Inheritance and Polymorphism

Uwe van Heesch
Richard van den Ham
Pieter van den Hombergh
Thijs Dorssers

Fontys Hogeschool voor Techniek en Logistiek

February 21, 2015

Topics

Java Inheritance
Class hierarchy - An example

Visibility

I can see what you can't
Constructors and inheritance
Polymorphism

Known

- Between objects, relationships exist

Known

- Between objects, relationships exist
- An association represents a *Has*-relationship
  - A human being has two legs
  - A chair has 4 chair legs
  - A car has 4 wheels

Known

- Between objects, relationships exist
- An association represents a *Has*-relationship
  - A human being has two legs
  - A chair has 4 chair legs
  - A car has 4 wheels
- Beside, a *Is-a*-relationship exists
  - Apples and pears are fruit species
  - Students and lecturers are humans
  - The type defines characteristics for its elements
    - Each human being has an eye color
    - Each human being has a hair color
    - Each human being has a size
  - Java represents this relationship by inheritance
  - *Parents* give their *Childs* characteristics

Java Inheritance

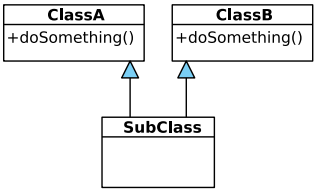
- Java defines classes in hierarchical relationships
- Therefore, classes have a *Is-a*-relationship with their parent class
- The keyword `extends` shows that a class is derived from another class
- Therefore, a class becomes a sub class
- The parent class is the super class
- The sub class inherits all **visible** characteristics of the super class

The implicit base class `Object`

- Classes without the `extends`-keyword automatically inherit from `Object`
- Object is an implicit super class
- Each class inherits either directly or indirectly from `java.lang.Object`
- All visible attributes and methods, like toString(), are inherited

No Multiple inheritance in Java

- Java only allows single inheritance
- Behind the `extends`-keyword, only *one* super class is allowed
- ao C++, Python and Perl allow Multiple Inheritance. Why does java not?



- `new SubClass().doSomething();` **Do what???**

Example *Person*

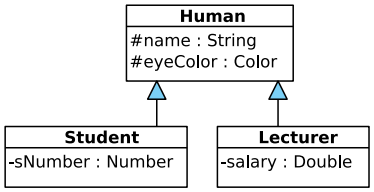


Figure : A class hierarchy for our school

- Students and Lecturers are Humans
- with ao a name and an eye color
- `TeamLeader` could have been a sub class of Lecturer

Example *Human*

```
package nl.fontys.pro2.week1;

import java.awt.Color;

public class Human {
    protected String name;
    protected Color eyeColor;
}
```

```
package nl.fontys.pro2.week1;

public class Student extends Human {
    private Number number;
}
```


Example *Human*

```
public static void main(String[] args) {
    Student theStud = new Student();
    theStud.name = "Richard Stallman";
}
```

- Class Student can use the inherited attributes `name` and `eyeColor`
- Changes in a super class are automatically represented in sub classes
- Therefore, a sub class has a very strong coupling to its super class
- Super classes *don't know* their sub classes

peekabo:I can see what you can't

Visibility

- Sub classes inherit all **visible** characteristics from their super class
- `public`: visible for all classes
- package-private: visible for all classes in the same package
- `private`: only visible within class (including contained inner class)
- In addition: `protected`:
  - protected attributes and methods are inherited by all sub classes
  - are visible for all classes in the same package
- `public` ➜ `protected` ➜ *package-private* ➜ `private`
- `Protected` -protection is weaker than package-private!

Visibility table

Table : Normalized visibility table.

Modifier	Class	Package	Subclass	World
public	y	y	y	y
protected	y	y	y	-
(default)	y	y	-	-
private	y	-	-	-

Usage of constructors with inheritance

- In contrast to methods, constructors are not inherited
- Sub classes need a constructor to enable objects to be created
- Within the constructor, Java automatically invokes the super class constructor
- When no constructor has explicitly been defined, the implicit default constructor is used

A super constructor

- `super()` invokes the super constructor explicitly
- The default constructor is always invoked implicitly
- `super` can invoke parameterized constructors
- Mandatory, if super class does not have default constructor

```
public class Student extends Human {
    public Student(Number number){ ... }
}
```

```
public class StudAssistant extends Student {
    public StudAssistant(Number sNumber){
        super(sNumber);
    }
}
```


Summary constructors and methods

- Constructors
  - can't be `abstract`, `final`, `native`, `static` or `synchronized`
  - don't have a return value, even not void
  - are not inherited
  - Can have all visibility attributes
  - `this(...)` refers to another constructor in the same class
  - `super(...)` invokes a constructor of super class
- Methods
  - can be public, protected, package-private, private, abstract, final, native, static or synchronized
  - can have return values, or void
  - Visible methods are inherited
  - Inherited visibility cannot be changed
  - `this` is a reference to the current instance of the class
  - With `super`, overridden methods of the superclass can be invoked

