Polymorphism

Lecture 6

Dr. Tamer ABUHMED
Java Programming Course (SWE2023)
College of Computing

SUNG KYUN KWAN UNIVERSITY
1398

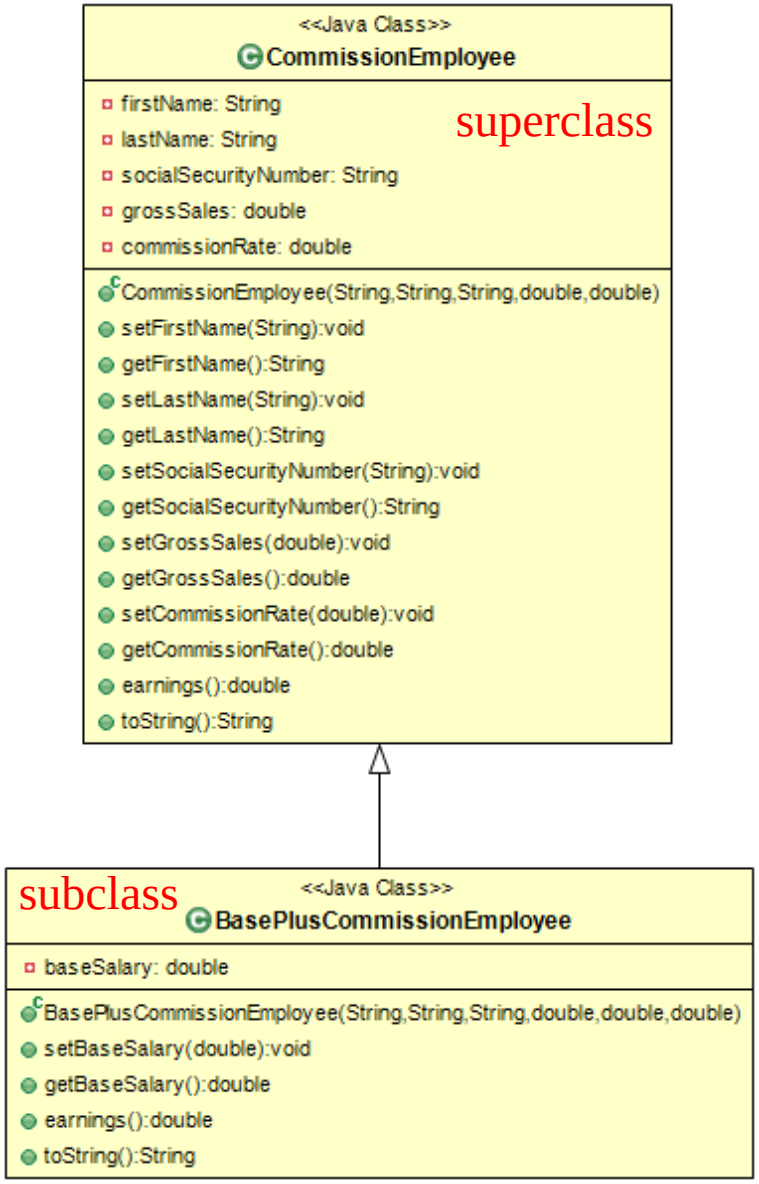
Outline

- What is **Polymorphism** ?
- Superclass reference at a subclass object
- Downcasting
- Polymorphism using **Abstract Classes**
  - Full Example
- Polymorphism using **Interface**
  - Full Example

Introduction

- Polymorphism
  - Enables you to “program in the general” rather than “program in the specific.”
  - Polymorphism enables you to write programs that process objects that share the same superclass as if they’re all objects of the superclass; this can simplify programming.

