SOFTWARE DESIGN
COSC 4353/6353

Dr. Raj Singh

OUTLINE

Software Architecture

Object and Class

Object Oriented Principles

OOP Example

Terms to Remember

SOFTWARE ARCHITECTURE

A high level structure of a software system.

Rules, heuristics and patterns for system design.

Partitioning the system into discrete pieces

Techniques

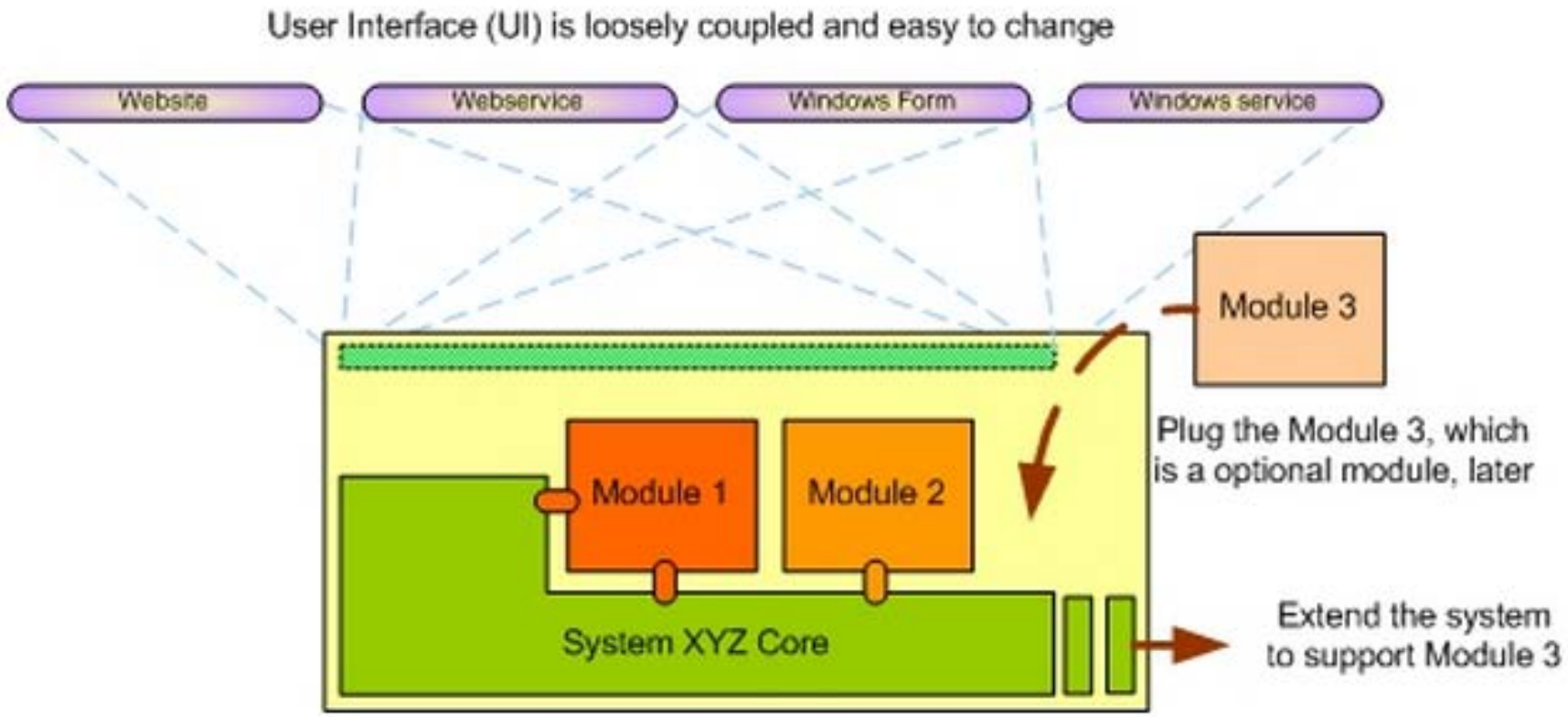
- used to create interfaces between the pieces
- manage overall structure and flow
- interface the system to its environment

Defines appropriate use of development and delivery techniques and tools.

ARCHITECTURE IMPORTANCE

- Controls complexity
- Enforces best practices
- Provides consistency and uniformity
- Increases predictability
- Enables reusability

ARCHITECTURE EXAMPLE



OBJECT-ORIENTED PROGRAMMING / DESIGN (OOP/OOD)

A paradigm that represents concepts as "objects" that have attributes that describe the object and associated procedures known as methods.

Objects, which are usually instances of classes, are used to interact with one another to design applications.

Objective-C, Smalltalk, Java and C# are examples of object-oriented programming languages.

OBJECT AND CLASS

An object can be considered as a "thing" that can perform a set of related activities.

The set of activities that the object performs defines the object's behavior.

