320341 Programming in Java

Fall Semester 2014

Lecture 7: Inheritance, Polymorphism, Abstract Classes

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Outline

- Objectives
- Inheritance
- Polymorphism
- Abstract Classes

Objectives

The objective of this lecture is to

- Introduce inheritance and polymorphism
- Introduce design guidelines for using inheritance and polymorphism

Reuse

Our Aim

- We want to apply previous knowledge to the current problem
- We want to reuse existing functionality to the current problem

We can achieve this by

- Using Composition (sometimes called `Black Box Reuse`)
  - ❑ obtain new functionality by using aggregation
  - ❑ new object is an aggregation of existing components
- Using Inheritance (sometimes called `While Box Reuse`)
  - ❑ obtain new functionality by using inheritance

Forms of Inheritance

Inheritance for describing taxonomies

- Detected by specialization
- Detected by generation

Inheritance for reuse

- Specification inheritance
  - ❑ Sometimes called subtyping

- Implementation inheritance (sometimes called class inheritance)
  - ❑ Similar class already exists e.g., wants to implement a Stack & List
  - ❑ Operations may exhibit undesired behavior

Introduction to Inheritance

Create a new class as a *type of* an existing class (**inheritance**)

- ❑ The new class inherits ***methods*** and ***fields*** of an existing class
- ❑ The new methods and fields can be ***added*** to the newly created class
- ❑ The inherited methods can be ***adapted*** to new class

The lecture focuses on **inheritance**

- *Inheritance is one of the cornerstones of OOP*
- *The "is-a" relationship is the hallmark of inheritance*

Introduction to Inheritance

The Java keyword extends is used to denote inheritance

superclass,
base class,
parent class

```
class Employee
    {
    // instance fields

    public String getName()  {…}
    public double getSalary()  {…}
    public Date getHireDay()  {…}
    public void raiseSalary(double byPercentage)  {…}

}

// inherit from Employee class

class Manager extends Employee {

        public setBonus (double bonus) {
                        this.bonus = bonus;
         }
    private double bonus;
    }
```

subclass,
derived class,
child class

- **Manager** is a new class that *derives* from the **Employee** class

Introduction to Inheritance

The subclass introduced a new field and method

```
// inherit from Employee class

class Manager extends Employee {
        private double bonus;
        …

        public setBonus (double bonus) {
                this.bonus = bonus;
        }
                                      }
```

- A **Manager** object can apply the *setBonus* method
- An **Employee** object *cannot* apply the *setBonus* method
  - ž *setBonus* is not among the set of methods in the **Employee** class

- However a **Manager** object can use **Employee** methods
- ***Methods Inherited from the superclass can be used by subclass objects***

Introduction to Inheritance

Observe that:

- A subclass normally *adds* its own fields and methods

- Therefore, a subclass is *more specific* than its superclass

- A subclass represents a *more specialized* group of objects

- Typically a subclass exhibits the behavior of its superclass and ***additional behaviors*** that are specific to the class

  - ❑ That is why inheritance is sometimes referred to as **specialization**

  - ❑ Superclasses tend to be “more general” and subclasses “more specific”

Introduction to Inheritance

Direct superclass

- The superclass from which the subclass explicitly inherits

Indirect superclass

- Any class above the direct superclass in the class hierarchy
- In Java the class hierarchy begins with class **Object** (**java.lang.Object**)

