Abstract Classes and Interfaces

Instructors: Ayça Tarhan, Fuat Akal, Gonenç Ercan, Vahid Garousi

Abstract Classes

- An abstract class is a class that is declared abstract
- An abstract class may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be sub-classed.

Abstract Classes: Revisiting the Shapes

Today

- Abstract Classes
  - Abstract methods
  - Polymorphism with abstract classes
  - Example project: Payroll System
- Interfaces
  - What is an Interface?
  - Defining an Interface
  - Implementing an Interface
  - Implementing Multiple Interfaces
  - Extending a Class and Implementing Interface(s)
  - Extending an Interface
  - Interfaces as Types
- Interfaces vs Abstract Classes
Abstract Classes

- Shapes all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common.
- Some of these states and behaviors are the same for all shapes (for example: position, fill color, and moveTo).
- Others require different implementations (for example, resize or draw).
- All Shapes must be able to draw or resize themselves; they just differ in how they do it.

Abstract Methods

- An abstract method is a method that is declared without an implementation
  - without braces, and followed by a semicolon, like this:
    ```java
    public abstract void draw();
    ```
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.
  - However, if it does not, then the subclass must also be declared abstract.
Abstract Classes

- Are part of the inheritance hierarchy
  - Circle extends Shape
  - Square extends Quadrilateral

- Can have constructor(s), but no objects of these classes can be created
  - Shape shape = new Shape("Shape");
    // compilation error! "Cannot instantiate the type Shape"

- Classes that can be used to instantiate objects are called concrete classes.

Example-1

```
public class Program {
    public static void main(String[] args) {
        Shape shape = new Shape("Shape");
        // compilation error! "Cannot instantiate the type Quadrilateral"
        Quadrilateral quadrilateral = new Quadrilateral("Quadrilateral");
        Square s = new Square("Square", 4);
        s.draw();
        Rectangle r = new Rectangle("Rectangle", 3, 7);
        r.draw();
        RightTriangle t = new RightTriangle("RightTriangle", 5);
        t.draw();
    }
}
```
Example-2

- Imagine there are several instruments, either stringed or wind.
- Design a class hierarchy for only two types of instruments, guitars and flutes.
- You have to design your model in a way that new instruments can be added in the hierarchy later on.
- Imagine there is only one feature for each instrument at the moment, which is the play feature.

```
public abstract class Instrument {
    protected String name;
    abstract public void play();
}

abstract class StringedInstrument extends Instrument {
    protected int numberOfStrings;
}

abstract class WindInstrument extends Instrument {
    //features
}

public class Guitar extends StringedInstrument{
    public void play(){
        System.out.println("Guitar is rocking!");
    }
}

public class Flute extends WindInstrument{
    public void play(){
        System.out.println("Flute is rocking!");
    }
}
```

Example Project: Payroll System
Overview of the classes

<table>
<thead>
<tr>
<th>Employee</th>
<th>earnings</th>
<th>toString</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>firstName lastName&lt;br&gt;social security number: SSN</td>
<td></td>
</tr>
<tr>
<td>Salaried-Employee</td>
<td>weeklySalary&lt;br&gt;salaried employee: firstName lastName&lt;br&gt;social security number: SSN&lt;br&gt;weekly salary: weeklySalary</td>
<td></td>
</tr>
<tr>
<td>Hourly-Employee</td>
<td>if hours &lt;= 40&lt;br&gt;wage = hours&lt;br&gt;else&lt;br&gt;if hours &gt; 40&lt;br&gt;40 * wage + ( hours - 40 ) * wage = 1.5&lt;br&gt;hourly employee: firstName lastName&lt;br&gt;social security number: SSN&lt;br&gt;hourly wage: wage; hours worked: hours</td>
<td></td>
</tr>
<tr>
<td>Commission-Employee</td>
<td>commissionRate * grossSales&lt;br&gt;commission employee: firstName lastName&lt;br&gt;social security number: SSN&lt;br&gt;gross sales: grossSales&lt;br&gt;commission rate: commissionRate</td>
<td></td>
</tr>
<tr>
<td>BasePlus-Commission-Employee</td>
<td>( commissionRate * grossSales ) + baseSalary&lt;br&gt;base salaried commission employee: firstName lastName&lt;br&gt;social security number: SSN&lt;br&gt;gross sales: grossSales&lt;br&gt;commission rate: commissionRate&lt;br&gt;base salary: baseSalary</td>
<td></td>
</tr>
</tbody>
</table>

