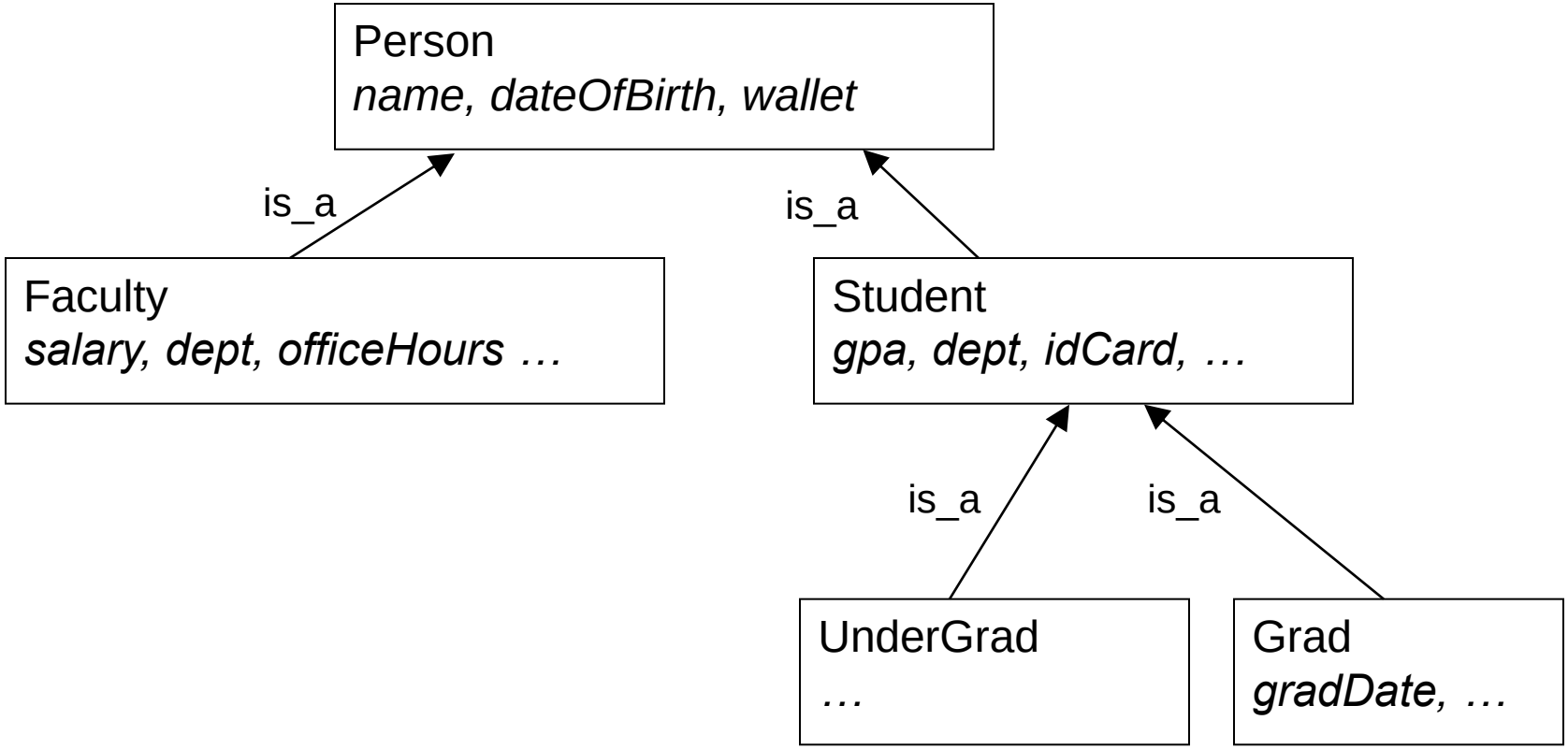
CHAPTER 12

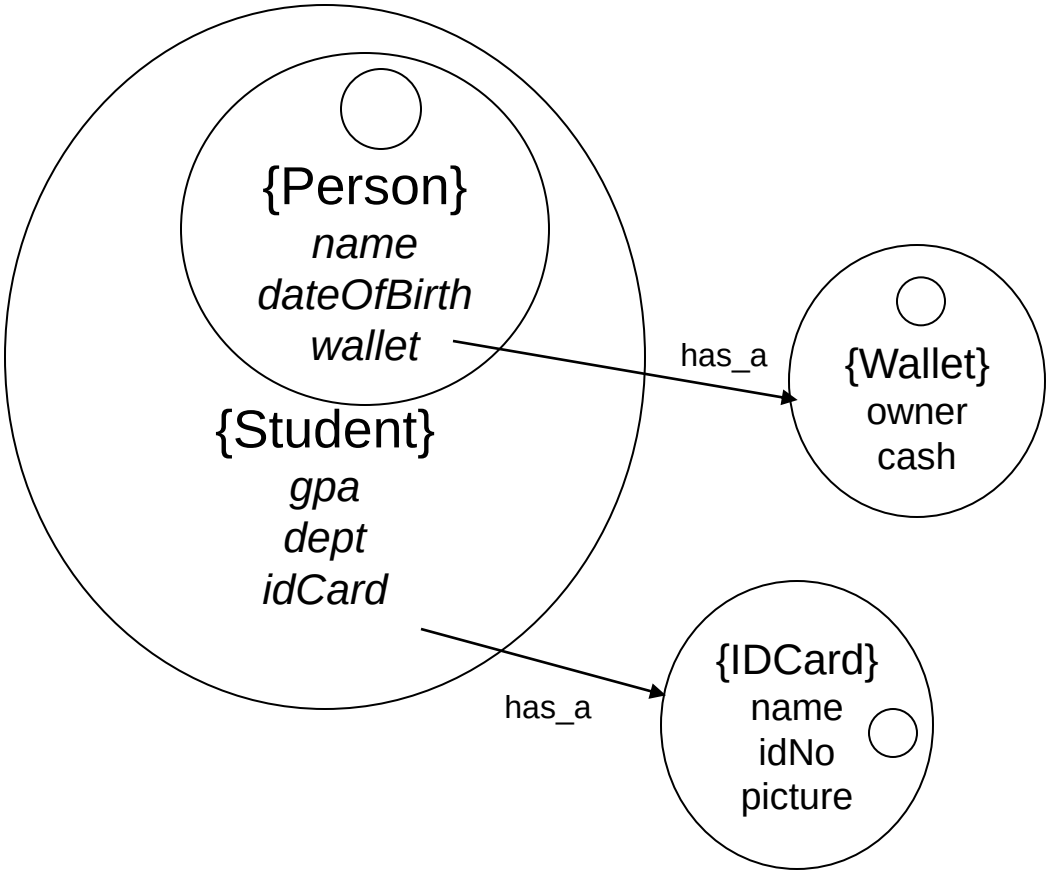
OBJECT-ORIENTED DESIGN

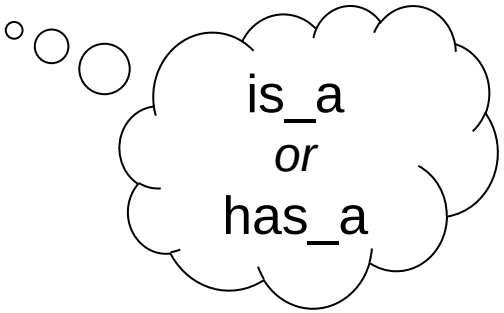
Slides by Rick Giles

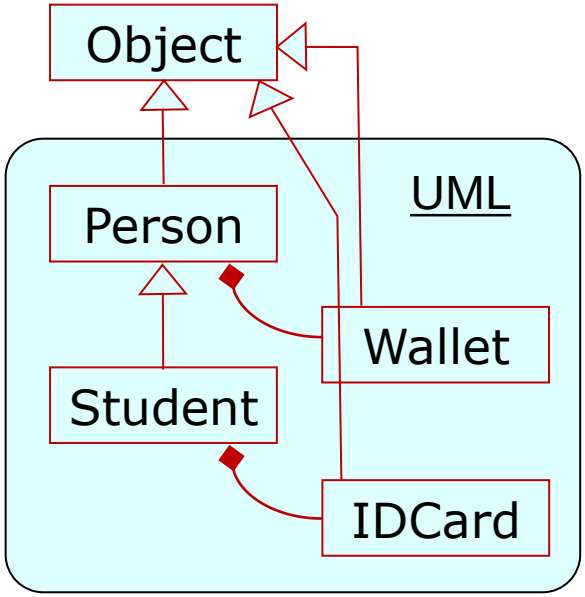
Inheritance Hierarchy



Inheritance & Composition







Example Java Code

```
public class Person {

        String  name;
        Date    dateOfBirth;
        Wallet  wallet;

        public Person ( String name, Date dob) {
              this.name = name;
              dateOfBirth = dob;
              wallet = null;
        }

        public String getName() {
              return name;
        }
}
```

Example Java Code

```
public class Student extends Person {

        double  gpa;
        String   dept;
        IDCard  id;

        public Student ( String name, Date dob,
                                IDCard id, String dept) {
            super( name, dob);
            this.id = id;
            this.dept = dept;
            gpa = 0;
        }

        public double getGpa() {
            return gpa;
        }
}
```

Student is_a Person

With additional properties

Call parent constructor to save writing code again!

