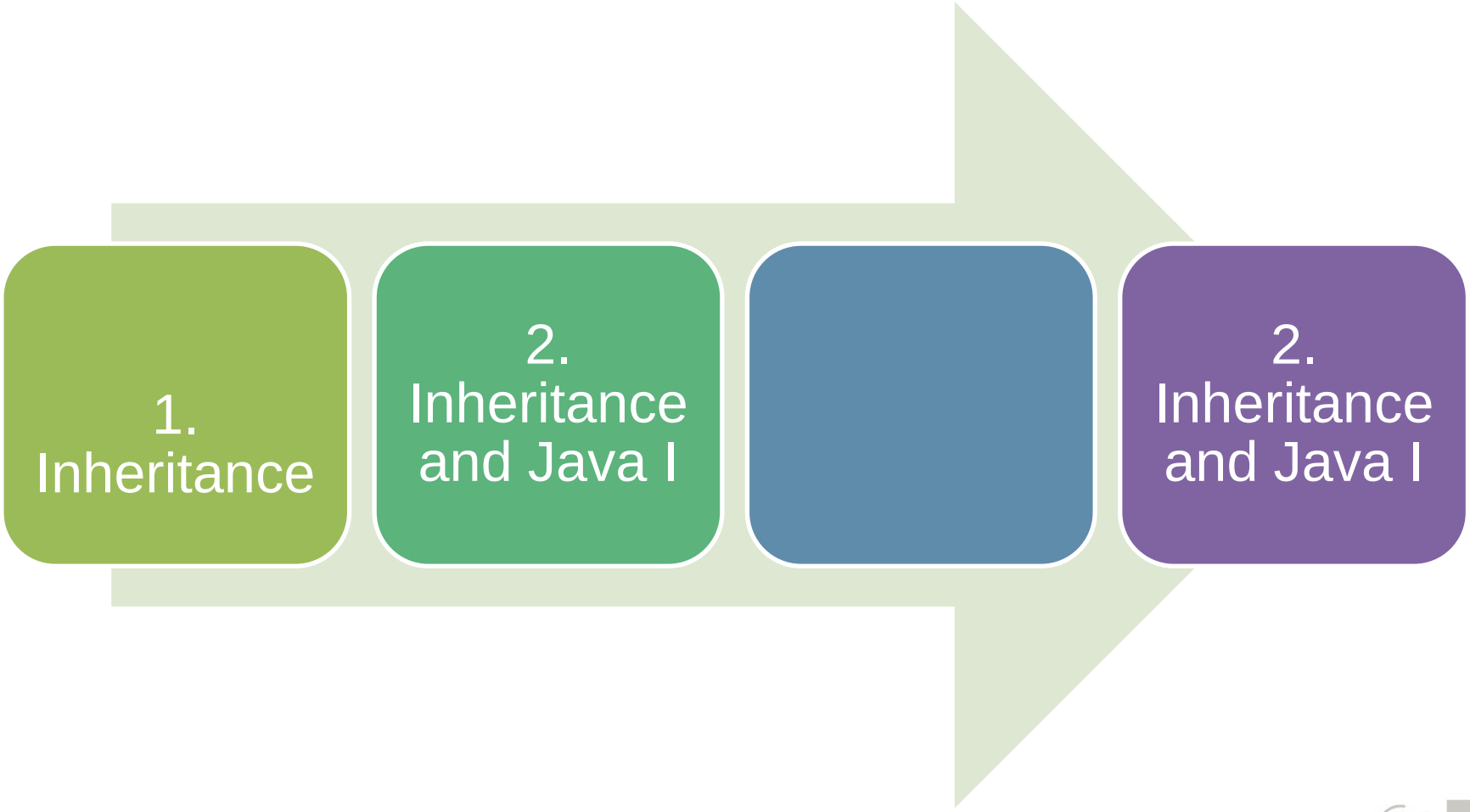
Inheritance

Christian Rodríguez Bustos
Edited by Juan Mendivelso

Object Oriented Programming

Agenda


1.
Inheritance
2.
Inheritance
and Java I
2.
Inheritance
and Java I

1. Inheritance

1.1 Hierarchy of Classes

1.2 UML Notation

1.1 Hierarchy of Classes

Actual Situation

```java
public class Student {

    private long id;
    private String user;

    @Override
    public String toString() {
        return "Student{" + "id="
                + this.getId() + "user="
                + this.getUser() + '}';
    }

    // ...
```

Typical Student has an id and a user

We @override **toString** method to print the Student data

New Requirements Arrives

The Academic Information System has to manage the information of **graduate students**:

- Undergraduate program
- Current place of employ

Your boss

Solution 1

Modify the Student Class

Solution 1 - Modify the Student Class

We can **add new parameters, setters and getters and modify the toString method** to print depending of type of student

Solution 1 - Modify the Student Class

```java
public class Student {

    private long id;
    private String user;
    private boolean graduateStudent;
    private String currentEmployePlace;
    private String undergraduateProgram;

    @Override
    public String toString() {
        if (this.isGraduateStudent()) {
            return "Student{" + "currentEmployePlace="
                    + this.getCurrentEmployePlace() + "undergraduateProgram="
                    + this.getUndergraduateProgram() + '}';
        } else {
            return "Student{" + "id="
                    + this.getId() + "user="
                    + this.getUser() + '}';
        }

    }

    // ...
```

New Requirements Arrives

The system must handle the graduate program being taken by graduate students:

- Current graduate program

Your boss

Solution 1 - Modify the Student Class again ???

