OOP with Java

16. Inheritance and Overriding

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

website

Introduction

- So far, we have basically treated as objects as data structures and used methods to work on them

Introduction

- So far, we have basically treated as objects as data structures and used methods to work on them
- But Object-Oriented Programming is much more

Introduction

- So far, we have basically treated as objects as data structures and used methods to work on them
- But Object-Oriented Programming is much more
- The true power of OOP arises from class inheritance and extension (often called specialization)

Extending Classes

- A class `Student` can *extend* another class `Person`

Extending Classes

- A class `Student` can *extend* another class `Person`
- `Student` inherits all the fields and functionality of the original class `Person`

Extending Classes

- A class `Student` can *extend* another class `Person`
- `Student` inherits all the fields and functionality of the original class `Person`
- We specify this via `class Student extends Person` in the declaration of class `Student`

Extending Classes

- A class `Student` can *extend* another class `Person`
- `Student` inherits all the fields and functionality of the original class `Person`
- We specify this via `class Student extends Person` in the declaration of class `Student`
- `Student` can add own, new data and methods

Inheritance and Compatibility

- Instances of `Student` can be used in any expression where instances of type `Person` are expected, as they inherit all the fields and methods from `Person`

Inheritance and Compatibility

- Instances of `Student` can be used in any expression where instances of type `Person` are expected, as they inherit all the fields and methods from `Person`
- But not the other way around: instances `Person` cannot be used when instances of `Student` are expected, as they may have less fields/functionality

Inheritance and Compatibility

- Instances of `Student` can be used in any expression where instances of type `Person` are expected, as they inherit all the fields and methods from `Person`
- But not the other way around: instances `Person` cannot be used when instances of `Student` are expected, as they may have less fields/functionality
- We can store an instance of `Student` in a variable of type `Person`

Inheritance and Compatibility

- Instances of `Student` can be used in any expression where instances of type `Person` are expected, as they inherit all the fields and methods from `Person`
- But not the other way around: instances `Person` cannot be used when instances of `Student` are expected, as they may have less fields/functionality
- We can store an instance of `Student` in a variable of type `Person`
- With the keyword `a instanceof X` we can check if a variable `a` stores a value compatible to type `X`

Inheritance and Compatibility

- Instances of `Student` can be used in any expression where instances of type `Person` are expected, as they inherit all the fields and methods from `Person`
- But not the other way around: instances `Person` cannot be used when instances of `Student` are expected, as they may have less fields/functionality
- We can store an instance of `Student` in a variable of type `Person`
- With the keyword `a instanceof X` we can check if a variable `a` stores a value compatible to type `X`
- The special value `null` is compatible to all classes

Inheritance and Compatibility

- Instances of `Student` can be used in any expression where instances of type `Person` are expected, as they inherit all the fields and methods from `Person`
- But not the other way around: instances `Person` cannot be used when instances of `Student` are expected, as they may have less fields/functionality
- We can store an instance of `Student` in a variable of type `Person`
- With the keyword `a instanceof X` we can check if a variable `a` stores a value compatible to type `X`
- The special value `null` is compatible to all classes
- Mathematically speaking, `Student` is a subset of `Person`, not all `Person`s are `Student`s, but all `Student`s are `Person`s

Person class with toString

Listing: A Person class with toString Method

```java
/** A class representing a person with constructor and toString method. */
public class Person {
  /** the family name of the person */
  String familyName;
  /** the given name of the person */
  String givenName;

  /** create a person record and set its name */
  Person(String _familyName, String _givenName) {
    this.familyName = _familyName;
    this.givenName = _givenName;
  }

  /** return a string representation of this person record */
  public String toString() {
    return this.givenName + '␣' + this.familyName;
  }
}
```