Employee.java (1)

```java
public abstract class Employee {
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;
    public Employee(String firstName, String lastName, String ssn) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.socialSecurityNumber = ssn;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return String.format("%s %s %s", firstName, lastName, socialSecurityNumber);
    }
}
```

Employee.java (2)

```java
// return last name
public String getLastName() {
    return lastName;
}

// set social security number
public void setSocialSecurityNumber(String ssn) {
    socialSecurityNumber = ssn; // should validate
}

// return social security number
public String getSocialSecurityNumber() {
    return socialSecurityNumber;
}

// return string representation of Employee object
public String toString() {
    return String.format("%s %s %s", getFirstName(), getLastName(), getSocialSecurityNumber());
}
```

SalariedEmployee.java

```java
public class SalariedEmployee extends Employee {
    private double weeklySalary;
    public SalariedEmployee(String firstName, String lastName, String ssn, double salary) {
        super(firstName, lastName, ssn);
        weeklySalary = salary;
    }

    public double getWeeklySalary() {
        return weeklySalary;
    }

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

Earnings will be calculated in subclasses.

Overridden methods.
HourlyEmployee.java (1)

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

    public HourlyEmployee(String firstName, String lastName, String ssn,
                           double hourlyWage, double hoursWorked)
    {
        super(firstName, lastName, 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;
    } // end method setWage

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

    // set hours worked
    public void setHours(double hoursWorked)
    {
        hoursWorked = hoursWorked;
        if (hoursWorked > 0.0 && hoursWorked <= 160.0)
            dailyWage = (hoursWorked - 40) * (hourlyWage - 42.5f) + 42.5f;
    } // end method setHours

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

    // calculate earnings; override abstract method earnings in Employee
    public double earnings()
    {
        if (getHours() > 40) // overtime
            return getWage() * getHours() / 40;
        else
            return 40 * getWage() + (getHours() - 40) * (getWage() - 42.5f);
    } // end method earnings

    // return String representation of HourlyEmployee object
    public String toString()
    {
        return String.format("%nHourly employee: %s %s, %s%n%s, %s, %s, %s, %s, %s%n %nhourly wage: %s", firstName,
                             lastName, ssn, dailyWage,
                             getWage(), getHours(), earnings(),
                             "hours worked", getHours(),
                             "reimbursement String toString()",
                             String.format("%nhourly employee: %s %s, %s%n%s, %s, %s, %s, %s, %s%n %nhourly wage: %s", firstName,
                                 lastName, ssn, dailyWage,
                                 getWage(), getHours(), earnings(),
                                 "hours worked", getHours(),
                                 "reimbursement String toString()",
                                 String.format("%n"));
    } // end class HourlyEmployee
```

HourlyEmployee.java (2)

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

// calculate earnings; override abstract method earnings in Employee
public double earnings()
{
    if (getHours() > 40) // overtime
        return getWage() * getHours() / 40;
    else
        return 40 * getWage() + (getHours() - 40) * (getWage() - 42.5f);
} // end method earnings

// return String representation of HourlyEmployee object
public String toString()
{
    return String.format("%nHourly employee: %s %s, %s%n%s, %s, %s, %s, %s, %s%n %nhourly wage: %s", firstName,
                        lastName, ssn, dailyWage,
                        getWage(), getHours(), earnings(),
                        "hours worked", getHours(),
                        "reimbursement String toString()",
                        String.format("%n"));
} // end class HourlyEmployee
```

CommissionEmployee.java (1)

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

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

    // set commission rate
    public void setCommissionRate(double rate)
    {
        commissionRate = rate;
    } // end method setCommissionRate

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

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

    // 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 String.format("%nCommission employee: %s %s, %s%n%s, %s, %s, %s, %s, %s%n %ncommission rate", firstName,
                             lastName, ssn, dailyWage,
                             getWage(), getHours(), earnings(),
                             "hours worked", getHours(),
                             "reimbursement String toString()",
                             String.format("%n"));
    } // end class CommissionEmployee
```

