BBM 102 – Introduction to Programming II

Spring 2017

Abstract Classes and Interfaces

Instructors: *Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi*

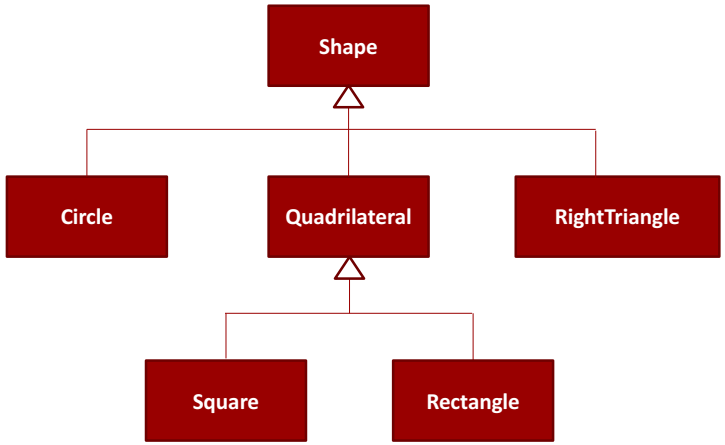
Today

- 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

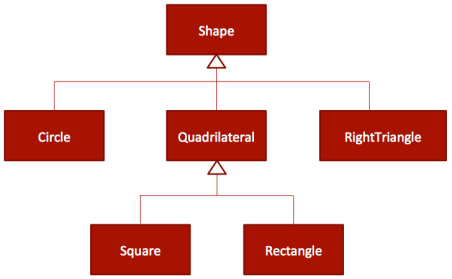
- An *abstract class* is a class that is declared **abstract**
- An *abstract class* may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.

Abstract Classes: Revisiting the Shapes



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 Classes

```
public class Shape {
    private String name;

    public Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void draw() {
        // what is the shape?
        // Code...?! Nothing!
    }
}
```

```
public abstract class Shape {
    private String name;

    public Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void draw();
}
```


Abstract Methods

- An *abstract method* is a method that is declared without an implementation
  - without braces, and followed by a semicolon, like this:

```
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

```
public class RightTriangle extends Shape {
    private int a;

    public RightTriangle(String name, int a) {
        super(name);
        this.a = a;
    }

    public int getA() {
        return a;
    }
    // override abstract method
    public void draw() {
        for (int line = 1; line <= a; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Abstract Classes

```
public abstract class Quadrilateral
                    extends Shape {

    public Quadrilateral(String name) {
            super(name);
    }

    // still nothing to draw!
    public abstract void draw();
}
```

```
public class Square extends Quadrilateral {
    private int a;

    public Square(String name, int a) {
            super(name);
            this.a = a;
    }
    public int getA() {
            return a;
    }

    // override abstract method
    public void draw() {
            for (int line = 0; line < a; line++) {
                for (int col = 0; col < a; col++) {
                        System.out.print("*");
                }
                System.out.println();
            }
    }
}
```

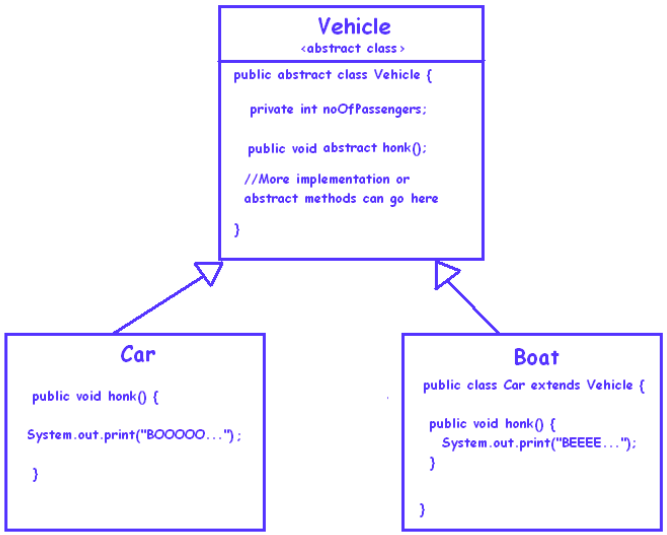

Abstract Classes

```
public class Program {

    public static void main(String[] args) {
            // compilation error!: "Cannot instantiate the type Shape"
            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();
    }
}
```


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



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.

