

10 - Inheritance

ICSI 201

Introduction to Computer Science

Qi Wang

Department of Computer Science

University at Albany

State University of New York

Course Materials

- All course materials are **copyrighted** and can be used by the students who are enrolled in the class only.
- Posting/sharing the course materials for other uses without permission is prohibited.

Outline

- Creating subclasses
- Overriding methods
- Class Hierarchies
- Visibility
- Design for inheritance
- Interface
- Polymorphism

Creating subclasses

Inheritance

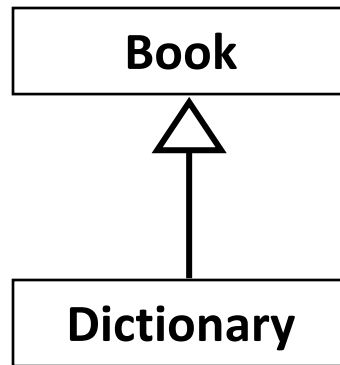
- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes.
- *Inheritance* allows a software developer to derive a new class from an existing one.

Inheritance

- The existing class is called the *parent class*, or *superclass*, or *base class*.
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent.
- That is, the child class inherits the methods and data defined by the parent class.

Inheritance

- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class.



- Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent.

Inheritance

- *Software reuse* is a fundamental benefit of inheritance.
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software.
- In Java, we use the reserved word `extends` to establish an inheritance relationship.

```
class Dictionary extends Book{  
    // class contents  
}
```


Inheritance

- See Book.java and Dictionary.java.
- No inheritance:

Book
- pages: int
+ getPages(): int + setPages(int): void + toString(): String + equals(Object): boolean

Dictionary
- pages: int - definitions: int
+ getPages(): int + setPages(int): void + getDefinitions(): int + setDefinitions(int): void + computeRatio(): double + toString(): String + equals(Object): boolean

The *super* Reference

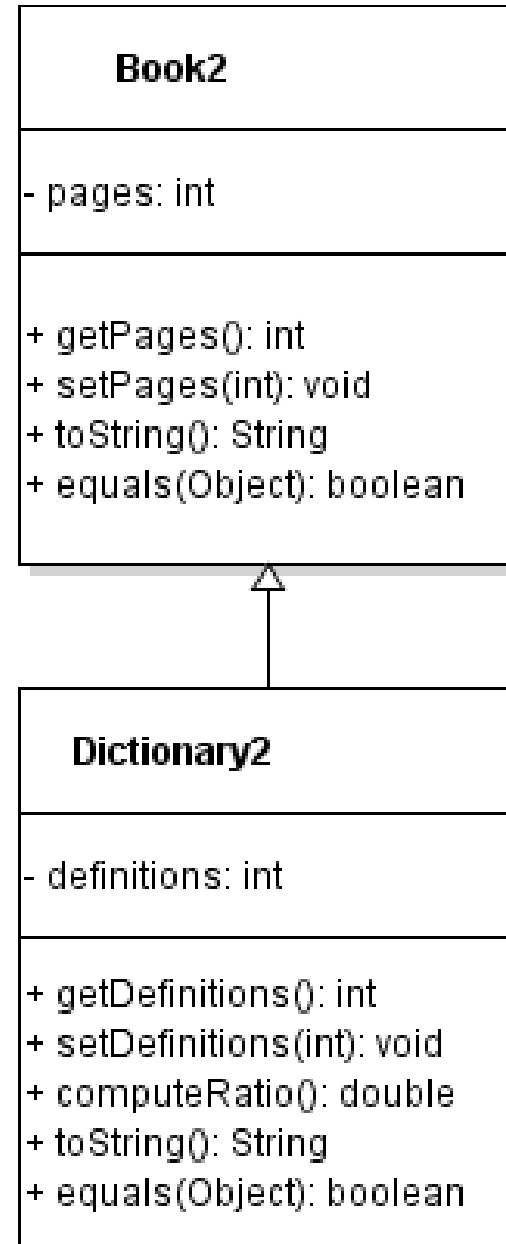
- Instance variables and public methods of a super class are inherited automatically by its sub classes.
- Constructors are not inherited, even though they have public visibility.
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object.
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor.

The super Reference

- A child's constructor is responsible for calling the parent's constructor.
- **The first line of a child's constructor should use the *super* reference to call the parent's constructor.**
- The *super* reference can also be used to reference other variables and methods defined in the parent's class.

Inheritance

- See Book2.java and Dictionary2.java.