CommissionEmployee.java (2)

```java
// 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 String.format("%nCommission employee: %s %s, %s%n%s, %s, %s, %s, %s, %s%n %ncommission rate", firstName,
                          lastName, ssn, dailyWage,
                          getWage(), getHours(), earnings(),
                          "hours worked", getHours(),
                          "reimbursement String toString()",
                          String.format("%n"));
} // end class CommissionEmployee
```
BasePlusCommissionEmployee.java

```java
public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary; // base salary per week
    // six-argument constructor
    public BasePlusCommissionEmployee(String firstName, String lastName, String socialSecurityNumber, double baseSalary, double salesAmount, double salesRate) {
        super(firstName, lastName, socialSecurityNumber, salesAmount, salesRate); // validate and store salesAmount
        this.baseSalary = baseSalary; // validate and store baseSalary
        // end six-argument BasePlusCommissionEmployee constructor
    }
    // get baseSalary
    public double getBaseSalary() {
        return baseSalary;
    }
    // return baseSalary
    public double getBaseSalary() {
        return baseSalary;
    }
    // override method earnings in CommissionEmployee
    public double getEarnings() {
        return baseSalary + super.getEarnings();
    }
    // return string representation of BasePlusCommissionEmployee object
    public String toString() {
        return String.format("%s %s: base-salary: $%,.2f,\n        ", firstName, lastName, baseSalary);
    }
}
```

PayrollSystemTest.java

```java
public class PayrollSystemTest {
    public static void main(String[] args) {
        // create subclass objects
        BasePlusCommissionEmployee john = new BasePlusCommissionEmployee("John", "Smith", "111-11-1111", 800.00);
        HourlyEmployee tina = new HourlyEmployee("Tina", " Jones", 2222.22, 16.75);
        CommissionEmployee maria = new CommissionEmployee("Maria", " During", 2222.22, 0.09);
        // create four-element Employee array
        Employee[] employees = new Employee[4];
        employees[0] = john;
        employees[1] = tina;
        employees[2] = maria;
```

PayrollSystemTest.java (2)

```java
        System.out.println("Employees processed individually:\n        ");
        for (Employee currentEmployee : employees) {
            System.out.println(currentEmployee + "\n            ");
        }
        // determine whether employee is a BasePlusCommissionEmployee
        for (Employee currentEmployee : employees) {
            EmployeeReference employeeReference = new EmployeeReference(currentEmployee);
            System.out.println("\n            BasePlusCommissionEmployee employee = \n            ");
            double newBaseSalary = employee.getBaseSalary();
            employee.setBaseSalary(1.10 * newBaseSalary);
            System.out.println("new base salary with 10% increase is: $%,.2f\n            ");
```

Interfaces

```java
GUI
LCD/LED TV
Laptop
```
Concept of Interface

- An interface is a **contract**. It guarantees that the system will have certain functionalities.
- An interface is an integration point between two systems.
- A system can have many interfaces, so it can be integrated to many other systems.

Defining an Interface

- **Keyword interface** is used to define an interface
- Methods in an interface must be **public** and **abstract**, these keywords are commonly omitted
- Interfaces can include **public static final** variables (constants), these keywords are commonly omitted

```java
public interface Shape {
    public abstract void draw();
    public static final double PI = 3.14;
}
```

Implementing an Interface

- An interface is implemented by the keyword **implements**
- Any class implementing an interface must either implement all methods of it, or be declared abstract

```java
public class RightTriangle implements Shape {
    // ....
    public void draw() {
        for (int line = 1; line <= a; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Implementing Multiple Interfaces

- More than one interface can be implemented by a class.
- Names of interfaces are separated by comma

```java
public class LedTv implements Usb, Hdmi, Scart, Vga {
    // ....
}
```

Question: What if at least two interfaces include the same method definition?
### Extending a Class and Implementing Interface(s)

```java
public class Car extends Vehicle
    implements Shape {
    public void draw() {
        // ....
    }
}
```

### Extending an Interface

- It is possible for an interface to extend another interface

```java
public interface I1 {
    void m1();
}

public class C2 implements I2 {
    public void m1() {
        // ...
    }
    public void m2() {
        // ...
    }
}

public interface I2 extends I1 {
    void m2();
}

public class C1 implements I1 {
    public void m1() {
        // ...
    }
}
```

### Interfaces as Types

- When you define a new interface, you are defining a new reference data type.

- You can use interface names anywhere you can use any other data type name.

- If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

### Interfaces as Types

```java
public class Program {
    public static void main(String[] args) {
        Shape shape;
        shape = new Square(4);
        shape.draw();
        shape = new Rectangle(3, 7);
        shape.draw();
        shape = new RightTriangle(5);
        shape.draw();
    }
}
```

```java
public class Program {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Square(5);
        shapes[1] = new Rectangle(2, 8);
        shapes[2] = new RightTriangle(3);
        for (Shape s : shapes) {
            drawIt(s);
        }
    }
}
```