Additional methods for new properties

Have direct access to non-private properties & methods of parent classes

Sub-class Constructors

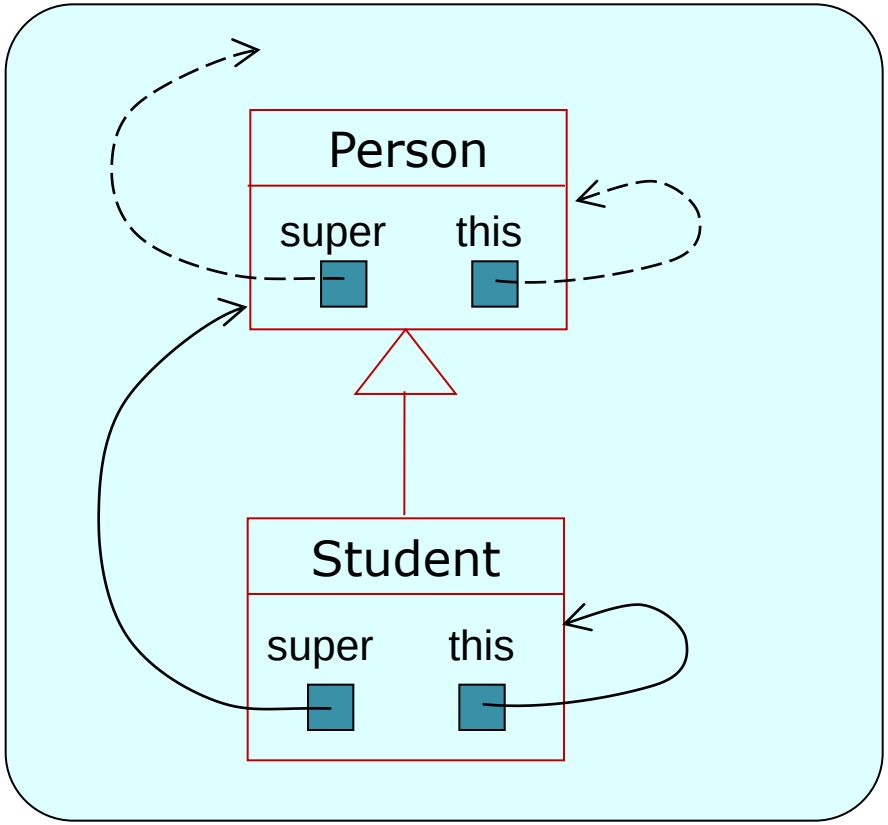
- Call to parent (super) constructor must be first statement in constructor (Java builds instances in layers, inside to out)

- If omitted Java automatically calls defaults constructor (one with no param's)

Note: Java classes can only have one parent.

Java is a *single-inheritance* language, as opposed to a *multiple-inheritance* language (such as C++) which can have many parents!

super vs. this



super refers to non-private constructors, properties & methods in parent class

this refers to properties & methods in current class

Extended Type Checking

- Can now match object of type or sub-type
- Distinguish type of reference vs. type of object

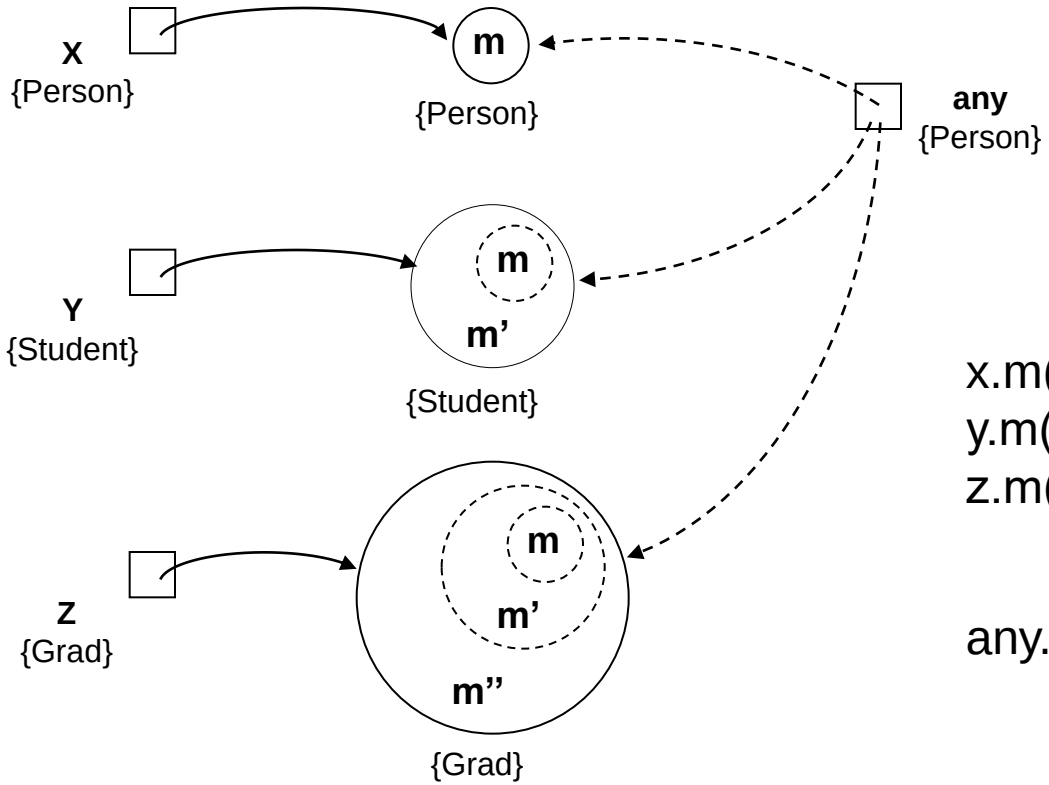
```
Object    x;    // can hold anything
Person    y;    // can hold Person, Student, …
Student   z;    // only Student and sub-classes
```

```
y = new Person( _____);
z = new Student( ____);

y.getName();

z.getName();
```

```
z.getGPA();

y.getGPA();

( (Student) y ).getGPA();
```