Every class in Java directly or indirectly extends the class Object

is-a Relationship versus *has-a* Relationship

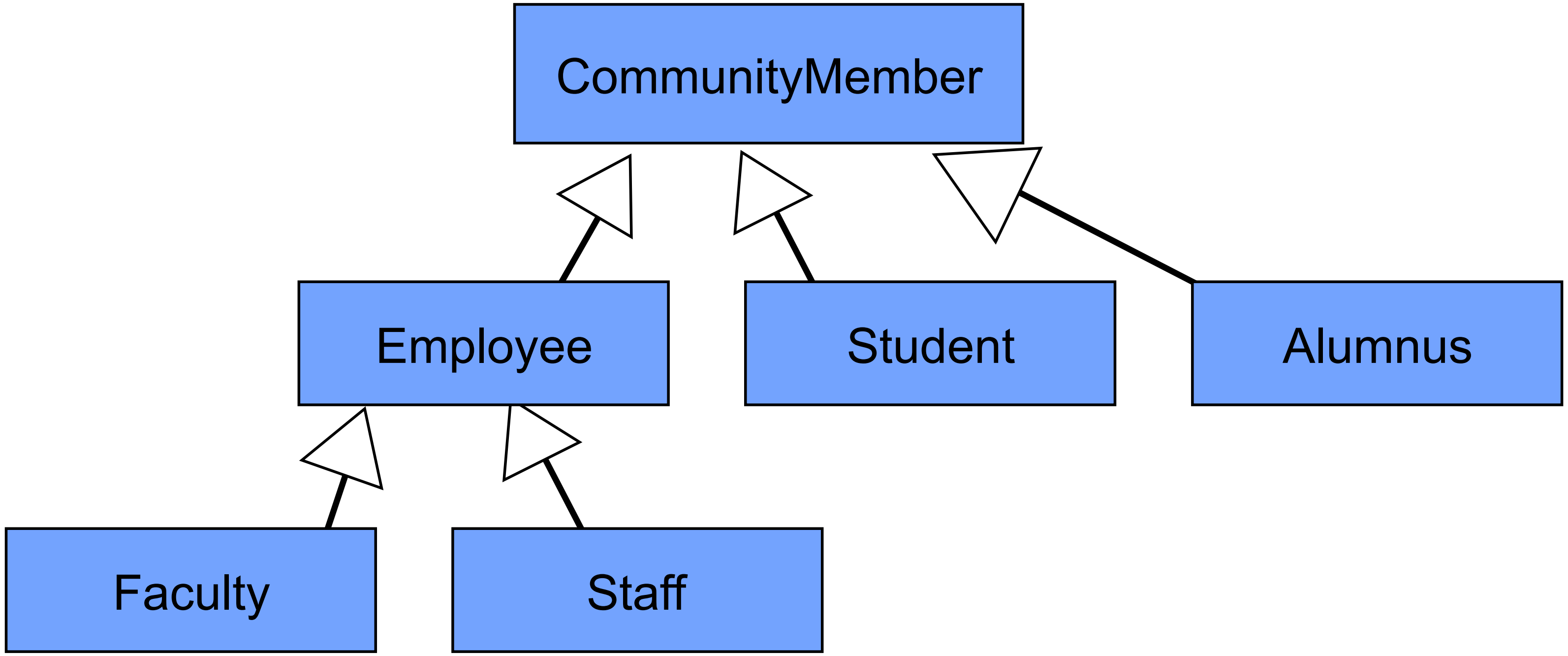
Is-a represents inheritance

- *An object of a class can also be treated as an object of its superclass*
- *Example: a car is a vehicle*
- *Superclass objects cannot be treated as objects of their subclasses*
- Ex. All cars are vehicles, but not all vehicles are cars

Has-a represents composition

- An object contains as its members, references to other objects
- Ex. a car *has a* steering wheel (and a car object has a reference to a steering wheel object)
- Ex. Given the classes **Employee**, **BirthDate**, **TelephoneNumber** we can say an **Employee** *has-a* **BirthDate** and an **Employee** *has-a* **TelephoneNumber**

Inheritance Hierarchy



Inheritance hierarchy for university CommunityMembers

Method Overriding

Adapt some superclass methods to specialized needs of the subclass

This is achieved by redefining appropriate superclass methods

The technique of redefining superclass methods in the subclass is called ***method overriding***

In summary, a subclass can:

1. *Add new fields*
2. *Add new methods*
3. *Override superclass methods*

A subclass cannot take away superclass fields or methods

Method Overriding

JACOBS UNIVERSITY

Example

```
class Employee {
        private String name;
        private double salary;
        private Date hireDay;
        …
        public Employee (String n, double s, int year, int month, int day) {
        …
        }
        public double getSalary() { return salary; }
}
```

```
class Manager extends Employee {
        …
        public double getSalary() {
                double baseSalary = super.getSalary();
                return baseSalary + bonus;
        }
        private double bonus;
                                                        }
```

Method getSalary() overridden in subclass

super calls a superclass method

Access Modifiers

A class's public members are accessible wherever the program has a reference to an object of that class or one of its subclasses

A class's private members are accessible only from within the class itself

A superclass's private members are not inherited by its subclasses!

