Introduction to Programming - Concepts and Tools

Lecture 5

Inheritance

Overview

- Extending a superclass to a subclass
- The subtype principle and polymorphism
- Generic methods
- Access modifiers (private, public, package, protected)
- Abstract classes and methods
- Interfaces
- Multiple inheritance
- Wrapper classes (if time permits)

Motivation

Three distinct classes like:

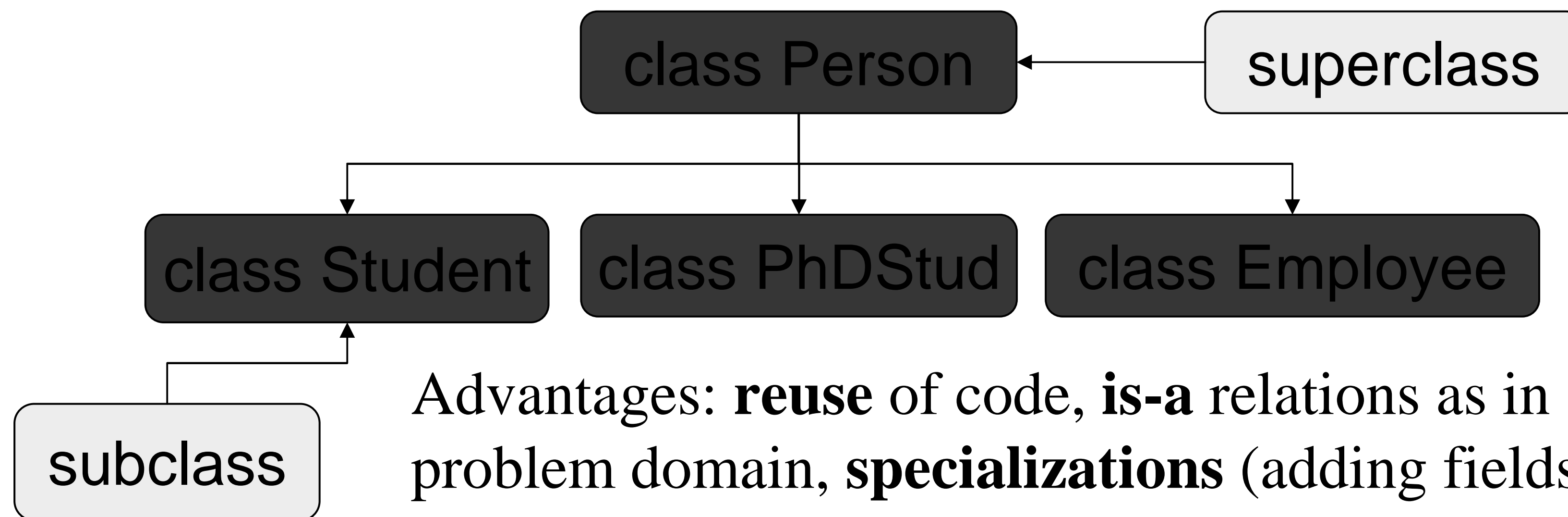
class Student | class PhDStud | class Employee

most likely have common fields and methods, e.g.

```
class Student {
  private int cpr;
  private String name;
  …
  String getName () { return name; }
}
```

Motivation

What about a **super class** from which the (sub) classes can **inherit** common fields and methods giving rise to an **inheritance hierarcy**.



Advantages: **reuse** of code, **is-a** relations as in problem domain, **specializations** (adding fields and methods in subclasses), **overriding** superclass methods.

```
class Person {

  private int cpr;
  private String name;
  private char gender; //male == 'M' , female == 'F'

  Person() {};
  Person(String name, int cpr, char gender) {
      this.name = name; this.cpr = cpr; this.gender =
    gender; }

  void setCPR(int cpr)        { this.cpr = cpr; }
  void setGender(char gender){ this.gender = gender; }
  void setName(String name)  { this.name = name; }
  int getCPR()               { return cpr; }
  char getGender()           { return gender; }
  String getName()           { return name; }
  public String toString() {
    return ("Name: " + name + ", Gender: " + gender);
  }
}
```

```
class Student extends Person  {

  private String studyline;
  private short year;

  Student() {}; // calls implicitly default constructor in Person
  Student(String name, int cpr, char gender, short year, String
     line) {
      super(name,cpr,gender); this.year = year; studyline = line; }
  // calls first explicitly constructor in Person

  void setLine(String studyline) { this.studyline = studyline; }
  void setYear(short year) { this.year = year; }
  String getLine() { return studyline; }
  short getYear() { return year; }
  public String toString() {  // overriding
    return(super.toString() + "\n  " +
      "Line: " + studyline + ", Year: " + year); }
}
```

