

Software Engineering Observation 10.6

In the Java API, the vast majority of classes are not *declared* `final`. *This enables inheritance and polymorphism. However, in some cases, it's important to declare classes* `final`*—typically for security reasons. Also, unless you carefully design a class for extension, you should declare the class as* `final` *to avoid (often subtle) errors.*

10.8 A Deeper Explanation of Issues with Calling Methods from Constructors

- Do not call overridable methods from constructors.
- When creating a *subclass* object, this could lead to an overridden method being called before the *subclass* object is fully initialized.
- Recall that when you construct a *subclass* object, its constructor first calls one of the direct *superclass's* constructors.
- If the *superclass* constructor calls an overridable method, the *subclass's* version of that method will be called by the *superclass* constructor—before the *subclass* constructor's body has a chance to execute.
- This could lead to subtle, difficult-to-detect errors if the *subclass* method that was called depends on initialization that has not yet been performed in the *subclass* constructor's body.
- It's acceptable to call a `static` method from a constructor.