Student class extending Person

Listing: A Student Class extending class Person

```java
/** A class representing a student */
public class Student extends Person { // class Student extends class Person
  /** the id of the student */
  String id; // Class student adds the new field 'id'

  /** create a student record and set its name and student id */
  Student(String _familyName, String _givenName, String _id) {
    super(_familyName, _givenName); // invoke the inherited constructor of Person setting up the name
    this.id = _id;
  }

  /** a new method */
  void inLecture() { System.out.println("Student␣" + this.toString() + "␣fell␣asleep.");} //$NON-NLS-1$//$NON-NLS-2$

  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static void main(String[] args) {
    Person person = new Person("Weise", "Thomas"); //$NON-NLS-1$//$NON-NLS-2$
    System.out.println(person); // print the result of person.toString()

    Student student = new Student("Chan", "Jacky", "S01"); //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
    System.out.println(student); // print the result of student.toString(): the inherited toString method
    System.out.println(student.id); // print the value of the id field, namely "S01"
    // System.out.println(person.id); // <- we cannot do that, because Person does not have such a field

    student.inLecture(); //invoke the method inLecture implemented by class Student
    //person.inLecture(); <- we cannot do this, because Person does not implement this method

    System.out.println(person instanceof Person); // true: variable person holds an instance of class Person
    System.out.println(person instanceof Student); // false: variable person does not hold an instance of class Student
    System.out.println(student instanceof Student); // true: every instance of Student is also an instance of Person
    System.out.println(student instanceof Person); // true: hence, variable student also holds an instance of person

    person = student; // we can do this, since variable student is guaranteed to hold an instance of Sstudent (or null)
    // student = person; <- but we can never do this, as some persons (like Weise above) are no students
    System.out.println(person); // print "Jacky Chan", the result of person.toString()
    System.out.println(person instanceof Person); // true: variable person holds an instance of class Person
    System.out.println(person instanceof Student); // true: variable person now also holds an instance of class Student
  }
}
```

Inheritance and Constructors

- In the constructor of `Student`, we first invoke the `super` constructor: the constructor inherited from `Person`

Inheritance and Overrides

- OK, we can inherit and override constructors

Inheritance and Overrides

- OK, we can inherit and override constructors
- But we can do more: We can inherit and override methods as well!

Inheritance and Overrides

- OK, we can inherit and override constructors
- But we can do more: We can inherit and override methods as well!
  - just implement the same method `y()` as already implemented in the extended super class again
  - inside the method, we can invoke the old implementation by doing `super.y()`

Inheritance and Overrides

- OK, we can inherit and override constructors
- But we can do more: We can inherit and override methods as well!
    - just implement the same method `y()` as already implemented in the extended super class again
    - inside the method, we can invoke the old implementation by doing `super.y()`
    - tag the old method with `@Override` to mark it as overriding an inherited method

Inheritance and Overrides

- OK, we can inherit and override constructors
- But we can do more: We can inherit and override methods as well!
  - just implement the same method `y()` as already implemented in the extended super class again
  - inside the method, we can invoke the old implementation by doing `super.y()`
  - tag the old method with `@Override` to mark it as overriding an inherited method
- The new method can be called like the old method

Inheritance and Overrides

- OK, we can inherit and override constructors
- But we can do more: We can inherit and override methods as well!
  - just implement the same method `y()` as already implemented in the extended super class again
  - inside the method, we can invoke the old implementation by doing `super.y()`
  - tag the old method with `@Override` to mark it as overriding an inherited method
- The new method can be called like the old method
- We can make a class `Professor`, override `String toString()`, and store an instance of `Professor` in a variable `person` of type `Person`

Inheritance and Overrides

- OK, we can inherit and override constructors
- But we can do more: We can inherit and override methods as well!
  - just implement the same method `y()` as already implemented in the extended super class again
  - inside the method, we can invoke the old implementation by doing `super.y()`
  - tag the old method with `@Override` to mark it as overriding an inherited method
- The new method can be called like the old method
- We can make a class `Professor`, override `String toString()`, and store an instance of `Professor` in a variable `person` of type `Person`
- If we call `person.toString()`, it will call the implementation of `Professor.toString()`, since `person` references an object of type `Professor`!

