Example: O-O Payroll Program (§11.4)

Object-Oriented Design

Behavior. Our program should read a sequence of employees from an input file, (managers, secretaries, programmers, and consultants). It should compute their pay and print a paycheck for each employee showing the employee's name, ID, and pay.

Description of Problem's Object	Type	Kind	Name
Our program	`PayrollGenerator`	--	--
Employee sequence	`Employee []`	varying	*employee*
Input file (stream)	`BufferedReader( FileReader( fileName))`	varying	*empFile*
Input file name	`String`	varying	*args*[0]
Employee	`Employee`	varying	*employee*[*i*]
Managers	`Manager`	varying	--
Secretaries	`Secretary`	varying	--
Programmers	`Programmer`	varying	--
Consultants	`Consultant`	varying	--
Pay	`double`	varying	*employee*[*i*].*pay*( )
Paycheck	`Paycheck`	varying	*paycheck*
Employee's name	`String`	varying	*employee*[*i*].*name*( )

OOD Analysis

Kind of employee — **Common Attributes**

Managers, programmers — *salary*

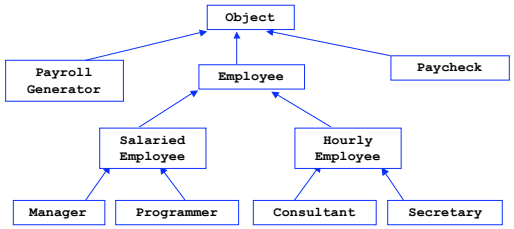
- Create a **`SalariedEmployee`** class
- Make **`Manager`** and **`Programmer`** subclasses.

Secretaries, consultants — *hourly wage, hours*

- Create an **`HourlyEmployee`** class
- Make **`Secretary`** and **`Consultant`** subclasses.

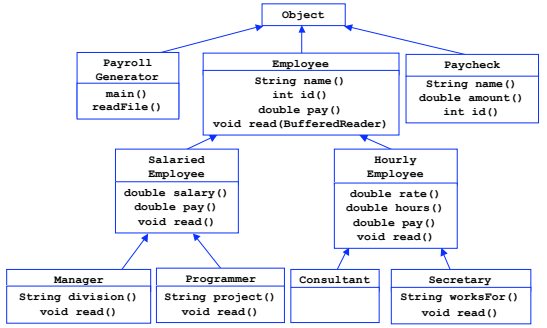
Salaried employee, hourly employees — *name, ID number, pay*, etc.

- Create an **`Employee`** class
- Make **`SalariedEmployee`** and **`HourlyEmployee`** subclasses.



Operation	Responsibility of
i. Read a sequence of employees from a file	`PayrollGenerator`
(open a stream to a file, read an `Employee` from a stream, close a stream to a file)	`Employee`, subclasses
ii. Compute an employee's pay	`Employee`
iii. Construct a paycheck	`Paycheck`
iv. Access an employee's name	`Employee`
v. Access an employee's ID number	`Employee`
vi. Access an employee's pay	`Employee`, subclasses

5



6

File Format

Number of employees	`7`
Employee #1 kind	`Manager`
Employee #1 name	`Grumpy`
Employee #1 id	`4`
Employee #1 salary	`950.00`
Employee #1 division	`Javadoc`
Employee #2 kind	`Secretary`
Employee #2 name	`Bashful`
Employee #2 id	`1`
Employee #2 wage	`8.75`
Employee #2 hours	`40`
Employee #2 works for	`Grumpy`
. . .	. . .

