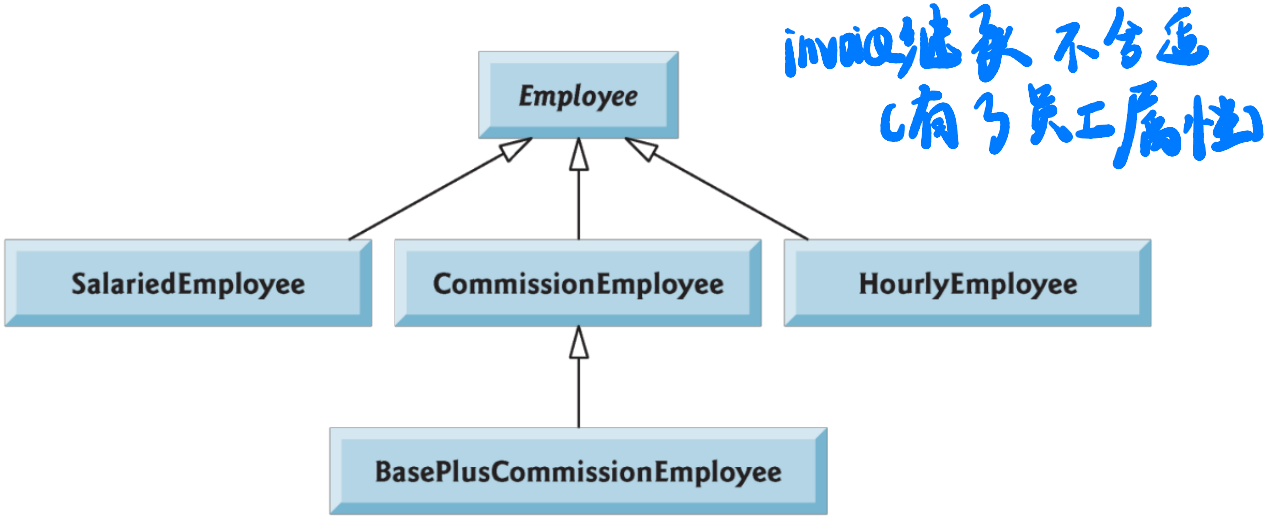
Chapter 12: Interface

TAO Yida

taoyd@sustech.edu.cn

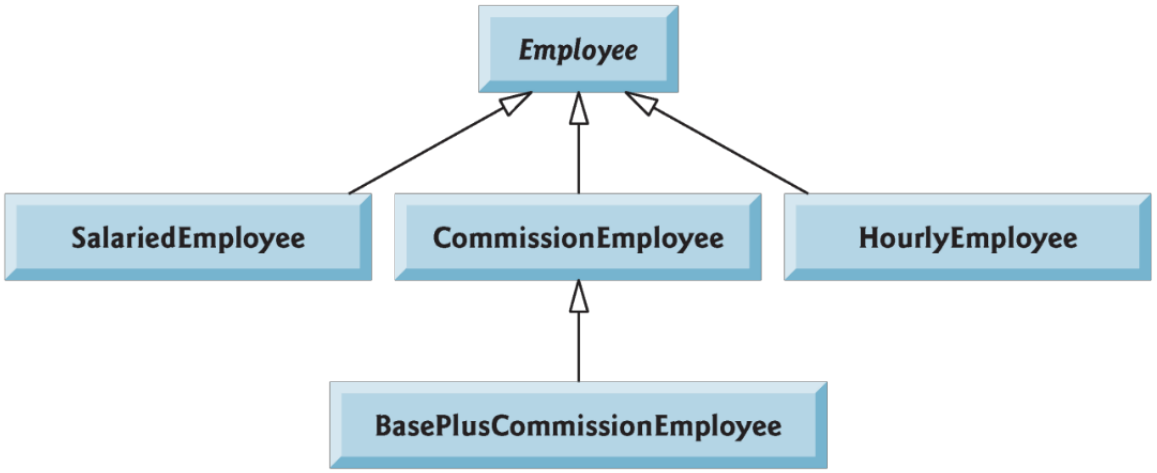
Extending the Payroll System

- Suppose the company wants to use the system to calculate the money it needs to pay not only for employees but also for invoices
  - For an employee, the payment refers to the employee's earnings.
  - For an invoice (发票), the payment refers to the total cost of the goods listed.