Access Modifiers

A class's protected members can be accessed by

(1) members of that superclass,

(2) members of its subclasses,

(3) members of other classes in the same package

- *protected members also have package access*

All public and protected superclass members retain their original access modifier when they become members of the subclass

Access Modifiers Summary

Visible to the class only (private)

Visible to the world (public)

Visible to the package and all subclasses (protected)

Visible to the package – the (default). No modifiers are needed

Access Modifiers Summary

A subclass cannot access the private fields of its superclass

Sometimes you want to restrict a method to subclasses only

Declare the class feature as protected

protected features in Java are visible to all subclasses as well as to all other classes in the same package!!

Superclass Constructor

The superclass constructor is called using special **super** syntax

- *If a superclass constructor is not called explicitly, the default (no parameter) constructor is invoked*

Subclass constructor

```
class Manager extends Employee {
        public Manager(String n, double s, int year, int mth, int day) {
                super(n, s, year, mth, day);
                bonus = 0;
        }
        private double bonus;
                                                }
```

- Calls the superclass constructor
- Must be the first statement in subclass constructor

Example

```
class Employee {
        public Employee (String n, double s, int year, int month, int day) {
        …
                }
        public double getSalary() {return salary;}
        …
        private String name;
        private double salary;
        private Date hireDay;
}
```

```
class Manager extends Employee {
        private double bonus;

        public Manager(String n, double s, int year, int mth, int day) {
                super(n, s, year, mth, day);    bonus = 0;
        }

        public setBonus (double bonus) {
                this.bonus = bonus;

        }

}
```

Polymorphism

Example: populate the array with a mix of managers & employees

Manager object created

```
Manager boss = new Manager(„Carl“, 80000, 1987, 12, 15);
boss.setBonus(5000);
```

Employee objects

```
Employee [] staff = new Employee[3];
staff[0] = boss;  // The actual type of staff[0] is Manager
staff[1] = new Employee(„Harry“, 50000, 1989, 10, 1);
staff[2] = new Employee(„Tommy“, 40000, 1990, 3, 15);
```

picks the correct *getSalary()* method

```
for (Employee e : staff)                    System.out.println
(e.getName()+“ „ +e.getSalary());
```

```
// Salary printout
Carl 85000      // base salary + bonus printed – Manager object
Harry 50000     // base salary printed – Employee object
Tommy 40000 // base salary printed – Employee object
```

Polymorphism

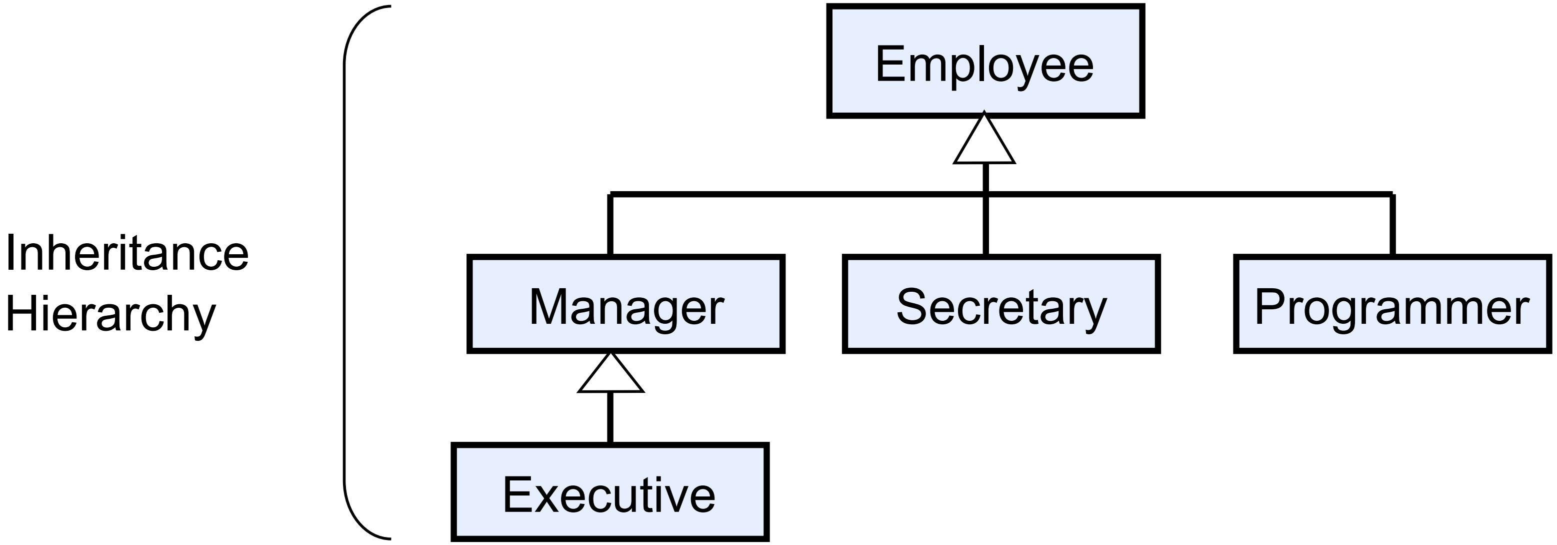
- The virtual machine knows about the ***actual type of an object***, hence invokes the correct method