The protected Modifier

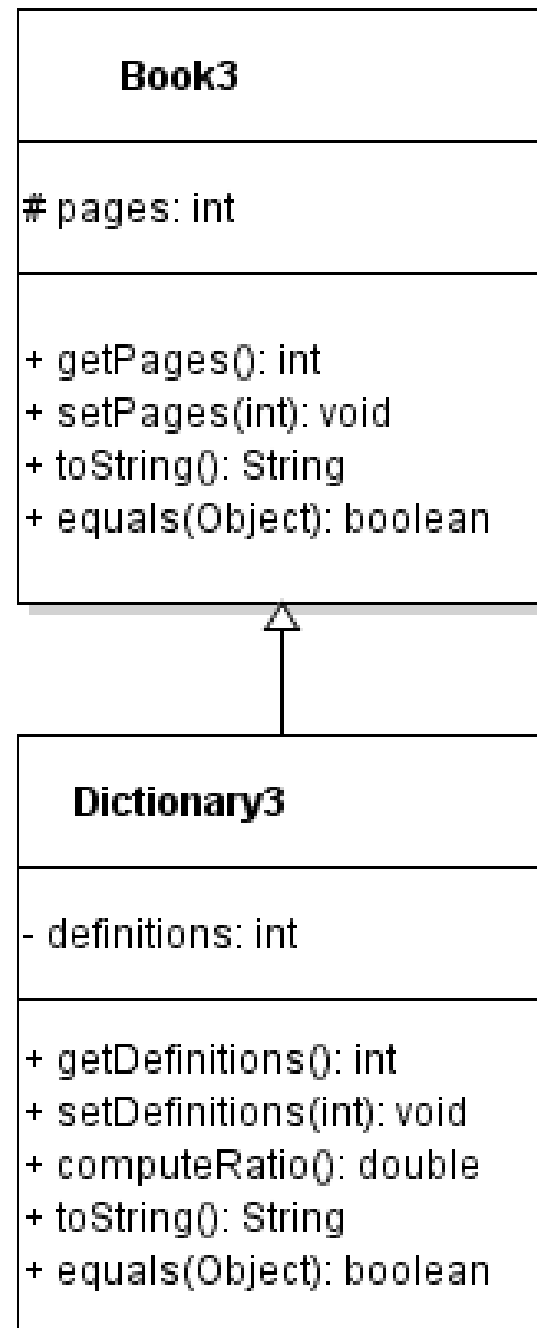
- Visibility modifiers affect the way that class members can be used in a child class.
- Variables and methods declared with private visibility cannot be referenced by name in a child class.
- They can be referenced in the child class if they are declared with public visibility -- but **public variables violate the principle of encapsulation.**
- There is a third visibility modifier that helps in inheritance situations: `protected`.

The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method directly in the child class.
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility.
- A protected variable is visible to all classes in the same package and all subclasses.
- Protected variables and methods can be shown with a `#` symbol preceding them in UML diagrams.

Inheritance

- See Book3.java and Dictionary3.java.



Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class.
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents.
- Java does not support multiple inheritance.

Overriding methods

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own.
- The new method must have the same signature as the parent's method, but can have a different body.
- The type of the object executing the method determines which version of the method is invoked.

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library.
- All classes are derived from the `Object` class.
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class.
- Therefore, the `Object` class is the ultimate root of all class hierarchies.

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes.
- For example, method `toString` and method `equals`.
- Every time we define the `toString` method and `equals` method, we are actually overriding an inherited definition.

Overriding

- A method in the parent class can be invoked explicitly using the `super` reference.
- If a method is declared with the `final` modifier, it cannot be overridden.
- The concept of overriding can be applied to data and is called *shadowing variables*.
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code.