Professor class extending Person

Listing: A Professor Class extending class Person

```java
/** A class representing a professor */
public class Professor extends Person { // class Processor extends class Person
  /** create a person record and set its name */
  Professor(String _familyName, String _givenName) {
    super(_familyName, _givenName); // invoke the inherited constructor of Person setting up the name
  }

  /** return "Prof. " + the result of super.toString() = Person.toString() */
  @Override // mark this method explicitly as overridden: explicitly remind programmers about this
  public String toString() {
    return "Prof.␣" + super.toString(); // "Prof. " + super implementation of toString() from Person //$NON-NLS-1$
  }

  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static void main(String[] args) {
    Person person = new Person("Chan", "Jacky"); //$NON-NLS-1$//$NON-NLS-2$
    System.out.println(person); // print the result of person.toString()

    Professor professor = new Professor("Weise", "Thomas"); //$NON-NLS-1$//$NON-NLS-2$
    System.out.println(professor); // print the result of professor.toString(): "Prof. " + the result of Person.toString()

    System.out.println(person instanceof Person); // true: variable person holds an instance of class Person
    System.out.println(person instanceof Professor); // false: variable person does not hold an instance of class Professor
    System.out.println(person instanceof Student); // false: variable person does not hold an instance of class Student

    System.out.println(professor instanceof Professor); // true: every instance of Professor is also an instance of Person
    // System.out.println(professor instanceof Student); // will always be false, but compiler sees this and warns us
    System.out.println(professor instanceof Person); // true: hence, variable professor also holds an instance of person

    person = professor; // we can do this, since variable professor is guaranteed to hold an instance of professor (or null)
    // professor = person; <- but we can never do this, as some persons (like Weise above) are no professor
    System.out.println(person); // print "Prof. Thomas Weise", the result of the toString() of the class Professor
    System.out.println(person instanceof Person); // true: variable person holds an instance of class Person
    System.out.println(person instanceof Professor); // true: variable person now also holds an instance of class Student
  }
}
```

Multi-Level Inheritance

- We can build a whole hierarchy of classes

Multi-Level Inheritance

- We can build a whole hierarchy of classes
- We can, e.g., create a new class `ForeignExchangeStudent` extending `Student`

Multi-Level Inheritance

- We can build a whole hierarchy of classes
- We can, e.g., create a new class `ForeignExchangeStudent` extending `Student`
- We can then override any method inherited from `Person`, `Student`, or `Object` (see a bit later)

Multi-Level Inheritance

- We can build a whole hierarchy of classes
- We can, e.g., create a new class `ForeignExchangeStudent` extending `Student`
- We can then override any method inherited from `Person`, `Student`, or `Object` (see a bit later)
- Inheritance is transitive: All `ForeignExchangeStudent`s are `Student`s and all `Student`s are `Person`s, then all `ForeignExchangeStudent`s are also `Person`s