You have to **add one more attribute, two methods and a extra validation** in the toString method!

Solution 1 is a bad solution

Your student class is not well delimited, at this moment: **your objects students can be understood as graduate and undergraduate**.

How can we distinguish between them?

Solution 2

Create GraduateStudent class

Solution 2 – Create GraduateStudent class

```java
public class GraduateStudent {

    private long id;
    private String user;
    private String currentEmployePlace;
    private String undergraduateProgram;

    @Override
    public String toString() {
        return "Student{" + "currentEmployePlace="
                + this.getCurrentEmployePlace() + "undergraduateProgram="
                + this.getUndergraduateProgram() + '}';

    }

    // ...
```

This makes sense, it could work!!!

Solution 2 – Create GraduateStudent class

Wait!!! , this code smells like a **cloned code!**

```java
public class Student {

    private long id;
    private String user;

    @Override
    public String toString() {
        return "Student{" + "id="
                + this.getId() + "user="
                + this.getUser() + '}';

    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }
```

```java
public class GraduateStudent {

    private long id;
    private String user;
    private String currentEmployePlace;
    private String undergraduateProgram;

    @Override
    public String toString() {
        return "Student{" + "currentEmployePlace="
                + this.getCurrentEmployePlace() + "undergraduateProgram="
                + this.getUndergraduateProgram() + '}';

    }

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }
    // ...
```

Clones!!!

Try to avoid cloning

Clones are hard to maintain and reflect poor designs.

Solution 3

Taking Advantage of Inheritance

Solution 3 - Taking Advantage of Inheritance

```java
public class GraduateStudent extends Student {

    private String currentEmployePlace;
    private String undergraduateProgram;

    @Override
    public String toString() {
        return "Student{" + "currentEmployePlace="
                + this.getCurrentEmployePlace() + "undergraduateProgram="
                + this.getUndergraduateProgram() + '}';

    }

    // ...
```

Graduate Student **inherits all the accessible methods and attributes** from Student class

Inheritance terms

Graduate Student

- ☐ is a **specialization** of Student
- ☐ is a **subclass** of Student

Student

- ☐ is a **generalization** of a Graduate Student
- ☐ is the **superclass** of Student