Example-2

Abstract class

```
public abstract class Instrument {

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

Still abstract

```
abstract class StringedInstrument extends Instrument {
    protected int numberOfStrings;
}
```

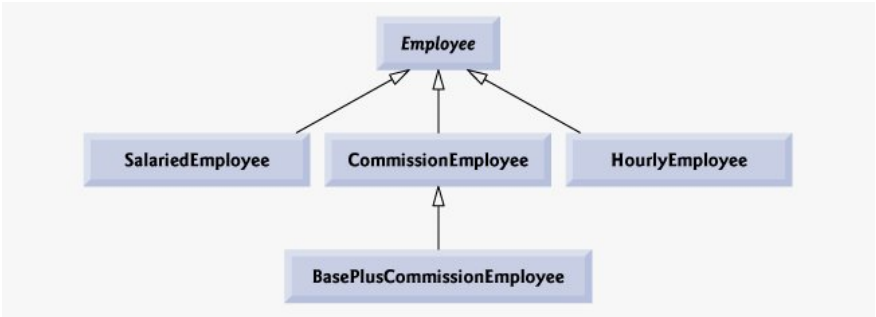
```
public class Guitar extends StringedInstrument{

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


Example-2

```
abstract class WindInstrument extends Instrument {
    //features
}
```

```
public class Flute extends WindInstrument{

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


Example Project: Payroll System



Overview of the 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.java (1)

```
1  // Fig. 10.4: 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
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
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
```


Employee.java (2)

```
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
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
```

Earnings will be calculated in subclasses

SalariedEmployee.java

```
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
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
```

Overridden methods

HourlyEmployee.java (1)

```
4  public class HourlyEmployee extends Employee
5  {
6     private double wage; // wage per hour
7     private double hours; // hours worked for week
8
9     // five-argument constructor
10    public HourlyEmployee( String first, String last, String ssn,
11       double hourlyWage, double hoursWorked )
12    {
13       super( first, last, ssn );
14       setWage( hourlyWage ); // validate hourly wage
15       setHours( hoursWorked ); // validate hours worked
16    } // end five-argument HourlyEmployee constructor
17
18    // set wage
19    public void setWage( double hourlyWage )
20    {
21       wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22    } // end method setWage
23
24    // return wage
25    public double getWage()
26    {
27       return wage;
28    } // end method getWage
29
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
```


HourlyEmployee.java (2)

```
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
```


CommissionEmployee.java (1)

```
4  public class CommissionEmployee extends Employee
5  {
6     private double grossSales; // gross weekly sales
7     private double commissionRate; // commission percentage
8
9     // five-argument constructor
10    public CommissionEmployee( String first, String last, String ssn,
11       double sales, double rate )
12    {
13       super( first, last, ssn );
14       setGrossSales( sales );
15       setCommissionRate( rate );
16    } // end five-argument CommissionEmployee constructor
17
18    // set commission rate
19    public void setCommissionRate( double rate )
20    {
21       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22    } // end method setCommissionRate
23
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
```


CommissionEmployee.java (2)

```
36    // return gross sales amount
37    public double getGrossSales()
38    {
39       return grossSales;
40    } // end method getGrossSales
41
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
```

BasePlusCommissionEmployee.java

```
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
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
```


PayrollSystemTest.java (1)

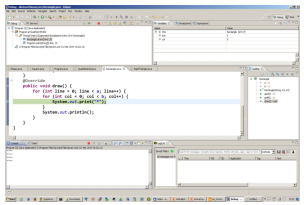
```
4  public class PayrollSystemTest
5  {
6     public static void main( String args[] )
7     {
8        // create subclass objects
9        SalariedEmployee salariedEmployee =
10          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11       HourlyEmployee hourlyEmployee =
12          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
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;
```


PayrollSystemTest.java (2)

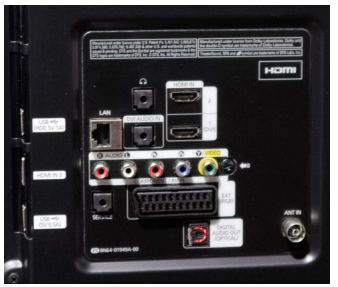
```
41       System.out.println( "Employees processed polymorphically:\n" );
42
43       // generically process each element in array employees
44       for ( Employee currentEmployee : employees )
45       {
46          System.out.println( currentEmployee ); // invokes toString
47
48          // determine whether element is a BasePlusCommissionEmployee
49          if ( currentEmployee instanceof BasePlusCommissionEmployee )
50          {
51             // downcast Employee reference to
52             // BasePlusCommissionEmployee reference
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
```


Interfaces

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

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

No need to write

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

```
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

```
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)

```
public class Car extends Vehicle
                 implements Shape {

    public void draw() {
        // ....
    }
}
```


Extending an Interface

- It is possible for an interface to extend another interface

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

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