7

Because an employee's pay is computed differently for salaried and hourly employees, we declare an abstract `pay()` method in class `Employee` so that `pay()` messages can be sent to an `Employee` handle.

```
abstract class Employee extends Object {
  //--- Employee constructors ---
  ...
  //--- Accessor methods ---
  ...
  //--- String converter (for output) ---
  ...
  //--- File input ---
  void read(BufferedReader reader) {
    try  {
      myName = reader.readLine();
      myID   = new Integer(reader.readLine()).intValue();
    }
    catch (Exception error)  {
      System.err.println("Employee:read(): " + error);
      System.exit(1);
    }
  }
  //--- Abstract pay method ---
  abstract public double pay();

  //--- Attribute variables ---
  private String myName;
  private int    myID;
}
```

8

But only subclasses **SalariedEmployee** and **HourlyEmployee** will know how to compute their pay, and so we leave it to them to supply a definition for this method:

```
class SalariedEmployee extends Employee {

  //--- Constructors ---
   ...
  //--- Accessor methods ---
  public double pay()    { return salary(); }
  public double salary() { return mySalary; }
  ...
  //--- String converter (for output) ---
  ...
  //--- File input ---
  ...
  //--- Attribute variables ---
   private double mySalary;
}
```

9

```
class HourlyEmployee extends Employee  {
  public final static double OVERTIME_THRESHOLD  = 40;
  public final static double OVERTIME_FACTOR     = 1.5;

  //--- Constructors ---
  ...
  //--- Accessor methods ---
  ...
  //--- Pay for hourly employee ---
  public double pay() {
    if (myHours <= OVERTIME_THRESHOLD)
      return myHours * myHourlyRate;
    else
      return OVERTIME_THRESHOLD * myHourlyRate +
             (myHours - OVERTIME_THRESHOLD) *
               myHourlyRate * OVERTIME_FACTOR;
  }
  ...
  //--- String converter (for output) ---
  ...
  //--- File input --- }
  ...
  //--- Attribute variables ---
  private double myHourlyRate;
  private double myHours;
}
```

pay() is defined differently in **SalariedEmployee** and **HourlyEmployee**. *Polymorphism* selects the correct version.

10

The **Manager** class is defined as a subclass of **SalariedEmployee**. Note that although **pay()** was an abstract method in the root class **Employee**, it was defined in **SalariedEmployee**, and it is this definition that is inherited by **Manager**.

```
class Manager extends SalariedEmployee
{
  //--- Constructors ---
  ...
  //--- Accessor method ---
  public String division() { return myDivision; }
  ...
  //--- String converter (for output) ---
  ...
  //--- File input ---

  //--- Attribute variable ---
  private String myDivision;
}
```

11

Similarly, the **Programmer** class is defined as a subclass of **SalariedEmployee**:

```
class Programmer extends SalariedEmployee
{
  //--- Constructors ---
  ...
  //--- Accessor method ---
  public String project() { return myProject; }
  ...
  //--- File input ---
  ...

 //--- Attribute variable ---
 private String myProject;
}
```

And **Secretary** and **Consultant** classes are defined as subclasses of **HourlyEmployee** and inherit it's methods, including the **pay()** method.

12

The **Paycheck** class is designed to model paychecks:

```
/** Paycheck.java provides a class to model paychecks.
 *  New attribute variables store a name, check amount,
 *  and ID number.
 *  Methods: Constructors: to construct a Paycheck from Employee
 *           accessors; to-string converter for output purposes;
 */

import java.text.*;         // NumberFormat

class Paycheck extends Object  {
 //--- Paycheck constructor ---
 public Paycheck(Employee employee) {
   myName   = employee.name();
   myAmount = employee.pay();
   myID     = employee.id();
 }
```



```
 //--- Accessor methods ---
 public String name()   { return myName; }
 public double amount() { return myAmount; }
 public int id()        { return myID; }

 //--- String converter (for output) ---
 public String toString() {
   NumberFormat cf = NumberFormat.getCurrencyInstance();
   String formattedAmount = cf.format(myAmount);
   return myName + "\t\t" + formattedAmount + "\n" + myID;
 }

 //--- Attribute variables ---
 private String myName;
 private double myAmount;
 private int    myID;
}
```

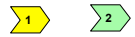
Note the use of the **NumberFormat** class method **getCurrencyInstance()** to create a number formatter for monetary values; it is then sent the **format()** message along with by the amount to be formatted to produce a **String** with the appropriate format.

Finally, there's the **PayrollGenerator** class that calculates wages and prepares paychecks:

```
class PayrollGenerator {
public static void main(String [] args)  {
   Employee [] employee = readFile(args[0]);

   for (int i = 0; i < employee.length; i++)  {
      Paycheck check = new Paycheck(employee[i]);
      System.out.println(check + "\n");
   }

public static Employee [] readFile(String fileName) {
  BufferedReader empFile = null;
  int numberOfEmployees = 0;
  Employee [] result = null;
  try {
    empFile = new BufferedReader(new FileReader(fileName));

    numberOfEmployees =
                  new Integer(empFile.readLine()).intValue();
    result = new Employee[numberOfEmployees];
    int i = 0;
```



```
   String className = "";
   for (;;) {
     String blankLine = empFile.readLine(); // eat blank line
     className = empFile.readLine();

     if (className == null || className == ""  // end of stream
         || i == result.length)                // end of array
       break;

     result[i] = (Employee)Class.forName(className).newInstance();
     result[i].read(empFile);
     i++;
   }
    empFile.close();
  }
  catch (Exception e)  {
     System.err.println(e);
     System.exit(1);
  }
  return result;
}
```



Java has a class named **`Class`** that provides various operations for manipulating classes. Two useful ones are **`forName()`** and **`newInstance()`**. If **`String`** ***`str`*** is the name of a class,

```
Class.forName(str)
```

returns a **`Class`** object with name ***`str`***; and

```
Class.forName(str).newInstance()
```

returns an instance of the class with this name, created using the default constructor of that class. It returns that instance as an **`Object`**, and so it must be cast to an appropriate type, usually the nearest ancestor.

For example,

```
(Employee)Class.forName(className).newInstance()
```

creates an object of type **`className`**.

employees.txt

```
7
Manager
Grumpy
4
950.00
Java

Secretary
Bashful
1
8.00
45
Happy

Programmer
Happy
5
850.00
Java IDE

Consultant
Doc
2
15.90
20

Programmer
Sneezy
7
850.00
Java Debug

Consultant
Dopey
3
0.50
40

Programmer
Sleepy
6
900.00
Java Threads
```

java PayrollGenerator employees.txt

```
Grumpy          $950.00
4

Bashful         $380.00
1

Happy           $850.00
5

Doc             $318.00
2

Sneezy          $850.00
7

Dopey           $20.00
3

Sleepy          $900.00
6
```

Example: Aviary Program (§11.1)

Some special features of the bird hierarchy:

- The root class **`Bird`** along with subclasses **`WalkingBird`** and **`FlyingBird`** are abstract classes with abstract method **`getCall()`**.
- It uses the **`getClass()`** method from class **`Object`**.
- It has a *random number generator* used to select a random phrase by talking parrots and a random number of "Hoo"s by a snow owl:


```
import java.util.Random;
  ...
abstract class Bird extends Object  {
  ...
 //--- Random number generator ---
 /** Static random integer generator
  *  Receive: int upperBound
  *  Return:  a random int from the range 0..upperBound-1
  */
  protected static int randomInt(int upperBound)  {
    return myRandom.nextInt(upperBound);
  }
  //--- Attribute variables ---
  ...
  private static Random myRandom = new Random();
}
```

```
/** SnowOwl.java provides a subclass of Owl that models a
 *  snow owl. It provides a constructor and a definition
 *  of getCall().
 */

class SnowOwl extends Owl
{
  public SnowOwl() { super("white"); }

  public String getCall()  {
    String call = "";
    int randomNumber = randomInt(4) + 1;   // 1..4

    for (int count = 1; count <= randomNumber; count++)
      call += "Hoo";

    return call + "!";
  }
}
```