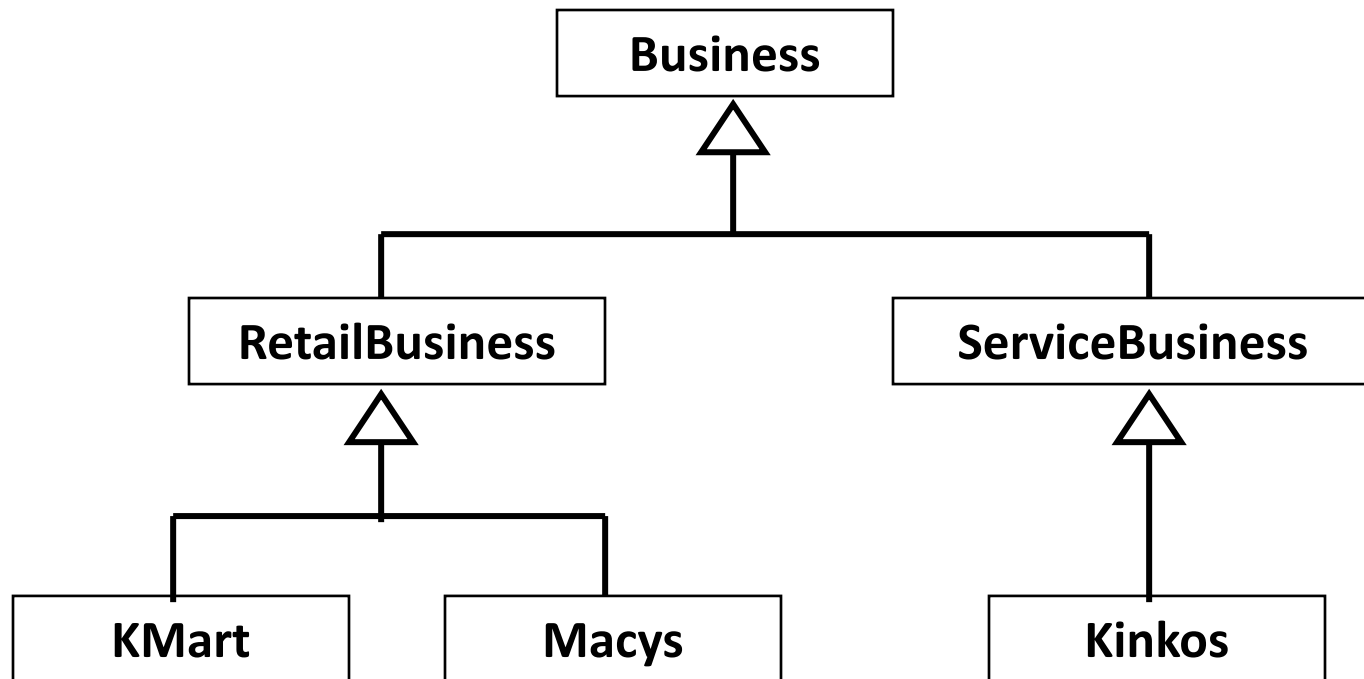
Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures.
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature.
- Overloading lets you define a similar operation in different ways for different parameters.
- Overriding lets you define a similar operation in different ways for different object types.

Class Hierarchies

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*.



Class Hierarchies

- Two children of the same parent are called *siblings*.
- Common features should be put as high in the hierarchy as is reasonable.
- An inherited member is passed continually down the line.
- Therefore, a child class inherits from all its ancestor classes.
- There is no single class hierarchy that is appropriate for all situations.

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept.
- An abstract class cannot be instantiated.
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product{  
    // contents  
}
```

Abstract Classes

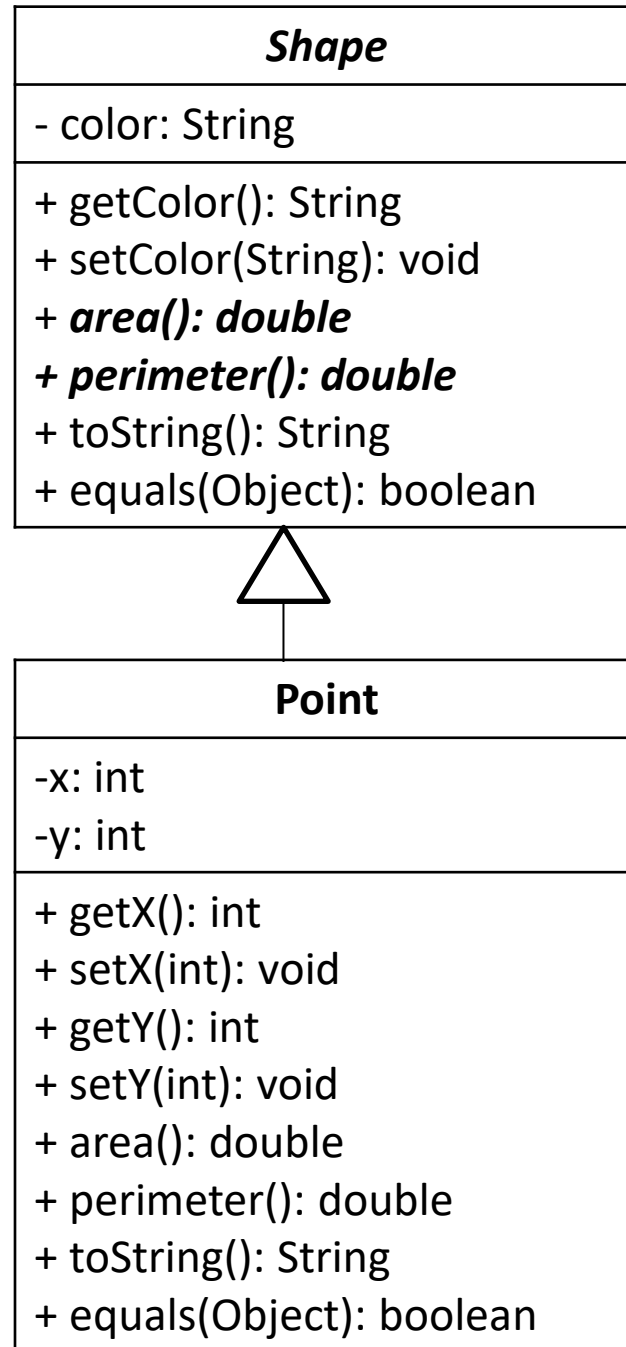
- An abstract class often contains abstract methods with no definitions.
- The `abstract` modifier must be applied to each abstract method.
- Also, an abstract class typically contains non-abstract methods with full definitions.
- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so.

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract.
- An abstract method cannot be defined as `final` or `static`.
- The use of abstract classes is an important element of software design – it allows us to **establish common elements in a hierarchy that are too generic to instantiate.**

Abstract Classes

- By convention, an abstract class name and its abstract members should be italicized in the design.
- See Shape.java, Point.java.



Visibility

Visibility Revisited

- It's important to understand one subtle issue related to inheritance and visibility.
- All variables and methods of a parent class, even private members, are inherited by its children.
- As we've mentioned, private members cannot be referenced by name in the child class.
- However, private members inherited by child classes exist and can be referenced indirectly.

Visibility Revisited

- Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods.
- The `super` reference can be used to refer to the parent class, even if no object of the parent exists.

Visibility Revisited

- In Book and Dictionary:

```
public double computeRatio() {  
    return this.definitions/this.getPages();  
}
```

can be changed to

```
public double computeRatio() {  
    return this.definitions/super.getPages();  
}
```

Design for inheritance

Designing for Inheritance

- As we've discussed, taking the time to create a good software design reaps long-term benefits.
- Inheritance issues are an important part of an object-oriented design.
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software.

Inheritance Design Issues

- Every derivation should be an is-a relationship.
- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible.
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate.
- Override methods as appropriate to tailor or change the functionality of a child.
- Add new variables to children, but don't redefine (shadow) inherited variables.

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data.
- Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions.
- Use abstract classes to represent general concepts that lower classes have in common.
- Use visibility modifiers carefully to provide needed access without violating encapsulation.

Restricting Inheritance

- The `final` modifier can be used to curtail inheritance.
- If the `final` modifier is applied to a method, then that method cannot be overridden in any descendent classes.

Restricting Inheritance

- If the `final` modifier is applied to an entire class, then that class cannot be used to derive any children at all.
 - Thus, an abstract class cannot be declared as final.
- These are key design decisions, establishing that a method or class should be used as is.

Interface

Interface

- An *interface* is similar to an abstract class that has all abstract methods.
 - Modifier `abstract` is not required although all methods are abstract.
 - An interface cannot be instantiated.
 - All the methods listed in an interface must be written elsewhere.
 - All methods specified by an interface are public by default.
- The purpose of an interface is to specify behavior for other classes.
- It is often said that an interface is like a “contract,” and when a class implements an interface it must adhere to the contract.

Interface

- An interface looks similar to a class, except:
 - the keyword `interface` is used instead of the keyword `class`, and
 - the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons.

Interface

- An interface looks similar to a class, except:
 - the keyword `interface` is used instead of the keyword `class`, and
 - the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons.

- The general format of an interface definition:

```
public interface InterfaceName{  
    (Method headers...)  
}
```

- A class can implement one or more interfaces.

Interface

- If a class implements an interface, it uses the `implements` keyword in the class header.

```
public class MyClass implements MyInterface
```

- A class can extend a class and implement an interface.

```
public class MyClass extends MySuperclass  
    implements MyInterface
```

Interface

- A class can implement multiple interfaces.

```
public class MyClass extends MySuperclass  
    implements MyInterface1, MyInterface2
```

- When a class implements multiple interfaces, it must provide the methods specified by all of them.

Fields in Interfaces

- An interface can contain field declarations.
- All fields in an interface are treated as `final` and `static`.
- You must provide an initialization value for each field.
- Any class that implements this interface has access to these variables(constants).

Fields in Interfaces

- For example,

