BBM 102 – Introduction to Programming II

Spring 2017

Abstract Classes and Interfaces

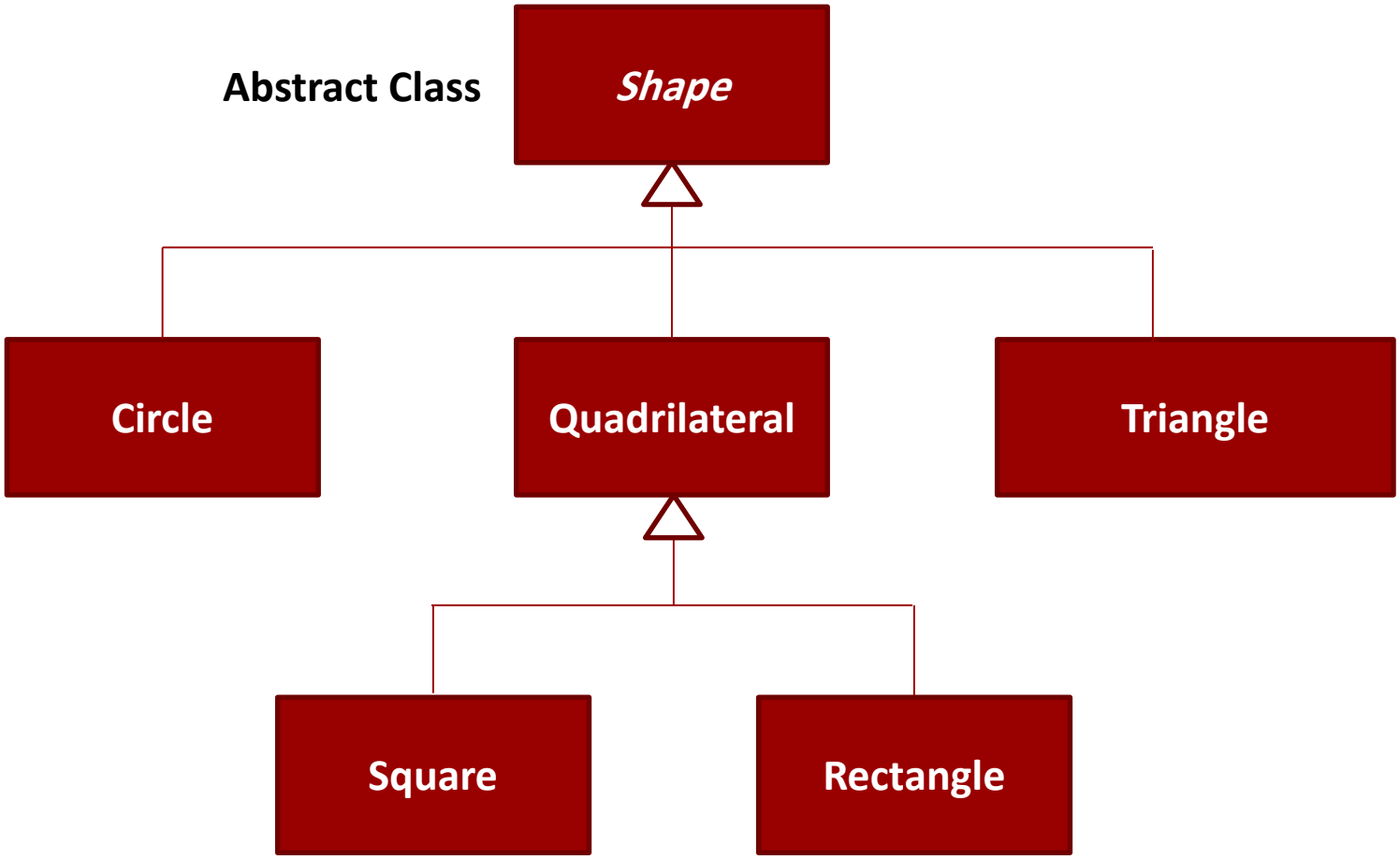
Today

- 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

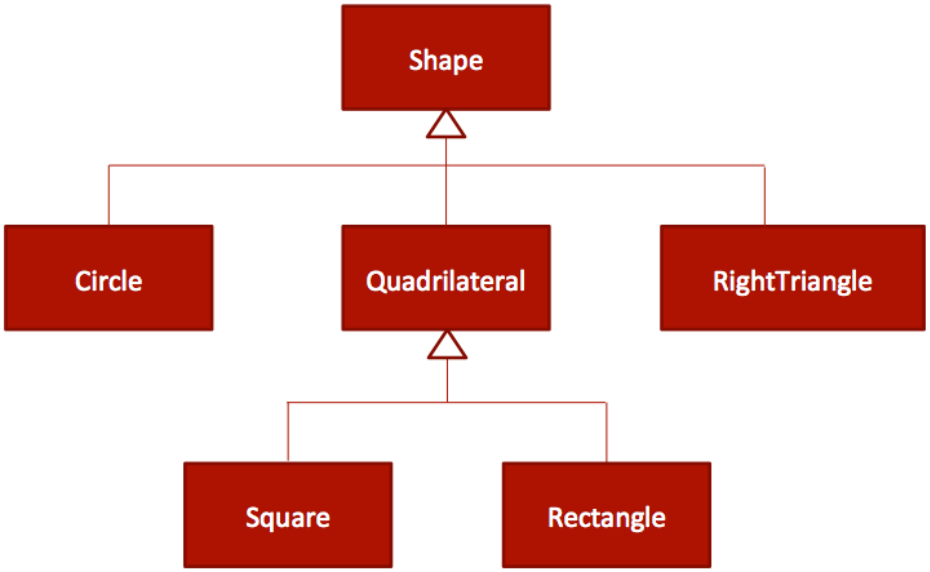
- An *abstract class* is a class that is declared **abstract**
- An *abstract class* may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.

Abstract Classes: Revisiting the Shapes



Abstract Classes

- All shapes have certain attributes (e.g.: position, orientation, line color, fill color) and behaviors (e.g.: moveTo, rotate, resize, draw) in common.
- Some of these attributes and behaviors are the same for all shapes (e.g. : position, fill color, and moveTo).