Polymorphism

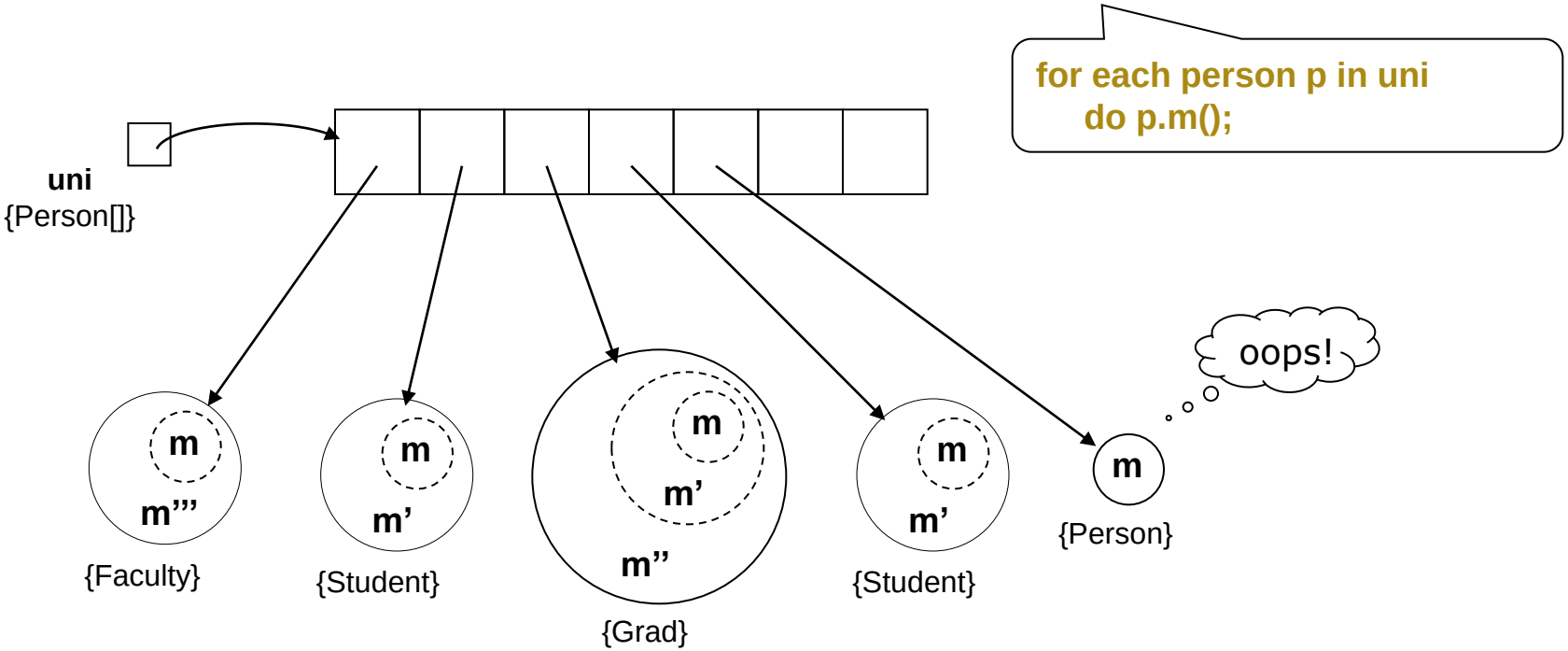
- Everyone does *it* their way!




```
x.m();  // does m
y.m();  // does m'
z.m();  // does m''

any.m();  // depends on
                object type!
```

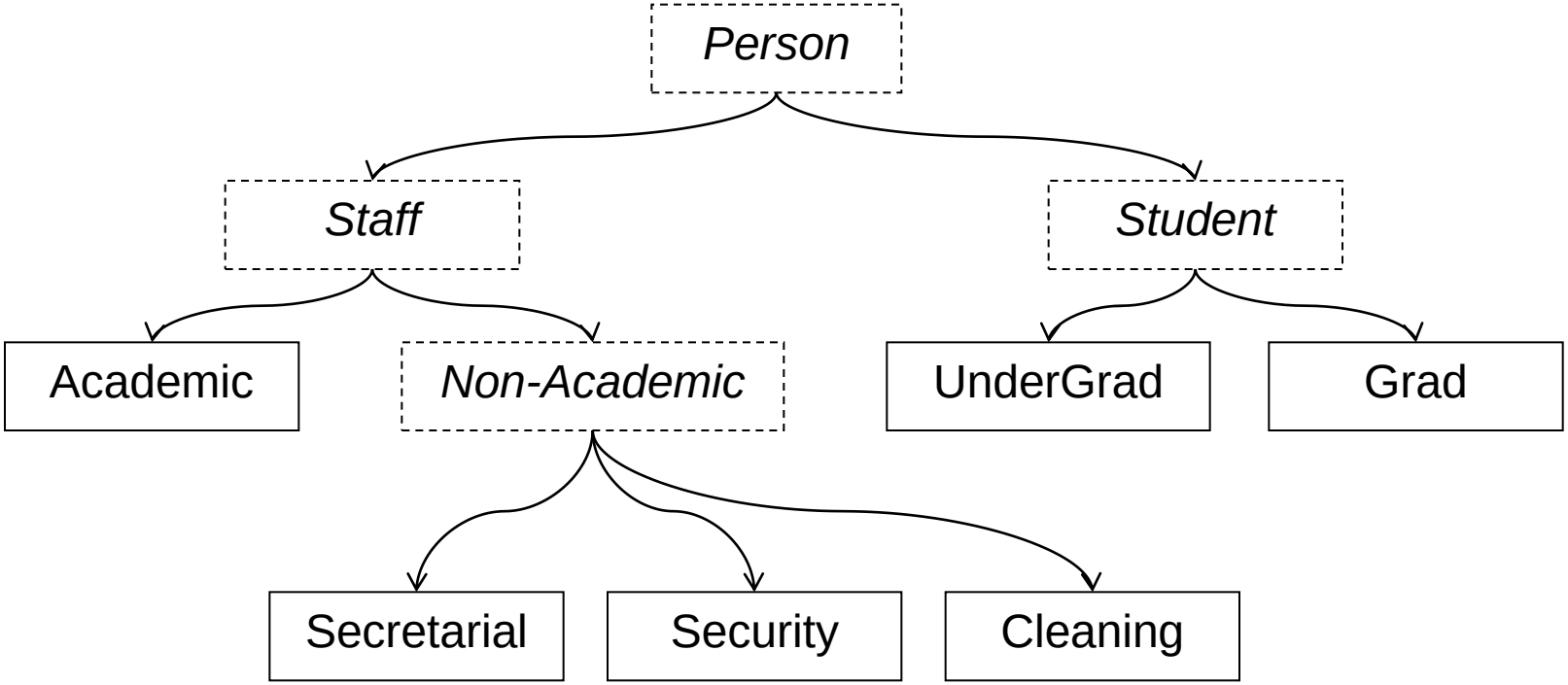
Polymorphic collections

- Objects in a collection all do *it* their way!



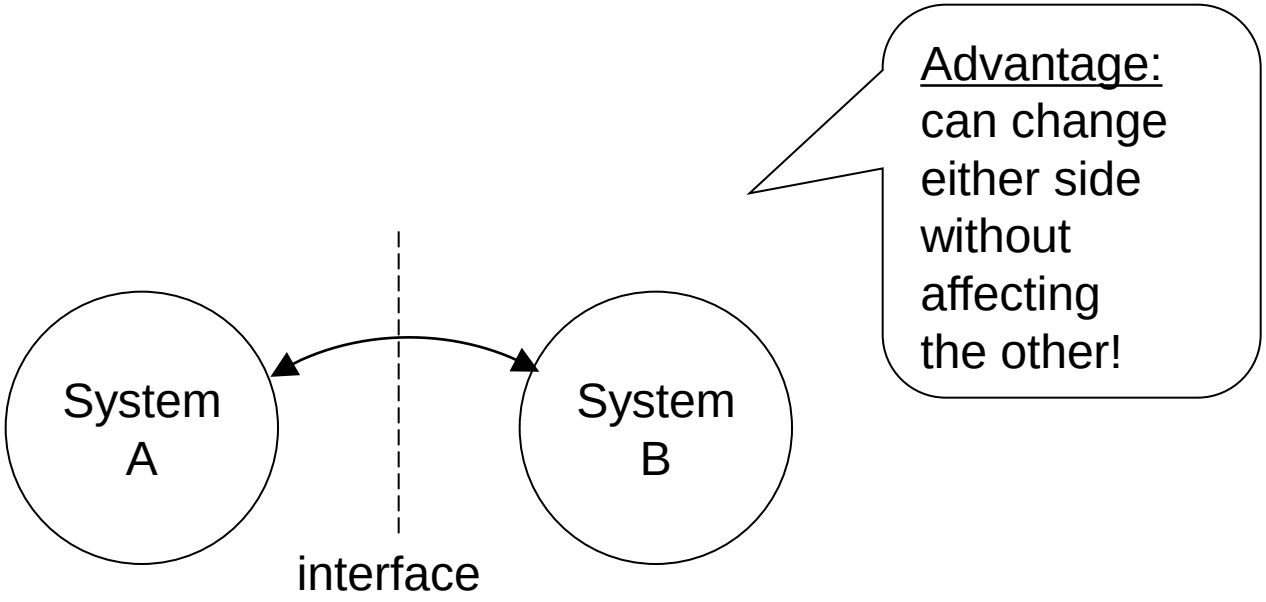
University People…

- Abstract vs. Concrete classes



Interfaces

- An interface
  is the boundary between two systems
  through which they connect/communicate
- Often standardised



Java Interfaces

- Declare with *public interface X {…}*
  - restricted to constants & abstract methods only
  - cannot be instantiated
  - guarantees any *implementing* class
    has specified methods (else cannot be instantiated)

- Classes extend one class & implement interfaces

  e.g. *public class U extends V implements X,Y,Z {…}*

- Can view as a special form of class

  so *class U is_a V, is_a X, is_a Y, is_a Z*

  i.e. a form of multiple inheritance

Simple example…

```
public interface Pointable {
   public boolean contains( int x, int y);
}

public class Circle extends TwoDShape implements Pointable {

   int   radius;

   public Circle( int radius) {
      super();
      this.radius = radius;
   }

   public int getRadius() {
      return radius;
   }

   public boolean contains( int x, int y) {
      // set result true iff x,y inside circle...
      return result;
   }
}
```

Define *interface* with abstract method

Define class that *implements* interface

Required method... won't compile if omitted

Another example…

- Multiple inheritance?