```
public interface Doable{  
    final static int FIELD1 = 1;  
    final static int FIELD2 = 2;  
    (Method headers...)  
}
```

- In this interface, FIELD1 and FIELD2 are final static int variables.

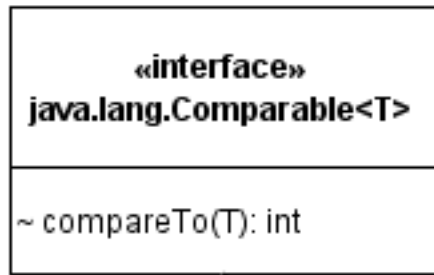
Java *Comparable* Interface

- Defines the *compareTo* method for comparing objects

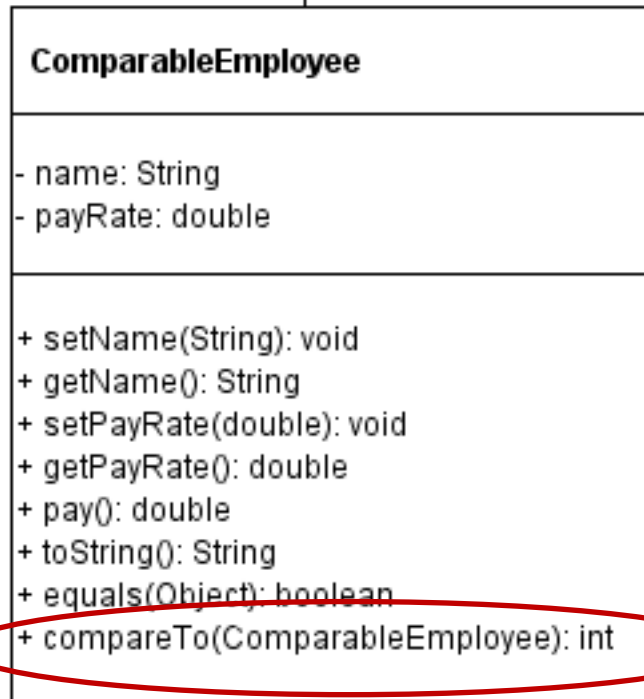
```
public interface Comparable<T>{           In package java.lang.  
    int compareTo(T o);  
}
```

- This interface is implemented by classes that need to compare their objects according to some natural order.
 - Only one natural order can be defined.
- The generic type `T` is replaced by a concrete type when implementing this interface.

Interface class in a UML



A dashed line with an arrow indicates implementation of an interface.

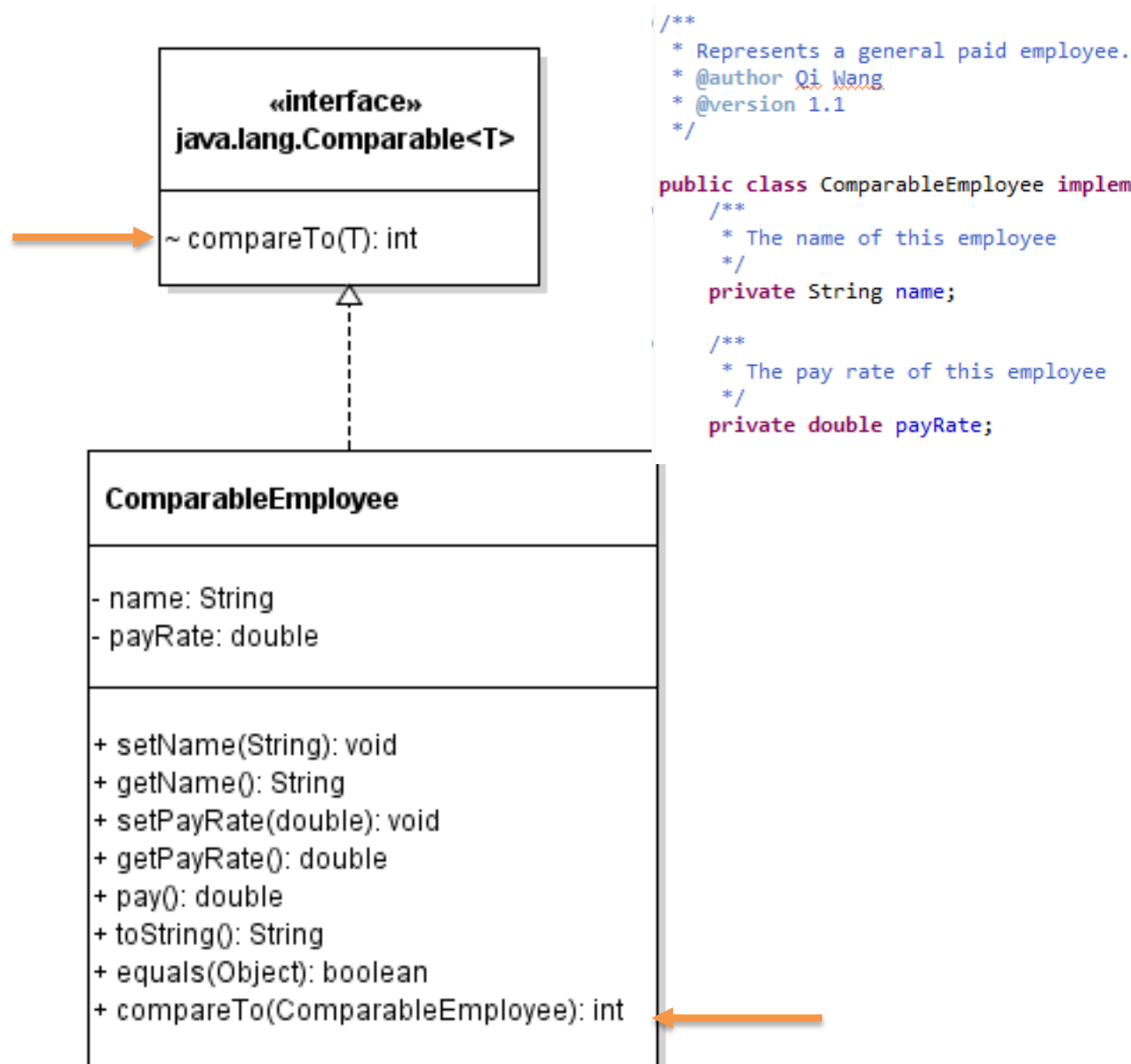



Defines the *compareTo* method in *ComparableEmployee* class to specify a natural ordering of employee objects.

Java *Comparable* Interface

The *compareTo* method:

- returns a negative integer if the calling object is “less than” the other object.
- returns 0 if the calling object is “equal” to the other object.
- returns a positive integer if the calling object is “greater than” the other object.