Superclass reference at a subclass object




```
public class PolymorphismTest
{
   public static void main( String[] args )
   {
      // assign superclass reference to superclass variable
      CommissionEmployee commissionEmployee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // assign subclass reference to subclass variable
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
"Call CommissionEmployee's toString with superclass reference ",
         "to superclass object", commissionEmployee.toString() );

      // invoke toString on subclass object using subclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call BasePlusCommissionEmployee's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString() );

      // invoke toString on subclass object using superclass variable
      CommissionEmployee commissionEmployee2 =
         basePlusCommissionEmployee;
      System.out.printf( "%s %s:\n\n%s\n",
         "Call BasePlusCommissionEmployee's toString with superclass",
         "reference to subclass object", commissionEmployee2.toString()
);
   } // end main
} // end class PolymorphismTest
```

superclass variable assigned to subclass variable

Call method in subclass class

Superclass reference at a subclass object

- In the previous example, we invoking a method on a subclass object via a superclass reference.
- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- Cannot do assignment of subclass reference to super class object
  basePlusCommissionEmployee = commissionEmployee2;
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.

Downcasting

```
public class Parent{}
public class Child extends Parent{
public static void main(String args[]) {
    Parent parent = new Child();
// a variable holding a value of type Child
    Child child = (Child)parent;
// OK since parent variable is currently
// holding Child instance
}
}
```

downcasting is the act of casting a reference of a super class to one of its subclasses

```
public class Parent{}
public class Child extends Parent{
public static void main(String args[]) {
    Parent parent = new parent();
// a variable holding a value of type parent
    Child child = (Child) parent;
// Not OK since parent variable is currently
// holding parent instance
}
}
```

Output

```
Exception in thread "main"
java.lang.ClassCastException: Parent cannot
be cast to Child
at Child.main(Parent.java:8)
```

Abstract Classes and Methods

- Abstract classes
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called abstract superclasses.
  - Cannot be used to instantiate objects—abstract classes are incomplete.
  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects;
  - Thus, Classes that can be used to instantiate objects are called concrete classes.

```
CommissionEmployee e = new CommissionEmployee ()
```

  - An abstract class provides a superclass from which other classes can inherit and thus share a common design.

Abstract Classes and Methods

- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses.
- You make a class abstract by declaring it with keyword abstract.

```
public abstract class Employee{} // abstract class
```

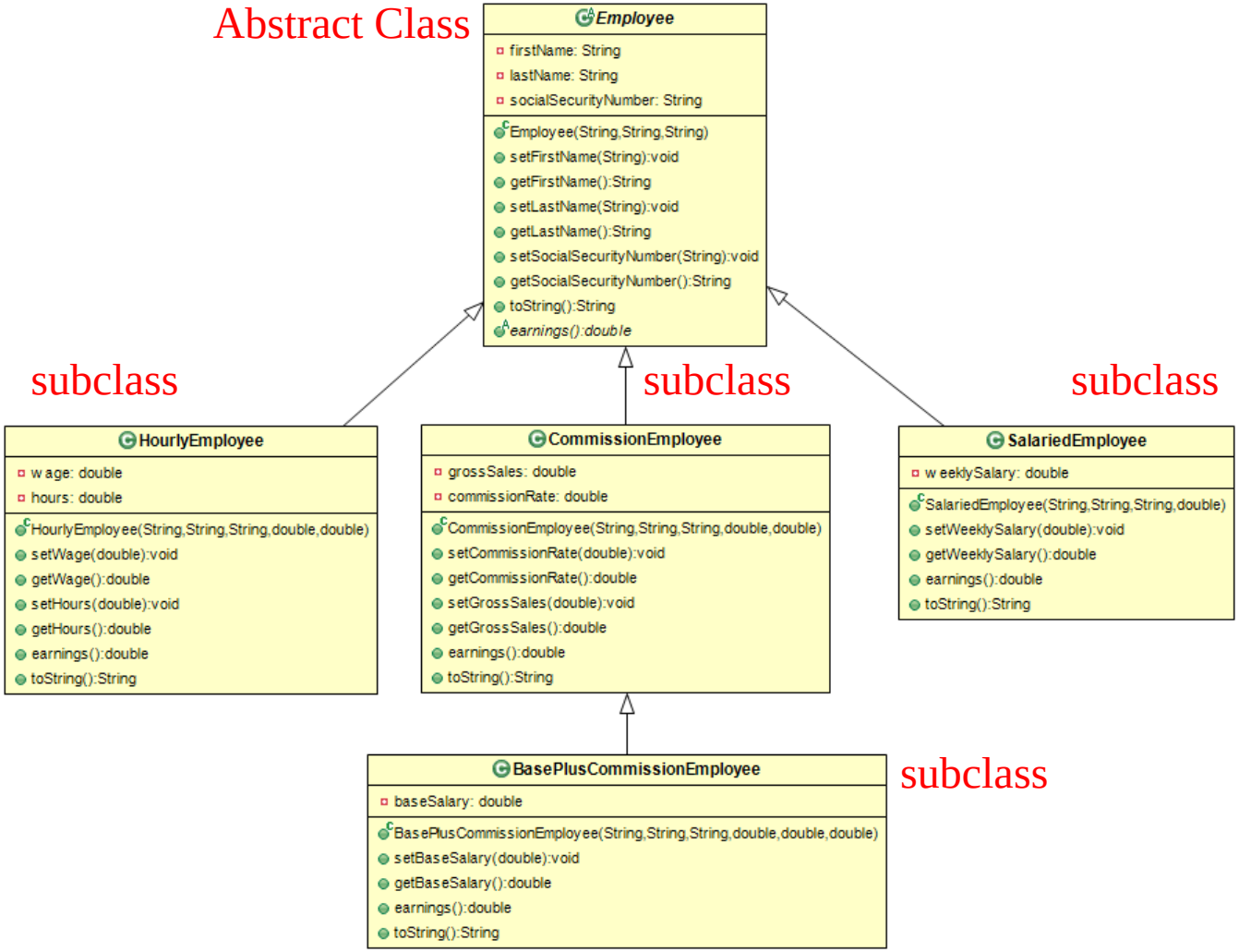
- An abstract class normally contains one or more abstract methods.
  - An abstract method is one with keyword `abstract` in its declaration, as in

