Polymorphism II

CS102A Lecture 12

James YU

Nov. 30, 2020

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

`final` methods and static binding

- A `final` method in a superclass cannot be overridden in a subclass. You might want to make a method `final` if it has an implementation that should not be changed and it is critical to the consistent state of the object.
- `private` methods are implicitly `final`. It's not possible to override them in a subclass (not inherited).
- `static` methods are implicitly `final`. Non-`private static` methods are inherited by subclasses, but cannot be overridden (they are `final`). They are hidden if the subclass defines a `static` method with the same signature.
- A `final` method's declaration can never change and therefore calls to `final` methods are resolved at compile time, known as *static binding*.

Static binding and dynamic binding

- These terms are used to describe whether the superclass' version or the subclass' version of method are called.
- *Static binding* in Java occurs during compile time while *dynamic binding* occurs during runtime.
- *Static binding* bases on variable type while *dynamic binding* bases on object's type.
- `private`, `final`, and `static` methods and variables use *static binding*.

final methods and static binding

```java
public class TestFinalMethod {
  public static void test() {
    System.out.println("hello from superclass");
  }
  public static void main(String[] args) {
    TestFinalMethod obj = new TestFinalMethod2();
    obj.test(); // which test will be called?
  }
}
```

```java
public class TestFinalMethod2 extends TestFinalMethod {
  public static void test() { // this is hiding, not overriding
    System.out.println("hello from subclass");
  }
}
```

`final` classes

- A `final` class cannot be a superclass (cannot be extended).
    - All methods in a `final` class are implicitly `final`.
- Class `String` is a good example of a `final` class.
    - If you were allowed to create a subclass of `String`, the subclass can override `String` methods in certain ways to make its object mutable. Since the subclass objects can be used wherever `String` objects are expected, this would break the contract that `String` objects are immutable.
    - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions (e.g., by overriding superclass methods).

Java `interface`

- We have shown that objects of related classes can be processed polymorphically by responding to the same method call in their own way (they implement common methods in their own way).
- Sometimes, it requires unrelated classes to implement a set of common methods. What should we do?

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, the payment refers to the total cost of the goods listed.
- In the earlier version of the system, every employee type directly or indirectly extends the abstract superclass `Employee`. The system can then manipulate different types of employee objects polymorphically.
  - Think about this: Can we make `Invoice` class extend `Employee`?
  - This is unnatural, the `Invoice` class would inherit inappropriate members (e.g., methods to obtain employee names, which have nothing to do with invoices).
- Interfaces are useful in such cases.

Java `interface`

- What is `interface`? Interfaces define and standardize the ways that objects interact with one another.
  - Controls on a radio serve as an `interface` between users and the radio's internal components (e.g., electrical wiring).
- Interfaces describe a set of methods that can be called on an object, but do not provide concrete implementations for all the methods.
- Different classes (radios) may implement the interfaces (controls) in different ways (e.g., using push buttons, dials, voice commands).

Java `interface`

- An `interface` is often used when disparate (i.e., unrelated) classes need to share common methods and constants.
  - An `interface` is a reference type.
  - You can create an `interface` that describes the desired functionality, then implement this `interface` in any classes that require that functionality.
  - A class can implement any number of `interface`s (making objects polymorphic beyond the constraints of single inheritance).
  - When a class implements an `interface`, it has an **is-a** relationship with the interface data type.

Java `interface`

- Implementing an `interface` allows a class to promise certain behaviors, i.e., forming a contract with the outside world. This contract is enforced at build time by the compiler.
- Interfaces are useful since they capture similarity between unrelated objects without forcing a `class` relationship.

Declaring `interface`

- Like `public abstract` classes, interfaces are typically `public` types.
- A `public interface` must be declared in a `.java` file with the same name as the interface.

Declaring `interface`

- An `interface` declaration begins with the keyword `interface` and contains only **constants** and **abstract methods**.
  - All fields are implicitly `public`, `static` and `final`.
  - All methods declared in an `interface` are implicitly `public abstract`.

```java
public interface Payable {
  double getPaymentAmount(); //calculate payment
} // 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.
- Java 8 introduced `default` and `static` methods in interfaces, we will not consider these new features in this course.

Using interface

- To use an interface, a concrete class must specify that it implements the interface and must implement each method in the interface with specified signature.
- A class that does not implement all the methods of the interface is an **abstract class** and must be declared abstract.

```
public class Invoice implements Payable {
  // must override and implement the getPaymentAmount() method
}
```

Using interface

- An interface extends an interface.
- A class extends a class.
- A class implements an interface.