Inheritance terms

```
public class GraduateStudent extends Student
```

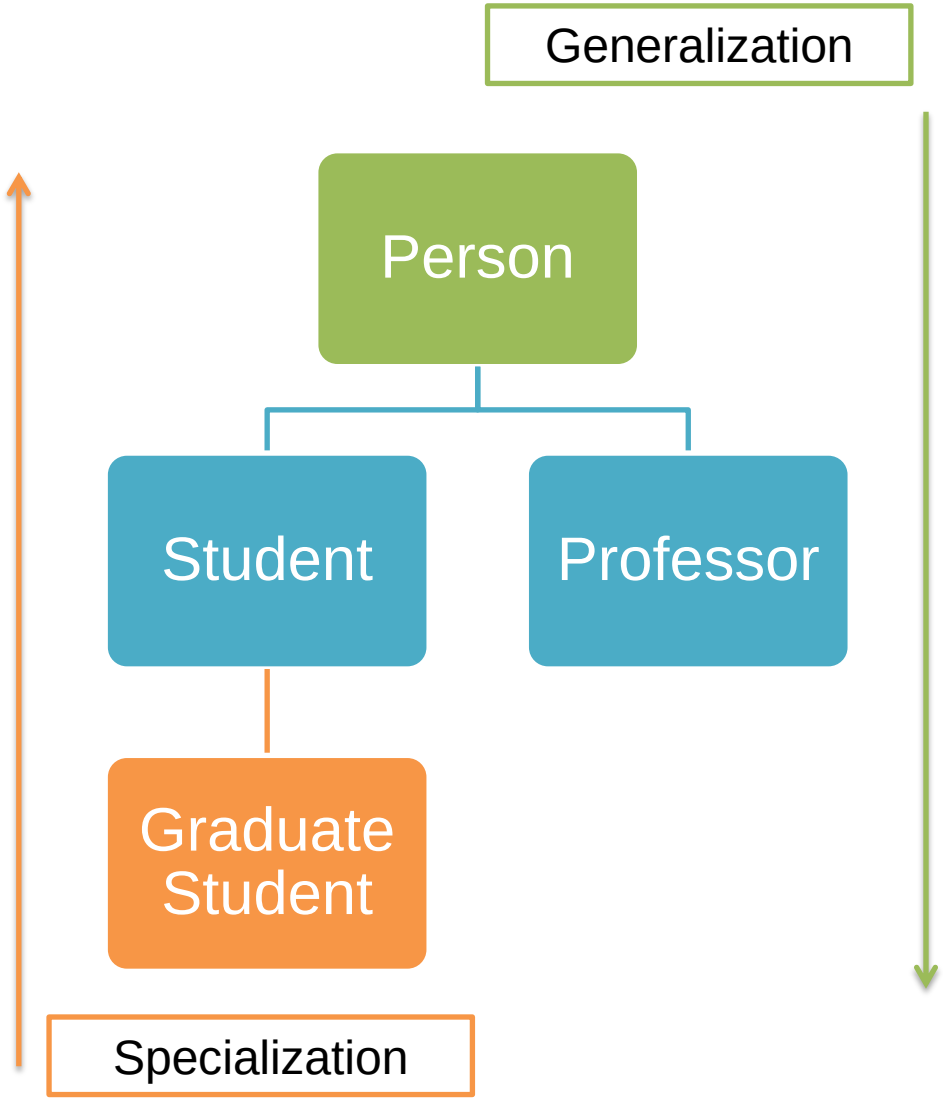
A _____ *Is a* _____

subclass superclass

A **Graduate Student** *is a* **Student**

A **Graduate Student** *is a specialization of a* **Student**

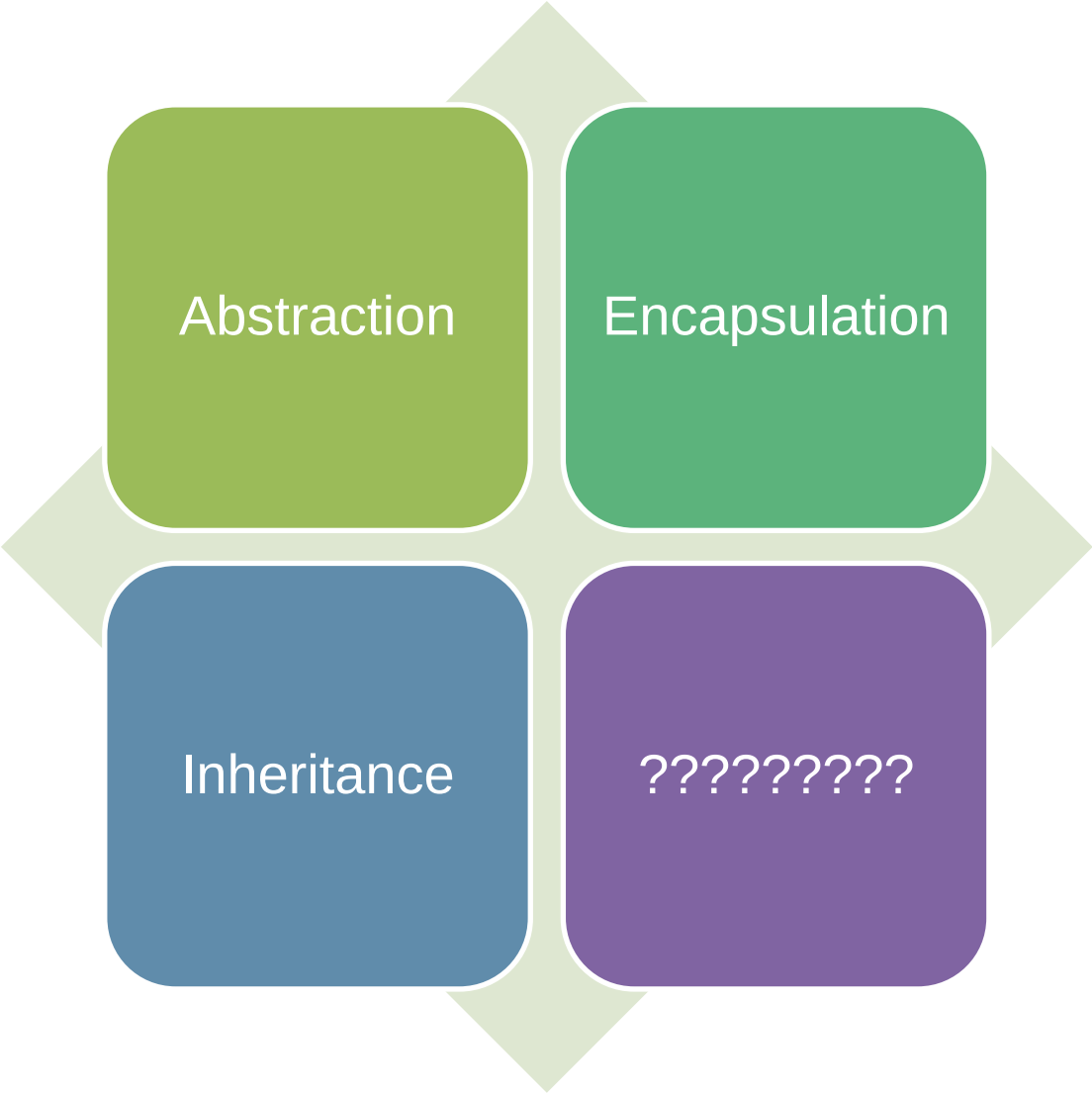
Class Hierarchies



We manage knowledge in terms of inheritance hierarchies

In a POO language we can abstract the real world relation into **class hierarchies**

Inheritance is one of the four principles of OOP



Inheritance benefits

Reduction of code redundancy

- Maintenance
- Avoid “Ripple Effects”