- The ability of object variables like `e` to refer to multiple ***actual types*** is called **polymorphism**

- Automatically selecting appropriate method at *runtime* is called **dynamic binding**

Polymorphism

A collection of classes extending from a common superclass is called an *inheritance hierarchy*



The path from a particular class to its ancestors in the inheritance hierarchy is called its *inheritance chain*

Java does not support multiple inheritance!!

Polymorphism

How do we determine if inheritance is the correct design tool?

- The **“is-a”** rules states that every object of the subclass is an object of the superclass
- You can also apply the substitution principle which states:
- ***“You can use a subclass object whenever the program expects superclass object”***

Example*:*

```
Employee emp;

emp = new Employee(…); // Employee object expected

emp = new Manager(…); // OK, Manager can e used as well.
```

Polymorphism

Object variables are polymorphic

- A variable of type **Employee** can refer to objects of type **Employee** or to an object of any subclass of the **Employee** class i.e., **Manager**, **Executive**, **Secretary**, etc.

- It is illegal in Java to assign a superclass reference to a subclass variable

```
Manager m = staff[i]; // ERROR – not all employees are managers
```

Dynamic Binding: The Principle

When a method call is applied to an object, the compiler:

- Enumerates all class methods plus superclass public methods with the same name

- Performs **overloading resolution** by finding methods with matching **signatures**

Dynamic Binding: The Principle

- If the method is **private**, **static**, **final** or a constructor, the compiler knows exactly which method to call (**static binding**);

- otherwise **dynamic binding (late binding, run-time binding)** is used

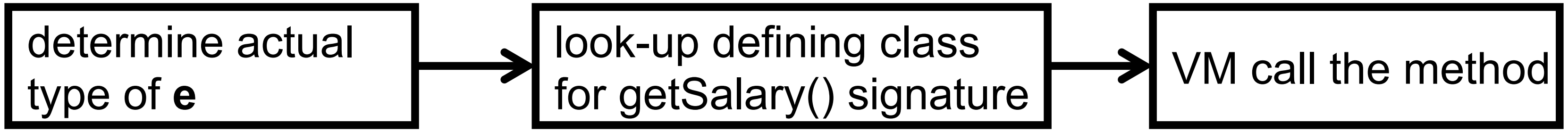
- When using **dynamic binding**, the VM must use the version of the method appropriate for the ***actual type*** of the referred to object

Example

Employee: **{**(getName(), Employee.getName());
(getSalary(), Employee.getSalary());
(getHireDay(), Employee.getHireDay()); (raiseSalary (double), Employee.raiseSalary(double)) **}**

Manager: **{**(getName(), Employee.getName());
(getSalary(), Manager.getSalary());
(getHireDay(), Employee.getHireDay()); (raiseSalary (double), Employee.raiseSalary(double)), (setBonus(double), Manager.setBonus(double)) **}**

- Employee class inherits from Object class; we've ignored these methods
- Method e.getSalary() is resolved at runtime as follows:



determine actual type of **e**

look-up defining class for getSalary() signature

VM call the method

Final Classes and Methods

Preventing Inheritance

- Use keyword final to prevent a class from being extended

```
final class Executive extends Manager {
                        …..
                                                                }
```

Preventing a method from being overridden

- Use final modifier to prevent a method from being overridden

```
class Employee {
           …..
           public final String getName() {
                      return name;
           }
           …
                                                                }
```

Final Classes and Methods

Note:

- A final field cannot be changed after the object is created
- If a class is declared final, only the methods, not the fields are automatically final

Why define final methods and class?