```java
public interface Doable extends Payable {
  // can define some specific abstract methods
}
```

Using `interface`

- 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`

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

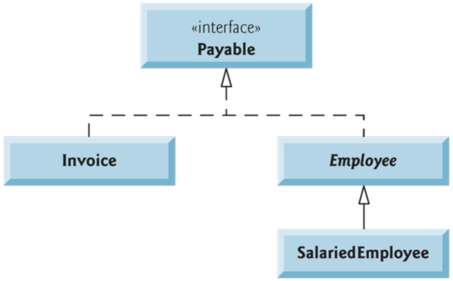
interface vs. abstract class

abstract class	interface
An abstract class can extend only one class or one abstract class.	An interface can extend any number of interfaces.
An abstract class can extend another concrete class or abstract class.	An interface can only extend another interface.
An abstract class can have both abstract and concrete methods.	An interface can have only abstract methods.
In abstract class keyword “abstract” is needed to declare an abstract method.	In an interface keyword “abstract” is optional to declare an abstract method.
An abstract class can have constructors.	An interface cannot have a constructor.
An abstract class can have protected and public abstract methods.	An interface can have only have public abstract methods.
An abstract class can have static, final or static final variables with any access.	An interface can only have public static final (constant) variable.

Example: Developing a `Payable` hierarchy

- Extend the earlier payroll system to make it able to determine payments for both employees and invoices.
  - Classes `Invoice` and `Employee` both represent things for which the company must be able to calculate a payment amount.
  - We can make both classes implement the `Payable` interface, so a program can invoke method `getPaymentAmount` on both `Invoice` and `Employee` objects.
  - Enables the polymorphic processing of `Invoice`s and `Employee`s.

The UML class diagram



- The UML expresses the relationship between a `class` and an `interface` as realization.
    - A `class` is said to “realize” or implement the methods of an `interface`.
- A subclass inherits its superclass’ realization relationships

Interface `Payable`

- Interface methods are always public and abstract, so they do not need to be explicitly declared as such.
- Interfaces can have any number of methods (no implementation is allowed).
- Interfaces may also contain fields that are implicitly `final` and `static`.

```
public interface Payable {
  double getPaymentAmount(); //calculate payment
} // end interface Payable
```

Class `Invoice`

- 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.
- To implement more than one interface, use a comma-separated list of interface names after keyword implements in the class declaration, as in:

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

Class Invoice

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

  // constructor
  public Invoice(String partNumber, String partDescription, int quantity,
      double pricePerItem) {
    if (quantity < 0) // validate quantity
      throw new IllegalArgumentException("Quantity must be >= 0");

    if (pricePerItem < 0.0) // validate pricePerItem
      throw new IllegalArgumentException("Price per item must be >= 0");
```

Class Invoice

```java
      this.quantity = quantity;
      this.partNumber = partNumber;
      this.partDescription = partDescription;
      this.pricePerItem = pricePerItem;
   } // end constructor
   // get part number
   public String getPartNumber() { return partNumber; } // should validate
   // get description
   public String getPartDescription() { return partDescription; }
   // set quantity
   public void setQuantity(int quantity) {
      if (quantity < 0) // validate quantity
         throw new IllegalArgumentException("Quantity must be >= 0");

      this.quantity = quantity;
   }
```

Class Invoice

```java
   // get quantity
   public int getQuantity() { return quantity; }

   // set price per item
   public void setPricePerItem(double pricePerItem) {
     if (pricePerItem < 0.0) // validate pricePerItem
       throw new IllegalArgumentException(
         "Price per item must be >= 0");

     this.pricePerItem = pricePerItem;
   }

   // get price per item
   public double getPricePerItem() { return pricePerItem; }
```

Class Invoice

```java
  // 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());
  }

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

Class `Employee`

- When a `class` implements an `interface`, it makes a contract with the Java compiler:
  - The `class` will implement each of the methods in the `interface` or that the `class` will be declared `abstract`.
  - If the latter, we do not need to declare the `interface` methods as `abstract` in the `abstract class` (they are already implicitly declared as such in the `interface`).
  - Any concrete subclass of the `abstract class` must implement the `interface` methods to fulfill the contract (the unfulfilled contract is inherited).
  - If the subclass does not do so, it too must be declared `abstract`.

Class Employee

```java
// Employee abstract superclass that implements Payable.
public abstract class Employee implements Payable {
  private final String firstName;
  private final String lastName;
  private final String socialSecurityNumber;

  // constructor
  public Employee(String firstName, String lastName,
      String socialSecurityNumber) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.socialSecurityNumber = socialSecurityNumber;
  }

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

Class Employee

```java
   // return last name
   public String getLastName() { return lastName; }

   // return social security number
   public String getSocialSecurityNumber() { return socialSecurityNumber;
      }

   // return String representation of Employee object
   @Override
   public String toString() {
      return String.format("%s %s%nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber());
   }

   // 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
```

Class SalariedEmployee

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

```java
// SalariedEmployee class that implements interface Payable.
// method getPaymentAmount.
public class SalariedEmployee extends Employee {
  private double weeklySalary;

  // constructor
  public SalariedEmployee(String firstName, String lastName,
      String socialSecurityNumber, double weeklySalary) {
    super(firstName, lastName, socialSecurityNumber);

    if (weeklySalary < 0.0)
      throw new IllegalArgumentException(
        "Weekly salary must be >= 0.0");
```

Class SalariedEmployee

```
      this.weeklySalary = weeklySalary;
   }

   // set salary
   public void setWeeklySalary(double weeklySalary) {
      if (weeklySalary < 0.0)
         throw new IllegalArgumentException(
            "Weekly salary must be >= 0.0");

      this.weeklySalary = weeklySalary;
   }

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

Class SalariedEmployee

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

   // 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 class SalariedEmployee
```

`SalariedEmployee` and `Invoice`

- 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.
- `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.

SalariedEmployee and Invoice

```java
// Payable interface test program processing Invoices and
// Employees polymorphically.
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);
```

SalariedEmployee and Invoice

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

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

SalariedEmployee and Invoice

```
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
```

SalariedEmployee and Invoice

```
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
```

Common Java `interface`s

- The Java API's interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program.
- The framework code would call certain methods defined in the interfaces and the method calls will be eventually dispatched to the methods implemented in your own classes.

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.

Example: The Comparable interface

```
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);
  }
```

Example: The `Comparable` interface

```java
  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;
  }
}
```