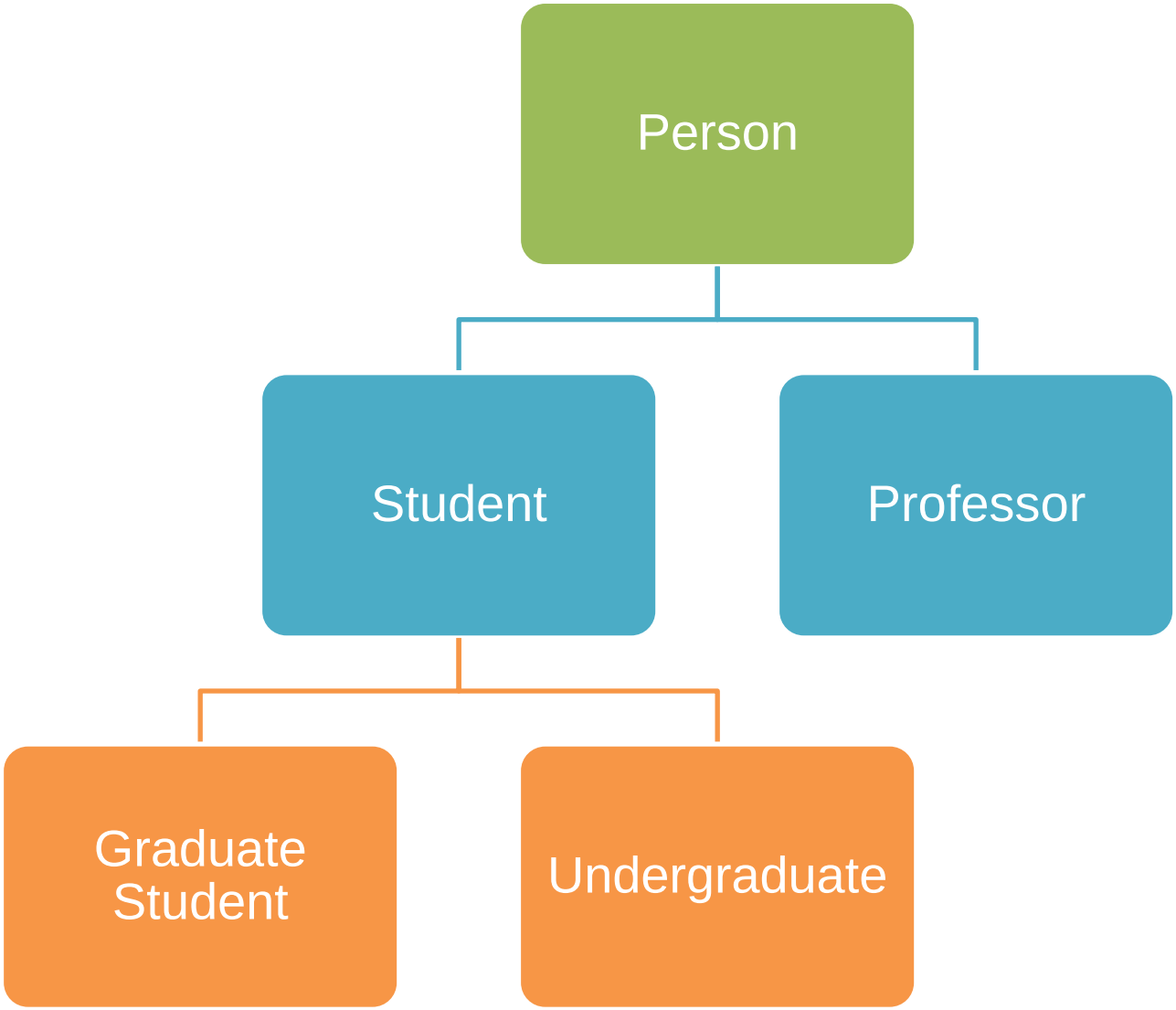
Subclasses are more **concise**

We can **reuse and extend** code that has already been tested

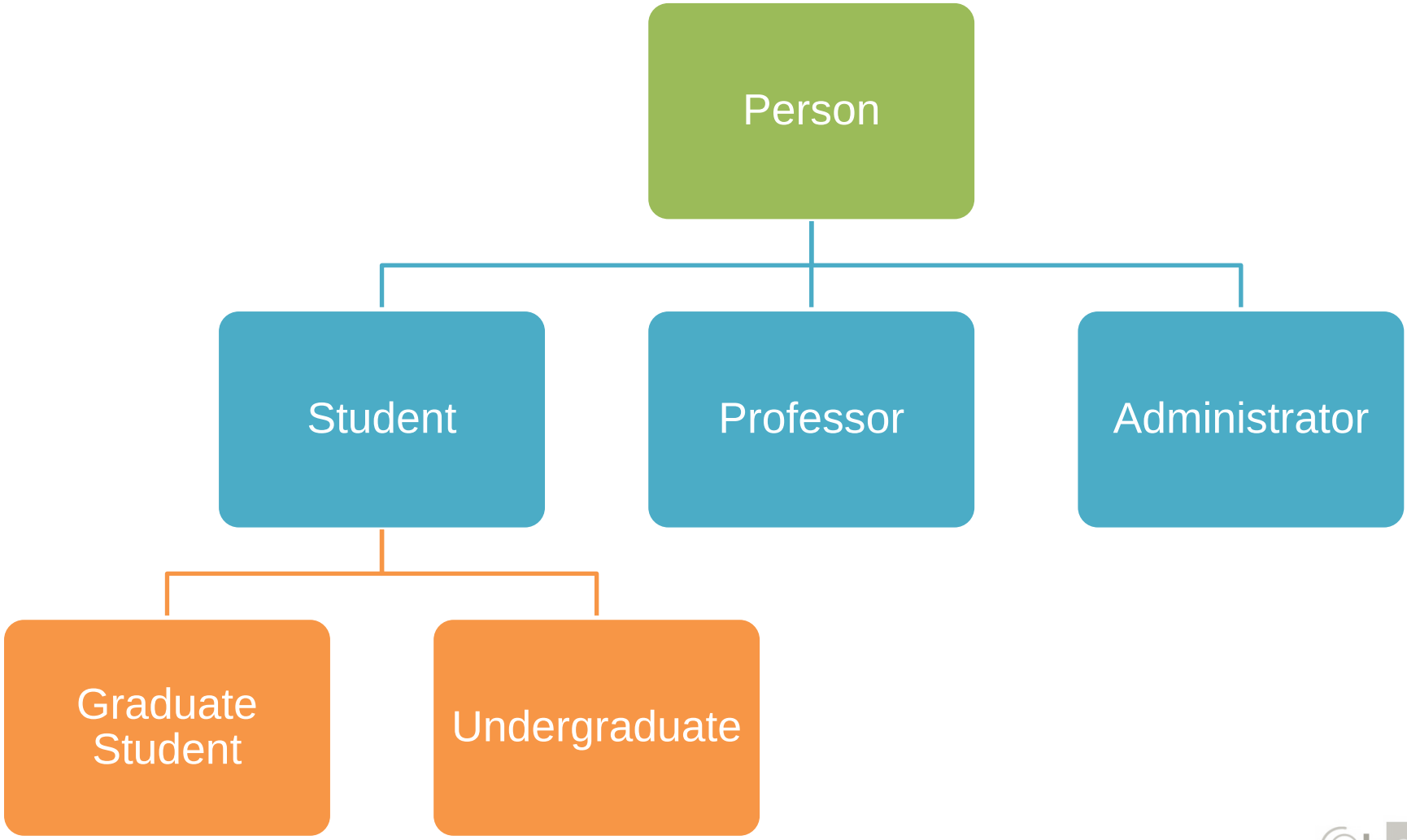