- Others require different implementations (e.g., 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();
    // sub-classes will define it
}
```

Abstract Methods

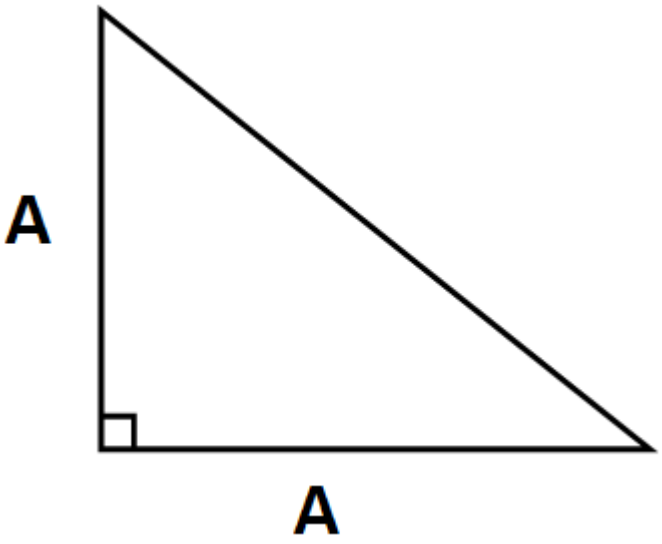
- An *abstract method* is a method that is declared without an implementation
  - without braces, and followed by a semicolon, like this:

```
public abstract void draw();
```

- When an abstract class is subclassed, the subclass should provide implementations for all of the abstract methods in its parent class.
  - However, if they do not, then those subclasses must also be declared abstract.

Abstract Classes

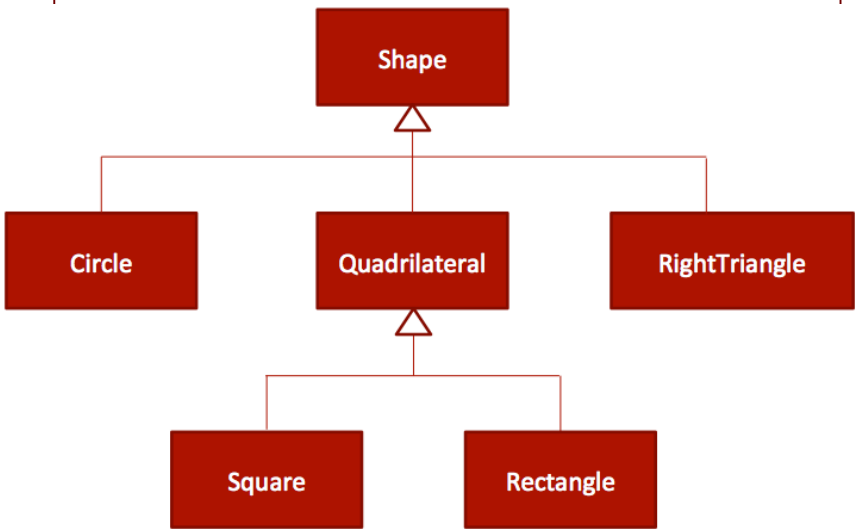
```
public class RightTriangle extends Shape {
    private int a;