```

/**
 * Represents a general paid employee.
 * @author Qi Wang
 * @version 1.1
 */

public class ComparableEmployee implements Comparable<ComparableEmployee>{
    /**
     * The name of this employee
     */
    private String name;

    /**
     * The pay rate of this employee
     */
    private double payRate;

    public int compareTo(ComparableEmployee o) {
        //Order of names
        return this.name.compareTo(o.name);
    }
}

```

A natural ordering based on employee names

```

/**
 * Represents a general paid employee.
 * @author Qi Wang
 * @version 1.1
 */

public class ComparableEmployee implements Comparable<ComparableEmployee>{
    /**
     * The name of this employee
     */
    private String name;

    /**
     * The pay rate of this employee
     */
    private double payRate;

```



A natural ordering based on
employee pay rates

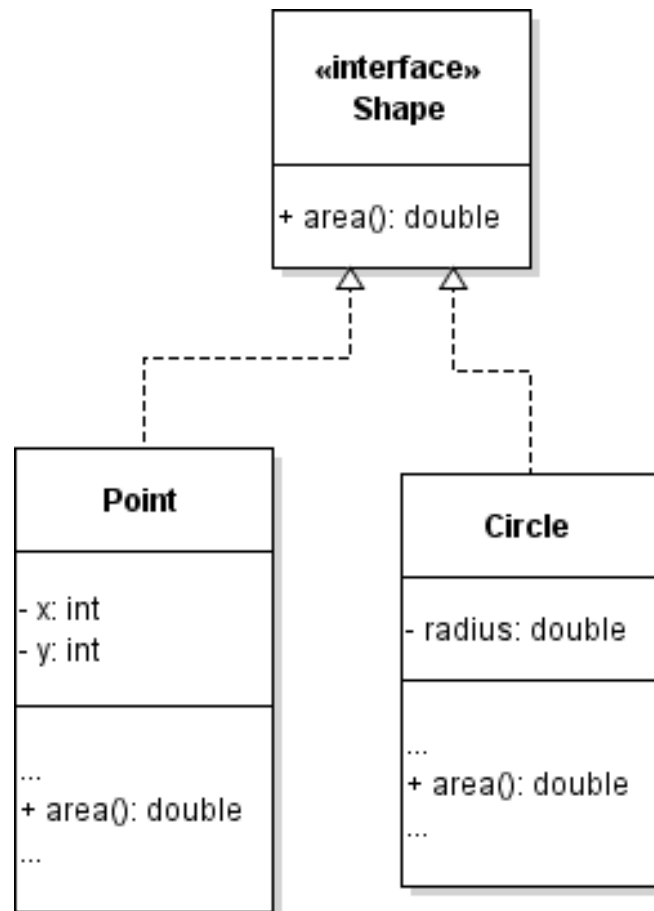
```

public int compareTo(ComparableEmployee2 o){
    //Compare long bit representations of double pay rates
    //Double.doubleToRawLongBits(this.payRate) vs. Double.doubleToRawLongBits(o.payRate)
    if(Double.doubleToRawLongBits(this.payRate) > Double.doubleToRawLongBits(o.payRate)){
        return 1;
    }else if(Double.doubleToRawLongBits(this.payRate) < Double.doubleToRawLongBits(o.payRate)){
        return -1;
    }else{
        //(Double.doubleToRawLongBits(this.payRate) == Double.doubleToRawLongBits(o.payRate)){
        return 0;
    }
}

```

Shape Interface

- We can write our own interface class.