We can **derive** a new class from an existing class

Inheritance **is a natural way to manage knowledge**

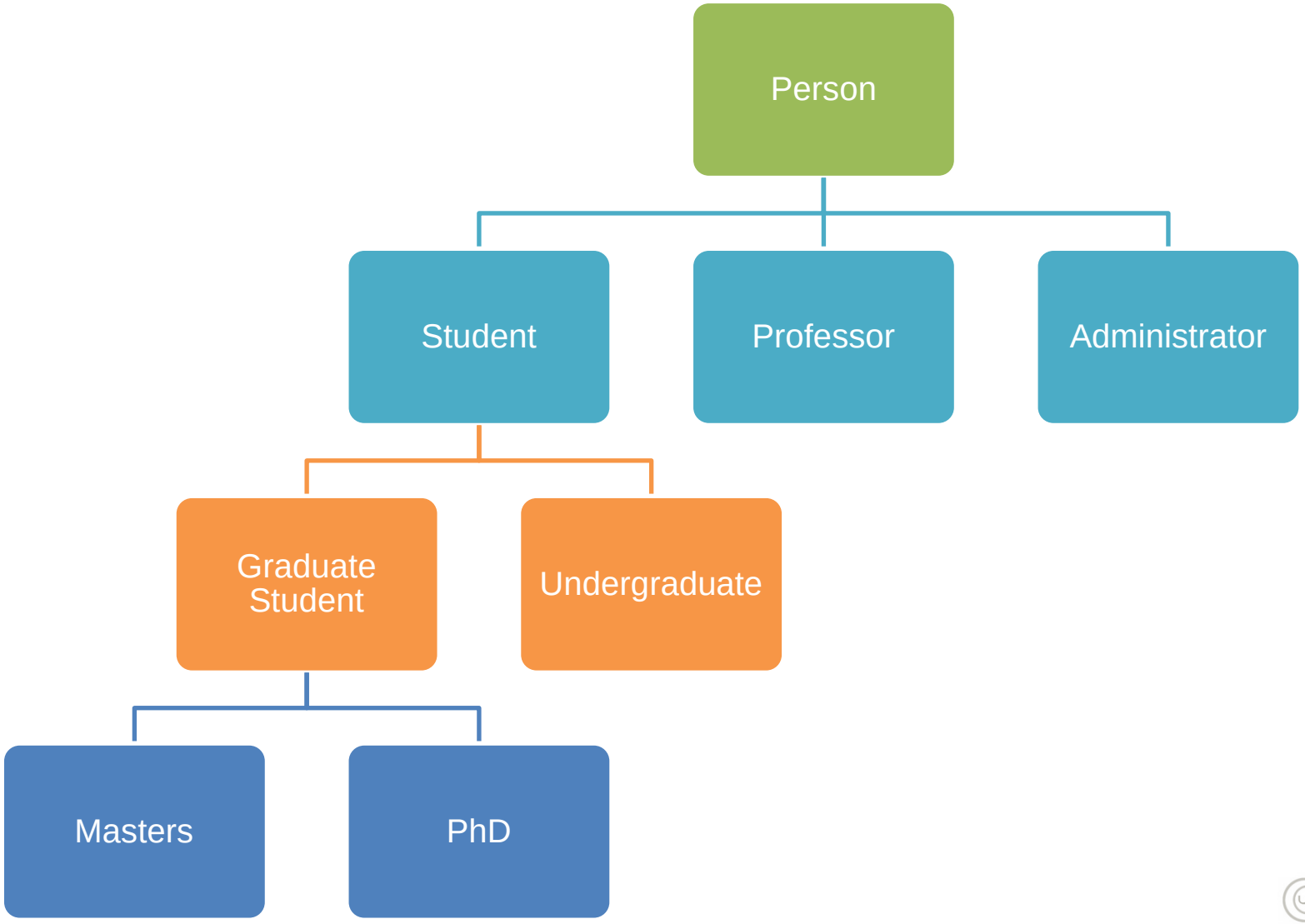
Class Hierarchies inevitably expand over time