    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

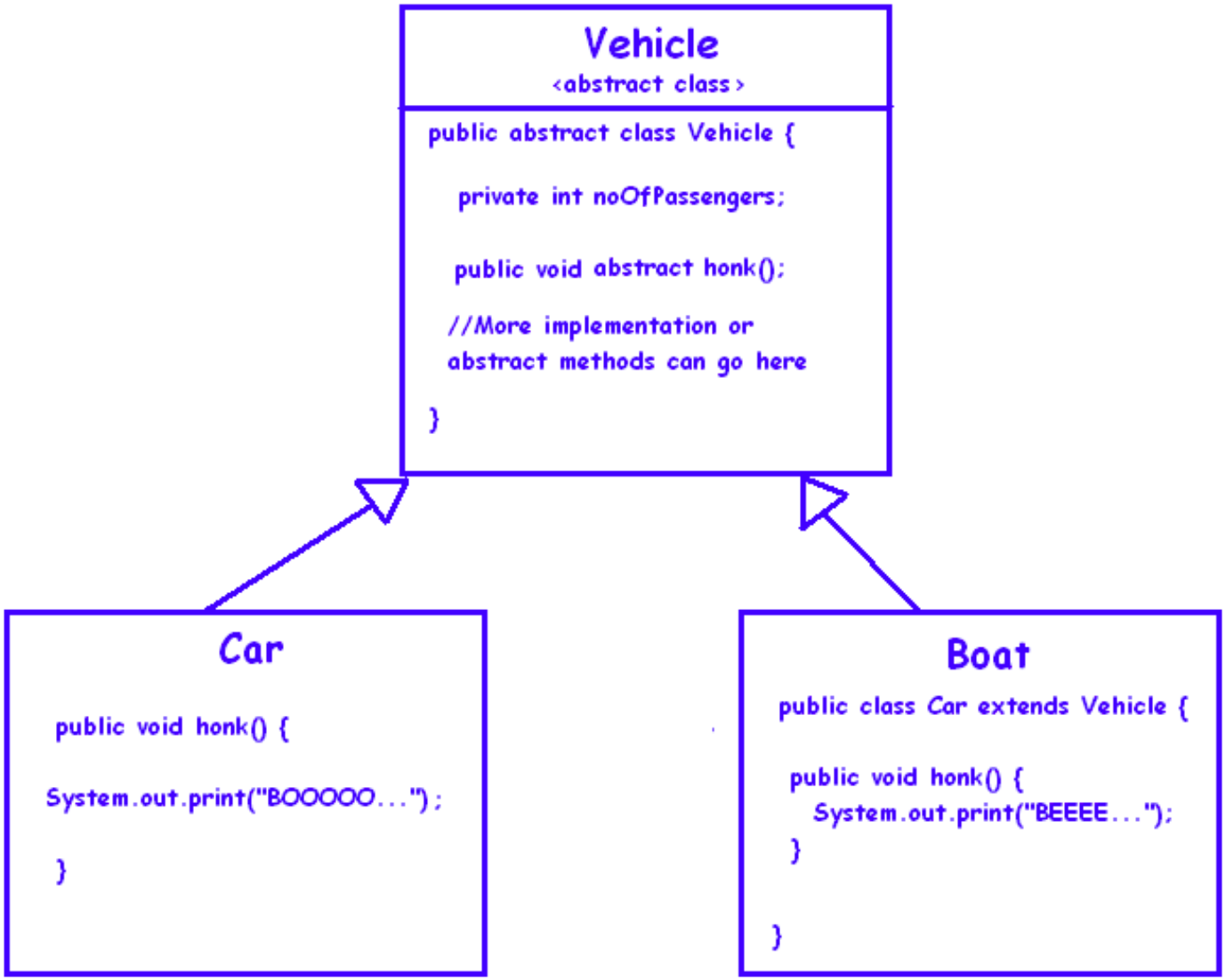
  Square extends Quadrilateral

- 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“
  ```

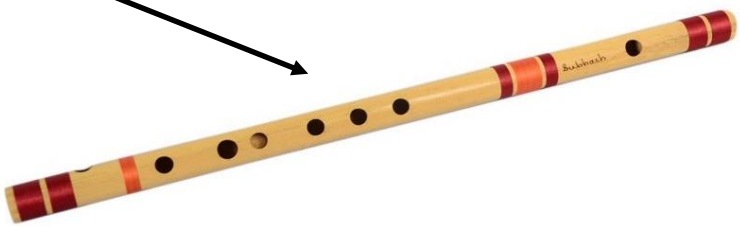
- Classes that can be used to instantiate objects are called **concrete classes**.

Example-1


Vehicle
<abstract class>
public abstract class Vehicle {
private int noOfPassengers;
public void abstract honk();
//More implementation or
abstract methods can go here
}
Car
public void honk() {
