Exercises „Software Development I“

10 Inheritance, Polymorphism
Dynamic Binding // Abstract Classes and Methods

January 9, 2013

Software Development I
Winter term 2012/2013

Institute for Pervasive Computing
Johannes Kepler University Linz
{farrahi,riener,schmittner}@pervasive.jku.at
Inheritance: Motivation

"Real world objects" often exist in various, similar variants
- Attributes (size, weight, color)
- Behavior
- Etc.

Objects are often grouped/arranged hierarchically
- Corporate structure (head → department chief → group leader → employee)
- Means of transportation
Inheritance: Motivation (2)

Example

```java
public class Car {
    Date regDate;
    int color;
    float luggageSpace;

    void drive () {...}
    void openTrunk () {...}
}
```

```java
public class Truck {
    Date regDate;
    int color;
    float loadingSpace;

    void drive () {...}
    void loading () {...}
}
```

Goals of inheritance

- Express commonalities
- Classification
- Specialization
- Re-use of code ("only" a positive side effect, not the main aim of inheritance!)
Inheritance: Terms and Definitions

Explicit inheritance

- Using the keyword "extends", a Java class can be extended
  \( \rightarrow \) subclass extend superclass, for example

```java
class Cabrio extends Car {  // Car extended from Vehicle
    int foldingTopTime;
    ...
}

Class TwoSeatCabrio extends Cabrio {
    boolean auxilaryRearSeats;
    ...
}
// Object \rightarrow \text{Transport} \rightarrow \text{Vehicle} \rightarrow \text{Car} \rightarrow \text{Cabrio} \rightarrow \text{TwoSeatCabrio}
```

Instantiation

```java
TwoSeatCabrio car1 = new TwoSeatCabrio();
car1.color = "red";  // Instance variable of "Vehicle"
car1.foldingTopTime = 20;  // Instance var. of "Cabrio"
car1.auxilaryRearSeats = false;  // Instance var. of "TwoSeatCabrio"
```
Polymorphism: Objects with Different Forms

Polymorphism refers (in general) to a principle in biology in which an organism or species can have many different forms or stages.

- This principle can also be applied to object-oriented programming (OOP) and the Java language.
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Demonstration of polymorphism with ComicCharacter

- Object of a subclass (SuperHero, Wizard) "is-a" object of the superclass (ComicCharacter)
- Objects of subclasses can be used instead of objects of the base class
- Object variables might be polymorph (having different "types")
  - Static data type (type of declaration)
  - Dynamic data type (corresponds to the actual class affiliation at run time)

→ Object of a subclass can be assigned to a variable of (declared, static) type of the parent class (multilevel!)
Polymorphism: Example ComicCharacter

```java
public static void main(String[] args) {
    ComicCharacter cc;
    cc = new SuperHero(); // A SuperHero "is-a" ComicCharacter
    cc.dance();           // Each ComicCharacter is able to dance
    // cc.fight();        // A ComicCharacter can't fight!
}
```

Attention! Polymorphism applies only to one direction

```java
public static void main(String[] args) {
    SuperHero sh;
    ComicCharacter cc;
    cc = new ComicCharacter();
    // sh = cc;            // Not every ComicCharacter "is-a" SuperHero!
    sh = new SuperHero();  // Other way round: ok (see above)
    cc = sh;
}
```

- Due to polymorphism, any object is assignable to a variable of type `java.lang.Object` (but not the other way round!)
- To check the dynamic data type of a variable the `instanceof` operator can be used
Polymorphism: Example ComicCharacter

```java
public static void main(String[] args) {
    ComicCharacter cc;
    cc = new SuperHero(); // A SuperHero "is-a" ComicCharacter

    System.out.println(cc instanceof ComicCharacter); // true
    System.out.println(cc instanceof SuperHero); // true
    System.out.println(cc instanceof Object); // true

    cc.dance(); // Each ComicCharacter is able to dance
    cc.fight(); // A ComicCharacter can't fight!
}
```

Exception in thread "main" java.lang.Error:
Unresolved compilation problem: The method fight() is undefined for the type ComicCharacter

Only attributes and methods of the superclass (ComicCharacter) are universally accessible!
Dynamic Binding: Introduction

Suppose you have 3 Java classes: A, B, and C where class B extends A and class C extends B (class inheritance hierarchy: A on top, then B in the middle, and finally C on the bottom); all 3 classes implement the instance method `void doIt();`

```java
public static void main(String[] args) {
    A x = new B();
    x.doIt();
}
```

What version of the `doIt()` method is actually executed and why?

- Given the class structure, the question is which version of the `doIt()` method will be executed – the one in class A, B, or C?
- Var. `x` is an object of type A, but it is instantiated as an object of class B
- The version of the `doIt()` method that’s executed by the object `x` is the one in class B because of what is known as dynamic binding in Java.
- Dynamic binding basically means that the method implementation that is actually called is determined at run time, and not at compile time.
- Dynamic binding (also known as late binding): the method that will be run is chosen at run time.
Dynamic Binding: Example ComicCharacter

Method that will be called is determined at run time (depending on dynamic type of the object)

```java
public static void main(String[] args) {
    ComicCharacter cc = new ComicCharacter();
    cc.setName("Papasmurf");
    cc.sing(); // sing() method of class ComicCharacter

    cc = new Smurf();
    cc.sing(); // sing() method of class Smurf; different behavior (method replaced in class Smurf)

    cc = new SuperHero();
    cc.sing(); // sing() method of class SuperHero; the method is not overridden in that class,
    // i.e. the sing method of the superclass ComicCharacter is invoked
}
```

Papasmurf sings.
Blue Smurfs don't sing!
null sings.
Abstract Methods/Classes: Declaration w/o Implementation

An **abstract method** is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
public abstract void draw(int x, int y);
```

An **abstract class** is a class that is declared abstract

- It may or may not include abstract methods
- Abstract classes cannot be instantiated, but they can be subclassed
  - when an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract (and implementation of the remaining methods must be provided in further subclasses)
- If a class includes abstract methods, the class itself must be declared abstract
- **Abstract class ≠ interface!**
  - unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods
  - such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation
  - if an abstract class contains only abstract method declarations, it should be declared as an interface instead…
Abstract Methods/Classes: Example GraphicalObjects

```java
abstract class GraphicalObject {
    abstract void draw();
}

class Circle extends GraphicalObject {
    public void draw() {
        ... // here follows the implementation!
    }
}

GraphicalObject go = new GraphicalObject(); // not allowed; abstract classes cannot be instantiated

GraphicalObject go = new Circle(); // ok; Circle "is-a" GraphicalObject

go.draw(); // calls the (concrete) draw method of Circle
```