- Semantics preservation of class and methods
- Example getTime and setTime methods of Calender class are final
- The String class is a final class

Object Class

The ultimate ancestor – Every class in Java extends the Object class

```
Object obj = new Employee(„Harry Hacker“, 35000);
```

- Use a variable of type Object to refer to objects of any type

Services offered by the Object class

Method	Description
equals	Tests if one object references are identical
getClass	Returns the class of an object
hashCode	Returns an integer derived from the object
toString	Returns a String representation of an object
clone	Creates a clone of an object

Casting

Forcing conversion from one type to another is **casting**

```
double x = 3.405;
int nx = (int) x;
```

Converts value of expression **x** into an int, discarding the fractional part

Converting object reference from one class to another

- Cast in order to use an object in its full capacity after its actual type is temporarily forgotten

```
Manager boss = (Manager) staff[0];
```

Upcasting

Upcasting is always safe

- The superclass cannot have a bigger interface than the subclass

Every message sent through superclass interface is guaranteed to be accepted. Why?

Example

```
Manager boss = (Manager) staff[0];    // OK
```

Downcasting

Might promise too much in a downcast

The virtual machine checks every cast operation

- **ClassCastException** at run-time if cast operation is not type-safe

Example

```
Manager boss = (Manager) staff[1];    // ERROR. WHY?
```

```
if (staff[1] instanceof Manager)    // always check if cast will succeed
        boss = (Manager) staff[1];
```

Use `instanceof` to check if cast will succeed

Casting Notes

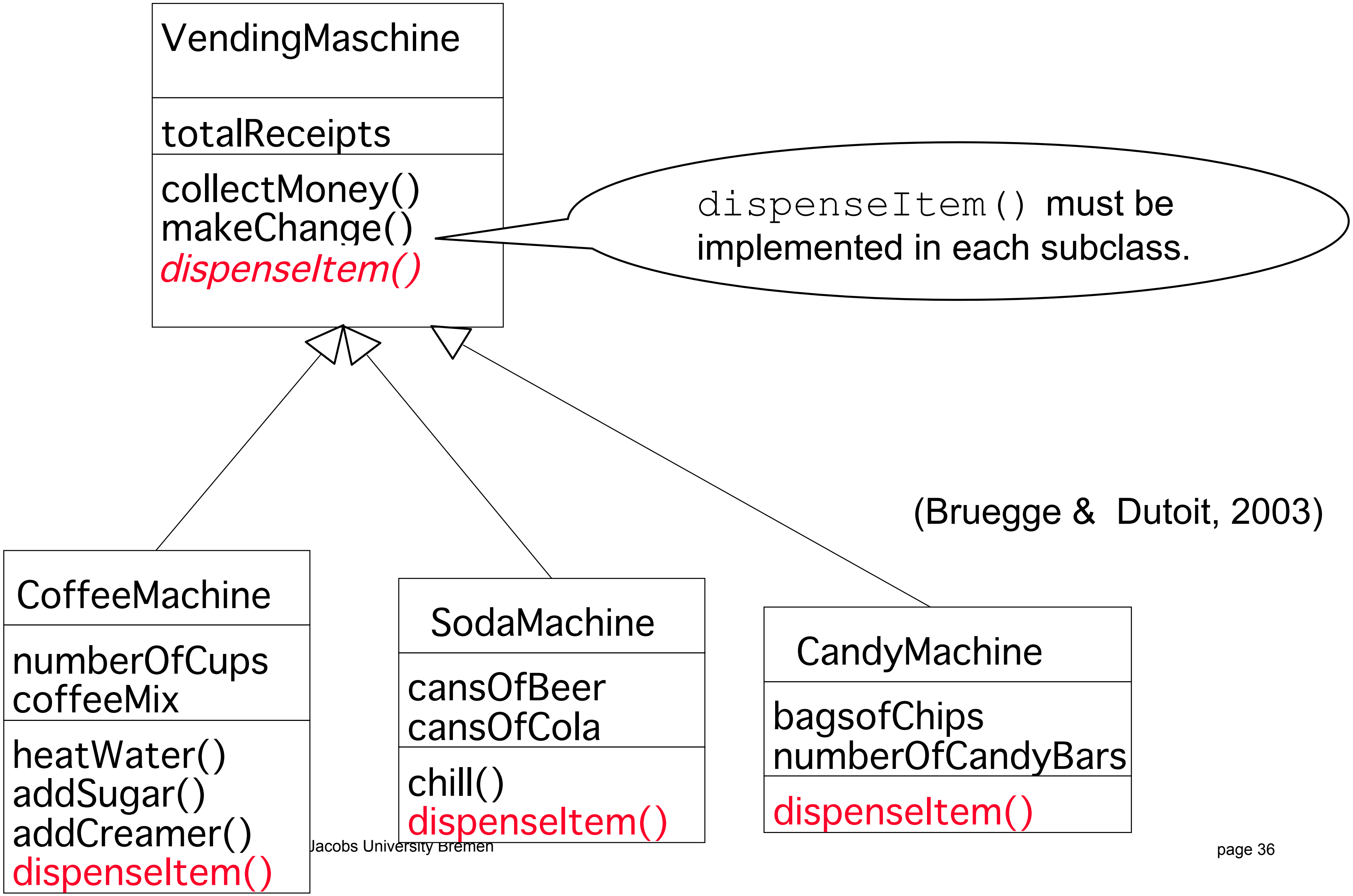
Compiler error if no chance that the cast will succeed

```
Date c = (Date) staff[1];    // Compile-time error  : Date not a subclass of Employee
```

