Object-Oriented Programming

Inheritance and polymorphism

Anders Jonsson & Federico Heras
2023-24

Course topics

Topic 1 Introduction and the concept of objects

Topic 2 The object-oriented programming paradigm

Topic 3 Object modelling and relations between objects

Topic 4 Inheritance and polymorphism

Topic 5 Abstract classes and interfaces

Topic 6 Reuse and study of problems solved using objects

Inheritance

- Makes it possible to specialize the definition of an existing superclass
- All instance members of the superclass are inherited
- Reuse: the definition of the superclass is reused
- Inheritance also makes it possible to group common members
- Reuse: avoids duplication of attributes and methods

Java code

```java
public class Person {
     protected String name; // attribute "name"
     protected int age;     // attribute "age"

     public Person( String n, int a ) {
          name = n;
          age = a;
     }

     public String getName() {
          return name;
     }

     public int getAge() {
          return age;
     }
}
```

Inheritance

```
public class Student extends Person {
    private int numberOfCourses;

    public Student( String n, int a, int c ) {
        super( n, a );
        numberOfCourses = c;
    }

    public int getNumberOfCourses() {
        return numberOfCourses;
    }
}
```

Program

```
public class TestStudent {
    public static void main( String[] args ) {
        Student s1 = new Student( "Juan", 19, 5 );
        Student s2 = new Student( "Eva", 20, 4 );

        String name = s1.getName();
        int courses = s2.getNumberOfCourses();
    }
}
```

Theory session 6

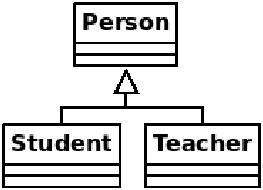
Polymorphism

Casting

Polymorphism

- Greek word, literally “many forms”
- The same object can behave in different ways depending on the context

Intuition



- Each instance of Student is also an instance of Person
  - has all the attributes of a Person
  - can apply all methods of Person
- To what class does a Student instance belong?

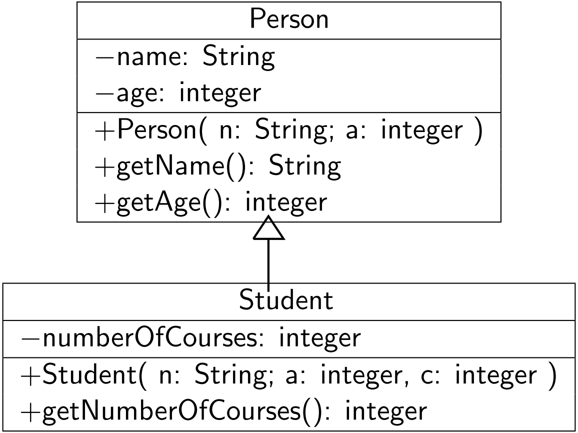
Polymorphism

- Based on two mechanisms:
  1. Declared type of variable $\neq$ type of instance stored
  2. Overriding methods in subclasses
- If we send a message to an object stored in a variable
  - the type of the instance is unknown at compile time
  - the interpretation of the message depends on the concrete instance!
- Another form of abstraction: actual type of instance is hidden!

Polymorphic variables

- Each variable has two associated types:
    - Declared type: the type specified when defining the variable
    - Instantiated type: the type of the instance stored
- The two types are not necessarily the same!
- Instantiated type: subclass (direct or indirect) of the declared type
- We can only call methods of the declared type

Example



Person

−name: String

−age: integer

+Person(n: String; a: integer)

+getName(): String

+getAge(): integer

Student

−numberOfCourses: integer

+Student(n: String; a: integer, c: integer)

+getNumberOfCourses(): integer

Program

```
public class TestStudent {
    public static void main( String[] args ) {
        Student s = new Student( "Eva", 20, 4 );
        Person person = s; // permitted
        person.getName(); // permitted
        person.getNumberOfCourses(); // prohibited

        Person p = new Person( "Juan", 19 );
        Student student = p; // prohibited
    }
}
```