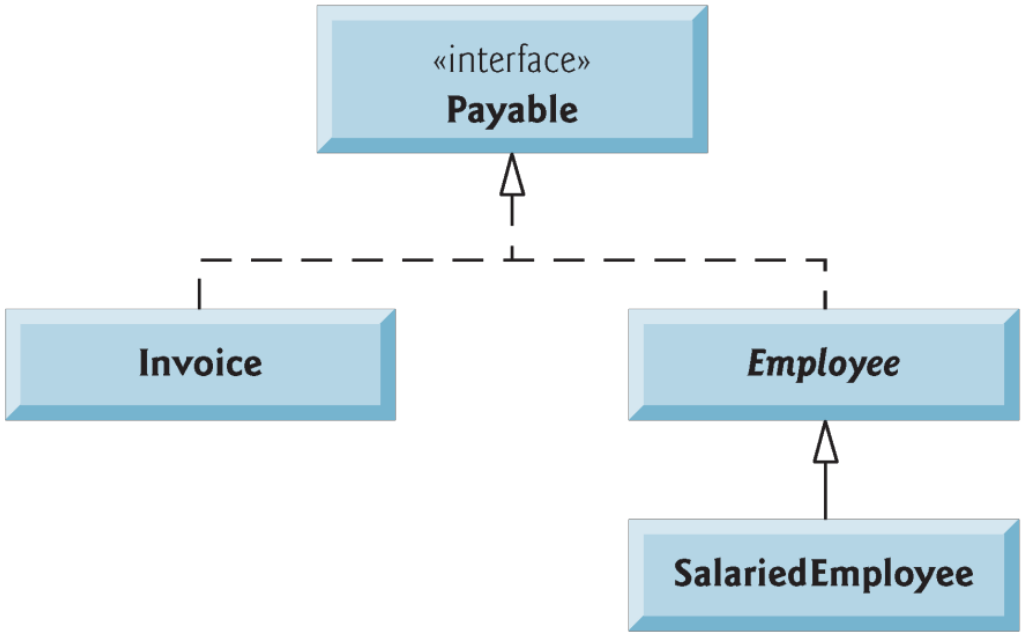
Extending the Payroll System

- Can we make `Invoice` class extend `Employee`?
  - This is unnatural, the `Invoice` class would inherit inappropriate members (e.g., employee names)



