Chapter 4

Polymorphism

CSC 113
King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI
Objectives

• After you have read and studied this chapter, you should be able to

  – Write programs that are easily extensible and modifiable by applying polymorphism in program design.
  – Define reusable classes based on inheritance and abstract classes and abstract methods.
  – Differentiate the abstract classes and Java interfaces.
  – Define methods, using the protected modifier.
  – Parse strings, using a String Tokenizer object.
Introduction to Polymorphism

• There are three main programming mechanisms that constitute object-oriented programming (OOP)
  – Encapsulation
  – Inheritance
  – Polymorphism

• **Polymorphism** is the ability to associate many meanings to one method name
  – It does this through a special mechanism known as *late binding* or *dynamic binding*

• A **polymorphic method** is one that has the same name for different classes of the same family, but has different implementations, or behavior, for the various classes.
Introduction to Polymorphism

• **Polymorphism**
  
  – 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
  
  – Facilitates adding new classes to a system with minimal modifications to the system’s code
A polymorphic method (ex: display())

- A method that has multiple meanings
- Created when a subclass overrides a method of the superclass

**Example: Demonstrating Polymorphic Behavior**

```
Base
  + display()

Doubler
  + display()

Tripler
  + display()

Squarer
  + display()
```
Example: Demonstrating Polymorphic Behavior

```java
double class Base {
    protected int i = 100;

    ...
    public void display() { System.out.println( i );
    }
}

double class Doubler extends Base {
    ...
    public void display() { System.out.println( i*2 );
    }
}

double class Tripler extends Base {
    ...
    public void display() {
        System.out.println(i*3);
    }
}

double class Squarer extends Tripler {
    ...
    public void display() { System.out.println( i*i );
    }
}
```
Example: Demonstrating Polymorphic Behavior

Case: Static binding

Some main program

<table>
<thead>
<tr>
<th></th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base B = new Base();</td>
<td>100</td>
</tr>
<tr>
<td>B. display();</td>
<td>100</td>
</tr>
<tr>
<td>Doubler D = new Doubler();</td>
<td>200</td>
</tr>
<tr>
<td>D. display();</td>
<td>200</td>
</tr>
<tr>
<td>Tripler T = new Tripler();</td>
<td>300</td>
</tr>
<tr>
<td>T. display();</td>
<td>300</td>
</tr>
<tr>
<td>Squarer S = new Squarer();</td>
<td>10000</td>
</tr>
<tr>
<td>S. display();</td>
<td>10000</td>
</tr>
</tbody>
</table>

*Static binding* occurs when a method is defined with the same name but with different headers and implementations. The actual code for the method is attached, or bound, at compile time. Static binding is used to support overloaded methods in Java.
**Example: Demonstrating Polymorphic Behavior**

**Case: Dynamic binding**

• A superclass reference can be aimed at a subclass object
  
  — This is possible because a subclass object *is a* superclass object as well
  
  — When invoking a method from that reference, the type of the actual referenced object, not the type of the reference, determines which method is called

---

**Some main program**

<table>
<thead>
<tr>
<th>Base</th>
<th>B = new Base();</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B. <code>display()</code>; output: 100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base D;</th>
</tr>
</thead>
<tbody>
<tr>
<td>D = new Doubler();</td>
</tr>
<tr>
<td>D. <code>display()</code>; output: 200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base T;</th>
</tr>
</thead>
<tbody>
<tr>
<td>T = new Tripler();</td>
</tr>
<tr>
<td>T. <code>display()</code>; output: 300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base S;</th>
</tr>
</thead>
<tbody>
<tr>
<td>S = new Squarer();</td>
</tr>
<tr>
<td>S. <code>display()</code>; output: 10000</td>
</tr>
</tbody>
</table>