Notebook
- *is_a* Computer
- *is_a* ElectricalDevice
- *is_a* NetworkedDevice

ElectricCooker
- *is_a* Cooker
- *is_a* ElectricalDevice

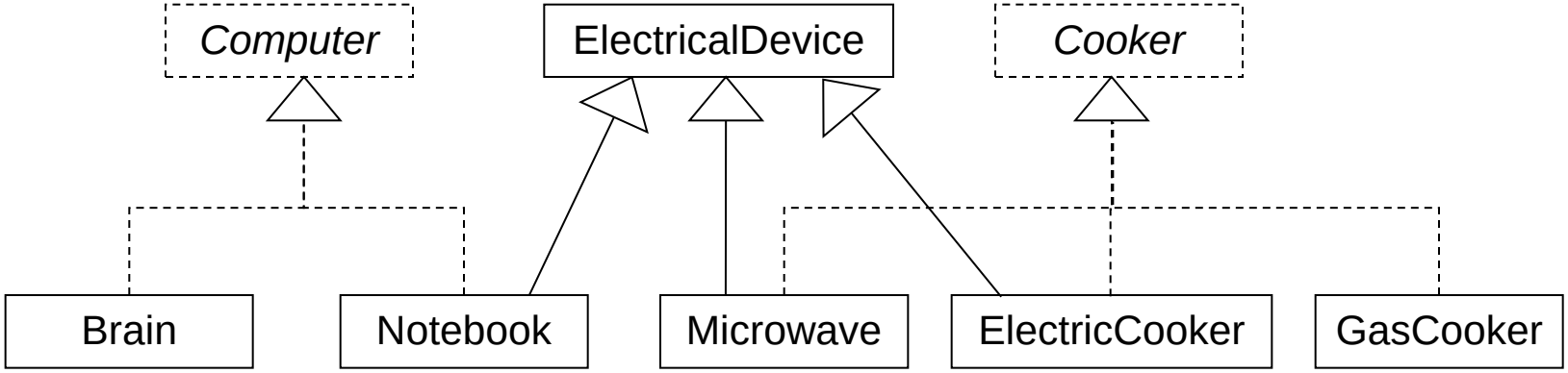
Microwave
- *is_a* Cooker
- *is_a* ElectricalDevice

SmartPhone
- *is_a* Phone
- *is_a* Computer
- *is_a* NetworkedDevice

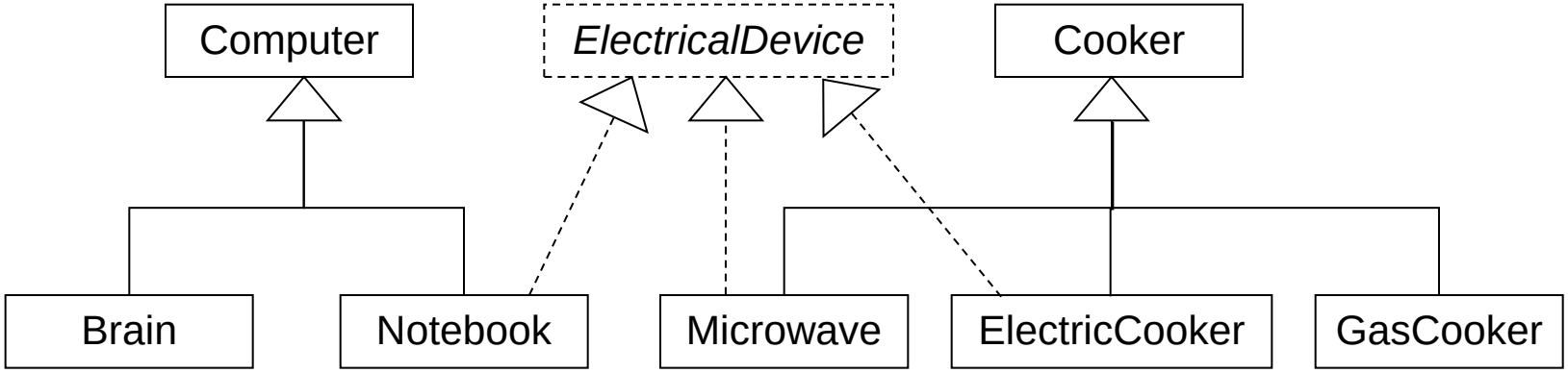
Brain
- *is_a* Computer
- *is_a* BiologicalDevice

GasCooker
- *is_a* Cooker
- *is_a* GasDevice

One soln with interfaces

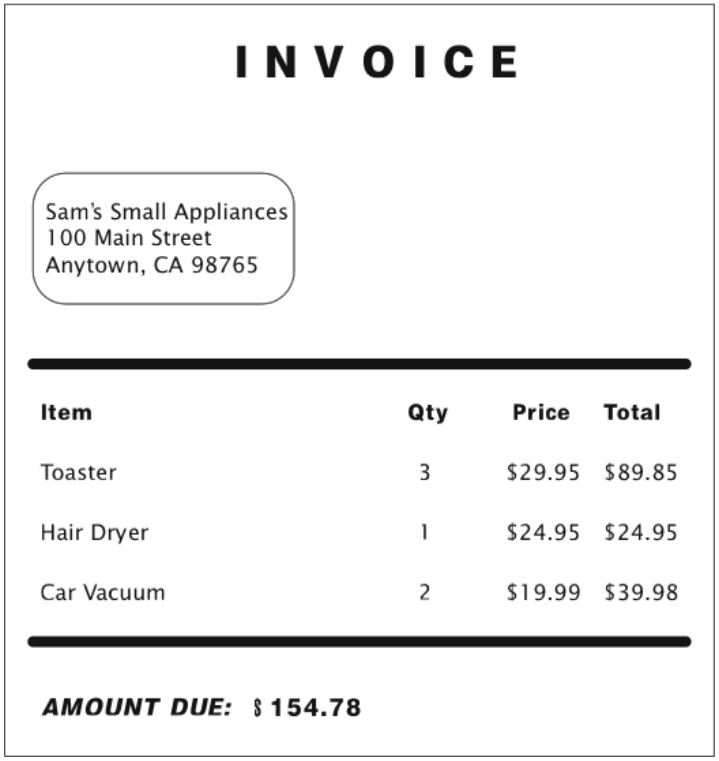


Another soln with interfaces



12.1 Classes and Their Responsibilities (1)

- To discover classes, look for nouns in the problem description
  - Example: Print an invoice
  - Candidate classes:
    - `Invoice`
    - `LineItem`
    - `Customer`



INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Item	Qty	Price	Total
Toaster	3	$29.95	$89.85
Hair Dryer	1	$24.95	$24.95
Car Vacuum	2	$19.99	$39.98

AMOUNT DUE: **$ 154.78**

12.2 Relationships Between Classes

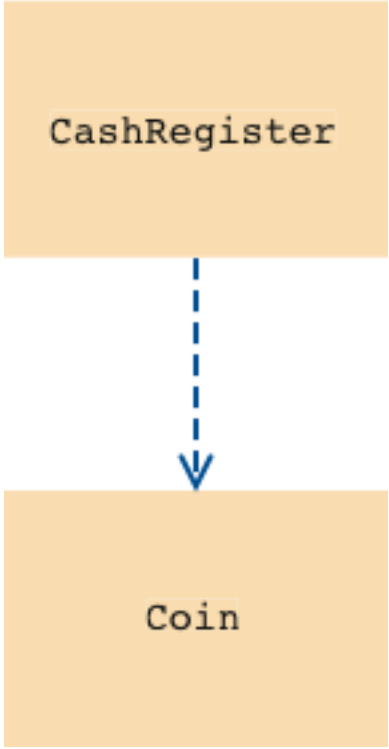
- A class **depends** on another if it uses objects of that class
  - "knows about" relationship
- `CashRegister` depends on `Coin` to determine the value of the payment
- To visualize relationships, draw class diagrams
- **UML**: Unified Modeling Language
  - Notation for object-oriented analysis and design

Dependency Relationship

Dependency (references)
It means there is no conceptual link between two objects.