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

- Abstract methods do not provide implementations
- A class that contains abstract methods must be an abstract class even if that class contains some concrete (nonabstract) methods.
- Constructors and `static` methods cannot be `abstract` methods.

Polymorphism using Abstract Classes

Abstract Class - Full Example



Abstract Class - Full Example (1/5)

```
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; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName
```

Abstract Class

```
   // set last name
   public void setLastName( String last )
   {
      lastName = last; // should validate
   } // 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
   @Override
   public String toString()
   {
      return String.format( "%s %s\nsocial security
number: %s",
         getFirstName(), getLastName(),
getSocialSecurityNumber() );
   } // end method toString

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

Abstract Method

Abstract Class - Full Example (2/5)

Inheritance

```
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 constructor

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0"
);
   } // end method setCommissionRate
   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate
```

```
  // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

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

   // calculate earnings; override abstract method
earnings in Employee
   @Override
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

// return String of CommissionEmployee object
   @Override
   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
```

override abstract method

override toString method

Abstract Class - Full Example (3/5)

```
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 )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0" );
   } // end method setBaseSalary
```

Inheritance

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

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

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

override earnings method

override toString method

Abstract Class - Full Example (4/5)

```
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 );
      setHours( hoursWorked );
   } // end five-argument HourlyEmployee constructor

   // set wage
   public void setWage( double hourlyWage )
   {
      if ( hourlyWage >= 0.0 )
         wage = hourlyWage;
      else
         throw new IllegalArgumentException(
            "Hourly wage must be >= 0.0" );
   } // end method setWage

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

Inheritance

```
   // set hours worked
   public void setHours( double hoursWorked )
   {
      if ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) )
         hours = hoursWorked;
      else
         throw new IllegalArgumentException(
            "Hours worked must be >= 0.0 and <= 168.0" );
   } // end method setHours

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

   // override abstract method earnings in Employee
   @Override
   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
   @Override
   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
```

override abstract method

override toString method

Abstract Class - Full Example (5/5)

```
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 ); // call Employee
constructor
      setWeeklySalary( salary );
} // end four-argument constructor

   // set salary
public void setWeeklySalary( double salary )
   {
      if ( salary >= 0.0 )
         weeklySalary = salary;
      else
         throw new IllegalArgumentException(
            "Weekly salary must be >= 0.0" );
   } // end method setWeeklySalary
```