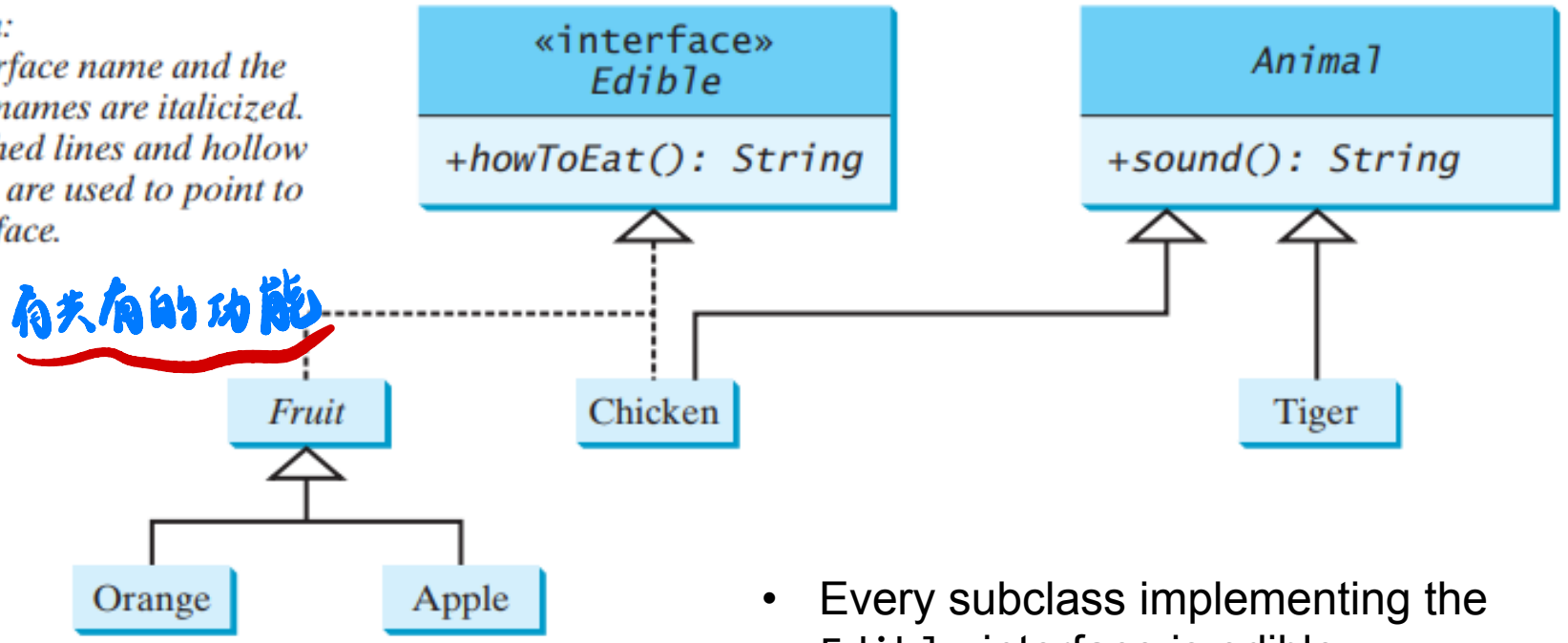
Extending the Payroll System

Interfaces are used to specify common behavior for objects of related classes or unrelated classes.