- For ex: CashRegister object references Coin object (as method parameters or return types)

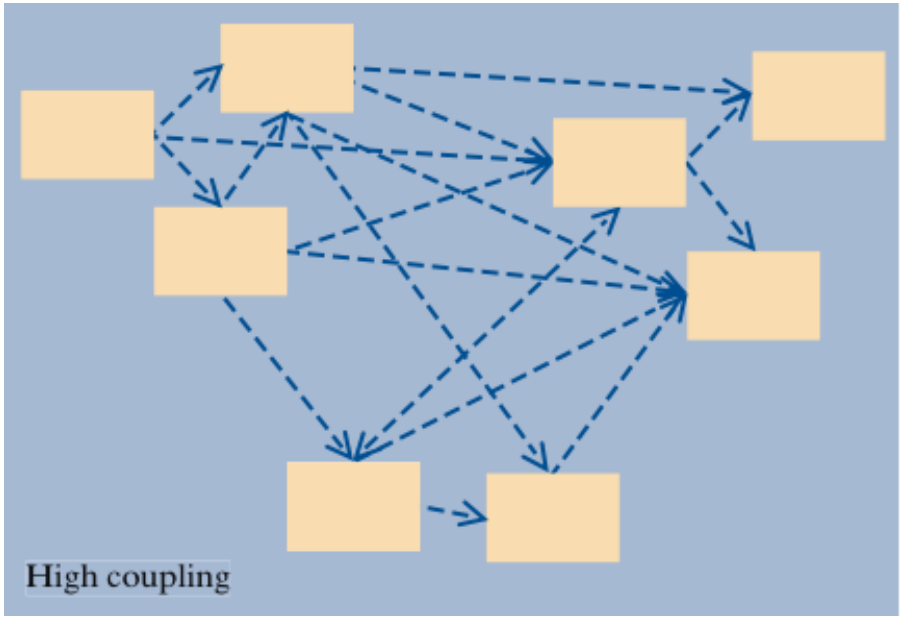
```
public class CashRegister{

        public void deposit(Coin c){}

}
```

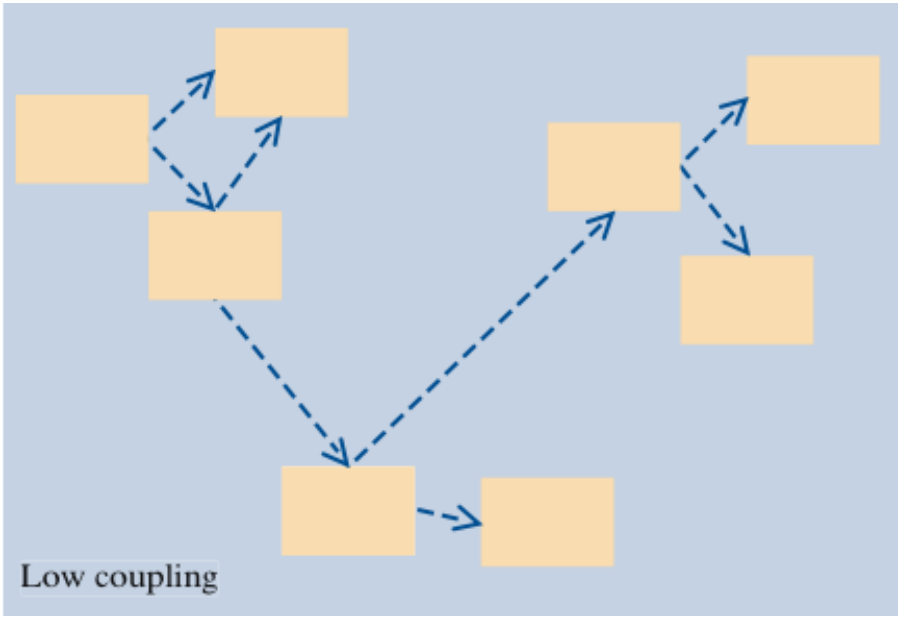


Coupling (1)

- If many classes depend on each other, the **coupling** between classes is high
- Good practice: minimize coupling between classes
  - Change in one class may require update of all coupled classes
  - Using a class in another program requires using all classes on which it depends

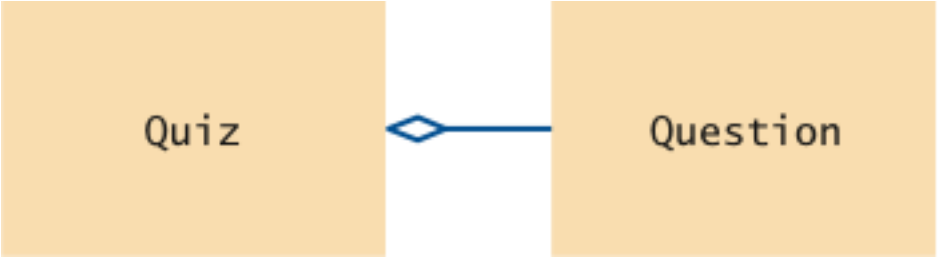
Coupling (2)





Aggregation (1)

- A class **aggregates** another of its objects contain objects of another class
  - "has-a" relationship
- Example: a quiz is made up of questions
  - Class `Quiz` aggregates class `Question`



Aggregation (2)

- Finding out about aggregation helps in implementing classes
- Example: since a quiz can have any number of questions, use an array or array list for collecting them

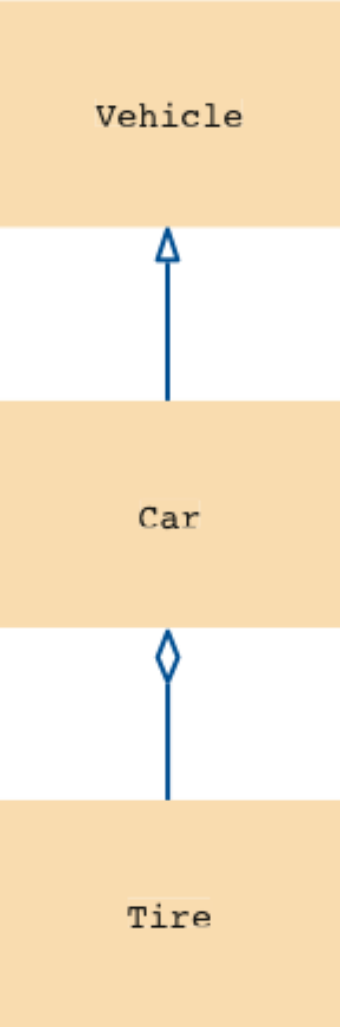
```
public class Quiz
{
   private ArrayList<Question> questions;
   . . .
}
```

Inheritance (1)

- **Inheritance** is the relationship between a more general class (**superclass**) and a more specialized class (**subclass**)
  - "is-a" relationship
- Example: every car *is a* vehicle; every car has tires
  - Class `Car` is a subclass of class `Vehicle`; class car aggregates class `Tire`

Inheritance (2)

```
public class Car extends Vehicle
{
   private Tire[] tires;
   . . .
}
```