```
public interface Shape{  
  
    /**  
     * Returns the area of this shape.  
     * @return The area of this shape  
     */  
    double area();  
}
```

Shape Interface

```
public class Point implements Shape {  
    /**  
     * The x coordinate of this point  
     */  
    private int x;  
    /**  
     * The y coordinate of this point  
     */  
    private int y;  
  
    /**  
     * Returns the area of this point.  
     *  
     * @return A double value specifying the area of this point  
     */  
    public double area() {  
        return 0;  
    }  
}
```

Class Types

Reference Types	Instantiation
Class (Concrete class)	A reference type. Can be instantiated. <code>Point p = new Point();</code>
Interface	A reference type. Can not be instantiated. <code>Shape s, s1;</code> <code>s = new Shape(); //NO!</code> <code>s1 = new Point(); //YES!</code>
Abstract Class	A reference type. Can not be instantiated. Has constructors that are called by its subclasses.

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes.
- That is, one interface can be derived from another interface.
- The child interface inherits all abstract methods of the parent.
- A class implementing the child interface must define all methods from both the ancestor and child interfaces.
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap).

Polymorphism

Polymorphism

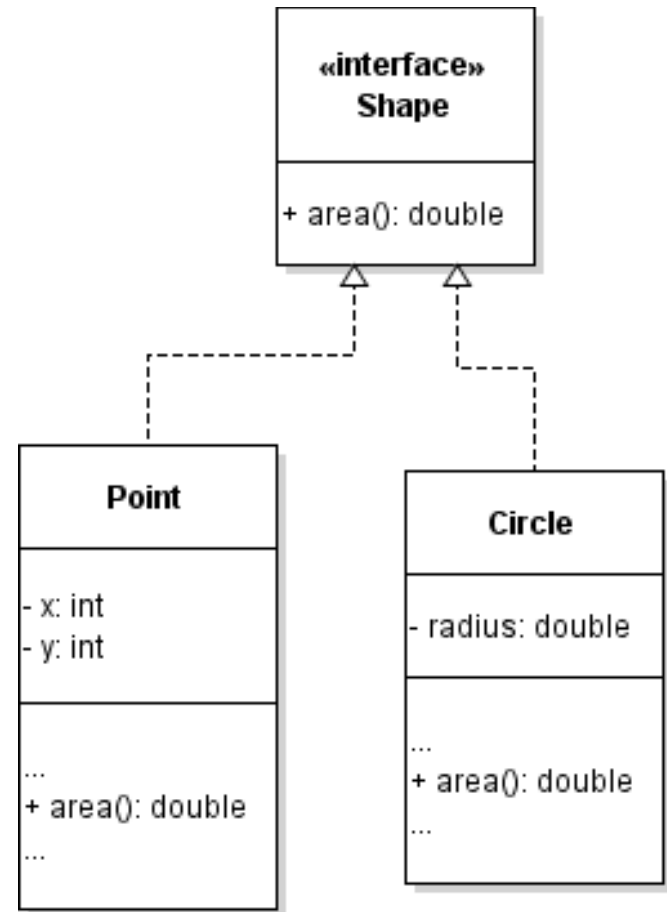
- The term *polymorphism* literally means "having many forms".
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time.
- The method invoked through a polymorphic reference can change from one invocation to the next.
- All object references in Java are potentially polymorphic.

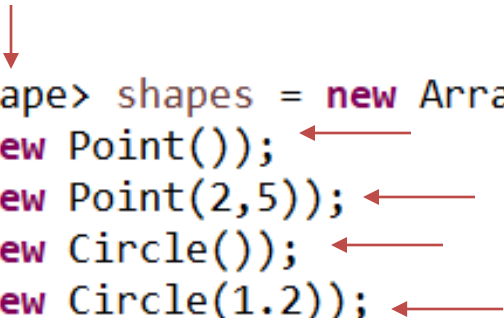
Polymorphism

- Java allows a variable to reference an object of any compatible type.
- This compatibility can be established using interfaces or inheritance.
- Careful use of polymorphic references can lead to elegant, robust software designs.

Polymorphism via Interface

- Shape example shows how this compatibility can be established using interfaces.
- A Shape variable can reference a Circle or a Rectangle. Not the other way around.