Class Hierarchies inevitably expand over time

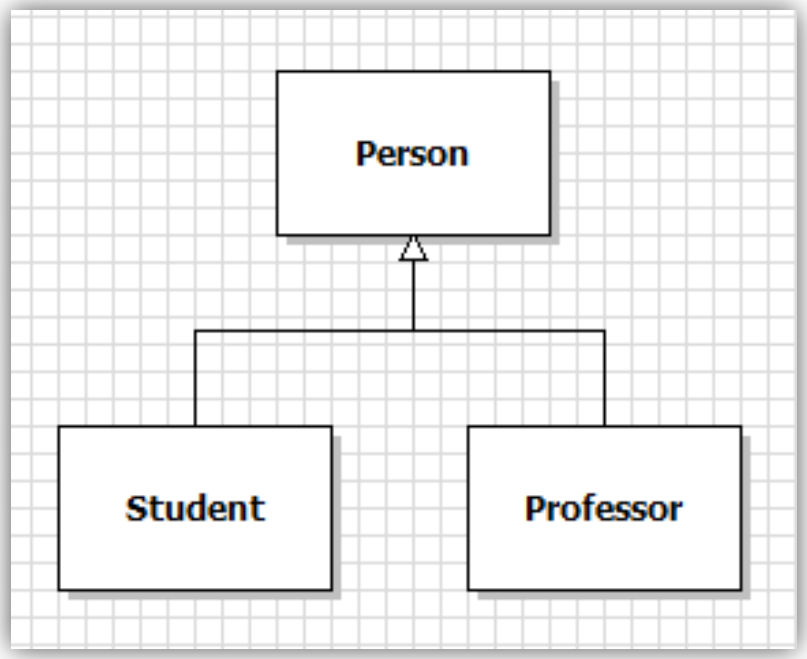


Class Hierarchies inevitably expand over time



1.2 UML Notation

UML notation



- A Student is a Person
- A Professor is a Person

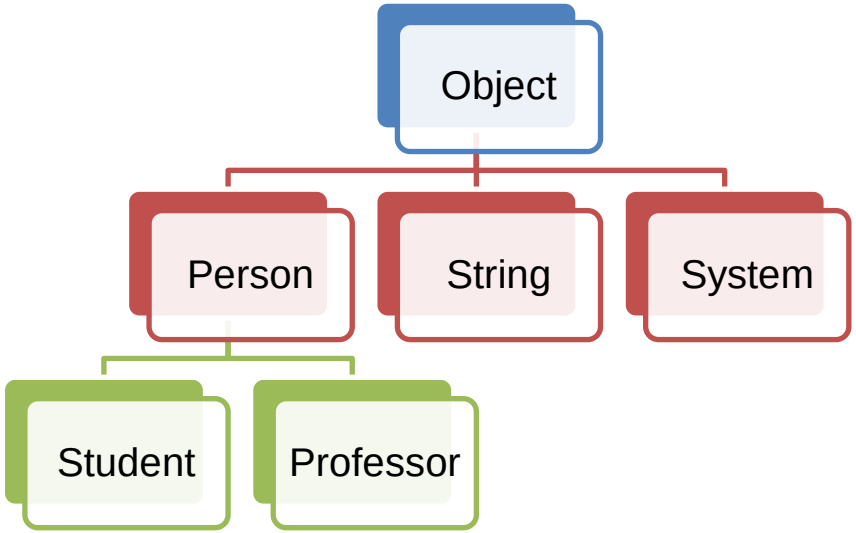
4. Java Inheritance

All classes are subclasses of the Object class

In Java **Class Object** is the root of the class hierarchy.

Every class has Object as a superclass.

All objects, including arrays, inherit the methods of this class.