Cast Notes

- Can cast only within inheritance hierarchy
- Use `instanceof` to check before casting from superclass to a subclass
- Its usually not a good idea to convert object types using casting
- Correct methods are automatically located using dynamic binding

Abstract Classes



VendingMaschine

totalReceipts

collectMoney()
makeChange()
dispenseItem()

`dispenseItem()` must be implemented in each subclass.

CoffeeMachine

numberOfCups
coffeeMix

heatWater()
addSugar()
addCreamer()
dispenseItem()

SodaMachine

cansOfBeer
cansOfCola

chill()
dispenseItem()

CandyMachine

bagsofChips
numberOfCandyBars

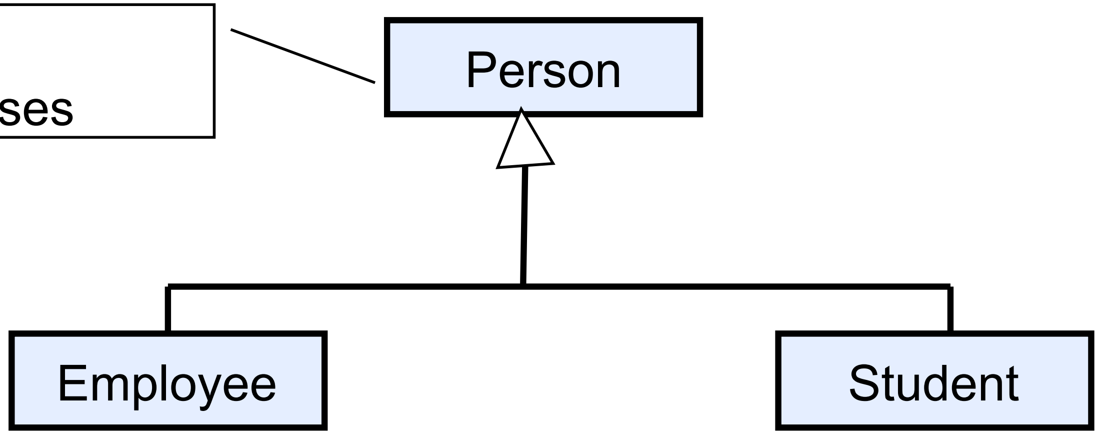
dispenseItem()

(Bruegge & Dutoit, 2003)