```
ArrayList<Shape> shapes = new ArrayList<Shape>();
shapes.add(new Point());
shapes.add(new Point(2,5));
shapes.add(new Circle());
shapes.add(new Circle(1.2));

for(int i = 0; i < shapes.size(); i++){
    //A Shape variable can reference an object of compatible types such as
    //Point or Circle
    Shape s = shapes.get(i);

    //The object type, not the reference type(Shape), determines which
    //version of the method to be invoked.
    //area() of Point is called when i is 0 or 1.
    //area() of Circle is called when i is 2 or 3.
    System.out.println(s.area());
}
```

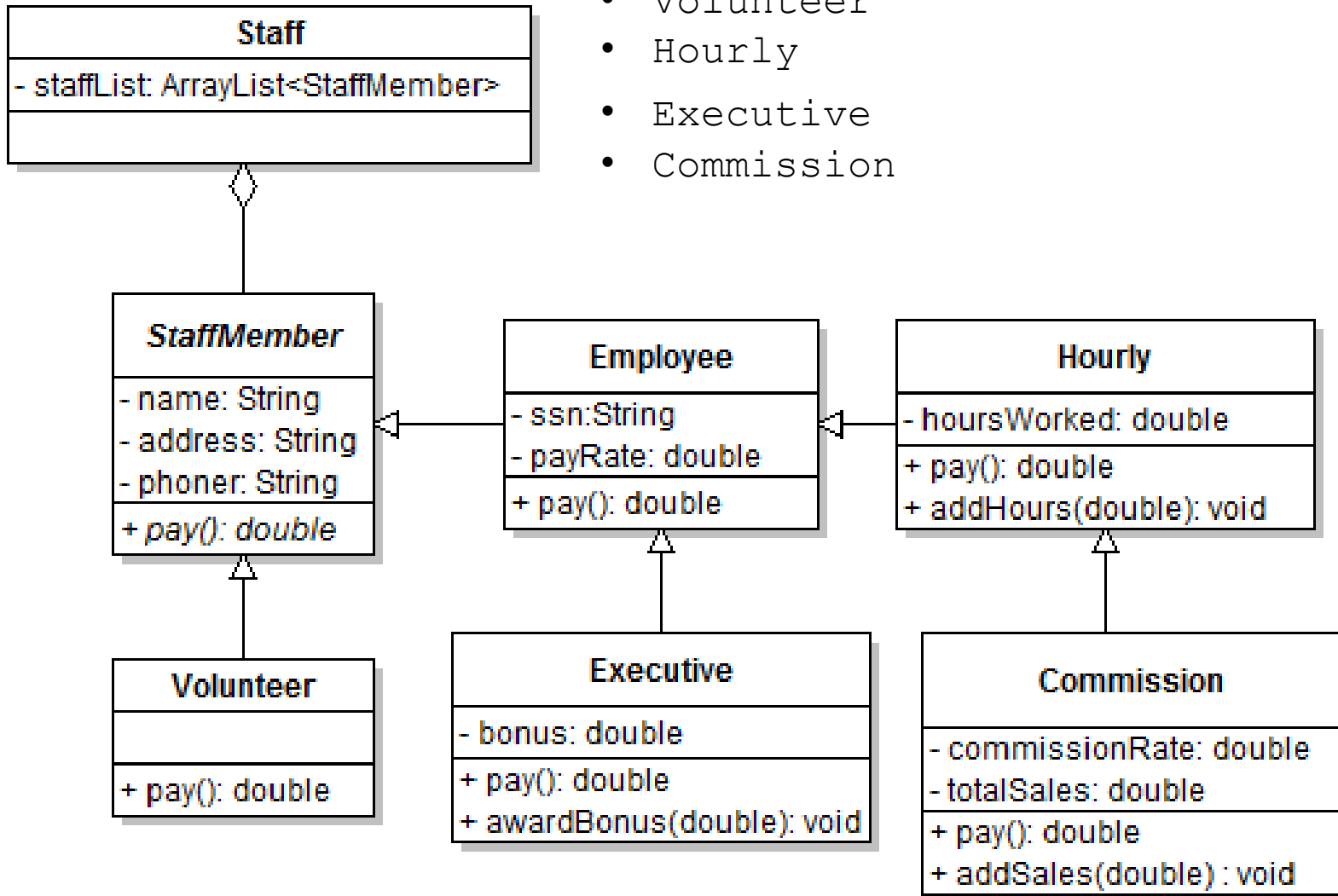
- The method *area* invoked through a polymorphic reference *s* can change from one invocation to the next.

Polymorphism via Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance.
- Assigning a child object to a parent reference is a widening conversion and can be performed by simple assignment.
- Assigning a parent object to a child reference can be done also, but it is a narrowing conversion and must be done with a cast.
- The widening conversion is the most useful.

The compatible types of `StaffMember`:

- `Employee`
- `Volunteer`
- `Hourly`
- `Executive`
- `Commission`



Polymorphism

dynamic binding or late binding

- It is the type of the object being referenced, not the reference type, that determines which method is invoked.
- Java defers method binding until run time -- this is called *dynamic binding or late binding*.

Summary

- Creating subclasses
- Overriding methods
- Class Hierarchies
- Visibility
- Design for inheritance
- Interface
- Polymorphism