Overriding

- Override: redefine an existing method of the superclass in the subclass
- Conserves the signature but changes the implementation
- Signature of a method:
  - Identifier
  - Return type
  - Argument list with types

Motivation

"Now Speak!"
"Quack"
"Woof"
"Meow"

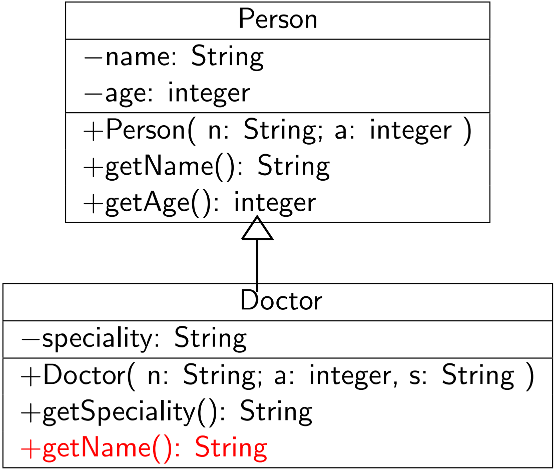
Overloading vs overriding

- Overload:
  - Define multiple methods with the same name
  - Different signature: effectively treated as different methods

```
public void println( int value );
public void println( String text );
```

- Override:
  - Redefine a method of the superclass in the subclass
  - Same signature: effectively treated as two versions of the same method

Overriding - example



Implementation

```
public class Person {
     ...
     public String getName() {
          return name;
     }
}

public class Doctor extends Person {
     ...
     // overriding
     public String getName() {
          return "Dr. " + name;
     }
}
```

Program

```
public class TestDoctor {
     public static void main( String[] args ) {
          Doctor d = new Doctor( "Maria", 35,
                                 "Neurology" );
          Person person = d;
          person.getName();
     }
}
```

Messages

- When sending a message to an object stored in a variable
  - The type of the instance is unknown at compile time
  - The subclasses may define different versions of the method!
- How do we determine which version of the method to execute?
- When do we determine which version of the method to execute?

Polymorphic messages

- How do we determine which version of a method to execute?
- Solution: first search for the method in the class of the instantiated type
- If the method is not defined there, search in the superclass
- Keep searching upwards in the hierarchy until method found
- Guarantee: method is defined in the class of the declared type
- Exception: constructor methods

Program

```
public class TestDoctor {
     public static void main( String[] args ) {
          Doctor d = new Doctor( "Maria", 35,
                                 "Neurology" );
          Person person = d;
          person.getName();
          person.getAge();
     }
}
```

Overriding - Consequences

- The code of the method in the superclass is no longer reused!
- However, the subclass reuses the API of the superclass
- We haven't changed what an instance is capable of doing (its methods), but we have changed how it does it (the code of the methods)

The keyword `super`

- Already discussed the semantics for constructor methods
- Can also be used to access instance members of the superclass:
  - `super.name; // attribute of the superclass`
  - `super.getName(); // method of the superclass`
- Similar to `this`, but treats an instance as an object of the superclass

Implementation

```
public class Person {
     ...
     public String getName() {
          return name;
     }
}

public class Doctor extends Person {
     ...
     // overriding
     public String getName() {
          return "Dr. " + super.getName();
     }
}
```

The keyword `final`

- For attributes: the value of the attribute cannot change
- For methods: the method can not be overridden
- For classes: the class can not be extended (inherited from)

Binding

- Binding = determine which version of a method to associate with a method call
- When do we determine which version of the method to execute?
  - Static binding: at compile time
  - Dynamic binding: at execution time
- Dynamic binding is necessary to achieve polymorphism!

Binding

- Java: dynamic binding by default
  - static binding for methods that are `final`, `private` and/or `static`
- C++: static binding by default
  - dynamic binding is achieved by explicitly including the keyword `virtual`
- Python: different way of implementing polymorphism since variables have no declared type!