System.out.print("BOOOOO...");
}
Boat
public class Car extends Vehicle {
public void honk() {
System.out.print("BEEEE...");
}
}

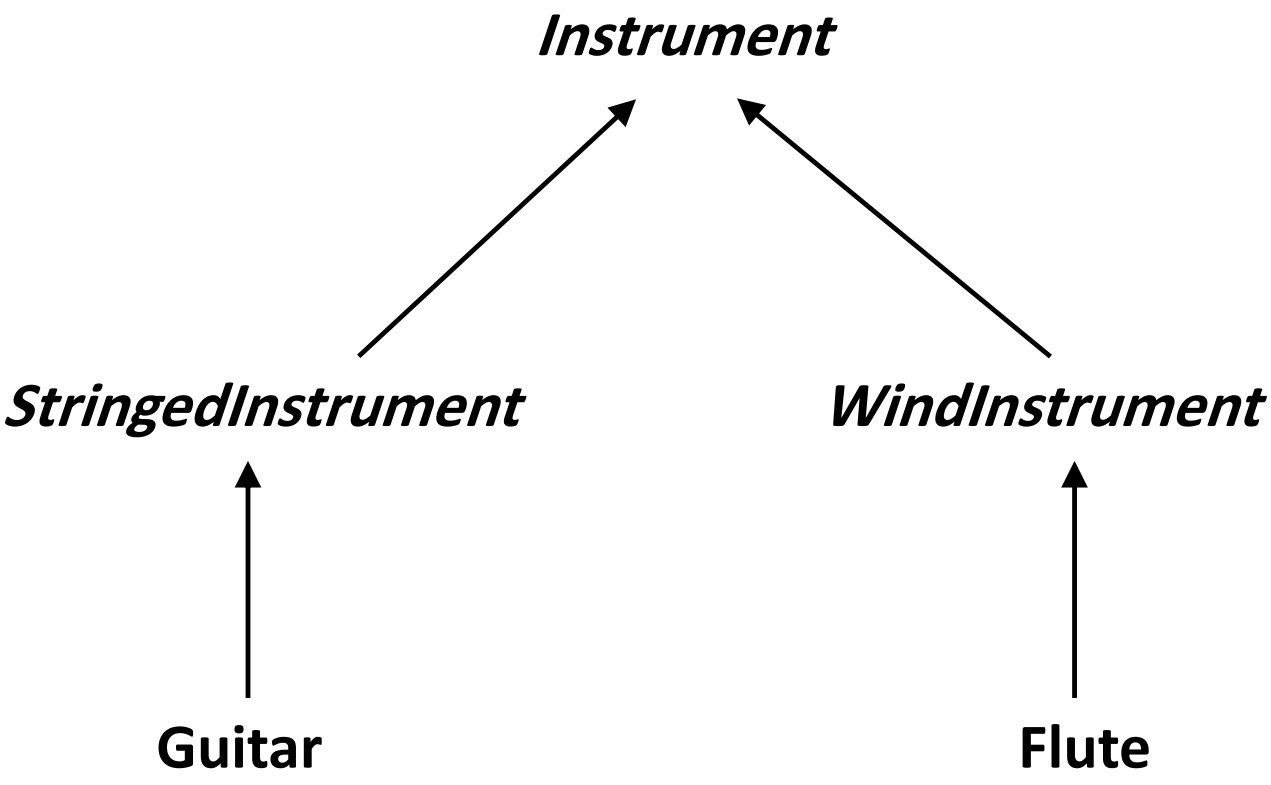
Example-2: music instruments

- 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: music instruments



Example-2: music instruments

Abstract class

```
public abstract class Instrument {

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

Still abstract

```
abstract public class StringedInstrument extends Instrument {
    private int numberOfStrings;
}
```