**Late binding or dynamic binding:**

The appropriate version of a polymorphic method is decided at execution time.
Example: Inheritance Hierarchy of Class Student: Polymorphism case

```
#NUM_OF_TESTS : int = 3
#name : string
#test [] : int
+Student()
+Student(in studentName : string)
+setScore(in s1 : int, in s2 : int, in s3 : int)
+setName(in newName : string)
+getTestScore() : int
+getCoursegrade() : string
+setTestScore(in testNumber : int, in testName : string)
+getName() : string
+computeCourseGrade()
```

```
Student

GraduateStudent
+computeCourseGrade()

UnderGraduateStudent
+computeCourseGrade()
```
Example: Inheritance Hierarchy of Class Student: Polymorphism case

Creating the roster Array

• We mentioned in array definition that an array must contain elements of the same data type. For example, we can’t store integers and real numbers in the same array.

• To follow this rule, it seems necessary for us to declare two separate arrays, one for graduate and another for undergraduate students. This rule, however, does not apply when the array elements are objects using the polymorphism. We only need to declare a single array.

• We can create the roster array combining objects from the Student, UndergraduateStudent, and GraduateStudent classes.

```java
Student roster = new Student[40];
...
roster[0] = new GraduateStudent();
roster[1] = new UndergraduateStudent();
roster[2] = new UndergraduateStudent();
...
```
State of the roster Array

- The roster array with elements referring to instances of \texttt{GraduateStudent} or \texttt{UndergraduateStudent} classes.
Sample Polymorphic Message

• To compute the course grade using the roster array, we execute

```java
for (int i = 0; i < numberOfStudents; i++) {
    roster[i].computeCourseGrade();
}
```

• If roster[i] refers to a GraduateStudent, then the computeCourseGrade method of the GraduateStudent class is executed.

• If roster[i] refers to an UndergraduateStudent, then the computeCourseGrade method of the UndergraduateStudent class is executed.
The instanceof Operator

- The `instanceof` operator can help us learn the class of an object.

- The following code counts the number of undergraduate students.

```java
int undergradCount = 0;
for (int i = 0; i < numberOfStudents; i++) {
    if (roster[i] instanceof UndergraduateStudent) {
        undergradCount++;
    }
}
```
Implementation **Student** in Java

**Case Study :**

class **Student** {
    protected final static int NUM_OF_TESTS = 3;
    protected String name;
    protected int[] test;
    protected String courseGrade;

    public **Student**() { this("No Name"); }
    public **Student**(String studentName) {
        name = studentName;
        test = new int[NUM_OF_TESTS];
        courseGrade = "****";
    }

    public void **setScore**(int s1, int s2, int s3) {
        test[0] = s1; test[1] = s2; test[2] = s3;
    }

    public void **computeCourseGrade**() {
        courseGrade="";
    }

    public String **getCourseGrade**( ) {
        return courseGrade;
    }

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

    public int **getTestScore**(int testNumber) {
        return test[testNumber-1];
    }

    public void **setName**(String newName) {
        name = newName;
    }

    public void **setTestScore**(int testNumber, int testScore) {
        test[testNumber-1]=testScore;
    }
}

class **GraduateStudent** extends **Student** {

    public void **computeCourseGrade**() {
        int total = 0;
        for (int i = 0; i < NUM_OF_TESTS; i++) {
            total += test[i];
        }
        if (total >= 80) {
            courseGrade = "Pass";
        } else {
            courseGrade = "No Pass";
        }
    }
}

class **UnderGraduateStudent** extends **Student** {

    public void **computeCourseGrade**() {
        int total = 0;
        for (int i = 0; i < NUM_OF_TESTS; i++) {
            total += test[i];
        }
        if (total >= 70) {
            courseGrade = "Pass";
        } else {
            courseGrade = "No Pass";
        }
    }
}

Dr. S. HAMMAMI
Implementation \textit{StudentTest} in Java

\textbf{Case Study :}

```java
public class StudentTest {
    public static void main(String[] args) {
        Student roster[] = new Student[2];
        roster[0] = new GraduateStudent();
        roster[1] = new UnderGraduateStudent();

        roster[0].setScore (20, 30, 50);
        roster[1].setScore (10, 17, 13);

        for (int i=0; i<roster.length; i++) {
            System.out.println("The name of the class is : "+ roster[i].getClass().getName());
            roster[i].computeCourseGrade();
            System.out.println(" Pass or Not  : "+ roster[i].getCourseGrade());
        }
    }
}
```

------ execution------------------

If \texttt{roster[i]} refers to a \texttt{GraduateStudent}, then the \texttt{computeCourseGrade} method of the \texttt{GraduateStudent} class is executed.