Polymorphism in C++

```
class Person {
     ...
     virtual std::string getName() { return name; }
};

class Doctor : public Person {
     ...
     std::string getName() { return "Dr. " + name; }
};

int main() {
     Person * person = new Doctor( "Maria", 35,
                                   "Neurology" );
     person->getName();
}
```

Shadowing

- The equivalent of overriding for attributes
- Redefine an existing attribute of the superclass in the subclass
- Different effect: an instance now has two attributes with the same name!
- Mostly just causes confusion, better to avoid

Polymorphism - Application

- Generic programming: treat a collection of instances as if they were objects of the same class

```
Person[] people = new Person[5];
...
for ( int i = 0; i < 5; ++i )
     System.out.println( people[i].getName() );
```

- The overridden methods are automatically executed!

Polymorphism - Advantages

- More generic and simple code
  - Augments the level of abstraction
- Makes it possible to call a (possibly overridden) method without worrying about the specific instantiated type
  - Augments the level of encapsulation

Theory session 6

Polymorphism

Casting

Casting

- Casting = store the same instance in a different variable with different declared type
- The declared type changes, but not the instantiated type
- Upcast: change the declared type to a more general type (superclass)
- Downcast: change the declared type to a more specific type (subclass)

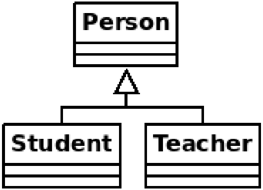
Casting - Motivation

- Store an instance in a variable with different declared type: why?
- Upcast: generic programming, treat instances as if they were from the same class

```
Person[] people = new Person[5];
...
for ( int i = 0; i < 5; ++i )
     System.out.println( people[i].getName() );
```

- Downcast: to call methods that are not defined in the class of the declared type

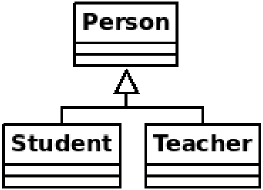
Upcast



- Always permitted: an instance of the subclass is also an instance of the superclass

```
Student s = new Student( "Eva", 20, 4 );
Person person = s; // upcast
```

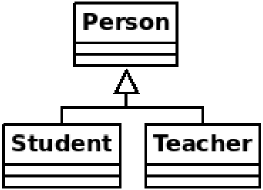
Downcast



- Can fail: an instance of the superclass may not be of the correct subclass

```
Teacher t = new Teacher( "Oscar", 38 );
Person person = t; // upcast
Student student = person; // prohibited
```

Downcast



- Solution: add an explicit cast

```
Student s = new Student( "Eva", 20, 4 );
Person person = s; // upcast
Student student = (Student)person; // can fail!
person.getNumberOfCourses(); // prohibited
student.getNumberOfCourses(); // permitted
```

The keyword instanceof

- Allows to test whether an instance is of a certain subclass
- Returns a boolean value (true or false)
- Contributes to avoiding problems with downcast

```
Person person = new Student( "Eva", 20, 4 );
if ( person instanceof Student ) {
     Student student = (Student)person;
     student.getNumberOfCourses();
}
person instanceof Person; // true
person instanceof Teacher; // false
```

Exercise

- Adapt the example of the classes Ellipse-Circle to override the method to compute the circumference
- It is easier to compute the circumference of a circle than that of an ellipse

Exercise

- Define a class `MyVector` that represents vectors of objects
- Implement a method `contains` that checks whether the vector already contains a given object
- Define a class `MyClass` that overrides the method `equals` of Object

The method equals

- Method of the class Object (superclass of all other classes)
- Useful to test whether two instances are equal
- Example:

```
MyClass a = new MyClass( 1 );
MyClass b = new MyClass( 1 );
if ( a == b )// false (compares memory addresses)
if ( a.equals( b ) )// true (uses method equals)
```

Summary

- Polymorphism
- Polymorphic variable
- Overriding
- Casting