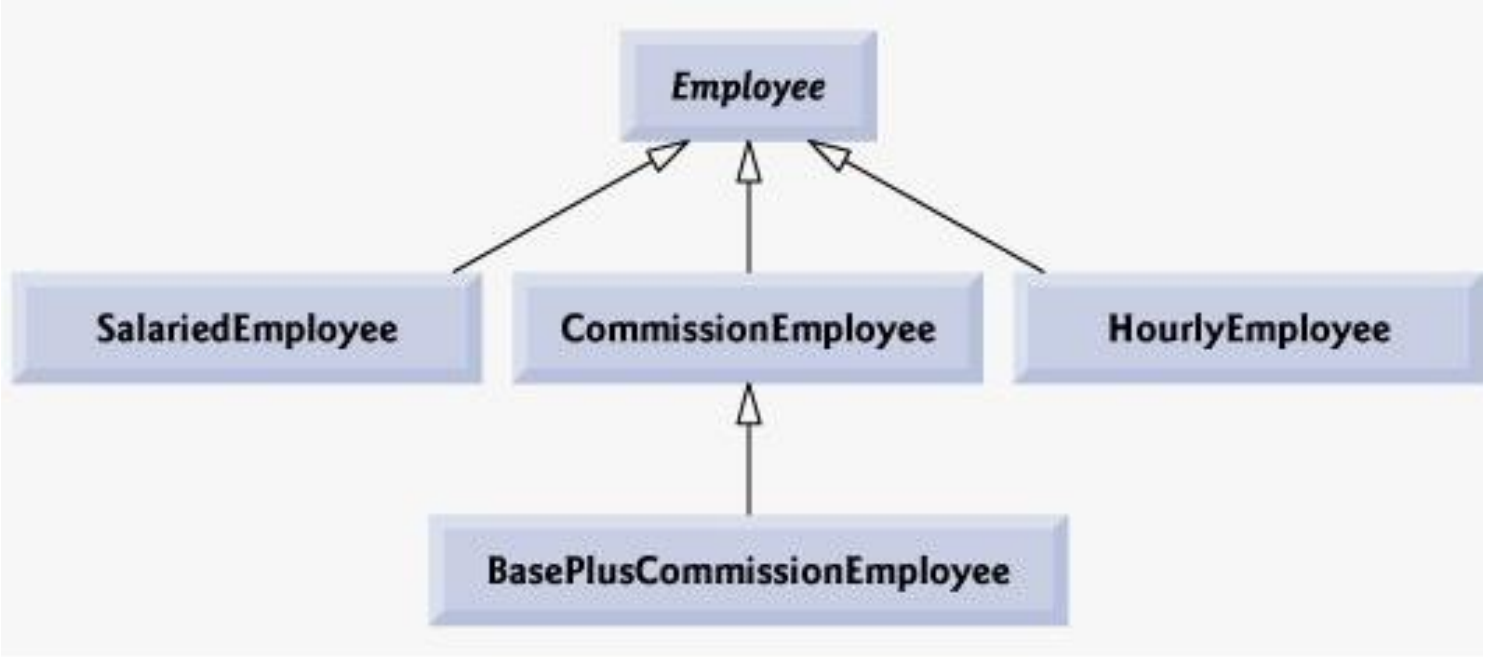
```
public class Guitar extends StringedInstrument{

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

Example-2: music instruments

```
publlic 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)

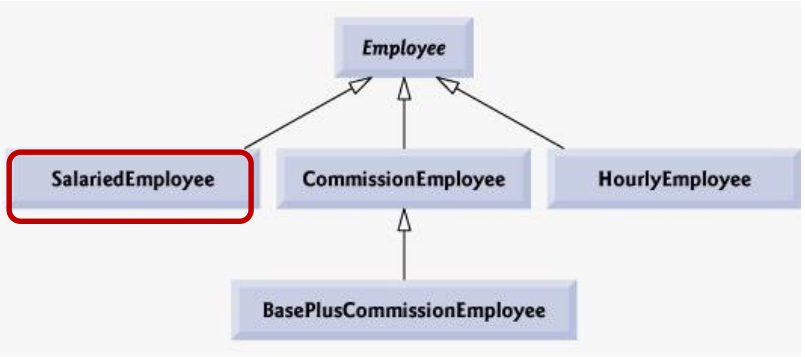
```java
// Fig. 10.4: Employee.java
// Employee abstract superclass.

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

```

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

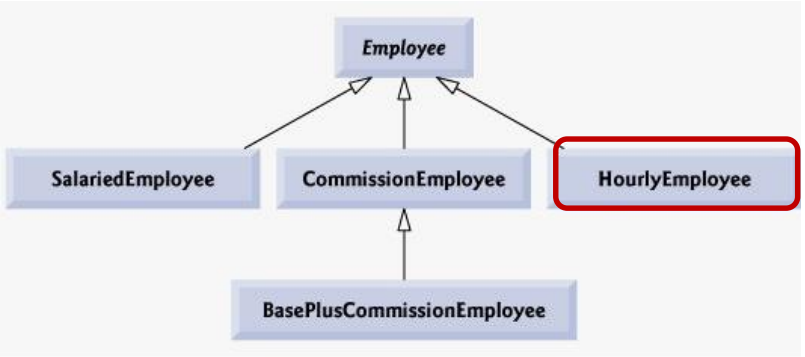
SalariedEmployee.java




```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;
   } // 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
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Overridden methods

HourlyEmployee.java (1)




```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 );
      setWage( hourlyWage ); // validate hourly wage
      setHours( hoursWorked ); // validate hours worked
   } // end five-argument HourlyEmployee constructor

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

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

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

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 ) // 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()
    {
       return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
          super.toString(), "hourly wage", getWage(),
          "hours worked", getHours() );
    } // end method toString
 } // end class HourlyEmployee
```

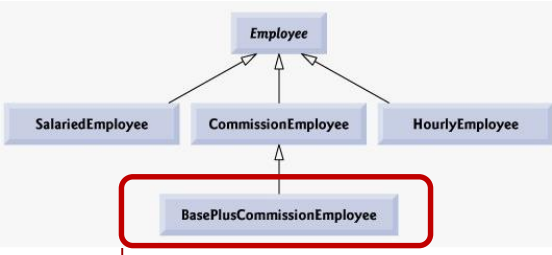
CommissionEmployee.jav


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

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales
```