ForeignExchangeStudent class extending Student

Listing: A ForeignExchangeStudent Class extending class Student

```java
/** A class representing a foreign exchange student */
public class ForeignExchangeStudent extends Student { // class ForeignExchangeStudent extends class Student
  /** the home country of the student */
  String homeCountry; // we add a new field

  /** create a student record and set its name, student id, and home country */
  ForeignExchangeStudent(String _familyName, String _givenName, String _id, final String country) {
    super(_familyName, _givenName, _id); // invoke the inherited constructor of Student setting up the name and id
    this.homeCountry = country;
  }

  /** override method inLecture() from Student */
  @Override
  public void inLecture() { super.inLecture(); System.out.println("Then␣wakes␣up."); super.inLecture();}  //$NON-NLS-1$

  /** override toString() from Person */
  @Override
  public String toString() { return super.toString() + "␣from␣" + this.homeCountry; }//$NON-NLS-1$

  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static void main(String[] args) {
    ForeignExchangeStudent student = new ForeignExchangeStudent("Onegin", "Eugene", //$NON-NLS-1$//$NON-NLS-2$
        "S02", "Russia"); //$NON-NLS-1$//$NON-NLS-2$

    System.out.println(student); // print the result of student.toString(): the inherited toString method
    System.out.println(student.id); // print the value of the id field, namely "S02"
    System.out.println(student.homeCountry); // print the value of the home country d field, namely "Russia"

    student.inLecture(); //invoke method inLecture originally overridden by class ForeignExchangeStudent over class Student

    System.out.println(student instanceof ForeignExchangeStudent); // true: ForeignExchangeStudent instances are instances of Student
    System.out.println(student instanceof Student); // true: instances of Student are instances of Person
    System.out.println(student instanceof Person); // true: hence, variable student also holds an instance of person

    Person person = student; // we can do this, variable student is guaranteed to hold an instance of ForeignExchangeStudent (or null)
    // student = person; <- but we can never do this, as some persons are no students
    System.out.println(person); // print "Eugene Onegin from Russia", the result of person.toString()
    System.out.println(person instanceof Person); // true: variable person holds an instance of class Person
    System.out.println(person instanceof Student); // true: variable person now holds an instance of class Student
    System.out.println(person instanceof ForeignExchangeStudent); // true: variable person  holds an instance of class ForeignStudent
  }
}
```

The Class `Object`

- The class `Object` is the base class in Java, the root of the class hierarchy

The Class `Object`

- The class `Object` is the base class in Java, the root of the class hierarchy
- If we do not specify `extends` when creating a class (as we did until this lesson), the class extends `Object`

The Class `Object`

- The class `Object` is the base class in Java, the root of the class hierarchy
- If we do not specify `extends` when creating a class (as we did until this lesson), the class extends `Object`
- The class `Object` provides the method `String toString()`, which returns the `String` representation of the object

The Class `Object`

- The class `Object` is the base class in Java, the root of the class hierarchy
- If we do not specify `extends` when creating a class (as we did until this lesson), the class extends `Object`
- The class `Object` provides the method `String toString()`, which returns the `String` representation of the object
- All instances of all classes can be stored in a variable of type `Object`

The Class `Object`

- The class `Object` is the base class in Java, the root of the class hierarchy
- If we do not specify `extends` when creating a class (as we did until this lesson), the class extends `Object`
- The class `Object` provides the method `String toString()`, which returns the `String` representation of the object
- All instances of all classes can be stored in a variable of type `Object`
- All arrays are instance of `Object`

The Class `Object`

- The class `Object` is the base class in Java, the root of the class hierarchy
- If we do not specify `extends` when creating a class (as we did until this lesson), the class extends `Object`
- The class `Object` provides the method `String toString()`, which returns the `String` representation of the object
- All instances of all classes can be stored in a variable of type `Object`
- All arrays are instance of `Object`
- `String`s are instances of `Object`

Variables of Type `Object`

Listing: Variables of Type `Object`

```java
/** test the interplay of Strings and objects in the class hierarchy */
public class ObjectTest {

  /** The main routine
   *  @param args
   *           we ignore this parameter */
  public static void main(String[] args) {
    Person person = new Professor("Weise", "Thomas"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(person); // "Prof. Thomas Weise"

    String text = person.toString();
    System.out.println(text); // "Prof. Thomas Weise"

    Object object = person; // store person in an object variable
    System.out.println(object); // "Prof. Thomas Weise"

    System.out.println(person == object);// true, both variables reference same object
    System.out.println(text == object);// false, text references a String, object is a Professor

    object = text;
    System.out.println(person == object);// false, object is now a String, person is a Professor
    System.out.println(text == object);// true, text and object reference the same object

    object = new int[34];
    System.out.println(person == object);// false, object is now an int array, person is a Professor
    System.out.println(text == object);// false, object is now an int array, text is a String
  }
}
```

`static` vs. instance methods

- Oh god, this is a tricky one. Prepare yourself.