```
public class C1 implements I1 {

    public void m1() {
        // ...
    }
}
```

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


Interfaces as Types

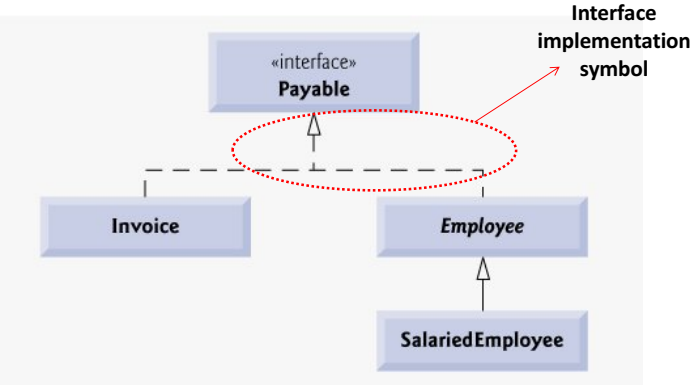
- When you define a new interface, you are defining a new reference data type.
- You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

Interfaces as Types

```
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();
    }
}
```

```
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);
        }
    }

    public static void drawIt(Shape s) {
        s.draw();
    }
}
```

Example Project: Payroll System Revisited



Payable.java

```
1  // Fig. 10.11: Payable.java
2  // Payable interface declaration.
3
4  public interface Payable
5  {
6     double getPaymentAmount(); // calculate payment; no implementation
7  } // end interface Payable
```


Invoice.java (1)

```
4  public class Invoice implements Payable
5  {
6     private String partNumber;
7     private String partDescription;
8     private int quantity;
9     private double pricePerItem;
10
11    // four-argument constructor
12    public Invoice( String part, String description, int count,
13       double price )
14    {
15       partNumber = part;
16       partDescription = description;
17       setQuantity( count ); // validate and store quantity
18       setPricePerItem( price ); // validate and store price per item
19    } // end four-argument Invoice constructor
20
21    // set part number
22    public void setPartNumber( String part )
23    {
24       partNumber = part;
25    } // end method setPartNumber
26
27    // get part number
28    public String getPartNumber()
29    {
30       return partNumber;
31    } // end method getPartNumber
32
33    // set description
34    public void setPartDescription( String description )
35    {
36       partDescription = description;
37    } // end method setPartDescription
38
39    // get description
40    public String getPartDescription()
41    {
42       return partDescription;
43    } // end method getPartDescription
```


Invoice.java (2)

```
45    // set quantity
46    public void setQuantity( int count )
47    {
48       quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
49    } // end method setQuantity
50
51    // get quantity
52    public int getQuantity()
53    {
54       return quantity;
55    } // end method getQuantity
56
57    // set price per item
58    public void setPricePerItem( double price )
59    {
60       pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
61    } // end method setPricePerItem
62
63    // get price per item
64    public double getPricePerItem()
65    {
66       return pricePerItem;
67    } // end method getPricePerItem
68
69    // return String representation of Invoice object
70    public String toString()
71    {
72       return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
73          "invoice", "part number", getPartNumber(), getPartDescription(),
74          "quantity", getQuantity(), "price per item", getPricePerItem() );
75    } // end method toString
76
77    // method required to carry out contract with interface Payable
78    public double getPaymentAmount()
79    {
80       return getQuantity() * getPricePerItem(); // calculate total cost
81    } // end method getPaymentAmount
82 } // end class Invoice
```

Employee.java

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

```
4  public abstract class Employee implements Payable
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
18       /* Rest of the class is same as the previous example
19       except there is no earnings() method! */
20
21
22
23
```


SalariedEmployee.java

```
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
22    // return salary
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // end method getWeeklySalary
27
28    // calculate earnings; implement interface Payable method that was
29    // abstract in superclass Employee
30    public double getPaymentAmount()
31    {
32       return getWeeklySalary();
33    } // end method getPaymentAmount
34
35    // return String representation of SalariedEmployee object
36    public String toString()
37    {
38       return String.format( "salaried employee: %s\n%s: $%,.2f",
39          super.toString(), "weekly salary", getWeeklySalary() );
40    } // end method toString
41 } // end class SalariedEmployee
```


PayableInterfaceTest.java

```
4  public class PayableInterfaceTest
5  {
6     public static void main( String args[] )
7     {
8        // create four-element Payable array
9        Payable payableObjects[] = new Payable[ 4 ];
10
11       // populate array with objects that implement Payable
12       payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13       payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14       payableObjects[ 2 ] =
15          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16       payableObjects[ 3 ] =
17          new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19       System.out.println(
20          "Invoices and Employees processed polymorphically:\n" );
21
22       // generically process each element in array payableObjects
23       for ( Payable currentPayable : payableObjects )
24       {
25          // output currentPayable and its appropriate payment amount
26          System.out.printf( "%s \n%s: $%,.2f\n\n",
27             currentPayable.toString(),
28             "payment due", currentPayable.getPaymentAmount() );
29       } // end for
30    } // end main
31 } // end class PayableInterfaceTest
```


Interfaces vs Abstract Classes

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

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	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
- http://www.javatpoint.com/difference-between-abstract-class-and-interface