All fields and methods (not constructors) are **inherited**.
Class Person is **extended** with new fields and methods.

```
class StudentTest {

  public static void main(String[] args) {

     Student s1 = new Student();
     Student s2 =
          new Student("Bob",12345678,'M',(short)2004,"INT");
     s1.setCPR(87654321);
     s1.setGender('F');
     s1.setName("Alice");
     s1.setLine("Open University");
     s1.setYear((short)2003);
     System.out.println(s1);
        // System.out.println(s1.toString());
  }
}
```

Methods definded in a superclass, e.g. `setCPR`, can be called by instances of its subclasses.

`println` calls the *overridden* `toString`.

The Subtype Principle

Because all methods needed for a superclass object are available in a subclass object:

A subclass object can always be used where an object of its superclass is expected.

```
Person p = new Student("Bob",12345678,'M',(short)2004,"INT");
p.getName();  // available methods in class Person are also
   available in class Student
p.getLine();  // illigal, not available in class Person
p.toString(); // which method is called?
```

Pure Polymorphism

```
Person p = new Person("Cliff",11223344,'M');
Person q = p;
System.out.println(q); //Name: Cliff, Gender: M

q = new Student("Bob",12345678,'M',(short)2004,"INT");
// valid because of the subtype principle
System.out.println(q); //Name: Bob, Gender: M
                       //  Line: INT, Year: 2004
```

When calling an overridden method, like `toString`, JVM selects at runtime the method defined for the type (class) of the actual object.

Pure Polymorphism

```
class PolymorphismTest { …

   Person p1 = new Person("Cliff",11223344,'M');
   Student s1 = new
    Student("Bob",12345678,'M',(short)2004,"INT");
   Student s2 = new
    Student("Bob",12345678,'M',(short)2004,"INT");

   Person[] arrayOfPersons = { p1, s1, s2 }; // an
    array, details in next lecture

   for( int i=0; i < arrayOfPersons.length; i++)
      System.out.println(arrayOfPersons[i]);
```

`class Object`

`Class Object` is the root of any class hierarchy, a class **implicitly** inherits from `class Object`.

`class Object` contains (among others) the **generic** methods `toString()` and

`equals(Object obj)`

which can be called from any instance of a class.

`o1.equals(o2)` returns true if `o1` and `o2` refers to the same object. This may be too strict so often `equals()` is overridden.

Using the standard `equal`

```
Person p = new Person("Cliff",11223344,'M');
Person q = p;
//p and q referes to the same object

if (q.equals(p))
  System.out.println("equal");
else System.out.println("not equal");

q = new Person("Cliff",11223344,'M');
// p and q refers to similar although distinct
     objects

if (q.equals(p))
  System.out.println("equal");
else System.out.println("not equal");
```

Using the standard equal

```
Student s1 = new
    Student(”Alice",87654321,’F',(short)2004,”MMT");
Student s2 = new
    Student("Bob",12345678,'M',(short)2004,"INT");
Person p1 = new Person("Bob",12345678,'M');
//p1 and s2 are the same person

Person[] arrayOfPersons = { p1, s1, s2 };

int count = 0; //counts persons identical to p1
for(int i=0; i < arrayOfPersons.length; i++) {
  if (p1.equals(arrayOfPersons[i]))
      count++;
}
System.out.println(count); // returns 1, not 2
```

Overriding `equals`

```
class Person { …

public boolean equals(Object obj) {
  if (obj instanceof Person)
       return this.cpr == ((Person)obj).cpr;
  else return super.equals(obj);
}

…

class EqualsTest { …

   int count = 0; //counts persons identical to p1
   for( int i=0; i < arrayOfPersons.length; i++) {
     if (p1.equals(arrayOfPersons[i]))
       count++;
   }
   System.out.println(count); // returns 2
```

Overriding vs. Overloading

- **Overloading** means two methods in a class with the same name distinguished by their signature (e.g. `Person()` and `Person(String name, int cpr, char gender)`
- **Overriding** is to redefine a superclass method in a subclass keeping the same signature.