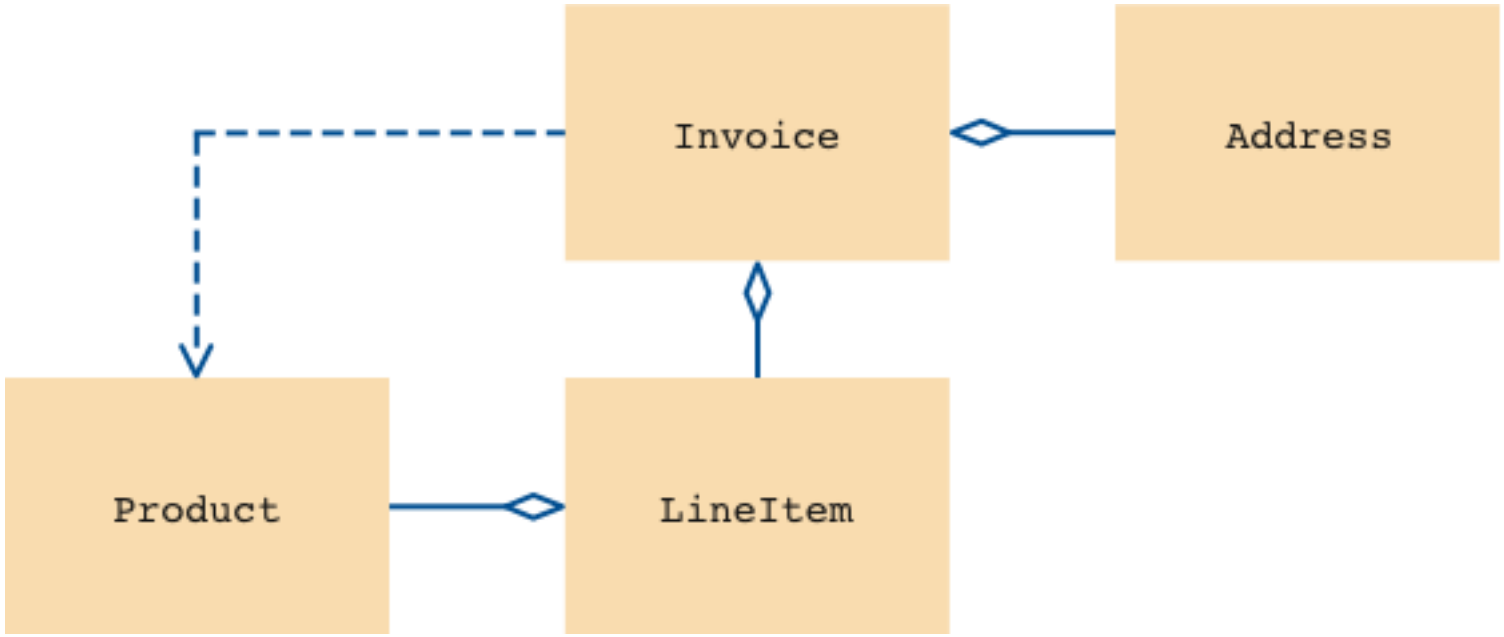


UML Relationship Symbols

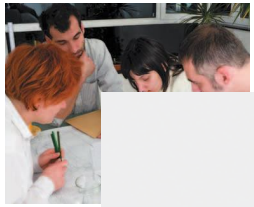
Table 1 UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance	——▷	Solid	Triangle
Interface Implementation	- - - - ▷	Dotted	Triangle
Aggregation	◇——	Solid	Diamond
Dependency	- - - - >	Dotted	Open

UML Class Diagram



Polymorphism Example



	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 *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.

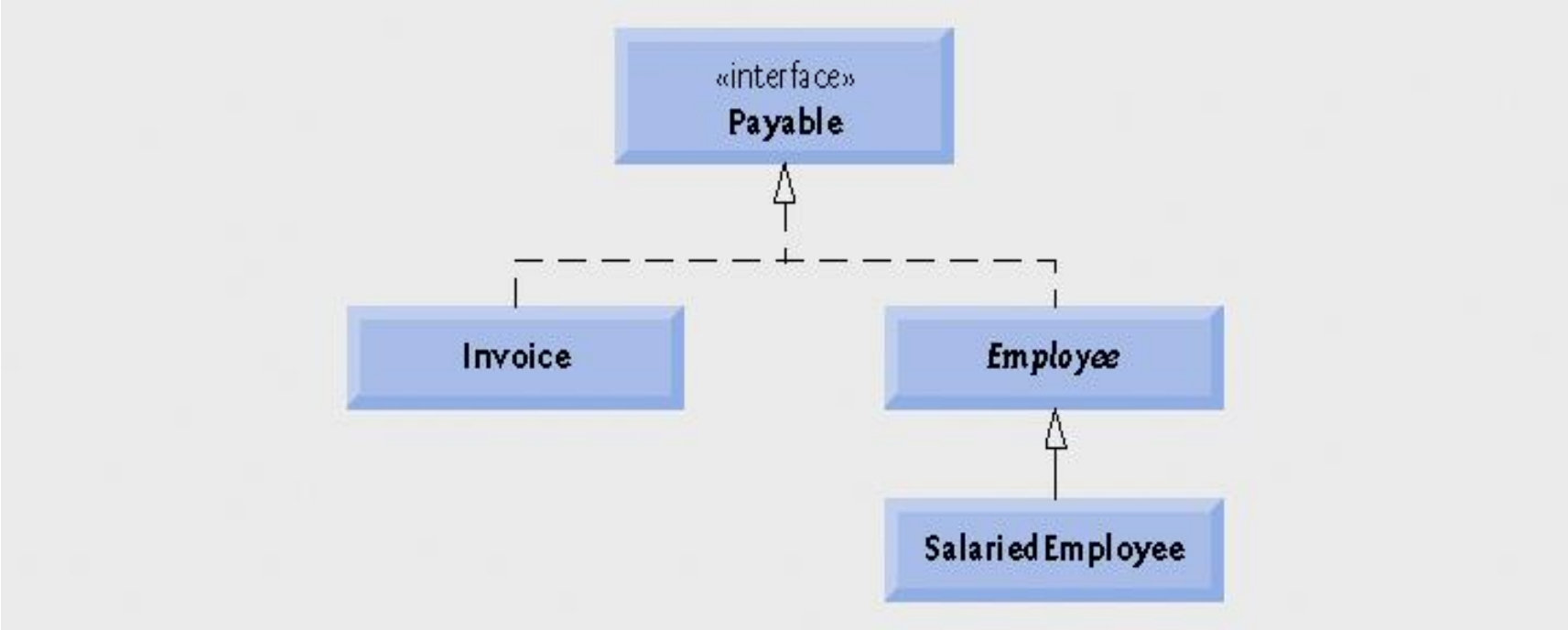


Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

Outline

- Payable.java
-

```
// Fig. 10.11: Payable.java
// Payable interface declaration.

public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Declare interface **`Payable`**

Declare **`getPaymentAmount`** method which is implicitly **`public`** and **`abstract`**

Outline

- Invoice.java
  - (1 of 3)

```java
// Fig. 10.12: Invoice.java
// Invoice class implements Payable.

public class Invoice implements Payable
{
   private String partNumber;
   private String partDescription;
   private int quantity;
   private double pricePerItem;

   // four-argument constructor
   public Invoice( String part, String description, int count,
      double price )
   {
      partNumber = part;
      partDescription = description;
      setQuantity( count ); // validate and store quantity
      setPricePerItem( price ); // validate and store price per item
   } // end four-argument Invoice constructor

   // set part number
   public void setPartNumber( String part )
   {
      partNumber = part;
   } // end method setPartNumber

```

Class **Invoice** implements interface **Payable**

Outline

- Invoice.java
  - (2 of 3)

```
   // get part number
   public String getPartNumber()
   {
      return partNumber;
   } // end method getPartNumber

   // set description
   public void setPartDescription( String description )
   {
      partDescription = description;
   } // end method setPartDescription

   // get description
   public String getPartDescription()
   {
      return partDescription;
   } // end method getPartDescription

   // set quantity
   public void setQuantity( int count )
   {
      quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
   } // end method setQuantity

   // get quantity
   public int getQuantity()
   {
      return quantity;
   } // end method getQuantity

```

Outline

- Invoice.java
  - (3 of 3)

```java
   // set price per item
   public void setPricePerItem( double price )
   {
      pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
   } // end method setPricePerItem

   // get price per item
   public double getPricePerItem()
   {
      return pricePerItem;
   } // end method getPricePerItem

   // return String representation of Invoice object
   public String toString()
   {
      return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
         "invoice", "part number", getPartNumber(), getPartDescription(),
         "quantity", getQuantity(), "price per item", getPricePerItem() );
   } // end method toString

   // method required to carry out contract with interface Payable
   public double getPaymentAmount()
   {
      return getQuantity() * getPricePerItem(); // calculate total cost
   } // end method getPaymentAmount
} // end class Invoice
```

Declare **`getPaymentAmount`** to fulfill contract with interface **`Payable`**

Outline

- Employee.java
- (1 of 3)

```java
// Fig. 10.13: Employee.java
// Employee abstract superclass implements Payable.

public abstract class Employee implements Payable
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

```

Class **Employee** implements interface **Payable**

```
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

```

Outline

- Employee.java
  - (2 of 3)

Outline

- Employee.java
  - (3 of 3)

```
   // 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

   // return String representation of Employee object
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // Note: We do not implement Payable method getPaymentAmount here so
   // this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

`getPaymentAmount` method is not implemented here

Outline

```java
// Fig. 10.14: SalariedEmployee.java
// SalariedEmployee class extends Employee, which implements Payable.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

```

Class **SalariedEmployee** extends class **Employee** (which implements interface **Payable**)