In pure OOP terms an object is an instance of a class.

OBJECT AND CLASS

- A class is a representation of a type of object.
- It describe the details of an object.
- Class is composed of three things: name, attributes, and operations.

```
public class Student {
        private String name;
        …
        private void method1(){
        …
        }
        …
}
Student objectStudent = new Student();
```

- student object, named objectStudent, is created out of the Student class.

CLASS DESIGN PRINCIPLE

 SRP - The Single Responsibility Principle	A class should have one, and only one, reason to change.
 OCP - The Open Closed Principle	You should be able to extend a classes behavior, without modifying it.
 LSP - The Liskov Substitution Principle	Derived classes must be substitutable for their base classes.
 DIP - The Dependency Inversion Principle	Depend on abstractions, not on concretions.
 ISP - The Interface Segregation Principle	Make fine grained interfaces that are client specific.

Main OOP/OOD concepts

Encapsulation	information hiding
Abstraction	define, don't implement
Inheritance	extensibility
Polymorphism	one object many shapes

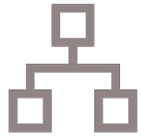
ASSOCIATION, AGGREGATION AND COMPOSITION

Association is a (*a*) relationship between two classes, where one class use another

Aggregation, a special type of an association, is the (*the*) relationship between two classes.

Association is non-directional, aggregation insists a direction.

Composition can be recognized as a special type of an aggregation.



Aggregation is a special kind of an association and composition is a special kind of an aggregation.

Association →Aggregation →Composition

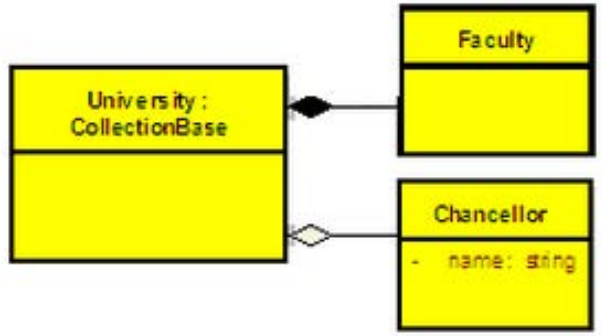
EXAMPLE

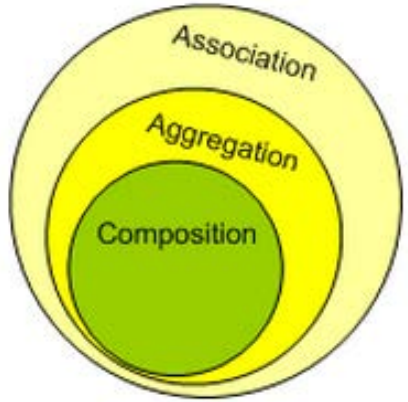
University aggregate Chancellor. University can exist without a Chancellor.

Faculties cannot exist without the University.

The life time of a Faculty is attached with the life time of the University .

University is composed of Faculties.





ABSTRACTION AND GENERALIZATION

Abstraction is an emphasis on the idea, qualities and properties rather than the particulars.

"What" rather than "How"

Generalization is the broadening of application to encompass a larger domain of objects of the same or different type.

Abstraction and generalization are often used together.

INHERITANCE

Software reusability

Reuse an existing class and it's behavior

Create new class from an existing class

Absorb existing class's data and behaviors

Enhance with new capabilities

Subclass extends superclass

More specialized group of objects

Behaviors inherited from superclass

ABSTRACT CLASSES AND METHODS

Classes that are too general to create real objects

Used only as abstract superclasses for concrete subclasses and to declare reference variables

Many inheritance hierarchies have abstract superclasses occupying the top few levels

Keyword abstract

- Use to declare a class abstract
- Also use to declare a method abstract

Abstract classes normally contain one or more abstract methods

All concrete subclasses must override all inherited abstract methods

INTERFACES

- Interfaces are used to separate design from coding as class method headers are specified but not their bodies.
- Interfaces are similar to abstract classes but all methods are abstract and all properties are static final.
- Interfaces can be inherited (i.e.. you can have a sub-interface).
- An interface is used to tie elements of several classes together.
- This allows compilation and parameter consistency testing prior to the coding phase.
- Interfaces are also used to set up unit testing frameworks.

POLYMORPHISM

Facilitates adding new classes to a system with minimal modifications

When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable

The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked

TYPES OF POLYMORPHISM

Overloading

More than one method in a class with same name different signature.

Does not depend on return type.

Overriding

Method in a subclass with same name and return type.