Private Access

```
class Person {

  private int cpr;
  private String name;
  private char gender; //male == 'M' , female == 'F’
    ...
}
```

Methods and fields in a class can be **private**. *They can be referred only from methods in the same class.*

E.g. methods in class `Student` cannot refer to `cpr`, `name`, and `gender` in class `Person`

Package Access

A **package** is a collection of classes.

```
package person;
class Person {
  int cpr; // package access is the default
           // cpr may be accessed by methods in Student
  ...
}

package person;
class Student extends Person { ... }
```

Fields and methods defined without any access modifiers can be accessed only from a method within the same package.

Hiding classes

A package may hide classes

```
package person;
class Person { // not accessible outside package
  int cpr;
  ...
}

package person;
public class Student extends Person { // externally accessible
 ...
}
```

A class is either declared public or with package access.

Public access

Methods and fields must be public to be accessible outside the package (see e.g. `class PackageTest`).

```
package person;
class Person { // not accessible outside package
  ...
   public void setCPR(int cpr) { this.cpr = cpr; }
  // accessible outside the package through instances of Student
  ...
}

package person;
public class Student extends Person { // accessible outside the
    package
  ...
   public Student(String name, int c, char g, short y, String l)
    { super(n,c,g); year = y; studyline = l; }
  // constructors must be public to be accessible outside
    package
  ...
```

Public access of `main()`

Any classes in a single directory that don't specify a package name are part of the same unnamed package.

That's why `main()` must be public, it must be exported outside its unnamed package. It's called by the JVM.

Access Modifiers and the Subtype Principle

When a method is overridden the access right cannot be restricted, otherwise the subtype principle would be violated.

E.g. `equals()` and `toString`() must be public in subclasses of `Object`.

Protected Access

Suppose we would like to export class `Student` from package `person`.

```
import person.*;
class PhDStudent extends Student {
  ...
}
```

A subclass may access a **protected** member of a superclass, if that access is made by a subclass reference.

Protected Access

```
package person;
public class Student extends Person  {
     protected String studyline;
     protected short year;
   …
}

import person.*;
class PhdStudent extends Student {
  ...
  public boolean isEqualYear(PhDStudent p)
    { return this.year == p.year; }
    // this and p are both subclass references
}

class ProtectedTest { …
  if (phd2.isEqualYear(phd1)) … }
```

Protected Access

- From within the same package protected access is the same as package acces.
- Outside the package a protected member of a class C is like a private member (unless the access is from a derived class of C)
- Avoid protected access, it weakens data hiding.

Abstract Classes

Suppose `Person` is a (local) root of a class hierarcy, e.g.

```
public class Student extends Person { … }
public class Employee extends Person { … }
```

If no instances of `Person` is needed it should be declared to be ***abstract***.

```
abstract class Person { … }
```

There can be no instances of an abstract class `C`, a reference to `C` always point to an object of a subclass of `C`.

Abstract Methods

An abstract class may leave methods undefined, they must be declared to be *abstract*, e.g.

```
abstract class Person {
    private int cpr;
    …
     public void setCPR(int cpr) { this.cpr = cpr; }
    …
    abstract public boolean sameType(Person p);
}
```

i.e. the implementation of `sameType` is *deferred* to subclasses.

Note the parameter (variable) of an abstract type.