- SalariedEmployee
- .java

- (1 of 2)

Outline

- SalariedEmployee
- .java

```
   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; implement interface Payable method that was
   // abstract in superclass Employee
   public double getPaymentAmount()
   {
      return getWeeklySalary();
   } // end method getPaymentAmount

   // return String representation of SalariedEmployee object
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Declare **`getPaymentAmount`** method instead of **`earnings`** method

Outline

- PayableInterface Test.java
- (1 of 2)

```java
// Fig. 10.15: PayableInterfaceTest.java
// Tests interface Payable.

public class PayableInterfaceTest
{
   public static void main( String args[] )
   {
      // create four-element Payable array
      Payable payableObjects[] = new Payable[ 4 ];

      // populate array with objects that implement Payable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Invoices and Employees processed polymorphically:\n" );

```

Declare array of **`Payable`** variables

Assigning references to **`Invoice`** objects to **`Payable`** variables

Assigning references to **`SalariedEmployee`** objects to **`Payable`** variables

Outline

- PayableInterface
- Test.java

```java
      // generically process each element in array payableObjects
      for ( Payable currentPayable : payableObjects )
      {
         // output currentPayable and its appropriate payment amount
         System.out.printf( "%s \n%s: $%,.2f\n\n",
            currentPayable.toString(),
            "payment due", currentPayable.getPaymentAmount() );
      } // end for
   } // end main
} // end class PayableInterfaceTest
```

Call **`toString`** and **`getPaymentAmount`** methods polymorphically

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

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

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

10.7.1 Developing a `Payable` Hierarchy

- `Payable` interface
  - Contains method `getPaymentAmount`
  - Is implemented by the `Invoice` and `Employee` classes
- UML representation of interfaces
  - Interfaces are distinguished from classes by placing the word “interface” in guillemets (« and ») above the interface name
  - The relationship between a class and an interface is known as realization
    - A class “realizes” the methods of an interface

Outline

- Employee.java
- (1 of 3)

```java
// Fig. 10.4: Employee.java
// Employee abstract superclass.

public abstract class Employee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

```

Declare **`abstract`** class **`Employee`**

Attributes common to all employees

Outline

- Employee.java
  - (2 of 3)

```
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

```

Outline

- Employee.java
  - (3 of 3)

```
   // 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

   // return String representation of Employee object
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // abstract method overridden by subclasses
   public abstract double earnings(); // no implementation here
} // end abstract class Employee
```

abstract method **earnings** has no implementation

Outline

- SalariedEmployee
- .java

- (1 of 2)

```java
// Fig. 10.5: SalariedEmployee.java
// SalariedEmployee class extends Employee.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); //
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

```

Class **SalariedEmployee** extends class **Employee**

Call superclass constructor

Call **setWeeklySalary** method

Validate and set weekly salary value

Outline

- SalariedEmployee
- .java

(2 of 2)

```
22    // return salary
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // end method getWeeklySalary
27
28    // calculate earnings; override abstract method earnings in Employee
29    public double earnings()
30    {
31       return getWeeklySalary();
32    } // end method earnings
33
34    // return String representation of S
35    public String toString()
36    {
37       return String.format( "salaried employe
38          super.toString(), "weekly salary", getWeeklySalary() );
39    } // end method toString
40 } // end class SalariedEmployee
```

Override **earnings** method so **SalariedEmployee** can be concrete

Override **toString** method

Call superclass's version of **toString**

Outline

- HourlyEmployee
- .java

- (1 of 2)

```java
// Fig. 10.6: HourlyEmployee.java
// HourlyEmployee class extends Employee.

public class HourlyEmployee extends Employee
{
   private double wage; // wage per hour
   private double hours; // hours worked for week

   // five-argument constructor
   public HourlyEmployee( String first, String last, String ssn,
      double hourlyWage, double hoursWorked )
   {
      super( first, last, ssn );
      setWage( hourlyWage ); // validate hourly wage
      setHours( hoursWorked ); // validate hours worked
   } // end five-argument HourlyEmployee constructor

   // set wage
   public void setWage( double hourlyWage )
   {
      wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
   } // end method setWage

   // return wage
   public double getWage()
   {
      return wage;
   } // end method getWage

```

Class **HourlyEmployee** extends class **Employee**

Call superclass constructor

Validate and set hourly wage value

Outline

- HourlyEmployee
- .java
- (2 of 2)

```
30    // set hours worked
31    public void setHours( double hoursWorked )
32    {
33       hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34          hoursWorked : 0.0;
35    } // end method setHours
36
37    // return hours worked
38    public double getHours()
39    {
40       return hours;
41    } // end method getHours
42
43    // calculate earnings; override abstract method earnings in Employee
44    public double earnings()
45    {
46       if ( getHours() <= 40 ) // no overtime
47          return getWage() * getHours();
48       else
49          return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
50    } // end method earnings
51
52    // return String representation of HourlyEmployee object
53    public String toString()
54    {
55       return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
56          super.toString(), "hourly wage", getWage(),
57          "hours worked", getHours() );
58    } // end method toString
59 } // end class HourlyEmployee
```

Validate and set hours worked value

Override **earnings** method so **HourlyEmployee** can be concrete

Override **toString** method

Call superclass's **toString** method

Outline

```java
// Fig. 10.7: CommissionEmployee.java
// CommissionEmployee class extends Employee.

public class CommissionEmployee extends Employee
{
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      super( first, last, ssn );
      setGrossSales( sales );
      setCommissionRate( rate );
   } // end five-argument CommissionEmployee constructor

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

```

Class **CommissionEmployee** extends class **Employee**

Call superclass constructor

Validate and set commission rate value

- CommissionEmployee.java
- (1 of 3)

Outline

- CommissionEmployee.java
  - (2 of 3)

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

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

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

```

Validate and set the gross sales value

Outline

```
   // calculate earnings; override abstract method earnings in Employee
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

   // return String representation of CommissionEmployee object
   public String toString()
   {
      return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
         "commission employee", super.toString(),
         "gross sales", getGrossSales(),
         "commission rate", getCommissionRate() );
   } // end method toString
} // end class CommissionEmployee
```

Override **`earnings`** method so **`CommissionEmployee`** can be concrete

- CommissionEmployee.java

Override **`toString`** method

Call superclass's **`toString`** method

Outline

```
// Fig. 10.8: BasePlusCommissionEmp
// BasePlusCommissionEmployee class

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

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
   } // end method setBaseSalary

```

Class **BasePlusCommissionEmployee** extends class **CommissionEmployee**

Call superclass constructor

Validate and set base salary value

- BasePlusCommissionEmployee.java
- (1 of 2)

Outline

```
22    // return base salary
23    public double getBaseSalary()
24    {
25       return baseSalary;
26    } // end method getBaseSalary
27
28    // calculate earnings; override method earnings in CommissionEmployee
29    public double earnings()
30    {
31       return getBaseSalary() + super.earnings();
32    } // end method earnings
33
34    // return String representation of BasePlusCommissionEmployee object
35    public String toString()
36    {
37       return String.format( "%s %s; %s: $%,.2f",
38          "base-salaried", super.toString(),
39          "base salary", getBaseSalary() );
40    } // end method toString
41 } // end class BasePlusCommissionEmployee
```

Override **earnings** method

Call superclass's **earnings** method

Override **toString** method

Call superclass's **toString** method

- BasePlusCommissionEmployee.java
- (2 of 2)

Outline

- PayrollSystemTest
- .java

- (1 of 5)

```java
// Fig. 10.9: PayrollSystemTest.java
// Employee hierarchy test program.

public class PayrollSystemTest
{
   public static void main( String args[] )
   {
      // create subclass objects
      SalariedEmployee salariedEmployee =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      HourlyEmployee hourlyEmployee =
         new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
      CommissionEmployee commissionEmployee =
         new CommissionEmployee(
         "Sue", "Jones", "333-33-3333", 10000, .06 );
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

      System.out.println( "Employees processed individually:\n" );

```

Outline

- PayrollSystemTest
- .java

- (2 of 5)

```
22      System.out.printf( "%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.println( "Employees processed polymorphically:\n" );
42
43      // generically process each element in array employees
44      for ( Employee currentEmployee : employees )
45      {
46         System.out.println( currentEmployee ); // invokes toString
47
```

Assigning subclass objects to supercalss variables

Implicitly and polymorphically call **toString**

Outline

```
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 ) curren
55
56             double oldBaseSalary = employee.getBaseSalary();
57             employee.setBaseSalary( 1.10 * oldBaseSalary );
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.earn
65       } // end for
66
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
```