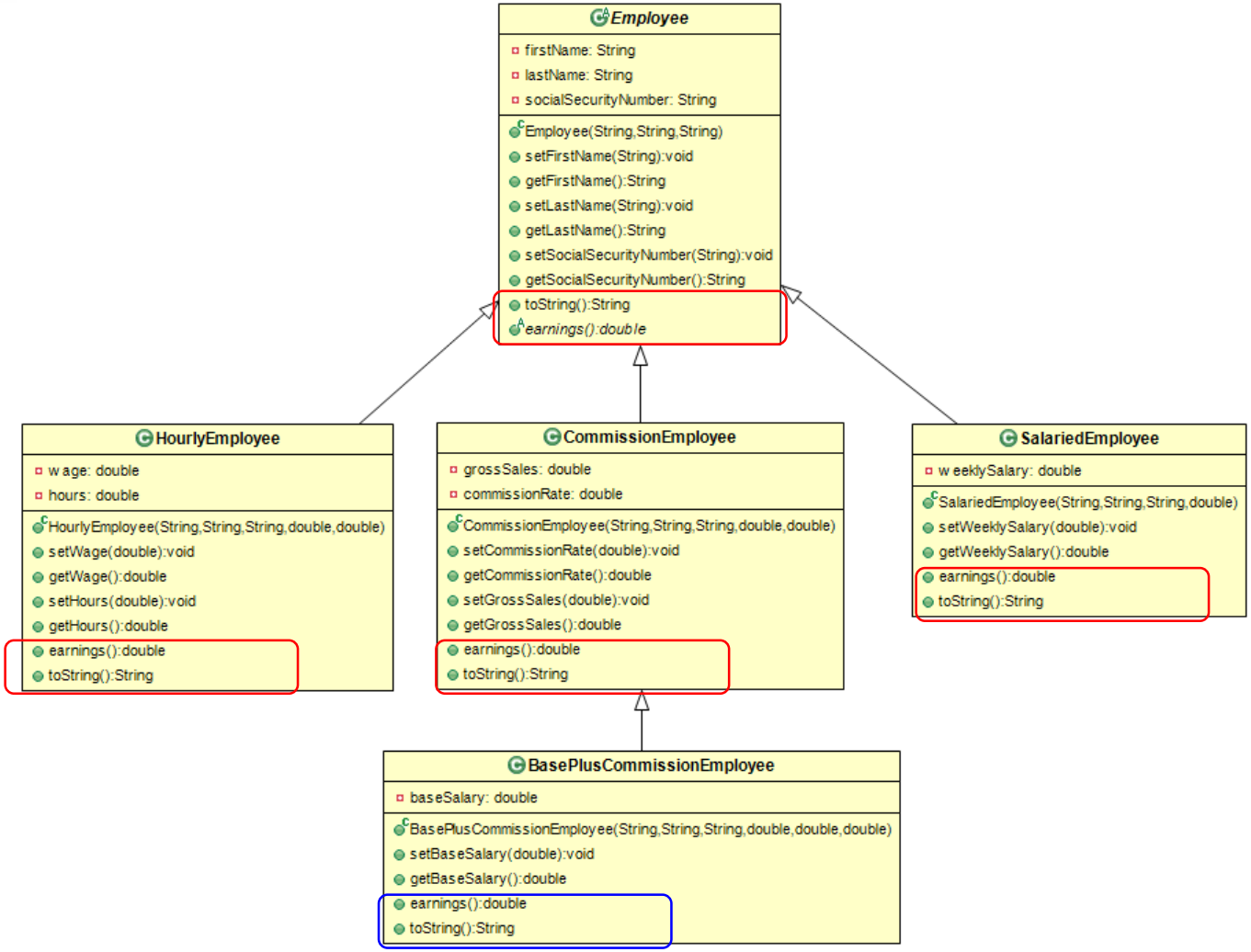
Inheritance

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

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

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

override abstract method

override toString method

Abstract Class - Full Example



Abstract Class - Full Example Test

```
public class PayrollSystemTest {

public static void main(String[] args) {

// create four-element Employee array

Employee[] employees = new Employee[4];


// initialize array with Employees

employees[0] =
new SalariedEmployee("John", "Smith", "111-11-1111",800.00);

employees[1] =
new HourlyEmployee("Karen", "Price", "222-22-2222", 16.75,
40);

employees[2] =
new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000,
.06);


employees[3] =
new BasePlusCommissionEmployee("Bob", "Lewis","444-44-4444",
5000, .04, 300);

System.out.println("Employees processed polymorphically:\n");
```