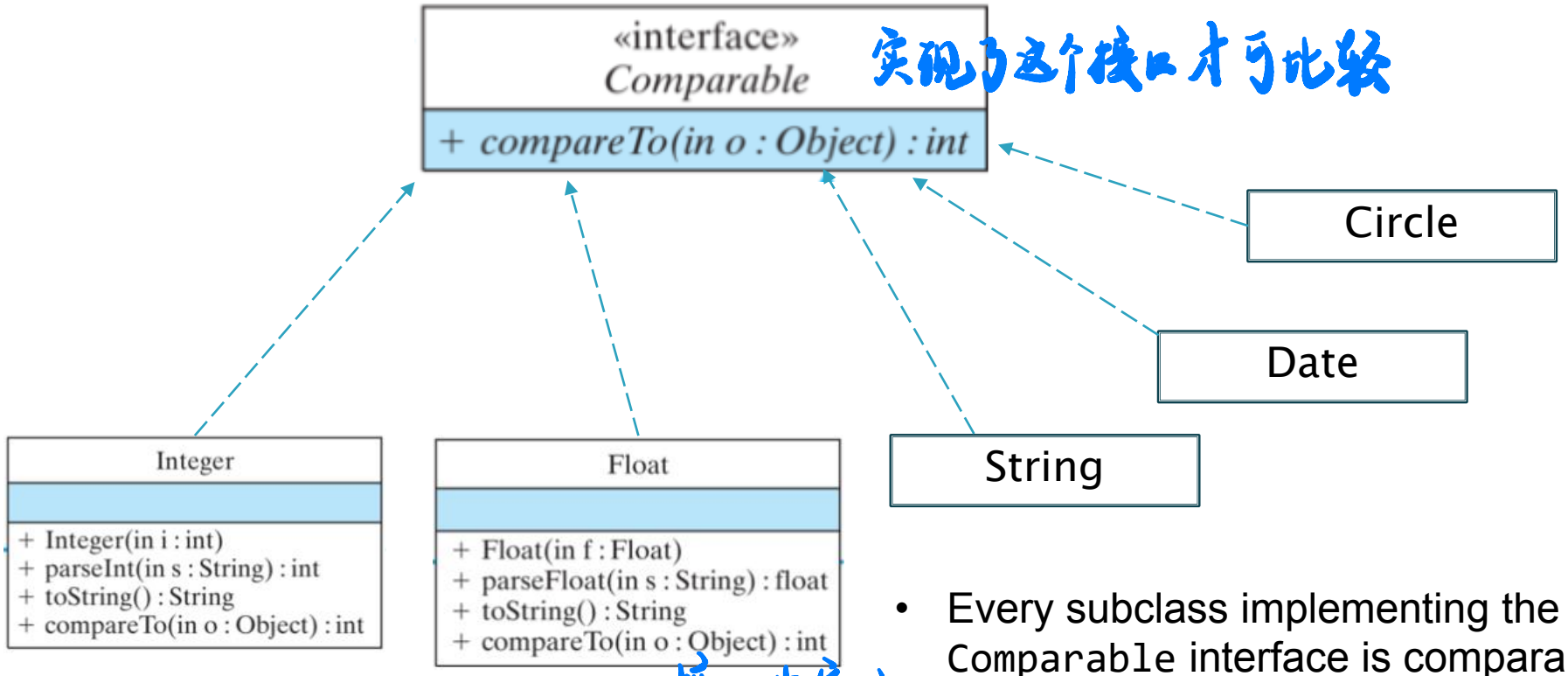
Examples of Interfaces

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.



- Every subclass implementing the `Edible` interface is edible
- Every subclass determines exactly how to eat.

Examples of Interfaces

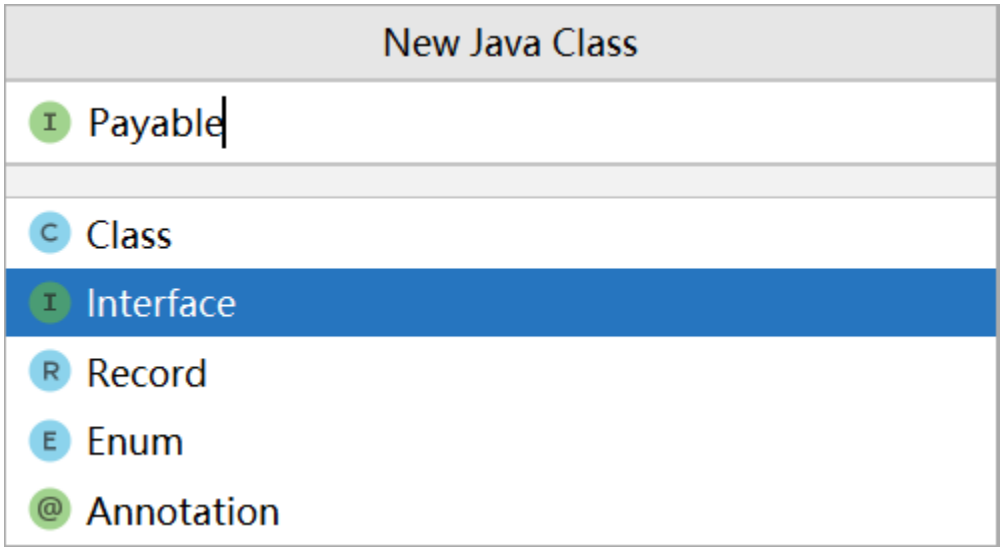


- Every subclass implementing the `Comparable` interface is comparable
- Every subclass determines exactly how to compare with each other.

各子类决定如何比较
(重写)

Declaring Interfaces

- In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects of related classes or unrelated classes.
- An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.



Declaring Interfaces

命名习惯：形容词

- Keyword `interface` is used in the declaration
- Like `public abstract` classes, interfaces are typically `public` types (can also be `package-private`). A `public` interface must be declared in a `.java` file with the same name as the interface.

一定是 public abstract

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

Interface names are often adjectives since interface is a way of describing what the classes can do. Class names are often nouns.

Fields in Interface

- An interface can contain **constant** declarations.
- All constant fields (values) defined in an interface are implicitly `public`, `static`, and `final`. You can omit these modifiers.

```
public interface MyInterface {
    int a = 1;
    char c = '2';
    boolean b;

}
```

一定要初始化

Variable 'b' might not have been initialized

Initialize variable 'b' Alt+Shift+Enter Mor

Fields in Interface

- An interface in Java doesn't have a constructor (as all fields are constants, there is no need to initialize them through the constructor)

Methods in Interface

- The interface body can contain abstract, default, and static methods.
- All abstract, default, and static methods in an interface are implicitly public, so you can omit the `public` modifier.
- Default methods are defined with the `default` modifier, and static methods with the `static` keyword; both default and static methods should have concrete implementations.
- Methods in an interface are implicitly `abstract` if they are not static or default

```
public interface MyInterface {

    default void foo(){
        System.out.println("Default method");
    }

    static void bar(){
        System.out.println("Static method");
    }

    void baz();
}
```


Methods in Interface

- Static methods
  - Typically used for utility methods
  - Can be invoked by `InterfaceName.staticMethodName(…)`
- Default methods
  - Classes that implement an interface can invoke the default methods in the interface
  - Default methods enable you to add new functionality to the interfaces of your libraries and ensure backward compatibility with code written for older versions of those interfaces.

Interface Payable

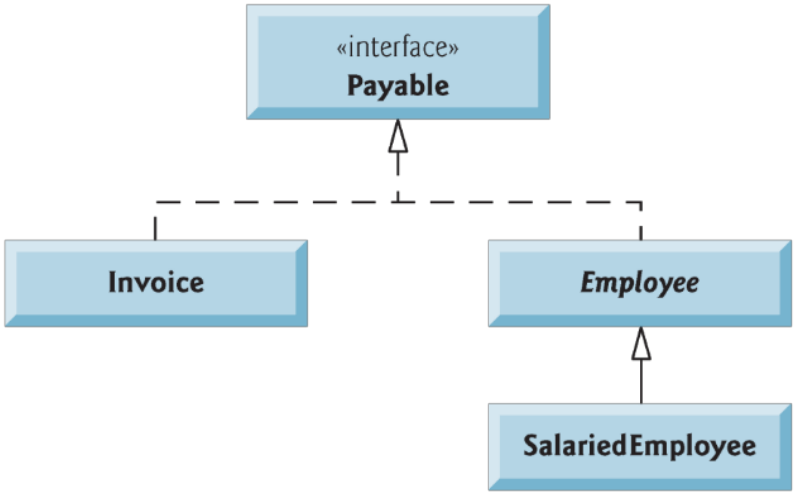
```
public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

协议

必须实现抽象方法，才可以用

The Payable interface has only one abstract method, getPaymentAmount(), to be implemented by its subclasses

编译器会检查是否 overwrite



Implementing an Interface

- To use an interface, a concrete class must specify that it `implements` the interface and must implement ALL abstract methods in the interface with specified signature. If not, the class must be abstract.

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

```
public class Invoice implements Payable {

    // must override and implement the getPaymentAmount() method

}
```

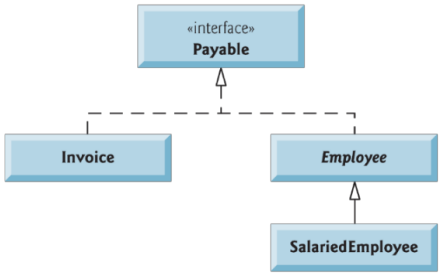

Implementing an Interface

- When a class implements an interface, it makes a contract (协议) with the Java compiler:
  - The class will implement each of the methods in the interface; If the subclass does not do so, it too must be declared `abstract`.
  - Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the contract (**the unfulfilled contract is inherited**).

```
class Invoice implements Payable{

      double getPaymentAmont(){
          // implementation
      }
}
```

Interface is a “capability”;
If a class claims to have a capability (implements an interface), it must override all its abstract methods before the compiler admits that it indeed has that capability

Class Invoice




```
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
```

The class extends `Object` (implicitly) and implements `Payable` interface

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

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

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

// get description
public String getPartDescription()
{
   return partDescription;
} // end method getPartDescription
```

```
// set quantity
public void setQuantity( int count )
{
   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
```

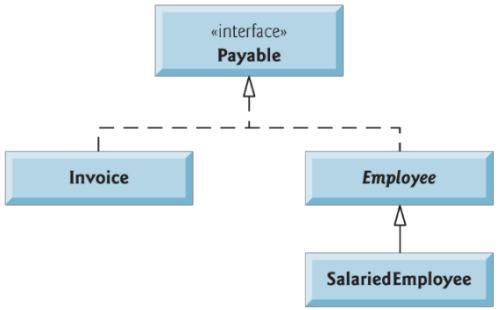
```
    // return String representation of Invoice object
    @Override
    public String toString()
    {
       return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
          "invoice", "part number", getPartNumber(), getPartDescription(),
          "quantity", getQuantity(), "price per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

    // method required to carry out contract with interface Payable
    @Override
    public double getPaymentAmount()
    {
       return getQuantity() * getPricePerItem(); // calculate total cost
    } // end method getPaymentAmount
} // end class Invoice
```

Providing an implementation of the interface's method(s) makes this class concrete

Class Employee

Abstract class extends `Object` (implicitly) and implements interface `Payable`

```
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
```



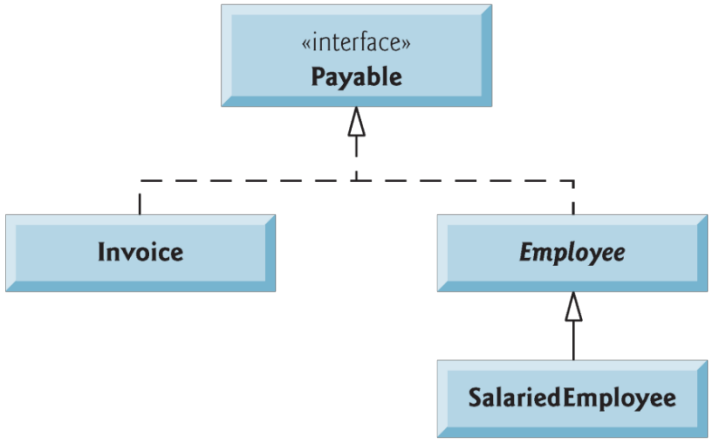
```
// return first name
public String getFirstName()
{
   return firstName;
} // end method getFirstName

// 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

    // Note: We do not implement Payable method getPaymentAmount here so
    // this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

We don't implement the method, so this class needs to be declared as `abstract`.

Class SalariedEmployee

- The `SalariedEmployee` class that extends `Employee` must fulfill superclass `Employee`'s contract to implement `Payable` method `getPaymentAmount`.



```
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
```

Providing an implementation of the method to make this class concrete and instantiable

```
   // 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
```

不用再写 abstract 了

Using an Interface as a Type

- An interface is a reference type. You can use interface names anywhere you can use any other data type name.
- We cannot instantiate an interface directly
- 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

* 父类引用指向子类对象

```
Payable payableObject = new Invoice(...);
```

同抽象类

Using an Interface as a Type

- Objects of a class (or its subclasses) that `implements` an interface can also be considered as **objects of the interface type**.
- Thus, just as we can assign the reference of a `SalariedEmployee` object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object to an interface `Payable` variable.
- `Payable payableObject = new SalariedEmployee(...);`
- `Invoice` implements `Payable`, so an `Invoice` object is also a `Payable` object, and we can assign the reference of an `Invoice` object to a `Payable` variable.
  - `Payable payableObject = new Invoice(...);`


```
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" );
```

An array of polymorphic objects

都要计算支出

Assigning the references of different types of objects to the `Payable` variables

```
        // generically process each element in array payableObjects
        for ( Payable currentPayable : payableObjects )
        {
           // output currentPayable and its appropriate payment amount
           System.out.printf( "%s \n%s: $%,.2f\n\n",
              currentPayable.toString(),
              "payment due", currentPayable.getPaymentAmount() );
        } // end for
     } // end main
} // end class PayableInterfaceTest
```

Objects are processed polymorphically

```
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
```

Implementing Interface vs. Extending class

- A class can inherit from only one superclass, but can implement as many interfaces as it needs.

```
public class ClassName extends SuperclassName
        implements FirstInterface, SecondInterface, ...
```

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```


Implementing Interface vs. Extending class

- An interface can `extend` other interfaces, just as a class extends another class.
- Whereas a class can extend only one other class, an interface can extend any number of interfaces (separated by comma)
- An interface cannot `extend` a class, and cannot `implement` other interfaces
  - This would cause a conflict with the fact that interfaces are "abstract"

注：接口与接口间是 extends (相同)

```
public interface Interface3 extends Interface1, Interface2{
}
```

不可继承类

Implementing Interface vs. Extending class

- Inheritance (extending a class)
  - Provide code reusability (subclasses reuse superclass's features)
  - Single inheritance

- Interface
  - Provides abstraction (used for design purposes, cannot be instantiated)
  - Multiple inheritance
  - Useful and more flexible since they capture similarity between unrelated objects without forcing a class relationship

Interface vs. Abstract Class

	Abstract Class	Interface
1	An abstract class can extend only one class or one abstract class	An interface can extend any number of interfaces
2	An abstract class can extend another concrete class or abstract class	An interface cannot extend classes
3	In abstract class keyword "`abstract`" is mandatory to declare a method as an abstract	In an interface keyword "`abstract`" is optional to declare a method as an abstract
4	An abstract class can have constructors	An interface cannot have a constructor
5	An abstract class can have `protected` and `public` abstract methods	An interface can only have `public` abstract methods
6	An abstract class can have `static`, `final` or `static final` variables with any access specifier	An interface can only have `public static final` (constant) variable

Example: The `Comparable` Interface

- Java contains several comparison operators (e.g., <, >=, ==) that allow you to compare primitive values.
- However, these operators cannot be used to compare objects.
- The interface `Comparable` is used to allow objects of a class that implements the interface to be compared to one another.
- `Comparable` is commonly used for ordering objects in a collection such as an array.


```
import java.util.Arrays;
public class Employee implements Comparable<Employee> {
    private String firstName, lastName;
    private int id;

    public Employee(String first, String last, int sid) {
        firstName = first;   lastName = last;   id = sid;
    }

    @Override
    public String toString() {
        return String.format("[%s %s ID: %d]", firstName, lastName, id);
    }

    public static void main(String[] args) {
        Employee[] employees = new Employee[3];
        employees[0] = new Employee("Jack", "Ma", 1);
        employees[1] = new Employee("Yanhong", "Li", 2);
        employees[2] = new Employee("Huateng", "Ma", 3);
        Arrays.sort(employees);
        System.out.println(Arrays.toString(employees));
    }

    @Override
    public int compareTo(Employee o) {
        return this.id - o.id;
    }
}
```

```
[[Jack Ma ID: 1], [Yanhong Li ID: 2], [Huateng Ma ID: 3]]
```