If \texttt{roster[i]} refers to a \texttt{UnderGraduateStudent}, then the \texttt{computeCourseGrade} method of the \texttt{UnderGraduateStudent} class is executed.

\textbf{Dr. S. HAMMAMI}
Implementation **StudentTest2** in Java

**Case Study:**

**Question:** Count the number of undergraduate students

```java
public class StudentTest2 {
    public static void main(String[] args) {
        Student roster[] = new Student[2];
        roster[0] = new GraduateStudent();
        roster[1] = new UnderGraduateStudent();

        roster[0].setScore (20, 30, 50);
        roster[1].setScore (10, 17, 13);
        int nb=0; // count the number of Under Graduate Students
        for (int i=0; i<roster.length; i++)
            if (roster[i] instanceof UnderGraduateStudent )
                nb++;

        System.out.println("The number of Under Graduate Students : " + nb);
    }
}
```

--- execution-------------------

*The number of Under Graduate Students : 1*

**Rule:** To determine the class of an object, we use the `instanceof` operator.
Example: Inheritance Hierarchy of Class Shape

- **Shape**
  - +area() : double
  - +perimeter() : double
  - +display()

- **Rectangle**
  - -width : double
  - -length : double
  - +Rectangle(in wid : double, in leng : double)
  - +getWidth() : double
  - +getLength() : double
  - +area() : double
  - +perimeter() : double

- **Square**
  - -side : double
  - +Square(in sid : double)
  - +getSide() : double
  - +area() : double
  - +perimeter() : double

Display method displays the area and perimeter of a shape object.

Dr. S. HAMMAMI
public class ShapeTest {
    public static void main(String[] args) {
        Shape shp = new Shape(); // shp is an object from class Shape
        Rectangle rect = new Rectangle(4.0, 5.0); // rect is an object from class Rectangle
        Square sqr = new Square(6.0); // sqr is an object from class Square

        shp.display(); //--- uses the method display() from the class Shape
        rect.display(); //--- object rect inherits method display() from Superclass Shape
        sqr.display(); //--- object sqr inherits method display() from Superclass Shape
    }
}

---- execution -------
The name of the class is : Shape
The area is : 0.0
The perimeter is : 0.0

The name of the class is : Rectangle
The area is : 20.0
The perimeter is : 13.0

The name of the class is : Square
The area is : 36.0
The perimeter is : 24.0
Implementation inheritance in Java

Case Study 3: Shape

```java
public class Shape {
    public double area() { return 0.0; }
    public double perimeter() { return 0.0; }
    public void display() {
        System.out.println("The name of the class is : " + this.getClass().getName());
        //--- getClass() a method inherits from the super class Object.
        //--- getName() a method from the class String.
        System.out.println("The area is :"+ area());
        System.out.println("The perimeter is :"+ perimeter()+"\n\n");
    }
}
```

```java
public class Rectangle extends Shape {
    private double width;
    private double length;
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    public double getLength() { return length; }
    public double getWidth() { return width; }
    public double area() {
        return (this.getLength()*this.getWidth());
    }
    public double perimeter() {
        return (2*this.getLength()+this.getWidth());
    }
}
```

```java
public class Square extends Shape {
    private double side;
    public Square(double side) {
        this.side = side;
    }
    public double getSide() { return side; }
    public double area() {
        return (this.getSide()*this.getSide());
    }
    public double perimeter() {
        return (4*this.getSide());
    }
}
```
Introduction to Abstract Classes

In the following example, we want to add to the Shape class a `display` method that prints the area and perimeter of a shape.
Introduction to Abstract Classes