If the **currentEmployee** variable points to a **BasePlusCommissionEmployee** obje

Downcast **currentEmployee** to a **BasePlusCommissionEmployee** reference

Give **BasePlusCommissionEmployee**s a 10% base salary bonus

Polymorphically call **earnings** method

Call **getClass** and **getName** methods to display each **Employee** subclass object's class name

Outline

```
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
```

- PayrollSystemTest
- .java

- (4 of 5)

Outline

- PayrollSystemTest
- .java

(5 of 5)

```
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

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
```

Same results as when the employees were processed individually

Base salary is increased by 10%

Each employee's type is displayed

Method Documentation (1)

```
/**
   Describes an invoice for a set of purchased products.
*/
public class Invoice
{
   /**
      Adds a charge for a product to this invoice.
      @param aProduct the product that the customer ordered
      @param quantity the quantity of the product
   */
   public void add(Product aProduct, int quantity)
   {
   }

   /**
      Formats the invoice.
      @return the formatted invoice
   */
   public String format()
   {
   }
}
```

Method Documentation (2)

```
/**
   Describes a quantity of an article to purchase.
*/
public class LineItem
{
   /**
      Computes the total cost of this line item.
      @return the total price
   */
   public double getTotalPrice()
   {
   }

   /**
      Formats this item.
      @return a formatted string of this item
   */
   public String format()
   {
   }
}
```

Method Documentation (3)

```
/**
   Describes a product with a description and a price.
*/
public class Product
{
   /**
      Gets the product description.
      @return the description
   */
   public String getDescription()
   {
   }

   /**
      Gets the product price.
      @return the unit price
   */
   public double getPrice()
   {
   }
}
```

Method Documentation (4)

```
/**
   Describes a mailing address.
*/
public class Address
{
   /**
      Formats the address.
      @return the address as a string with three lines
   */
   public String format()
   {
   }
}
```

Class Documentation in HTML Format

Invoice - Firefox

File Edit View History Bookmarks Tools Help

file:///home/cay/BigJava/ch12/invoice/index.html

All Classes

Address
Invoice
InvoicePrinter
LineItem
Product

Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Invoice

```
java.lang.Object
  └Invoice
```

```
public class Invoice
extends java.lang.Object
```

Describes an invoice for a set of purchased products.

Constructor Summary

	Invoice(Address anAddress) Constructs an invoice.

Method Summary

void	**add**(Product aProduct, int quantity) Adds a charge for a product to this invoice.
java.lang.String	**format**() Formats the invoice.
double	**getAmountDue**() Computes the total amount due.

InvoicePrinter.java

```
/**
   This program demonstrates the invoice classes by printing
   a sample invoice.
*/
public class InvoicePrinter
{
   public static void main(String[] args)
   {
      Address samsAddress
            = new Address("Sam&apos;s Small Appliances",
               "100 Main Street", "Anytown", "CA", "98765");

      Invoice samsInvoice = new Invoice(samsAddress);
      samsInvoice.add(new Product("Toaster", 29.95), 3);
      samsInvoice.add(new Product("Hair dryer", 24.95), 1);
      samsInvoice.add(new Product("Car vacuum", 19.99), 2);

      System.out.println(samsInvoice.format());
   }
}
```

Invoice.java

```
import java.util.ArrayList;

/**
   Describes an invoice for a set of purchased products.
*/
public class Invoice
{
   private Address billingAddress;
   private ArrayList<LineItem> items;

   /**
      Constructs an invoice.
      @param anAddress the billing address
   */
   public Invoice(Address anAddress)
   {
      items = new ArrayList<LineItem>();
      billingAddress = anAddress;
   }

```

Continued

Invoice.java (cont.)

```
21      /**
22          Adds a charge for a product to this invoice.
23          @param aProduct the product that the customer ordered
24          @param quantity the quantity of the product
25      */
26      public void add(Product aProduct, int quantity)
27      {
28          LineItem anItem = new LineItem(aProduct, quantity);
29          items.add(anItem);
30      }
31
```

Continued

Invoice.java (cont.)

```
    /**
       Formats the invoice.
       @return the formatted invoice
    */
    public String format()
    {
       String r =  "                          I N V O I C E\n\n"
             + billingAddress.format()
             + String.format("\n\n%-30s%8s%5s%8s\n",
                "Description", "Price", "Qty", "Total");

       for (LineItem item : items)
       {
          r = r + item.format() + "\n";
       }

       r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());

       return r;
    }

```

Continued

Invoice.java (cont.)

```
    /**
       Computes the total amount due.
       @return the amount due
    */
    public double getAmountDue()
    {
       double amountDue = 0;
       for (LineItem item : items)
       {
          amountDue = amountDue + item.getTotalPrice();
       }
       return amountDue;
    }
}
```

LineItem.java

```
/**
   Describes a quantity of an article to purchase.
*/
public class LineItem
{
   private int quantity;
   private Product theProduct;

   /**
      Constructs an item from the product and quantity.
      @param aProduct the product
      @param aQuantity the item quantity
   */
   public LineItem(Product aProduct, int aQuantity)
   {
      theProduct = aProduct;
      quantity = aQuantity;
   }

```

Continued

LineItem.java (cont.)

```
    /**
       Computes the total cost of this line item.
       @return the total price
    */
    public double getTotalPrice()
    {
       return theProduct.getPrice() * quantity;
    }

    /**
       Formats this item.
       @return a formatted string of this item
    */
    public String format()
    {
       return String.format("%-30s%8.2f%5d%8.2f",
          theProduct.getDescription(), theProduct.getPrice(),
          quantity, getTotalPrice());
    }
}
```

Product.java

```
/**
   Describes a product with a description and a price.
*/
public class Product
{
   private String description;
   private double price;

   /**
      Constructs a product from a description and a price.
      @param aDescription the product description
      @param aPrice the product price
   */
   public Product(String aDescription, double aPrice)
   {
      description = aDescription;
      price = aPrice;
   }

```

Continued

Product.java (cont.)

```
    /**
        Gets the product description.
        @return the description
    */
    public String getDescription()
    {
        return description;
    }

    /**
        Gets the product price.
        @return the unit price
    */
    public double getPrice()
    {
        return price;
    }
}
```

Address.java

```
/**
    Describes a mailing address.
*/
public class Address
{
    private String name;
    private String street;
    private String city;
    private String state;
    private String zip;

    /**
        Constructs a mailing address.
        @param aName the recipient name
        @param aStreet the street
        @param aCity the city
        @param aState the two-letter state code
        @param aZip the ZIP postal code
    */
```

Continued

Address.java (cont.)

```
    public Address(String aName, String aStreet,
          String aCity, String aState, String aZip)
    {
       name = aName;
       street = aStreet;
       city = aCity;
       state = aState;
       zip = aZip;
    }

    /**
       Formats the address.
       @return the address as a string with three lines
    */
    public String format()
    {
       return name + "\n" + street + "\n"
             + city + ", " + state + " " + zip;
    }
}
```

12.4 Packages

- **Package**: a set of related classes
- Important packages in the Java library:

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database Access	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`omg.w3c.dom`	Document Object Model for XML documents	`Document`

Organizing Related Classes into Packages (1)

- To put a class in a package, you must place

```
package packageName;
```

  as the first statement in its source

- Package name consists of one or more identifiers separated by periods

Organizing Related Classes into Packages (2)

- For example, to put the `BankAccount` class into a package named `com.horstmann`, the `BankAccount.java` file must start as follows:

```
package com.horstmann;

public class BankAccount
{
   . . .
}
```

- **Default package** has no name, no `package` statement

Importing Packages

- Can always use class without importing:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Tedious to use fully qualified name
- Import lets you use shorter class name:

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in);
```

- Can import all classes in a package:

```
import java.util.*;
```

- Never need to import classes in package `java.lang`
- Don’t need to import other classes in the same package

Package Names

- Use packages to avoid name clashes

  `java.util.Timer`

  vs.

  `javax.swing.Timer`

- Package names should be unambiguous
- Recommendation: start with reversed domain name:

  `com.horstmann`

- `edu.sjsu.cs.walters`: for Britney Walters' classes (`walters@cs.sjsu.edu`)

How Classes Are Located

- **Base directory:** holds your program's source files
- Path of a class source file, relative to base directory, must match its package name
- Example: if base directory is

  `/home/britney/assignments`

  place source files for classes in package `problem1` in directory

  `/homehome/britney/assignments/problem1`