All classes are subclasses of the Object class

```
public class Student {
```

Is equivalent
to

```
public class Student extends Object{
```

Constructor Summary

	Object()

Method Summary

protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collec
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

```
board.
    equals(Object obj)            boolean
    getBoard()                    Square[][]
    getClass()                    Class<?>
    hashCode()                    int
    notify()                      void
    notifyAll()                   void
    toString()                    String
    updateSquare(Square square)   void
    wait()                        void
    wait(long timeout)            void
    wait(long timeout, int nanos) void
```

All those methods are inherited by all classes

4. Accessibility

4.1 Java Access Modifiers
4.2 Inheritance and Accessibility

4.1 Java Access Modifiers

1.2 Inheritance and Accessibility

Java Access Modifiers

Modifier	Access Levels			
	Class	Package	Subclass	World
`public`	☺	☺	☺	☺
`protected`	☺	☺	☺	☹
Default (no modifier)	☺	☺	☹	☹
`private`	☺	☹	☹	☹

Access level modifiers determine **whether other classes can use a particular field or invoke a particular method**

4.2 Inheritance and Accessibility

Inheritance and Access Modifiers

```
public class Person {

    private long id;
    private String user;
    private String firstName;
    private String lastName;
    private Date birthDate;

    // ...
```

Encapsulation define that attributes are defined as **private**.

And private attributes cannot be inherited

So….

Inheritance and Access Modifiers

```
public class Student extends Person {
```

```
firstName has private access in sia.Person
--
(Alt-Enter shows hints)
```

```
Student student = new Student();
student.firstName = "Clark";
```

How can be accessed superclass attributes from a subclass?

Inheritance and Access Modifiers

We can access to private attributes through **public superclass methods**

```
Student student = new Student();
student.setFirstName("Clark");
```

Inheritance and Access Modifiers

Public or protected Person Methods are inherited by the Student subclass

```
public class Student extends Person {
```

```
equals(Object obj)                                   boolean
getAttends()                                         List<Group>
getBirthDate()                                       Date
getClass()                                           Class<?>
getFirstName()                                       String
getGradesReceived()                                  List<Grade>
getId()                                              long
getLastName()                                        String
getUser()                                            String
hashCode()                                           int
notify()                                             void
notifyAll()                                          void
setAttends(List<Group> attends)                      void
setBirthDate(Date birthDate)                         void
setFirstName(String firstName)                       void
setGradesReceived(List<Grade> gradesReceived)        void
setId(long id)                                       void
setLastName(String lastName)                         void
setUser(String user)                                 void
toString()                                           String
wait()                                               void
wait(long timeout)                                   void
wait(long timeout, int nanos)                        void
```

Inherited Person methods

Inheritance and Access Modifiers

Public or protected Person Methods are inherited by the Student subclass

```
public class Student extends Person {
```

equals(Object obj)	boolean
getAttends()	List<Group>
getBirthDate()	Date
getClass()	Class<?>
getFirstName()	String
getGradesReceived()	List<Grade>
getId()	long
getLastName()	String
getUser()	String
hashCode()	int
notify()	void
notifyAll()	void
setAttends(List<Group> attends)	void
setBirthDate(Date birthDate)	void
setFirstName(String firstName)	void
setGradesReceived(List<Grade> gradesReceived)	void
setId(long id)	void
setLastName(String lastName)	void
setUser(String user)	void
toString()	String
wait()	void
wait(long timeout)	void
wait(long timeout, int nanos)	void

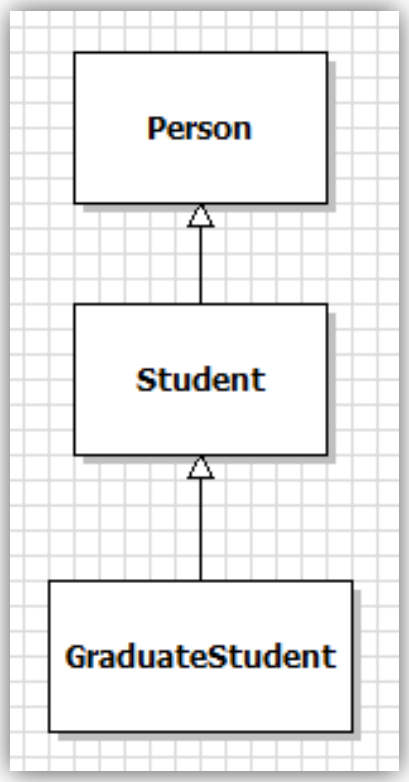
Student methods

Inheritance and Access Modifiers

```
equals(Object obj)                                  boolean
getAttends()                                   List<Group>
getBirthDate()                                         Date
getClass()                                        Class<?>
getCurrentWorkPlace()                               String
getFirstName()                                      String
getGradesReceived()                            List<Grade>
getId()                                                long
getLastName()                                       String
getUndergraduateCarreer()                           String
getUser()                                           String
hashCode()                                             int
notify()                                              void
notifyAll()                                           void
setBirthDate(Date birthDate)                          void
setCurrentWorkPlace(String currentWorkPlace)          void
setFirstName(String firstName)                        void
setGradesReceived(List<Grade> gradesRecei...          void
setId(long id)                                        void
setLastName(String lastName)                          void
setUndergraduateCarreer(String undergradu...          void
setUser(String user)                                  void
toString()                                          String
wait()                                                void
wait(long timeout)                                    void
wait(long timeout, int nanos)                         void
```

Public or protected Person and Student Methods are inherited by the **GraduateStudent** subclass

Inheritance and Access Modifiers



Public or protected
Person and Student
Methods are inherited
by the
GraduateStudent
subclass

5. Overriding methods

Overriding

Overriding involves “rewriting” how a method works internally, **without changing the signature** of that method.