`static` vs. instance methods

- Oh god, this is a tricky one. Prepare yourself.
- Overriding an inherited method from a super class works by defining a method with the same signature

`static` vs. instance methods

- Oh god, this is a tricky one. Prepare yourself.
- Overriding an inherited method from a super class works by defining a method with the same signature
- Inherited instance methods can be overridden this way

`static` vs. instance methods

- Oh god, this is a tricky one. Prepare yourself.
- Overriding an inherited method from a super class works by defining a method with the same signature
- Inherited instance methods can be overridden this way
- inherited `static` methods cannot be overriden, but are hidden this way

`static` vs. instance methods

- Oh god, this is a tricky one. Prepare yourself.
- Overriding an inherited method from a super class works by defining a method with the same signature
- Inherited instance methods can be overridden this way
- inherited `static` methods cannot be overriden, but are hidden this way
- Let us check examples

Instance Methods: The Classes

Listing: Base Class `A` implementing instance method

```java
/** A base class used for demonstrating instance method overriding */
public class A {
  void doSomething() {
    System.out.println('A');
  }
}
```

Listing: Subclass `B` overriding instance method

```java
/** A subclass used for demonstrating instance method overriding */
public class B extends A { // B extends A and overrides its method
  void doSomething() {
    System.out.println('B');
  }
}
```

Instance Methods: The Test

Listing: Testing the Instance Method Overriding

```
/** Test classes A and B */
public class ABTest {
  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static void main(String[] args) {
    A a = new A();    // create an instance of A
    a.doSomething(); // print 'A'
    B b = new B();    // create an instance of B
    b.doSomething(); // print 'B'

    a = b; // this is allowed, since B inherits from A
    a.doSomething(); // print 'B', since a now contains instance of B
  }
}
```

static Methods: The Classes

Listing: Base Class C implementing static method

```java
/** A base class used for demonstrating static method hiding */
public class C {
 static void doSomething() {
    System.out.println('C');
  }
}
```

Listing: Subclass D overriding static method

```java
/** A subclass used for demonstrating static method overriding */
public class D extends C { // D extends C  and overrides its method
  static void doSomething() {
    System.out.println('D');
  }
}
```

static Methods: The Test

Listing: Testing the static Method Hiding

```
/** Test classes C and D */
public class CDTest {
  /** The main routine
   *  @param args
   *            we ignore this parameter */
  public static void main(String[] args) {
    C c = new C();    // create an instance of C
    c.doSomething(); // print 'C'
  // ^- the Eclipse compiler will complain about that (and rightly so!)

    D d = new D();    // create an instance of D
    d.doSomething(); // print 'D'
  // ^- the Eclipse compiler will complain about that (and rightly so!)

    c = d; // this is allowed, since D inherits from C
    c.doSomething(); // print still 'C', since static methods are not
        overridden
  // ^- the Eclipse compiler will complain about that (and rightly so!)
  }
}
```

static Methods: What Actually Happens

Listing: static Method Hiding: What Happens

```
/** Test classes C and D: what actually happens */
public class CDActual {
  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static void main(String[] args) {
    C c = new C();   // create an instance of C, but never actually use it
   // ^- the Eclipse compiler will complain about that (and rightly so!)

    C.doSomething(); // print 'C'
    D d = new D();   // create an instance of D,  but never actually use it
    D.doSomething(); // print 'D'

    c = d; // this is allowed, since D inherits from C
    C.doSomething(); // print still 'C', since static methods are not
        overridden
  }
}
```

`static` vs. instance methods

- That was a tricky one

`static` vs. instance methods

- That was a tricky one
- Summary: You cannot override static methods.

Summary

- We have learned about the inheritance / subclassing / `extends`.
- It allows us to define a hierarchy of classes
- A subclass inherits methods and variabes from its super class
- It can add new methods and new variables
- It can override instance methods (but not `static` ones)
- All classes inherit from `Object`, so each object is an `instanceof Object`
- The class hierarchy allows us to share common variables and code and reduce program complexity

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog