Chapter 1

Object-Oriented Concepts

A *class* consists of variables called fields together with functions called methods that act on those fields.

Let's look at the

String

class

An *object* is a variable whose type is a class. An object has the fields and can call the methods of its class.

A String object is a variable that contains a string (a sequence of characters) and can call methods in the String class.

String s;

In this declaration, s is not a String object, but rather a String reference, that is, a variable that can hold the address of a String object.

To store the address of a String object in s, we will:

1. **Allocate space for a new String object.**
2. **Initialize the fields in that object.**
3. **Assign to s the address of that object.**

s = **new** String();

A method with the same name as the class is called a *constructor*.

The purpose of a constructor is to initialize the object's fields.

A class's default constructor has no parameters.

The String class's default constructor initializes the fields so that the String object represents an empty string.

Another constructor in the String class has a String parameter. Here is the heading

parameter

public String (String original)

String t = **new** String ("Aloha");

argument

is a reference to "Aloha"

Now the String objects referenced by s and t can invoke String methods:

s.length() // returns 0

t.toLowerCase() // returns (a reference to)

// "aloha" . t is still a

// reference to "Aloha"

```
/**
 *  Returns the index within this String object
 *  of the first occurrence of the specified
 *  substring.
 *  @param str – the specified substring
 *  @return the index of the first occurrence
 *                 of str in this String object, or –1
 *                 if str is not a substring of this
 *                 String object
```

```
 *  @throws NullPointerException – if str is
 *                 null
 */
public int indexOf (String str)
```

The JAVADOC comments plus the method heading constitute the *method specification* – A user's view of the method.

```
System.out.println (t.indexOf (“ha”));

System.out.println (t.indexOf (“a”));

System.out.println (s.indexOf (“ha”));
```

Hint: Indexes start at 0.

String w = **null**;

w **does not contain the address of any String object, so** w **cannot call any methods.**

The equals method tests for equality of objects, and the == operator tests for equality of references.

String z = **new** String (“Aloha”);

s.equals (“”)

s == “”

t.equals (“Aloha”)

t == “Aloha”

t.equals (null)

t.equals (z)

t == z

w.equals (null)

w == null

String y1 = "Aloha";

String y2 = "Aloha";

These statements create two references, y1 **and** y2**, to the same string object, so**

y1 == y2   // returns true

y1 == t   // returns false

but

y1.equals (t)   // returns true

So far, we have studied what a class does, not how the class does it.

That is, we have studied a class from the user's perspective (method specifications) rather than from a developer's perspective (fields and method definitions)

Principle of data abstraction:

A user's code should not access the implementation details of the class used.

Many of the classes we will study share the same method specifications.

When we abstract these specifications from the classes we get an interface. An *interface* consists of method specifications and constants only.

For example, here is an interface for the employees in a company. The information read in for each employee consists of the employee's name and gross pay.

```
public interface Employee
{
```

```
/**
 *   Determines if this Employee object's gross pay
 *   is greater than a specified employee's gross pay.
 *
 *  @param otherEmployee – the specified
 *         Employee object whose gross pay this
 *         Employee object's gross pay is compared to.
 *
*/
boolean makesMoreThan
                     (Employee otherEmployee);
```

```
    /**
     *  Returns a String representation of this
     *  Employee object with the name followed by a
     *  space followed by a dollar sign followed by the
     *  gross weekly pay, with two fractional digits.
     *
     *  @return a String representation of this
     *          Employee object.
     *
     */
    String toString();

} // interface Employee
```

Note: Each method is automatically public, and each method heading is followed by a semicolon.

To implement that interface, we will create a class with fields and, using those fields, definitions of at least the two methods.

```
import java.util.*;      // for StringTokenizer class
import java.text.*;      // for DecimalFormat class

public class FullTimeEmployee implements Employee
{
      private String name;

      private double grossPay;
```

```java
/**
 *  Initializes this FullTimeEmployee object to have an
 *  empty string for the name and 0.00 for the gross pay.
 *
 */
public FullTimeEmployee()
{
     final String EMPTY_STRING = "";

     name = EMPTY_STRING;
     grossPay = 0.00;
} // default constructor
```

```java
/**
 *  Initializes this FullTimeEmployee object's name
 *  and gross pay from a specified String object,
 *  which consists of a name and gross pay, with at
 *  least one blank in between.
 *
 *  @param s – the String object from which this
 *               FullTimeEmployee object is initialized.
 *
 */
```

```java
public FullTimeEmployee (String s)
{
     StringTokenizer tokens = new StringTokenizer (s);
     name = tokens.nextToken();
     grossPay = Double.valueOf (tokens.nextToken ());
} // constructor with String parameter
```

```java
/**
 *   Determines if this Employee object's gross pay
 *   is greater than a specified Employee object's
 *  gross pay.
 *
 *  @param otherEmployee – the specified
 *       Employee object whose gross pay this
 *       Employee object's gross pay is compared to.
 *
 *  @return true – if otherEmployee is a
 *        FullTimeEmployee object, and this
 *        if the calling object's gross
 *        pay is greater than otherEmployee's gross pay.
 *
 *
 */
```

```java
public boolean makesMoreThan
                (Employee otherEmployee)
  {
     if (!(otherEmployee instanceof FullTimeEmployee))
          return false;
     FullTimeEmployee full =
                    (FullTimeEmployee)otherEmployee;
     return grossPay > full.grossPay;
  } // method makesMoreThan
```

Note: The parameter type must be Employee **because that is the parameter type in the interface.**

```java
/**
 *  Returns a String representation of this Employee
 *  object with the name followed by a space followed
 *  by a dollar sign followed by the gross weekly pay,
 *  with two fractional digits.
 *
 *  @return a String representation of this
 *              Employee object.
 *
 */
```

```
public String toString()
{
    final String DOLLAR_SIGN = "  $";

    DecimalFormat d = new DecimalFormat ("0.00");

    return name + DOLLAR_SIGN +
                                d.format (grossPay);
} // method toString

} // class FullTimeEmployee
```

Suppose, in some other class, we have the following:

```
FullTimeEmployee emp1 =
        new FullTimeEmployee ("a 1000.00"),
              emp2 =
        new FullTimeEmployee ("b 885.00");

System.out.println (emp1.makesMoreThan (emp2));
```

What is compared here:

```
FullTimeEmployee full =
                (FullTimeEmployee)otherEmployee;
return grossPay > full.grossPay;
```

In a method definition, when a member (field or method) appears without an object reference, a reference to the calling object is assumed.

Now suppose we want to find the best-paid full-time employee in a company. We will create a Company class.

There are methods to initialize a Company object, to find the best-paid full-time employee, and to print that employee's name and gross pay.

There are two fields:

```
bestPaid  // to hold the best paid full-time employee

atLeastOneEmployee   // in case there are no full-
                     // time employees in the input
```

```
public class Company
{
   private FullTimeEmployee bestPaid;

   private boolean atLeastOneEmployee;


    /**
     *  Initializes this Company object.
     *
     */
    public Company()
    {
         bestPaid = new FullTimeEmployee();
         atLeastOneEmployee = false;
    } // default constructor
```

```
    /**
     *  Determines the best-paid full-time employee in
     *  this Company object.
     *
     */
    public void findBestPaid () throws IOException
    {
       final String SENTINEL = "***";

       final String INPUT_PROMPT =
           "\nPlease enter a name (with no blanks) " +
           "and gross pay, followed by the Enter key. " +
           "The sentinel is " + SENTINEL + "  ";

       FullTimeEmployee employee;
```

```
        String line;

        BufferedReader reader = new BufferedReader
                    (new InputStreamReader (System.in));

        while (true)
        {
              System.out.print (INPUT_PROMPT);
              line = reader.readLine();
              if (line.equals (SENTINEL))
                     break;
              employee = new FullTimeEmployee (line);
              atLeastOneEmployee = true;
              if (employee.makesMoreThan (bestPaid))
                    bestPaid = employee;
        }//while
    } // method findBestPaid
```

```
    /**
     *  Prints out the best-paid full-time employee in the
     *  input, or an error message if the only line of input
     *  is the sentinel.
     *
     */
    public void printBestPaid()
    {
       final String NO_INPUT_MESSAGE =
          "\n\n\nERROR: there were no " +
          "employees in the input.";

       final String BEST_PAID_MESSAGE =
          "\n\n\nThe best paid employee " +
          "(and gross pay) is ";
```

```
        if (atLeastOneEmployee)
            System.out.println (BEST_PAID_MESSAGE +
                                    bestPaid);
        else
            System.out.println (NO_INPUT_MESSAGE);
    } // method printBestPaid

 } // class Company
```

Finally, we need a main **method to get everything started.**

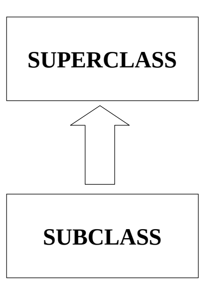
```java
import java.io.*;

public class CompanyMain
{
   /**
     *  Finds and prints out the best-paid full-time
     *  employee in the input.
    *
    */
   public static void main (String[ ] args)
                                                 throws IOException
   {
      Company company = new Company();

      company.findBestPaid();
      company.printBestPaid();
   } // method main

} // class CompanyMain
```

Exercise: Make up sample input, and the corresponding output.

Inheritance

***Inheritance* is the ability to define a new class that includes all the fields and some or all of the methods of an existing class.**

Existing class = **superclass** = base class
New class = **subclass** = derived class



The subclass may declare new fields and methods, and may *override* existing methods by giving them method definitions that differ from those in the superclass.

Example: **Find the best-paid hourly full-time employee with no overtime (40 hours)**

Input: **Name,**
Hours worked,
Pay rate

Modify FullTimeEmployee **class?**

The Open-Closed Principle

Every class should be

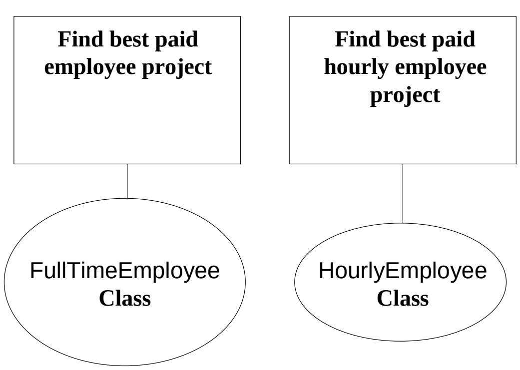
Open: **extendible through inheritance**

Closed: **stable for existing applications**

Specifically, the FullTimeEmployee **class should be stable for the existing application of finding the best-paid employee in a company.**

And extendible for this new application!

```
public class HourlyEmployee
                        extends FullTimeEmployee
{
```



Overridden methods?

The declarations of name **and** grossPay **must be altered in the** FullTimeEmployee **class: those fields cannot be private.**

Would it be a good idea to make them public?

```
public class FullTimeEmployee
{
    protected String name;

    protected double grossPay;
```

A superclass member (field or method) with protected visibility is accessible in any subclass method as if the member were declared in the subclass instead of in the superclass.

For the sake of Subclasses of HourlyEmployee**:**

```
protected int hoursWorked;

protected double payRate;
```

```
public class HourlyEmployee
                      extends FullTimeEmployee
                      implements Employee
{
       protected int hoursWorked;

       protected double payRate;
```

```
/**
 *  Initializes this HourlyEmployee object to have an
 *  empty string for the name, 0 for hours worked, 0.00
 *  for the pay rate and 0.00 for grossPay.
 *
 */
public HourlyEmployee()
{
        hoursWorked = 0;
        payRate = 0.00;
} // default constructor
```

```
/**
 *  Initializes this HourlyEmployee object's name
 *  and gross pay from a a specified String object,
 *  which consists of a name, hours worked and
 *  pay rate, with at least one blank between each
 *  of those three components.
 *
 *  @param s – the String object from which this
 *               HourlyEmployee object is initialized.
 *
 */
```

```
public HourlyEmployee (String s)
{
        StringTokenizer tokens =
                new StringTokenizer (s);
        name = tokens.nextToken();
        hoursWorked =
                Integer.parseInt (tokens.nextToken());
        payRate = Double.parseDouble
                (tokens.nextToken());

        grossPay = hoursWorked * payRate;
} // constructor with string parameter
```

```
/**
 *  Determines if this HourlyEmployee object's gross pay is
 *  greater than a specified Employee object's gross pay.
 *  @param otherEmployee – the specified Employee object
 *          whose gross pay this HourlyEmployee object's gross
 *          pay is compared to.
 *  @return true – if this HourlyEmployee object did not work
 *          any overtime, otherEmployee is a FullTimeEmployee
 *          object, and this HourlyEmployee object's gross pay
 *          is greater than otherEmployee's gross pay.
 *          Otherwise, return false.
 */
public boolean makesMoreThan (Employee otherEmployee)
{
      final int MAX_NORMAL_HOURS = 40;
      return hoursWorked <= MAX_NORMAL_HOURS
              && super.makesMoreThan (otherEmployee);
} // method makesMoreThan
} // class HourlyEmployee
```

For the project of finding the best-paid, non-overtime hourly employee, we will need HourlyCompany, a **Subclass of** Company**.**

```
import java.io.*;

public class HourlyCompany extends Company
{


    /**
     *  Initializes this HourlyCompany object.
     *
     */
    public HourlyCompany()
    {
    }
```

```
/**
 *  Determines the best-paid, non-overtime, full-time employee
 *    in this HourlyCompany object.
 *
 */
public void findBestPaid ()  throws IOException
{
      final String SENTINEL = "***";

      final String INPUT_PROMPT =
          "\n\nPlease enter a name, with no " +
          "blanks, hours worked and pay rate.  The sentinel is "
          + SENTINEL + "  ";

      HourlyEmployee hourly;

      String line;

      BufferedReader reader = new BufferedReader
                              (new InputStreamReader (System.in));
```

```
      while (true)
      {
          System.out.print (INPUT_PROMPT);
          line = reader.readLine();
          if (line.equals (SENTINEL))
                break;
          hourly = new HourlyEmployee (line);
           if (hourly.makesMoreThan (bestPaid))
           {
                atLeastOneEmployee = true;
                bestPaid = hourly;
            } // if
       } // while
   } // findBestPaid

} // class HourlyCompany
```

bestPaid = hourly?

FullTimeEmployee bestPaid;

HourlyEmployee hourly;

Subclass Substitution Rule:

When a

Reference-To-Superclass-Object

is called for in an evaluated expression, a

Reference-To-Subclass-Object

may be substituted.

So

bestPaid = hourly;

is legal. But

hourly = bestPaid;

would be illegal because the variable on the left-hand side of an assignment statement is not evaluated.

It is also legal to have a SubClass reference argument passed to a SuperClass reference parameter.

Data Abstraction:

A user's code should not access the implementation details of the class used.

Burden on user;
Helps user

Information Hiding:

Making the implementation details of a class inaccessible to user's code.

Burden on developer;
Helps user

Encapsulation:

Grouping of fields and methods into a single entity–the class–whose implementation details are hidden from users (for example, with the private and protected visibility modifiers.

Object-Oriented Essentials:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**

Polymorphism **is the ability of a reference to refer to different objects.**

Such a reference is called a ***Polymorphic*** **reference.**

```
public class X
{

    public String whatIAm( )
    {
        return "I'm an X.";
    } // method whatIAm

} // class X
```

```
public class Y extends X
{

    public String whatIAm()
    {
        return "I'm a Y.";
    } // method whatIAm

} // class Y
```

```
public static void main (String[ ] args)
                                throws IOException
{
    X x;    // x is of type reference-to-X

    BufferedReader reader = new BufferedReader
                    (new InputStreamReader (System.in);

    if (reader.readLine().equals ("Go with X"))
        x = new X();
    else
        x = new Y();
    System.out.println (x.whatIAm());
} // method main
```

What is printed?

In other words, which version of the whatIAm **method is invoked?**

When a message is sent, the version of the method called depends on

The type of the object,

***Not* on the type of the reference.**

How can the Java compiler decide which version of the whatIAm **method is to be called?**

The determination cannot **be made at compile time because the type of the object (X or Y) is not available until run-time.**

The "binding" of the method identifier to the method definition must be made at run time.

This is called

- **Late binding**
- **Dynamic binding**

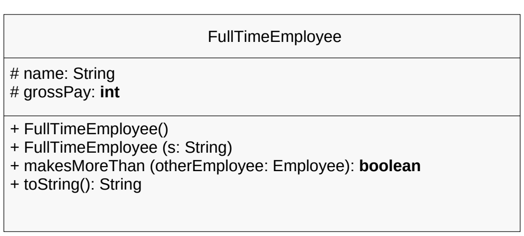
A *virtual method* is a method that is bound to its method identifier at run-time.

In Java, almost all methods are virtual.

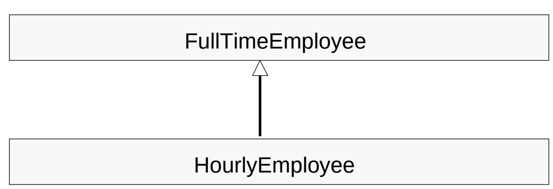
The Unified Modeling Language

UML

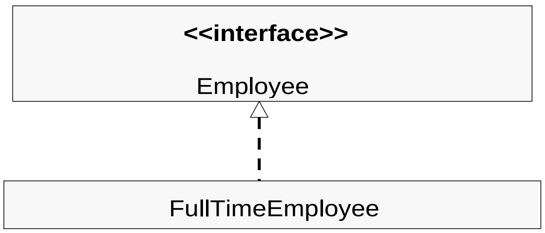
A Class-Level Documentation Tool



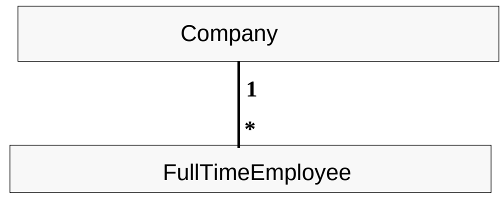
Inheritance: Solid arrow from Subclass to Superclass



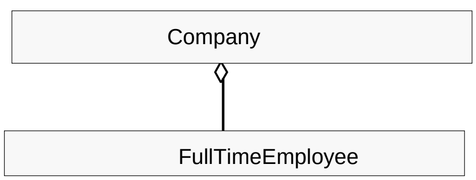
Interface: Dashed arrow from Class to Interface



Association between classes: Solid line



Aggregation (an association in which one class has a field whose type is the other class): Solid line with diamond



Exercise: Draw the UML diagram for the best-paid hourly-employee project. Include method headings (and fields) for Company, HourlyCompany, FullTimeEmployee, HourlyEmployee and Employee.