Abstract Classes

Classes become more general up the inheritance hierarchy

- General class
- Basis for other classes



Abstract class

```
abstract class Person {
                    .....
                                                  public abstract
String getDescription() ;
                                                    }
```

- An abstract method has no implementation inside the class of declaration
- A class with one or more abstract methods must itself be declared abstract
- abstract classes can have concrete data and methods

Abstract Classes

Abstract method:

- A method with a signature but without an implementation. Also called abstract operation

Abstract class:

- A class which contains *at least one abstract method* is called abstract class

Abstract Classes

Example

```
abstract class Person {
        private String name;

        public Person (String aName) {
                name = aName;
                                        }
        public abstract String getDescription();

        public String getName() { return name; }



}
```

- The *getDescription()* method has no implementation so it must be declared abstract
- The **Person** class must also be declared abstract since it contains an abstract method *getDescription()*

Abstract Classes

Abstract methods are placeholders for methods to be implemented in subclass

We make abstract classes concrete by extending them

If some or all abstract methods are not defined, then tag the subclass as abstract

Abstract Classes

Note: A class can be declared abstract though it has no abstract methods

Abstract classes cannot be instantiated

We can create object variables of abstract classes

Abstract Classes

Examples

```
new Person(“Vince Vu“); // Error
```

- You cannot create objects from an abstract class

Abstract Classes

Examples

```
class Student extends Person {
            public Student (String aName, String aMajor) {
                                        super(aName);
                                        major = aMajor;
                                                              }
            public String getDescription() {return „a student majoring in “+major;}
                        private String major;
                                                            }
```

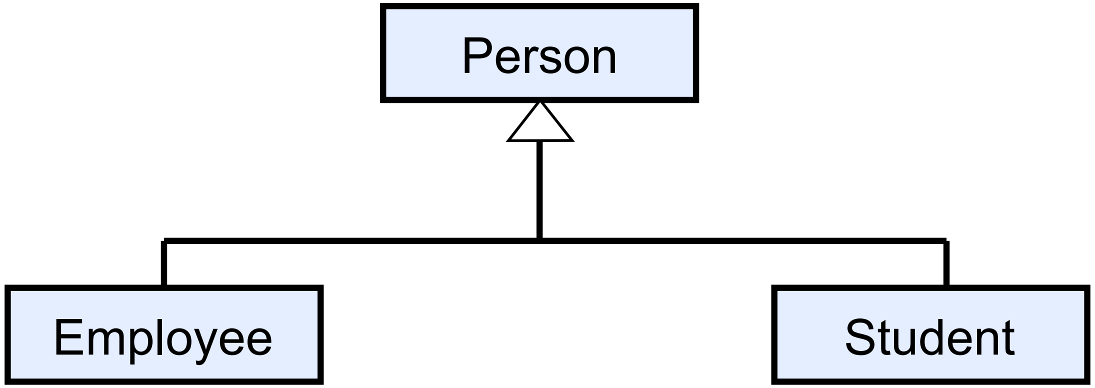
```
Person p = new Student(“Vince Vu“, „Economics“); // OK
```

- **p** is a variable of the abstract type **Person** that refers to an instance of a nonabstract subclass **Student**

- All methods in **Student** class are concrete, thus the class is no longer abstract

Abstract Classes

Example



```
Person[] person = new Person[2];
people[0]         = new Employee(…);
people[1]         = new Student(…);

// print names and descriptions of these objects

for (Person p : person )
    System.out.println(p.getName() + „, “ + p.getDescription());
```

- The variable *p* does not refer to a **Person** object
- *p* refers to an object of a concrete class such as **Employee** or **Student**
- For these objects, the method *getDescription* is defined

Design Hints

Hints

- Place common operations and fields in the superclass
- Don’t use protected fields if you can? WHY?
- *Use inheritance to model the “is-a” relationship*
- Don’t use inheritance unless all inherited methods make sense
- Don’t change expected behavior when you override a method
- Use polymorphism, not type information

Reading Assignment

Core Java 2 Volume I, Chapter 5. Inheritance by Horstmann and Cornell

Bruegge, B. & Dutoit, A. (2003) Object Oriented Software Engineering: Using UML, Patterns and Java. Prentice Hall .