Dynamic Binding

Also known as late binding

Calls to overridden methods are resolved at execution time, based on the type of object referenced

POLYMORPHISM EXAMPLE:
EMPLOYEE HIERARCHY CLASSES

	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*

EMPLOYEE CLASS

```
1  // Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     public Employee( String first, String last){
10       firstName = first;
11       lastName = last;
12    }
13    public Employee( String first, String last, String ssn){
14       firstName = first;
15       lastName = last;
16       socialSecurityNumber = ssn;
17    } // end three-argument Employee constructor
```

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

Attributes common to all employees

Method Overloading

EMPLOYEE CLASS ...

```
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last;
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
```

EMPLOYEE CLASS ...

```
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
48    // return social security number
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // end method getSocialSecurityNumber
53
54    // return String representation of Employee object
55    public String toString()
56    {
57       return String.format( "%s %s\nsocial security number: %s",
58          getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // end method toString
60
61    // abstract method overridden by subclasses
62    public abstract double earnings(); // no implementation here
63 } // end abstract class Employee
```

`abstract` method **`earnings`** has no implementation

SALARIED EMPLOYEE SUBCLASS

```
1  // SalariedEmployee.java
2  // SalariedEmployee class extends Employee.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // four-argument constructor
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // pass to Employee constructor
13       setWeeklySalary( salary ); // validate and store salary
14    } // end four-argument SalariedEmployee constructor
15
16    // set salary
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // end method setWeeklySalary
21
```

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

Call superclass constructor

Call **setWeeklySalary** method

Validate and set weekly salary value

SALARIED EMPLOYEE SUBCLASS

```
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 SalariedEmployee object
35    public String toString()
36    {
37       return String.format( "salaried employee: %s\n%s: $%,.2f",
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`**

HOURLY EMPLOYEE SUBCLASS

```java
// 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

HOURLY EMPLOYEE SUBCLASS

```
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

COMMISSION EMPLOYEE SUBCLASS

```java
// 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

COMMISSION EMPLOYEE SUBCLASS

```
24    // return commission rate
25    public double getCommissionRate()
26    {
27       return commissionRate;
28    } // end method getCommissionRate
29
30    // set gross sales amount
31    public void setGrossSales( double sales )
32    {
33       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34    } // end method setGrossSales
35
36    // return gross sales amount
37    public double getGrossSales()
38    {
39       return grossSales;
40    } // end method getGrossSales
41
```

Validate and set the gross sales value

COMMISSION EMPLOYEE SUBCLASS

```
42    // calculate earnings; override abstract method earnings in Employee
43    public double earnings()
44    {
45       return getCommissionRate() * getGrossSales();
46    } // end method earnings
47
48    // return String representation of CommissionEmployee object
49    public String toString()
50    {
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    } // end method toString
56 } // end class CommissionEmployee
```

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

Override **`toString`** method

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

BASE PLUS COMMISSION EMPLOYEE SUBCLASS

```
1  // BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class e
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       super( first, last, ssn, sales, rate );
13       setBaseSalary( salary ); // validate and store base salary
14    } // end six-argument BasePlusCommissionEmployee constructor
15
16    // set base salary
17    public void setBaseSalary( double salary )
18    {
19       baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
20    } // end method setBaseSalary
21
```

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

Call superclass constructor

Validate and set base salary value

BASE PLUS COMMISSION EMPLOYEE SUBCLASS

```
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

```
1  // 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:\n" );
21
```

TEST CLASS

```
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`**

TEST CLASS

```
48          // determine whether element is a BasePlusCommissionEmployee
49          if ( currentEmployee instanceof BasePlusCommissionEmployee )
50          {
51             // downcast Employee reference to
52             // BasePlusCommissionEmployee referenc
53             BasePlusCommissionEmployee employee =
54                ( BasePlusCommissionEmployee ) currentEmployee;
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.earnings() );
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** object

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

TEST CLASS

```
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
```

OUTPUT

OUTPUT

```
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

VOCABULARY I

- class – a description of a set of objects
- object – a member of a class
- instance – same as "object"
- field – data belong to an object or a class
- variable – a name used to refer to a data object
  - instance variable – a variable belonging to an object
  - class variable, static variable – a variable belonging to the class as a whole
  - method variable – a temporary variable used in a method

VOCABULARY II

- method – a block of code that can be used by other parts of the program
  - instance method – a method belonging to an object
  - class method, static method – a method belonging to the class as a whole
- constructor – a block of code used to create an object
- parameter – a piece of information given to a method or to a constructor
  - actual parameter – the value that is passed to the method or constructor
  - formal parameter – the name used by the method or constructor to refer to that value
- return value – the value (if any) returned by a method

VOCABULARY III

- hierarchy – a treelike arrangement of classes
- root – the topmost thing in a tree
- Object – the root of the class hierarchy
- subclass – a class that is beneath another in the class hierarchy
- superclass – a class that is above another in the class hierarchy
- inherit – to have the same data and methods as a superclass

HOMEWORK

Review class notes.

Additional reading:

Examples of UML diagrams

Start a discussion on Google Groups to clarify your doubts.