Static and dynamic types

- Seemingly trivial relationships
  - A Student is a Human
  - A Lecturer is a Human
  - A Human is an Object
  - A Student is a Student
- In Java:

```
Human   studentIsHuman   = new Student();
Human   lecturerIsHuman  = new Lecturer();
Object  humanIsObject    = new Human();
Student studentIsStudent = new Student();
```

Static and Dynamic types

- The declared type is the so-called **static type**
- The initialised type is the **dynamic type**
- The declared type or L-Value is configured with their static type. An object knows its dynamic type.
- Variables are treated as being of a static type

```
Human studHum = new Student();
System.out.println(studHum.getName());
// The following will not work:
System.out.println(studHum.getSNumber());
```


Static and dynamic types

- Variables can not simply be initialised with *lesser*[1] types.
- In case of the correct dynamic type, a typecast helps
- `instanceof` verifies dynamic types

```
Human stud = new Student();
// next line will not work:
Student stud2 = stud;
// This one will:
Student stud3 = (Student) stud;
// Because stud has the dynamic type  Student
if(stud instanceof Student){
    // instanceof tests the dynamic Typ
}
```

[1] less specific

Override a method

- Methods of Super classes are overwritten when
  - The name **and**
  - the list of parameters (number of params and their type, not their name)
  - **and** return type
  - **exactly** match that of the super classes method
- The annotation `@Override` lets the compiler also check whether a method is actually and correctly overwritten
- The method of the superclass is then ruled out, right?
- Example: override the `toString()` method of `Object`

Override a method

- makes calling overridden methods possible
- You can only reach one level higher with super: Chaining of `super` keywording at a deeper inheritance hierarchy is not possible
- `Super` methods are not accessible from the outside. A call to a `super....` method can only be made on `this`, inside the defining class, not any other object.

```
public class Student extends Human {
...
    @Override
    public void setName(String name) {
        if(isNiceName(name)){
            super.setName(name);
        }
    }
}
```


prohibit method override

- The keyword `final` prohibits overriding.
  - In classes, thereby prohibiting extension
    - example `final class Integer`
  - for methods, thereby prohibiting that they can be overridden
    - example: `public final wait()`[2]
- Quiz: could you come up with a use for a `protected final` method?

[2] Defined in this way in `java.lang.Object`

Not all understood? For next time read:

- BlueJ: Chapter 8.2 - 8.11

Questions?

Questions or remarks?

Inheritance and Polymorphism

Uwe van Heesch
Richard van den Ham
Pieter van den Hombergh
Thijs Dorssers

Fontys Hogeschool voor Techniek en Logistiek

February 21, 2015

Topics

Polymorphism
Abstract classes and methods
Interfaces

Java8 default methods
multiple inheritance of implementation
Example: Parents
Example: Child interfaces

Polymorphism

Polymorphism is the ability to provide a single interface to entities of different types

- Visible methods of super classes also exist in sub classes
- Super classes can predefine implementations, which can be overwritten, if required
- We can however be sure the methods exist

Polymorphism - Example

```
public class Human extends Object {
...
  @Override
  public String toString(){
    return "[Human:...";
  }
}
```

```
public class Student extends Human {
...
  @Override
  public String toString(){
    return "[Student:...";
  }
}
```


Polymorphism - Example

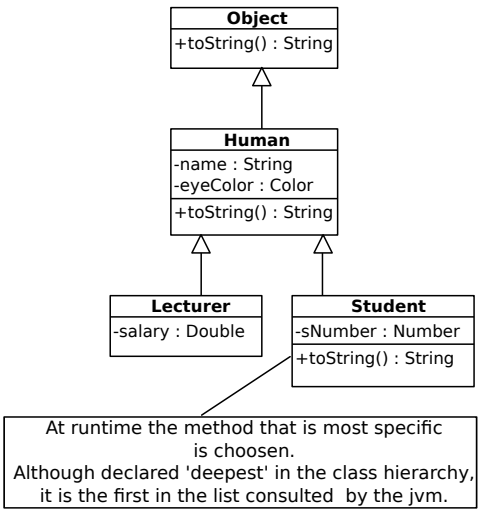
```
Student student = new Student();
System.out.println(student);

Human human = new Student();
System.out.println(human);

Object object = new Student();
System.out.println(object);
```

What will the output look like?

Polymorphism - Example

- During runtime, the method `toString()` from `Student` will always be used
- In contrast to the compiler, the runtime environment knows the dynamic type
- This is called **dynamic binding**, because the actual type of an object is only determined at runtime
- The method that is the deepest one in the class hierarchy is chosen

Polymorphism - Example



Exceptions on polymorphism

- Not all methods are dynamically bound
  - Only **overridden** methods participate
  - Methods which can't be overridden will bound statically
  - `private`, `static` and `final` methods fall in this category.