CommissionEmployee.java (2)

```
    // 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( "%s: %s\n%s: $%,.2f; %s: %.2f",
          "commission employee", super.toString(),
          "gross sales", getGrossSales(),
          "commission rate", getCommissionRate() );
    } // end method toString
 } // end class CommissionEmployee
```

BasePlusCommissionEmployee.java




```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();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee object
   public String toString()
   {
      return String.format( "%s %s; %s: $%,.2f",
         "base-salaried", super.toString(),
         "base salary", getBaseSalary() );
   } // end method toString
} // 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)

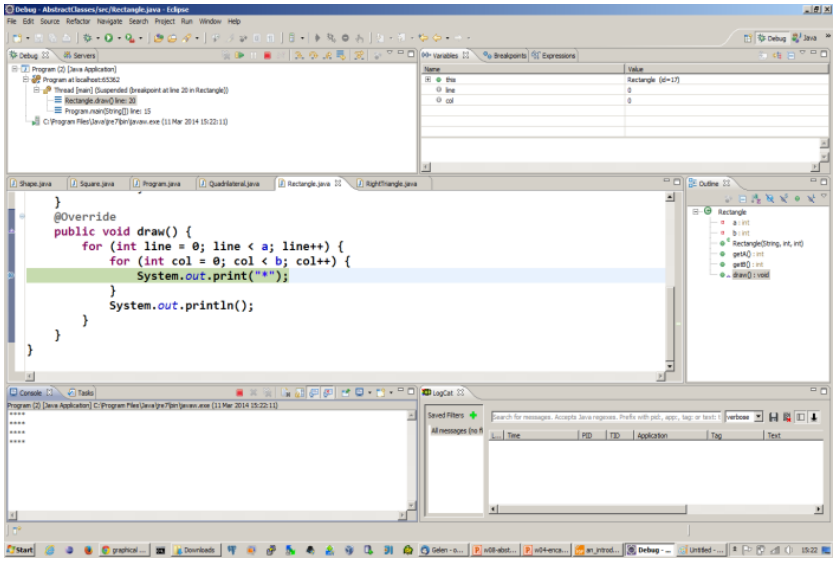
```
41         System.out.println( "Employees processed polymorphically:\n" );
42
43         // generically process each element in array employees
44         for ( Employee currentEmployee : employees )
45         {
46            System.out.println( currentEmployee ); // invokes toString
47
48            // 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
```

Today

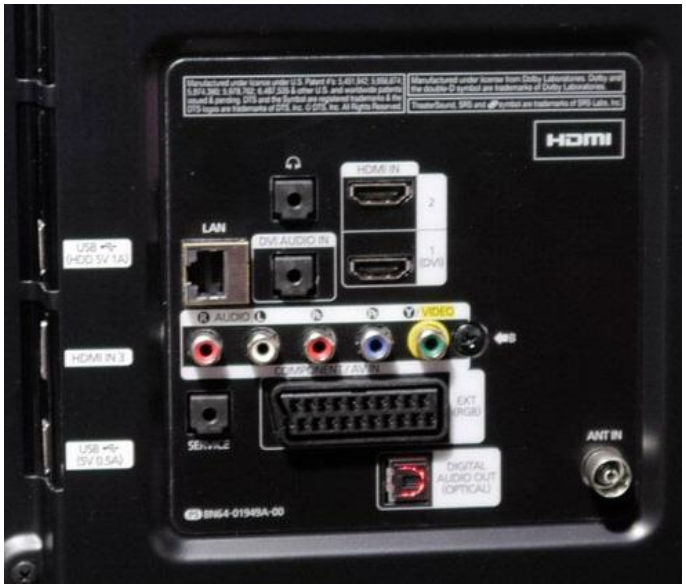
- Abstract Classes
  - Abstract methods
  - Polymorphism with abstract classes
  - Example project: Payroll System
- Interfaces
  - What is an Interface?
  - Defining an Interface
  - Implementing an Interface
  - Implementing Multiple Interfaces
  - Extending a Class and Implementing Interface(s)
 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

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 in Java

- 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, Vga {

    // .....

}
```