Abstract Methods

The abstract method `sameType()` may be implemented by (see `AbstractTest.java`)

```
class Employee extends Person{
    …
    public boolean sameType(Person p) {
      return p instanceof Employee; }
    …
}

class Student extends Person {
    …
    public boolean sameType(Person p) {
      return p instanceof Student; }
    …
}
```

Interfaces

An **interface** is a design specification containing *constants* and public *abstract methods,* it contains no instance variables.

```
interface Person {
  char Male = 'M'; // don’t need to write final
  char Female = 'F’;

  void setCPR(int c); // don’t need to write abstract public
  void setGender(char g);
  void setName(String n);
  int getCPR();
  char getGender();
  String getName();
  boolean sameType(Person person);
}
```

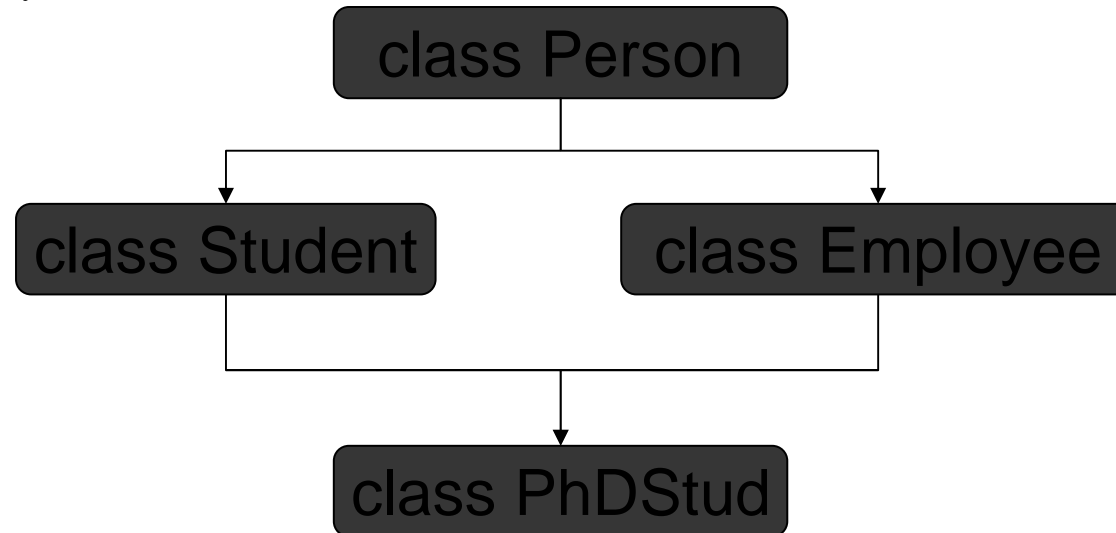
A class implementing interface `I` **must** implement its methods. There can be no instances of `I` (a reference to `I` points to an instance of a class implementing `I`).

Interfaces

```
public class Student implements Person  {

      private int cpr;
      private String name;
      private char gender;    // instance variables added
      private String studyline;
      private short year;

      public Student() {};
      public Student(String n, int c, char g, short y, String l) {
          name = n; cpr = c; gender = g; year = y; studyline = l; }

      // public set and get functions omitted

      public boolean sameType(Person person) {
          return person instanceof Student;
      }
  }
```

Multiple Inheritance

What if a PhD student is both a student and an employee?



Java does *not* allow to extend multiple classes, but it allows to implement multiple interfaces.

Implemting multiple interfaces

```
public interface Student extends Person  {
// extending interface Person
  void setLine(String s);
  void setYear(short y);
  String getLine();
  short getYear();
}

public interface Employee extends Person  {

   void setPosition(String position);
   void setSalary(short salary);
   String getPosition();
   short getSalary();
}
```

Implemting multiple interfaces

```
class PhDStudent implements Student, Employee {

  private String name, studyline, position;
  private int cpr;
  private char gender;
  private short year, salary;

  public PhDStudent(String n, int c, char g, String l, short y,
     short s, String p) { … }

  // all the set methods from Person, Student, Employee
  …
  // all the get methods from Person, Student, Employee
  …

  public boolean sameType(Person person) { // required by Person
      return person instanceof PhDStudent;
  }
}
```

Wrapper Classes

The *wrapper classes*: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` wraps a primitive type inside a class. They are *public final* and cannot be extended.

E.g. `Integer(int)` is a constructor for the wrapper class for `int`.

`Number` is the superclass for the wrapper classes for the primitive numeric types. It provides methods for converting to any primitive numeric type, e.g. `intValue()` and `doubleValue()`.

```
Number min = array[0];
for(int i=0; i < array.length; i++) {
  if (array[i].doubleValue() < min.doubleValue())
      min = array[i];
}
System.out.println(min.doubleValue());
```

The End