

Polymorphism I

CS102A Lecture 11

James YU

Nov. 23, 2020



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

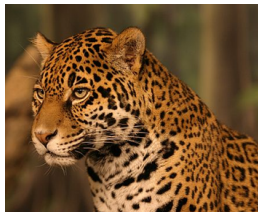
Objectives

- Polymorphism
- Override
- Abstract and concrete classes
- Determine an object's type
- Interface



Polymorphism

- The word polymorphism is used in various disciplines to describe situations where something occurs in several different forms.



Biology example: About 6% of the South American population of jaguars are dark-morph jaguars.



Polymorphism in OOP

- In Java, *polymorphism* is the ability of an object to take on many forms.
- Objects of different types can be accessed through the same interface. Each type can provide its own, independent implementation of this interface.
 - Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes *Fish*, *Frog* and *Bird* represent three types of animals under study.
 - Each class extends superclass *Animal*, which contains a method *move* and maintains an animal's current *location* as x-y coordinates. Each subclass implements (*overrides*) method *move*.



Polymorphism in OOP

- Each specific type of Animal responds to a **move** message in a unique way.
 - A fish might swim 3 feet.
 - A frog might jump 5 feet.
 - A bird might fly 10 feet.



Polymorphism in OOP

- An object of subclass can be treated as an object of the super class.
- 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*.



Polymorphism in OOP

- Polymorphism enables you to write programs to process objects that share the same superclass *as if* they're all objects of the superclass.
- With polymorphism, we can design and implement *extensible* systems.
 - 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 only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes (e.g., the part that creates the corresponding objects).



Another example: Quadrilaterals

- If **Rectangle** is derived from **Quadrilateral**, then a **Rectangle** object is a more specific version of a **Quadrilateral**.
- Any operation (e.g., calculating area) that can be performed on a **Quadrilateral** can also be performed on a **Rectangle**.
- These operations can also be performed on other **Quadrilaterals**, such as **Squares**, **Parallelograms** and **Trapezoids**.
- **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 exact type of the object.

Polymorphic behavior

- All Java objects are *polymorphic* since any object will pass the *IS-A* test for at least their own type and the class `Object`.
 - A bird is an instance of `Bird` class, also an instance of `Animal` and `Object`.
- Earlier, when we write programs, we aim super class variables at superclass objects and subclass variables at subclass objects



```
1 CommissionEmployee employee1 = new CommissionEmployee(...);  
2 BasePlusCommissionEmployee employee2 = new BasePlusCommissionEmployee  
  (...);
```

Such assignments are natural.



Polymorphic behavior

- In Java, we can also **aim a superclass reference at a subclass object** (the most common use of polymorphism).

```
1 CommissionEmployee employee = new BasePlusCommissionEmployee(...);
```

This is totally fine due to the *is-a* relationship (an instance of the subclass is also an instance of superclass)

```
1 BasePlusCommissionEmployee employee = new CommissionEmployee(...);
```

This will not compile, the is-a relationship only applies up the class hierarchy.



Polymorphic behavior

- Then the question comes...

```
1 CommissionEmployee employee = new BasePlusCommissionEmployee(...);  
2 double earnings = employee.earnings();
```

- Question: Which version of `earnings()` will be invoked? The one in the superclass or the one overridden by the subclass?
 - Which method is called is determined by the type of the *referenced object*, not the type of the variable.
 - 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.



Demo

```
1 // Assigning superclass and subclass references to superclass and
2 // subclass variables.
3
4 public class PolymorphismTest {
5     public static void main(String[] args) {
6         // assign superclass reference to superclass variable
7         CommissionEmployee commissionEmployee = new CommissionEmployee(
8             "Sue", "Jones", "222-22-2222", 10000, .06);
9
10        // assign subclass reference to subclass variable
11        BasePlusCommissionEmployee basePlusCommissionEmployee =
12            new BasePlusCommissionEmployee(
13                "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
```



Demo

```
11 // invoke toString on superclass object using superclass variable
12 System.out.printf("%s %s:%n%n%s%n%n",
13     "Call CommissionEmployee's toString with superclass reference ",
14     "to superclass object", commissionEmployee.toString());
15 // invoke toString on subclass object using subclass variable
16 System.out.printf("%s %s:%n%n%s%n%n",
17     "Call BasePlusCommissionEmployee's toString with subclass",
18     "reference to subclass object",
19     BasePlusCommissionEmployee.toString());
20 // invoke toString on subclass object using superclass variable
21 CommissionEmployee commissionEmployee2 = basePlusCommissionEmployee;
22 System.out.printf("%s %s:%n%n%s%n",
23     "Call BasePlusCommissionEmployee's toString with superclass",
24     "reference to subclass object", commissionEmployee2.toString());
25 } // end main
26 } // end class PolymorphismTest
```



Demo

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
```



Demo

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
```



Polymorphic behavior

- When the Java compiler encounters a method call made through a variable, it 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 will be successfully compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use (JVM will take care of this).
 - This process is called *dynamic binding*. Binding means “associating method calls to the appropriate method body”.



Polymorphic behavior

- What if the subclasses do not override the superclass' method to implement its own specific version (i.e., use the inherited one as is)?
 - Can we force a subclass to override a method inherited from superclass?
 - Yes, we can leverage the power of *abstract class*.



Concrete classes

- All classes we have defined so far provide implementations of every method they declare (some of the implementations can be inherited).
- They are called “*concrete classes*”.
- Concrete classes can be used to instantiate objects.

Abstract classes

- Sometimes it's useful to declare “incomplete” classes for which you never intend to create objects.
- Used only as superclasses in inheritance hierarchies.
- They are called “*abstract classes*”, cannot be used to instantiate objects.
- Subclasses must declare the “missing pieces” to become concrete classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.



Abstract classes

- An abstract class provides a superclass from which other classes can inherit and thus share a common design. **Not all hierarchies contain abstract classes.**
- Programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of subclass types (i.e., program in general not in specific).
 - `moveAnAnimal(Animal a) ...` (suppose `Animal` is an abstract class).
 - When called, such a method can receive an object of any concrete class that directly or indirectly extends the abstract superclass `Animal`.

Declaring abstract classes

- You make a class abstract by declaring it with keyword `abstract`.
- An abstract class normally contains one or more *abstract methods*, which are declared with the keyword `abstract` and provides no implementations.

```
1 public abstract class Animal {  
2     public abstract void move(); // Be careful, no brackets {}  
3     // ...  
4 }
```



Abstract method

```
1 public abstract class Animal {  
2     public abstract void move();  
3 }
```

- Abstract methods have the same visibility rules as normal methods, except that they cannot be `private`.
 - Private abstract methods make no sense since abstract methods are intended to be overridden by subclasses.
- Abstract methods have no implementations because the abstract classes are too general and only specify the common interfaces of subclasses.
 - Think about this: How can an `Animal` class provide an appropriate implementation for `move()` method without knowing the specific type of the animal? Every type of animal moves in a different way.

Abstract method

- A class that contains **abstract** methods must be declared as **abstract** even if that class contains some *concrete methods*.
- If a subclass does not implement all abstract methods it inherits from the superclass, the subclass must also be declared as **abstract** and thus **cannot be used to instantiate objects**.
- Constructors and **static** methods cannot be declared **abstract** (constructors are not inherited, non-**private static** methods are inherited but cannot be overridden).

Abstract class

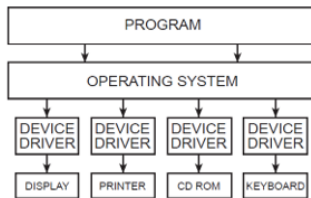
- Although abstract classes cannot be used to instantiate objects, they can be used to declare variables.
- Abstract superclass variables can hold references to objects of any concrete class derived from them.

```
1 Animal animal = new Frog(...); // assuming Animal is abstract
```

- Such practice is commonly adopted to manipulate objects polymorphically. Note that we can use abstract superclass names to invoke static methods declared in those abstract superclasses.

The usefulness of polymorphism

- Very effective for implementing layered software systems.
- Example: Operating systems (OS) and device drivers.
- Device drivers control all communication between the OS 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 (i.e., read/write commands to different devices may have uniformity)



The usefulness of polymorphism

- An object-oriented design of OS might use an abstract superclass to provide an “interface” appropriate for all device drivers.
 - The device-driver methods are declared as **abstract** 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 driver provided by the device vendor and is immediately operational after you connect it and install the driver (think about this: do we need to modify the operating system code?)

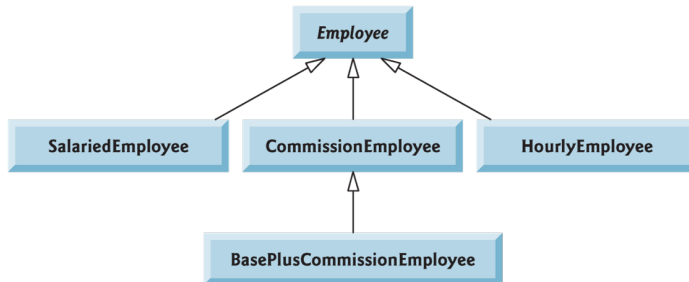


Case study: A payroll system using polymorphism

- The company pays its four types of employees on a weekly basis.
 - Salaried employees get a fixed weekly salary regardless of working hours.
 - Hourly employees are paid for each hour of work and receive overtime pay (i.e., 1.5x their hourly salary rate) for after 40 hours worked.
 - Commission employees are paid a percentage of their sales.
 - Salaried-commission employees get a base salary + 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 to write a Java application that performs its payroll calculations polymorphically.



The design: Main classes



The **Employee** abstract class

- Abstract superclass **Employee** declares the “interface”: the set of methods that a program can invoke on all **Employee** objects.
 - Each employee has a first name, a last name and a social security number. This applies to all employee types.
 - Set and get methods for each field. These methods are concrete and the same for all employee types.
 - A constructor for initializing the three fields.
 - Represent the employee's basic information as a string.
 - **earnings()**: abstract method that needs to be implemented by the subclasses (the **Employee** class does not have enough information to do the calculation).



The **Employee** abstract class

```
1 // Employee abstract superclass.
2 public abstract class Employee {
3     private final String firstName;
4     private final String lastName;
5     private final String socialSecurityNumber;
6
7     // constructor
8     public Employee(String firstName, String lastName,
9         String socialSecurityNumber) {
10         this.firstName = firstName;
11         this.lastName = lastName;
12         this.socialSecurityNumber = socialSecurityNumber;
13     }
14
15     public String getFirstName() { return firstName; }
16     public String getLastName() { return lastName; }
```



The **Employee** abstract class

```
17 public String getSocialSecurityNumber() {  
18     return socialSecurityNumber;  
19 }  
20  
21 // return String representation of Employee object  
22 @Override  
23 public String toString() {  
24     return String.format("%s %s\nsocial security number: %s",  
25         getFirstName(), getLastName(), getSocialSecurityNumber());  
26 }  
27  
28 // abstract method must be overridden by concrete subclasses  
29 public abstract double earnings(); // no implementation here  
30 } // end abstract class Employee  
31 } // end abstract class Employee
```



The `SalariedEmployee` class

- Defines a new field `weeklySalary`, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.

```
1 // SalariedEmployee concrete class extends abstract class Employee.
2 public class SalariedEmployee extends Employee {
3     private double weeklySalary;
4
5     // constructor
6     public SalariedEmployee(String firstName, String lastName,
7         String socialSecurityNumber, double weeklySalary) {
8         super(firstName, lastName, socialSecurityNumber);
9
10        if (weeklySalary < 0.0)
11            throw new IllegalArgumentException(
12                "Weekly salary must be >= 0.0");
```



The SalariedEmployee class

```
13     this.weeklySalary = weeklySalary;
14 }
15
16 // set salary
17 public void setWeeklySalary(double weeklySalary) {
18     if (weeklySalary < 0.0)
19         throw new IllegalArgumentException(
20             "Weekly salary must be >= 0.0");
21
22     this.weeklySalary = weeklySalary;
23 }
24
25 // return salary
26 public double getWeeklySalary() { return weeklySalary; }
```



The SalariedEmployee class

```
27 // calculate earnings; override abstract method earnings in Employee
28 @Override
29 public double earnings() {
30     return getWeeklySalary();
31 }
32
33 // return String representation of SalariedEmployee object
34 @Override
35 public String toString() {
36     return String.format("salaried employee: %s\n%s: $%,.2f",
37         super.toString(), "weekly salary", getWeeklySalary());
38 }
39 } // end class SalariedEmployee
```



The `CommissionEmployee` class

- Defines two new fields `grossSales` and `commissionRate`, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.

```
1 // CommissionEmployee class extends Employee.
2 public class CommissionEmployee extends Employee {
3     private double grossSales; // gross weekly sales
4     private double commissionRate; // commission percentage
5
6     // constructor
7     public CommissionEmployee(String firstName, String lastName,
8         String socialSecurityNumber, double grossSales,
9         double commissionRate) {
10         super(firstName, lastName, socialSecurityNumber);
```



The `CommissionEmployee` class

```
11     if (commissionRate <= 0.0 || commissionRate >= 1.0) // validate
12         throw new IllegalArgumentException(
13             "Commission rate must be > 0.0 and < 1.0");
14
15     if (grossSales < 0.0) // validate
16         throw new IllegalArgumentException("Gross sales must be >= 0.0");
17     this.grossSales = grossSales;
18     this.commissionRate = commissionRate;
19 }
20
21 // set gross sales amount
22 public void setGrossSales(double grossSales) {
23     if (grossSales < 0.0) // validate
24         throw new IllegalArgumentException("Gross sales must be >= 0.0");
25
26     this.grossSales = grossSales;
27 }
```



The `CommissionEmployee` class

```
28 // return gross sales amount
29 public double getGrossSales() { return grossSales; }
30
31 // set commission rate
32 public void setCommissionRate(double commissionRate) {
33     if (commissionRate <= 0.0 || commissionRate >= 1.0) // validate
34         throw new IllegalArgumentException(
35             "Commission rate must be > 0.0 and < 1.0");
36
37     this.commissionRate = commissionRate;
38 }
39
40 // return commission rate
41 public double getCommissionRate() { return commissionRate; }
```



The `CommissionEmployee` class

```
42 // calculate earnings; override abstract method earnings in Employee
43 @Override
44 public double earnings() {
45     return getCommissionRate() * getGrossSales();
46 }
47
48 // return String representation of CommissionEmployee object
49 @Override
50 public String toString() {
51     return String.format("%s: %s\n%s: $%,.2f; %s: %.2f",
52         "commission employee", super.toString(),
53         "gross sales", getGrossSales(),
54         "commission rate", getCommissionRate());
55 }
56 } // end class CommissionEmployee
```



The `HourlyEmployee` class

- Defines two new fields `wage` and `hours`, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.

```
1 // HourlyEmployee class extends Employee.
2 public class HourlyEmployee extends Employee {
3     private double wage; // wage per hour
4     private double hours; // hours worked for week
5
6     // constructor
7     public HourlyEmployee(String firstName, String lastName,
8         String socialSecurityNumber, double wage, double hours) {
9         super(firstName, lastName, socialSecurityNumber);
10
11         if (wage < 0.0) // validate wage
12             throw new IllegalArgumentException("Hourly wage must be >= 0.0");
```



The `HourlyEmployee` class

```
13     if ((hours < 0.0) || (hours > 168.0)) // validate hours
14         throw new IllegalArgumentException(
15             "Hours worked must be >= 0.0 and <= 168.0");
16
17     this.wage = wage;
18     this.hours = hours;
19 }
20
21 // set wage
22 public void setWage(double wage) {
23     if (wage < 0.0) // validate wage
24         throw new IllegalArgumentException("Hourly wage must be >= 0.0");
25
26     this.wage = wage;
27 }
```



The HourlyEmployee class

```
28 // return wage
29 public double getWage() { return wage; }
30
31 // set hours worked
32 public void setHours(double hours) {
33     if ((hours < 0.0) || (hours > 168.0)) // validate hours
34         throw new IllegalArgumentException(
35             "Hours worked must be >= 0.0 and <= 168.0");
36
37     this.hours = hours;
38 }
39
40 // return hours worked
41 public double getHours() { return hours; }
```



The HourlyEmployee class

```
42 // calculate earnings; override abstract method earnings in Employee
43 @Override
44 public double earnings() {
45     if (getHours() <= 40) // no overtime
46         return getWage() * getHours();
47     else
48         return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
49 }
50
51 // return String representation of HourlyEmployee object
52 @Override
53 public String toString() {
54     return String.format("hourly employee: %s\n%s: $%,.2f; %s: %%,.2f",
55         super.toString(), "hourly wage", getWage(),
56         "hours worked", getHours());
57 }
58 } // end class HourlyEmployee
```

The `BasePlusCommissionEmployee` class

- Extends `CommissionEmployee`. Defines a new field `baseSalary`, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.

```
1 // BasePlusCommissionEmployee class extends CommissionEmployee.
2 public class BasePlusCommissionEmployee extends CommissionEmployee {
3     private double baseSalary; // base salary per week
4
5     // constructor
6     public BasePlusCommissionEmployee(String firstName, String lastName,
7         String socialSecurityNumber, double grossSales,
8         double commissionRate, double baseSalary) {
9         super(firstName, lastName, socialSecurityNumber,
10            grossSales, commissionRate);
11
12         if (baseSalary < 0.0) // validate baseSalary
13             throw new IllegalArgumentException("Base salary must be >= 0.0");
```



The BasePlusCommissionEmployee class

```
14     this.baseSalary = baseSalary;
15 }
16
17 // set base salary
18 public void setBaseSalary(double baseSalary) {
19     if (baseSalary < 0.0) // validate baseSalary
20         throw new IllegalArgumentException("Base salary must be >= 0.0");
21
22     this.baseSalary = baseSalary;
23 }
24
25 // return base salary
26 public double getBaseSalary() { return baseSalary; }
```



The BasePlusCommissionEmployee class

```
27 // calculate earnings; override method earnings in CommissionEmployee
28 @Override
29 public double earnings() { return getBaseSalary() + super.earnings(); }
30
31 // return String representation of BasePlusCommissionEmployee object
32 @Override
33 public String toString() {
34     return String.format("%s %s; %s: $%,.2f",
35         "base-salaried", super.toString(),
36         "base salary", getBaseSalary());
37 }
38 } // end class BasePlusCommissionEmployee
```



Assignments between superclass and subclass variables

```
1 Employee e = new Employee();
```

```
1 CommissionEmployee e = new CommissionEmployee();
```

```
1 Employee e = new CommissionEmployee();
```

- Which version of method is actually called via `e.toString()`?
 - In this particular case, `CommissionEmployee`'s version is called.



Variable's Type and Object's Type

- Variable is the holder, object is the content
- In our example,

```
1 Employee e = new CommissionEmployee();
```

- `e` is a variable of `Employee`'s type, but it is carrying a `CommissionEmployee` object's reference.
- As `Employee` is the superclass of `CommissionEmployee`, this assignment is safe.
- But for other way round, you need *downcasting* to avoid compilation error.



Putting things together

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



Putting things together

```
16 System.out.println("Employees processed individually:");
17
18 System.out.printf("%n%s%n%s: $%,.2f%n%n",
19     salariedEmployee, "earned", salariedEmployee.earnings());
20 System.out.printf("%s%n%s: $%,.2f%n%n",
21     hourlyEmployee, "earned", hourlyEmployee.earnings());
22 System.out.printf("%s%n%s: $%,.2f%n%n",
23     commissionEmployee, "earned", commissionEmployee.earnings());
24 System.out.printf("%s%n%s: $%,.2f%n%n",
25     basePlusCommissionEmployee, "earned",
26     basePlusCommissionEmployee.earnings());
27
28 // create four-element Employee array
29 Employee[] employees = new Employee[4];
```



Putting things together

```
30 // initialize array with Employees
31 employees[0] = salariedEmployee;
32 employees[1] = hourlyEmployee;
33 employees[2] = commissionEmployee;
34 employees[3] = basePlusCommissionEmployee;
35
36 System.out.printf("Employees processed polymorphically:%n%n");
37
38 // generically process each element in array employees
39 for (Employee currentEmployee : employees) {
40     System.out.println(currentEmployee); // invokes toString
```



Putting things together

```
41 // determine whether element is a BasePlusCommissionEmployee
42 if currentEmployee instanceof BasePlusCommissionEmployee() {
43     // downcast Employee reference to
44     // BasePlusCommissionEmployee reference
45     BasePlusCommissionEmployee employee =
46         (BasePlusCommissionEmployee) currentEmployee;
47
48     employee.setBaseSalary(1.10 * employee.getBaseSalary());
49
50     System.out.printf(
51         "new base salary with 10%% increase is: $%,.2f%n",
52         employee.getBaseSalary());
53 } // end if
54
55 System.out.printf("earned $%,.2f%n%n", currentEmployee.earnings());
56 } // end for
```



Putting things together

```
57     // get type name of each object in employees array
58     for (int j = 0; j < employees.length; j++)
59         System.out.printf("Employee %d is a %s%n", j,
60             employees[j].getClass().getName());
61     } // end main
62 } // end class PayrollSystemTest
```



Putting things together

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



Putting things together

```
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:

```
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
```



Putting things together

```
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
```



Assignments between superclass and subclass variables

- Assigning a superclass object's reference to a superclass variable is natural.
- Assigning a subclass object's reference to a subclass variable is natural.
- Assigning a subclass object's reference to a superclass variable is safe, because the subclass object is also an object of its superclass (Java objects are polymorphic).
 - The superclass variable can be used to refer only to superclass members.
 - If a program refers to subclass-only members through the superclass variable, the compiler reports errors.



Assignments between superclass and subclass variables

- Attempting to assign a superclass object's reference to a subclass variable is a compilation error.
- To avoid this error, the superclass object's reference must be cast to a subclass type explicitly.
- At execution time, if the object to which the reference refers is not a subclass object, an exception will occur.
- Use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.