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

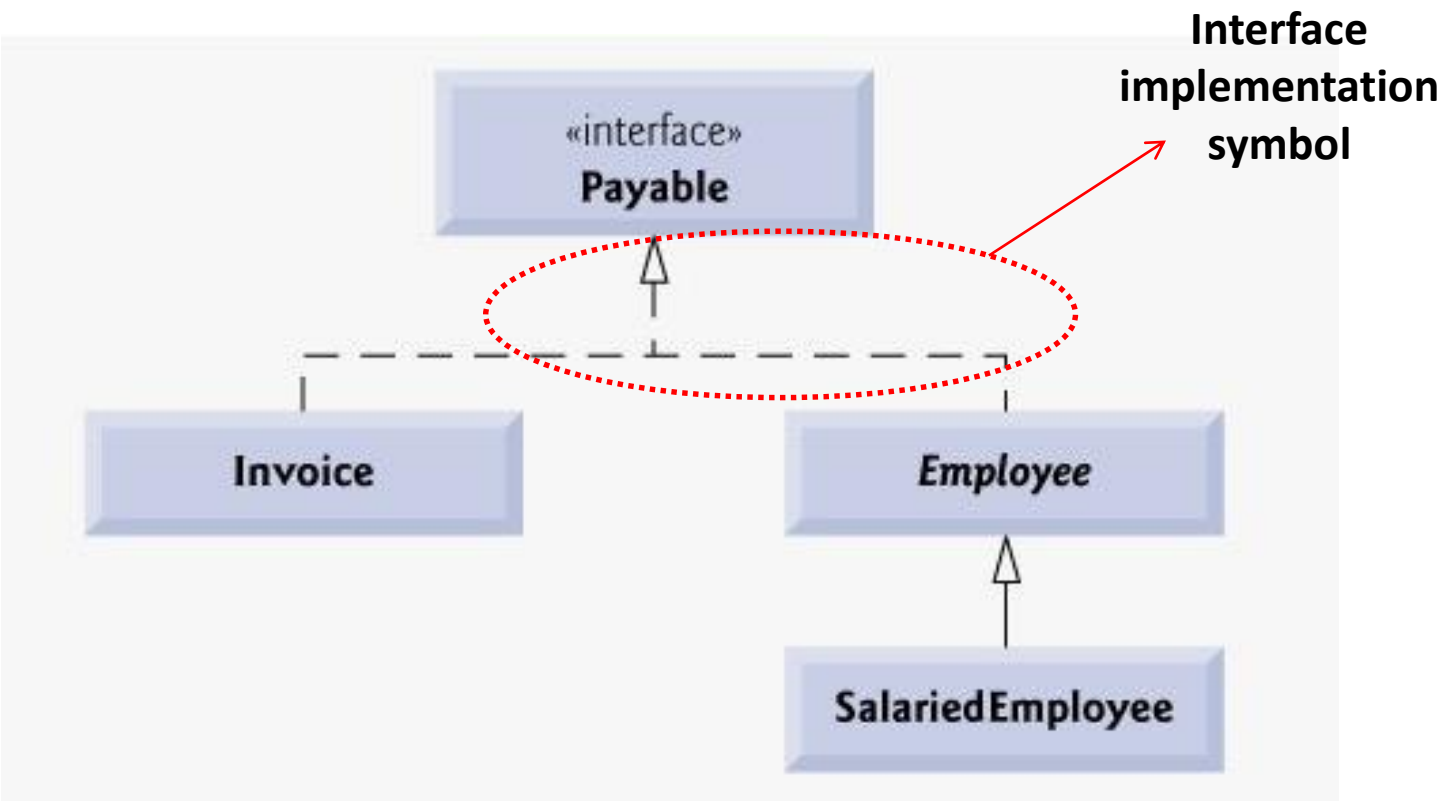
- When you define a new interface, you are defining a new reference data type.
- You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