Abstract Method

• The following method is added to the Shape class

    public void display()
    {
        System.out.println (this.area());
        System.out.println (this.perimeter());
    }

Introduction to Abstract Classes

Abstract Method

• There are several problems with this method:
  
  – The **area** and **perimeter** methods are invoked in the **display** method
  
  – There are **area** and **perimeter** methods in each of the subclasses
  
  – There is no **area** and **perimeter** methods in the **Shape** class, nor is there any way to define it reasonably without knowing whether the shape is Rectangle or Square.
Introduction to Abstract Classes

Abstract Class

• In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  
  – An abstract method has a heading, but no method body
  – The body of the method is defined in the subclasses

• The class that contains an abstract method is called an *abstract class*
Abstract Method

• An abstract method is like a **placeholder** for a method that will be fully defined in a descendent class

• It has a complete method heading, to which has been added the modifier **abstract**

• **It cannot be private**

• It has **no method body**, and ends with a semicolon in place of its body

```java
public abstract double area();
public abstract double perimeter();
```
Introduction to Abstract Classes

display() method displays the area and perimeter of a shape object

Shape
- color: string
  + area() : double
  + perimeter() : double
  + display()

Rectangle
- width : double
- length : double
  + Rectangle(in wid : double, in leng : double)
  + getWidth() : double
  + getLength() : double
  + area() : double
  + perimeter() : double
  + display()

Square
- side : double
  + Square(in sid : double)
  + getSide() : double
  + area() : double
  + perimeter() : double
  + display()
Introduction to Abstract Classes

• A class that has at least one abstract method is called an *abstract class*

  - An abstract class must have the modifier `abstract` included in its class heading:

```java
public abstract class Shape {
    protected String color;
    . . .
    public abstract double area();
    public abstract double perimeter();
    public void display() {
        System.out.println (this.area());
        System.out.println (this.perimeter());
    }
    . . .
}
```
Introduction to Abstract Classes

- An abstract class can have any number of abstract and/or fully defined methods

- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add `abstract` to its modifier

- A class that has no abstract methods is called a *concrete class*
Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss shape in general, in reality a shape must be a rectangle form or a square form

- An abstract class constructor cannot be used to create an object of the abstract class
  - However, a subclass constructor will include an invocation of the abstract class constructor in the form of `super`
Dynamic Binding and Abstract Classes

• Controlling whether a subclass can override a superclass method
  – Field modifier \texttt{final}
    • Prevents a method from being overridden by a subclass
  – Field modifier \texttt{abstract}
    • Requires the subclass to override the method

• Early binding or static binding
  – The appropriate version of a method is decided at compilation time
  – Used by methods that are \texttt{final} or \texttt{static}
Employee hierarchy UML class diagram.
Example: Inheritance Hierarchy of Class Employee

- Employee
  - fName : string
  - lName : string
  - ssn : string
  + Employee()
  + setFirstName(in fNa : string)
  + setLastName(in lName : string)
  + setSNN(in ssn : string)
  + getFirstName() : string
  + getLastName() : string
  + getSNN() : string
  + earnings() : double
  + toString()

- SalariedEmployee
  - weeklySalary : double
  + earnings() : double
  + toString() : string

- HourlyEmployee
  - wage : double
  - hours : double
  + earnings() : double
  + toString() : string

- CommissionEmployee
  - grossSales : double
  - commissionRate : double
  + earnings() : double
  + toString() : string

- BasePlusCommissionEmployee
  - baseSalary : double
  + earnings()
  + toString()
  + ....()
Implementation **Employee** in Java

**Case Study:**

```java
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

    // 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

    // 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 "The name is :"+ getFirstName()+" "+
                getLastName() + "nThe Social Security Number: "+
                getSocialSecurityNumber() ;
    } // end method toString

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

Case Study:

```java
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;
      // this mean that, if salary is <0 then put it 0 else put it salary
    } // end method setWeeklySalary