Abstract classes

- Abstract classes can't be instantiated
- An abstract class can be used as a static (or declaration) type.
- Useful, for example for classes which are only used as super class, to provide signatures for sub classes
- The keyword `abstract` identifies abstract classes
- Abstract classes are the opposite of concrete classes

```
public abstract class Human{
...
}
```

Abstract classes

- Abstract classes are often used in inheritance
- They behave like concrete classes
- A subclass can extend an abstract class and can be abstract itself again

```
Human stud1 = new Student();
Human[] humans = new Human[] {new Student(), new ↩
    Lecturer() };
```

Where could abstract classes be used for as well?

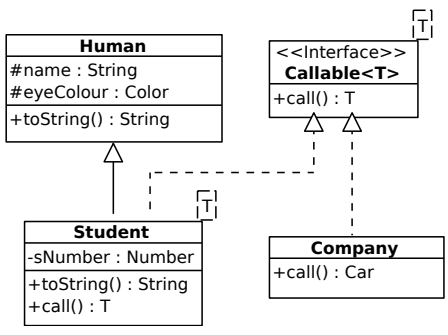
Abstract methods

- Methods in abstract classes *could* also be abstract
- They only define a method signature for the sub class
- *Concrete* sub classess have to implement these

```
public abstract class Human extends Object {
...
    protected abstract void dress();
}
```


Interfaces

- It's difficult to give classes multiple types by using inheritance
- Inheritance is always done in order, Human inherits from Object, Students inherits from Human, etc.
- Sometimes, classes need to have types from different hierarchies



Interfaces

- Instead of `class`, `interface` is used
- Methods in interfaces don't have a body
- All methods are automatically `abstract` and `public`
- Constructors are useless and therefore not allowed
- Instance variables are not allowed either, `static`-variables however are allowed (automatically `final`)

```
public interface Callable<T> {

    T call();

}
```


Interfaces

```
public class Student extends Human
                implements Callable{
    //...
    public void call() {...}
}
```

```
public class Student extends Human
    implements Callable, Annoyable{
    //...

    public void call() {...}

    public StressLevel annoy(double degree) {...}
}
```


Interfaces

```
public class Student extends Human
                implements Callable{
    //...
    public void call() {...}
}
```

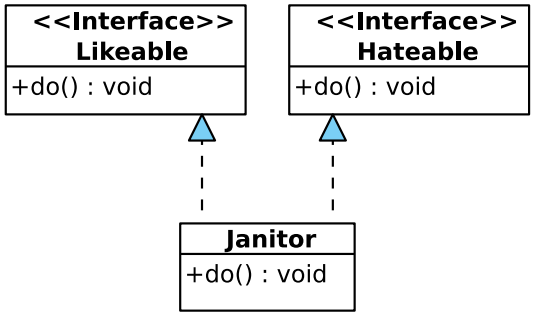
```
public class Student extends Human
    implements Callable, Annoyable{
    //...

    public void call() {...}

    public StressLevel annoy(double degree) {...}
}
```

Interfaces

- Classes can implement multiple interfaces
- Interfaces can extend zero, one or **multiple** other interfaces
- They allow objects to act in different roles
- A powerful object can be reduced to a single method
- Usage analogous to usage of abstract classes

```
Callable callable = new Student();
Human human = (Student) callable;
```

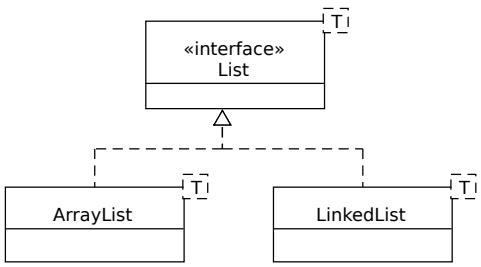

Interfaces



What will happen now?

Interfaces as specification

- Separation of functionality and implementation
- Clients communicate implementation-independent
- Alternative implementations can be used



Interfaces vs. Abstract classes

```
public interface Callable<T> {

    T call();

}
```

```
public abstract class Callable<T> {

    abstract T call();

}
```


Java8 default methods

To be able to extend of the framework without breaking existing interfaces and implementing classes, Java 8 introduces a few new concepts.

- `static` methods (implementations) in `interface`s.
- `default` method (implementations!) in `interface`s.
- It allows the extension of interfaces without breaking existing implementing classes.
- The static methods are typically helpers for the default methods.
- It also implies multiple inheritance for said default methods.
- If a conflict is possible, the implementation programmer must help the compiler by specifying which default variant method is to be taken.
- The first use case of this extension is the stream framework, introduced, in combination with λ expression.

Multiple inheritance example, parents

```
interface AnInterface {

    default String getName() {
        return "An";
    }
}

interface BnInterface {

    default String getName() {
        return "Bn";
    }
}
```

Multiple inheritance example, child

```
interface CnInterface extends AnInterface,
                              BnInterface {

    // resolve which super to use.
    // also works in implementing classes
    @Override
    default String getName() {
        return AnInterface.super.getName();
    }

}
```

Note that an class implementing two interfaces-methods with the same signature must also specify which variant to select, using the same construction as above.
See **DEMO**

Any open issues?

Not all understood? For next time read:

- BlueJ: Chapters 9 and 10
- Java 8 tutorial on default methods
  `http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`

Questions?

Questions or remarks?