Each employee reference has different initialization

```
// generically process each element in array
employees

for (Employee currentEmployee : employees) {

System.out.println(currentEmployee); // invokes
toString

System.out.printf("earned $%,.2f\n\n",
currentEmployee.earnings());

} // end for


// 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());

} // end main

} // end class PayrollSystemTest
```

Objects LOOP

Get object class type

Abstract Class - Full Example Test

Output

```
Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned $500.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Interface

- Interfaces offer a capability requiring that unrelated classes implement a set of common methods.
- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
- The interface specifies *what* methods are permitted to use but does not specify *how* the methods are performed.
  - A Java interface describes a set of methods that can be called on an object.

Interface

- An interface declaration begins with the keyword interface and contains only constants and `abstract` methods.
  - All interface members must be `public`.
  - Interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
  - All methods declared in an interface are implicitly `public abstract` methods.
  - All fields are implicitly `public`, `static` and `final`.

```
public interface Payable
{
  double getPaymentAmount();
}
```

Interface Payable

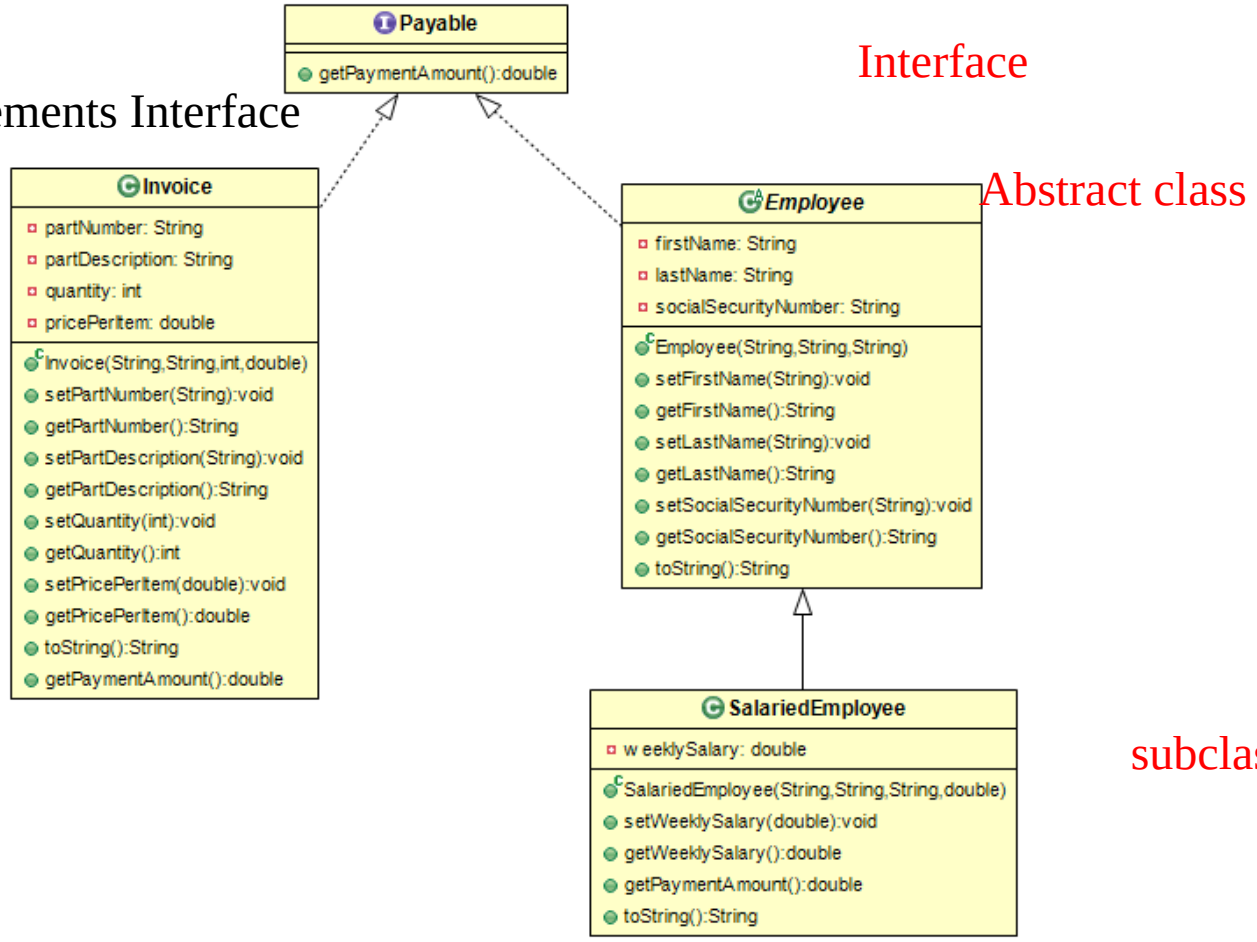
Abstract method

Interface

- An interface cannot extend a class
- A class cannot extend an interface
- To use an interface, a concrete class must specify that it implements the interface and must declare each method in the interface with specified signature.
  - Add the `implements` keyword and the name of the interface to the end of your class declaration's first line.
- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.
- Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs.

Polymorphism using Interfaces

Interface - Full Example



Interface- Full Example (1/3)