Interfaces as Types

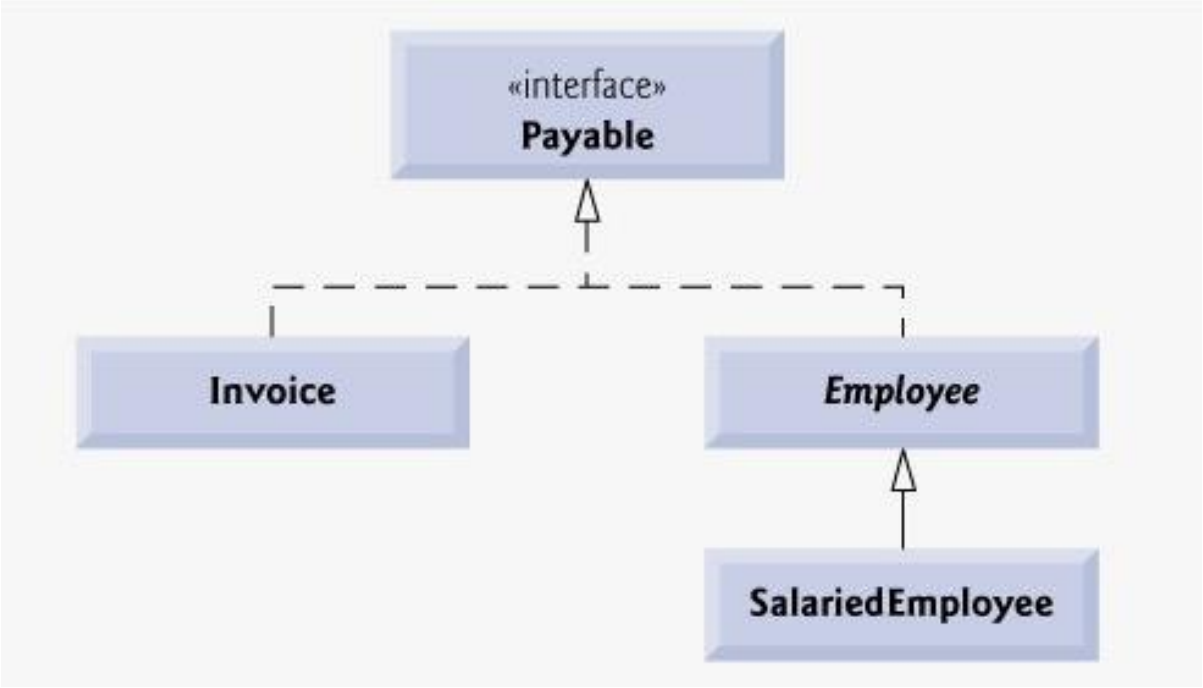
```
public class Program {
    public static void main(String[] args) {
        Shape shape;

        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

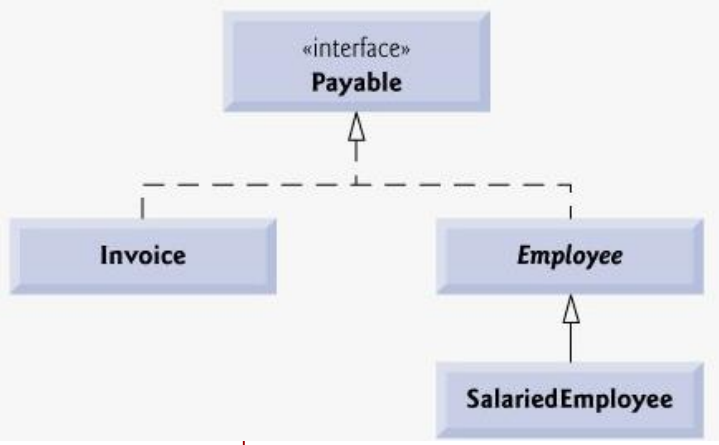


Payable.java




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

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

Invoice.java (1)

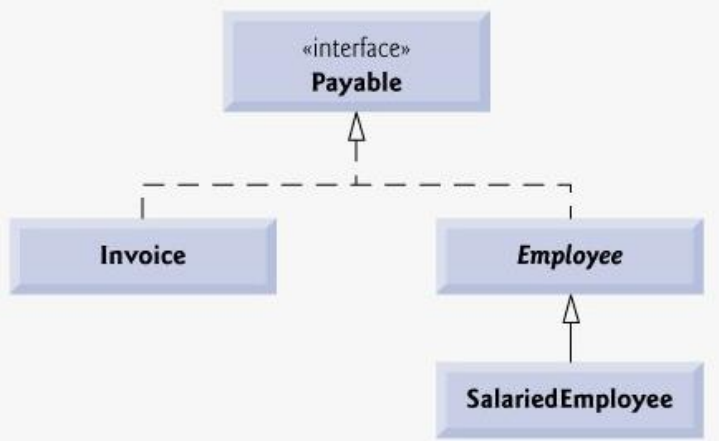



```java
public class Invoice implements Payable
{
   private String partNumber;
   private String partDescription;
   private int quantity;
   private double pricePerItem;

   // 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;
   } // end method setPartNumber

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

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

   // get description
   public String getPartDescription()
   {
      return partDescription;
   } // 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 class** `Employee` **does not implement it!**

```
public abstract class Employee implements Payable
{
   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

      /* Rest of the class is same as the previous example
      except there is no earnings() method! */
```

SalariedEmployee.java

```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;
   } // end method setWeeklySalary

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

   // calculate earnings; implement interface Payable method that was
   // abstract in superclass Employee
   public double getPaymentAmount()
   {
      return getWeeklySalary();
   } // end method getPaymentAmount

   // return String representation of SalariedEmployee object
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // 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

I will give you an example first:

```
public interface LoginAuth{
   public String encryptPassword(String pass);
   public void checkDBforUser();
}
```

Now suppose you have 3 databases in your application. Then each and every implementation for that database needs to define the above 2 methods:

```
public class DBMySQL implements LoginAuth{
          // Needs to implement both methods
}
public class DBOracle implements LoginAuth{
          // Needs to implement both methods
}
public class DBAbc implements LoginAuth{
          // Needs to implement both methods
}
```

But what if encryptPassword() is not database dependent, and it's the same for each class? Then the above would not be a good approach.

Instead, consider this approach:

```
public abstract class LoginAuth{
   public String encryptPassword(String pass){
            // Implement the same default behavior here
            // that is shared by all subclasses.
   }

   // Each subclass needs to provide their own implementation of this only:
   public abstract void checkDBforUser();
}
```

Now in each child class, we only need to implement one method - the method that is database dependent.

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