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

    // calculate earnings; override abstract method earnings in Employee
    public double earnings() {
      return getWeeklySalary();
    } // end method earnings

    // return String representation of SalariedEmployee object
    // this method override toString() of superclass method
    public String toString() {
      return super.toString() + "earnings = " + getWeeklySalary();
    } // end method toString
}
```
Case Study: **HourlyEmployee** in Java

```java
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 ) code (constructor) reuse
        super(first, last, ssn);
        setWage(hourlyWage); // validate and store hourly wage
        setHours(hoursWorked); // validate and store hours worked
    } // end five-argument HourlyEmployee constructor

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

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

    public void setHours(double hoursWorked) {
        hours = ((hoursWorked >= 0.0) && (hoursWorked <= 168.0)) ? hoursWorked : 0.0;
    } // end method setHours

    public double getHours() {
        return hours;
    } // end method getHours

    // calculate earnings; override abstract method earnings in Employee
    public double earnings() {
        if (getHours() <= 40) // no overtime
            return getWage() * getHours();
        else
            return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
    } // end method earnings

    // return String representation of HourlyEmployee object
    public String toString() { // here overriding the toString() superclass method */
        return (super.toString() + "Hourly wage: " + getWage() + 
                "Hours worked:");
    } // end method toString

} // end class HourlyEmployee
```
Implementation **CommissionEmployee** in Java

**Case Study:**

```java
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

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

    // 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 ( super.toString() + "
Gross sales: " + getGrossSales() + "Commission rate: " + getCommissionRate() + "earnings = " + earnings() );
    } // end method toString
}
```
Implementation **BasePlusCommissionEmployee** in Java

Case Study:

```java
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

    // return base salary
    public double getBaseSalary() {
        return baseSalary;
    } // end method getBaseSalary

    // calculate earnings; override method earnings in CommissionEmployee
    public double earnings() {
        return getBaseSalary() + super.earnings(); // code reuse form CommissionEmployee
    } // end method earnings

    // return String representation of BasePlusCommissionEmployee object
    public String toString() {
        return ("Base-salaried:
            + super.toString()
            + "Base salary: "+ getBaseSalary() + "earnings =" + earnings() );
    } // end method toString
} // end class BasePlusCommissionEmployee
```
public class PayrollSystemTest
{
    public static void main( String args[] )
    {
        // create subclass objects
        SalariedEmployee SA = new SalariedEmployee( "Ali", "Samer", "111-11-1111", 800.00 );
        HourlyEmployee HE = new HourlyEmployee( "Ramzi", "Ibrahim", "222-22-2222", 16.75, 40 );
        CommissionEmployee CE = new CommissionEmployee( "Med", "Ahmed", "333-33-3333", 10000, .06 );
        BasePlusCommissionEmployee BP = new BasePlusCommissionEmployee("Beji", "Lotfi", "444-44-4444", 5000, .04, 300 );

        System.out.println("Employees processed individually:
");
        System.out.println(SA.toString()+"earned: "+SA.earnings()+"\n\n");
        System.out.println(HE + "earned: "+HE.earnings()+"\n\n");
        System.out.println(CE + "earned: "+CE.earnings()+"\n\n");
        System.out.println(BP + "earned: "+BP.earnings()+"\n\n");

        // create four-element Employee array
        Employee employees[]= new Employee[ 4 ];
        System.out.println("Employees processed polymorphically:\n");

        // generically process each element in array employees
        for (Employee currentEmployee : employees) // invokes toString : here is polymorphism : call toString() of class at the execution time.
        {
            System.out.println(currentEmployee); // invokes toString : here is polymorphism : call toString() of class at the execution time.
            // called dynamic binding or late binding
            // Note : only methods of superclass can be called via superclass variable
        }

        // get type name of each object in employees array
        for ( int j = 0; j < employees.length; j++)
        {
            System.out.printf("Employee %d is a %s\n", j, employees[ j ].getClass().getName()); // display the name of the class whose
        // object is employee[j]
        }
    }
}

// end main  } // end class PayrollSystemTest