```
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 ); } // end constructor

   public void setPartNumber( String part )
   {
      partNumber = part; // should validate
   } // end method setPartNumber

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

   public void setPartDescription( String description )
   {
      partDescription = description; // should validate
   } // end method setPartDescription
  public String getPartDescription()
   {
      return partDescription;
   } // end method getPartDescription
```

implements Interface Payable

```
public void setQuantity( int count )
   {
quantity = count;
} // end method setQuantity

public int getQuantity()
   {
      return quantity;
   } // end method getQuantity

public void setPricePerItem( double price )
   {
      if ( price >= 0.0 )
         pricePerItem = price;
      else
         throw new IllegalArgumentException(
            "Price per item must be >= 0" );
   } // end method setPricePerItem

public double getPricePerItem()
   {
      return pricePerItem;
   } // end method getPricePerItem

@Override
   public String toString()
   {return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s:
$%,.2f","invoice", "part number", getPartNumber(),
getPartDescription(),
         "quantity", getQuantity(), "price per item",
getPricePerItem() );
   } // end method toString

@Override
   public double getPaymentAmount()
   {
    return getQuantity() * getPricePerItem(); // calculate
total cost
   } // end method getPaymentAmount
} // end class Invoice
```

override method

override abstract method

Interface- Full Example (2/3)

```
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

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName
```

We do not implement Payable method getPaymentAmount here so this class must be declared abstract

```
   public void setLastName( String last )
   {
      lastName = last; // should validate
   } // end method setLastName

   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   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
   @Override
   public String toString()
   {
      return String.format( "%s %s\nsocial security number:
%s",
         getFirstName(), getLastName(),
getSocialSecurityNumber() );
   } // end method toString

} // end abstract class Employee
```

override method

Interface- Full Example (3/3)

```
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 )
   {
      if ( salary >= 0.0 )
         weeklySalary = salary;
      else
         throw new IllegalArgumentException(
            "Weekly salary must be >= 0.0" );
   } // 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
   @Override
   public double getPaymentAmount()
   {
      return getWeeklySalary();
   } // end method getPaymentAmount

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

Inheritance

override abstract method

override method

```
public class PayableInterfaceTest
{
   public static void main( String[] args )
   {
      // create four-element Payable array
      Payable[] payableObjects = new Payable[ 4 ];

      // populate array with objects that implement Payable
payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00
);
payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95
);
payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-
1111", 800.00 );
payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-
8888", 1200.00 );

System.out.println( "Invoices and Employees processed
polymorphically:\n" );

for ( Payable currentPayable : payableObjects )
      {
System.out.printf( "%s \n%s: $%,.2f\n\n",
currentPayable.toString(),
  "payment due", currentPayable.getPaymentAmount() );
   } // end for
} // end main
} // end class PayableInterfaceTest
```