```java
public class Program {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        for (Shape s : shapes) {
            drawIt(s);
        }
    }
}
```
Example Project: Payroll System Revisited

Interface implementation symbol

Invoice

Employee

SalariedEmployee

Payable.java

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

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

Invoice.java (1)

```java
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;
  }

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

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

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

  // calculate total cost
  public double getTotalCost() {
    return getQuantity() * getPricePerItem(); // calculate total cost
  }
}
```

Invoice.java (2)

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

// end method setQuantity

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

// end method getQuantity

// 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
```

Payable.java
Employee.java

```java
5 public abstract class Employee implements Payable {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     // three-argument constructor
10     public Employee(String first, String last, String ssn) {
11         firstName = first;
12         lastName = last;
13         socialSecurityNumber = ssn;
14     } // end three-argument Employee constructor
15     // Rest of the class is same as the previous example except there is no earnings() method! */
```

Payable interface includes getPaymentAmount() method, but Employee class does not implement it!

PayableInterfaceTest.java

```java
4 public class PayableInterfaceTest {
5     public static void main(String args[]) {
6         // create four-element Payable array
7         Payable[] payables = new Payable[4];
8         // populate array with objects that implement Payable
9         payables[0] = new Invoice("01234", "test", 2, 375.00);
10        payables[1] = new Invoice("56789", "test", 1, 199.25);
11        payables[2] = new SalariedEmployee("John", "Smith", "111-11-1111", 800.00);
12        payables[3] = new SalariedEmployee("Lisa", "Barnes", "000-88-8888", 1200.00);
13        System.out.println("Invoices and Employees processed polymorphically:
14         " + System.currentTimeMillis() + "\n");
15        // generically process each element in array payables
16        for (Payable payee : payables) {
17            // output currentPayable and its appropriate payment amount
18            System.out.println("=" + payee.getInvoices().size() + "; Payable: 
19            " + payee.getClass().getName() + "; payment due: " + payee.getPaymentAmount());
20        } // end for
21    } // end main
22    } // end class PayableInterfaceTest
```

SalariedEmployee.java

```java
10 public class SalariedEmployee extends Employee {
11     public SalariedEmployee(String first, String last, String ssn, double salary) {
12         super(first, last, ssn); // pass to Employee constructor
13         setWeeklySalary(salary); // validate and store salary
14     } // end four-argument SalariedEmployee constructor
15     // set salary
16     public void setWeeklySalary(double salary) {
17         weeklySalary = salary;
18     } // end method setWeeklySalary
19     // return salary
20     public double getWeeklySalary() {
21         return weeklySalary;
22     } // end method getWeeklySalary
23     // calculate earnings; represent interface Payable method that was abstract in superclass Employee
24     public double getPaymentAmount() {
25         return getWeeklySalary();
26     } // end method getPaymentAmount
27     // return String representation of SalariedEmployee object
28     public String toString() {
29         return String.format("SalariedEmployee: Salaries: $%.2f",
30             super.toString(), weeklySalary, getWeeklySalary());
31     } // end method toString
32 } // end class SalariedEmployee
```

Interfaces vs Abstract Classes

<table>
<thead>
<tr>
<th>Abstract class</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Abstract class can have abstract and non-abstract methods.</td>
<td>Interface can have only abstract methods.</td>
</tr>
<tr>
<td>2) Abstract class doesn't support multiple inheritance.</td>
<td>Interface supports multiple inheritance.</td>
</tr>
<tr>
<td>3) Abstract class can have final, non-final, static and non-static variables.</td>
<td>Interface has only static and final variables.</td>
</tr>
<tr>
<td>4) Abstract class can have static methods, main method and constructor.</td>
<td>Interface can't have static methods, main method or constructor.</td>
</tr>
<tr>
<td>5) Abstract class can provide the implementation of interface.</td>
<td>Interface can't provide the implementation of abstract class.</td>
</tr>
<tr>
<td>6) The abstract keyword is used to declare abstract class.</td>
<td>The Interface keyword is used to declare interface.</td>
</tr>
</tbody>
</table>

Example:
```java
public abstract class Shape {
    public abstract void draw();
} 
```
Summary

- Abstract class is defined with the keyword `abstract`
- If a class includes an abstract method, it must be declared as `abstract`
- Objects of abstract classes cannot be created
- Interface is defined with the keyword `interface`
- A class can `implement` an interface, an interface can `extend` an interface
- A class can implement many interfaces
- Objects of interfaces cannot be created

Acknowledgements

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012