Objects LOOP

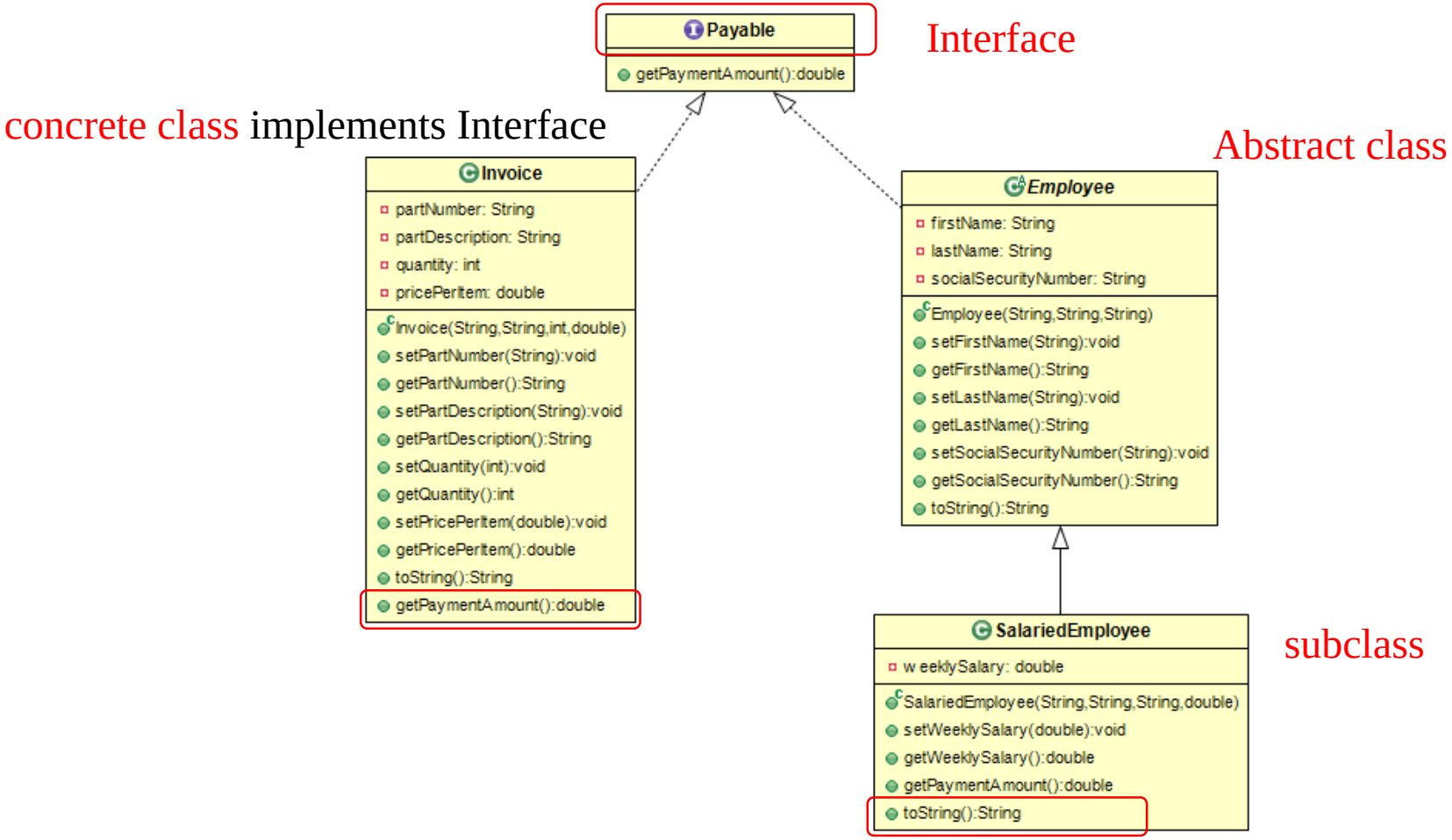
Interface- Full Example Test

Output

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Polymorphism using Interfaces

Interface - Full Example



Summary

- What is Polymorphism ?
- Superclass reference at a subclass object
- Downcasting
- Polymorphism using Abstract Classes
  - Full Example
- Polymorphism using Interface
  - Full Example