In the real life

Animals talk in different ways

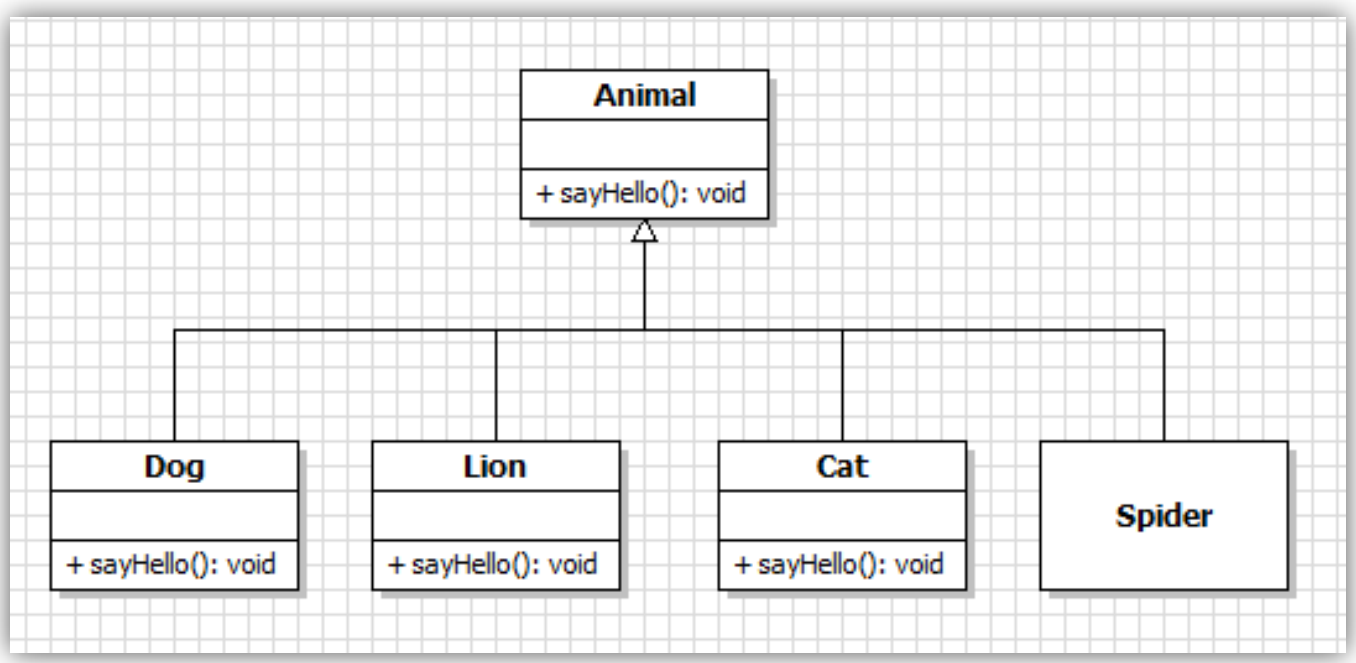
Dogs say: GUAU!!!

Cats say: MEOW!!!

Lions say: GRRR!!!

Spiders say: ……..

In UML life


Animal
+ sayHello(): void
Dog
+ sayHello(): void
Lion
+ sayHello(): void
Cat
+ sayHello(): void
Spider

I am the superclass

```java
public abstract class Animal {

    public void sayHello() {
        System.out.println("I have nothing to say");
    }
}
```

```java
public class Dog extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say GUAU!!!");
    }
}
```

```java
public class Lion extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say GRRRR!!!");
    }
}
```

```java
public class Cat extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say MEOW!!!");
    }
}
```

```java
public class Spider extends Animal {
}
```

Overriding example

```java
public class ZooTest {

    public static void main(String[] args) {

        Animal dog = new Dog();
        Animal cat = new Cat();
        Animal lion = new Lion();
        Animal spider = new Spider();

        dog.sayHello();
        cat.sayHello();
        lion.sayHello();
        spider.sayHello();

    }
}
```

Respective override `sayHello` method is called

```
run:
I say GUAU!!!
I say MEOW!!!
I say GRRRR!!!
I have nothing to say
BUILD SUCCESSFUL (total time: 0 seconds)
```

Overriding example

```java
public class Spider extends Animal {
}
```

```java
public abstract class Animal {

    public void sayHello() {
        System.out.println("I have nothing to say");
    }
}
```

```
run:
I say GUAU!!!
I say MEOW!!!
I say GRRRR!!!
I have nothing to say
BUILD SUCCESSFUL (total time: 0 seconds)
```

If no method is found, the JVM search for it in the superclass

Overriding example (another method)

This is a **valid override**, because both methods have the same signature

walk (int)

```
public abstract class Animal {

    public void walk(int centimeters) {
    }

    // ...
```

```
public class Dog extends Animal {

    @Override
    public void walk(int metters) {
    }

    // ...
```

Overriding example (another method)

This is a **invalid override**, because both methods have different signature

walk (int)

walk (long)

```
public abstract class Animal {

    public void walk(int centimeters) {
    }

    // ...
```

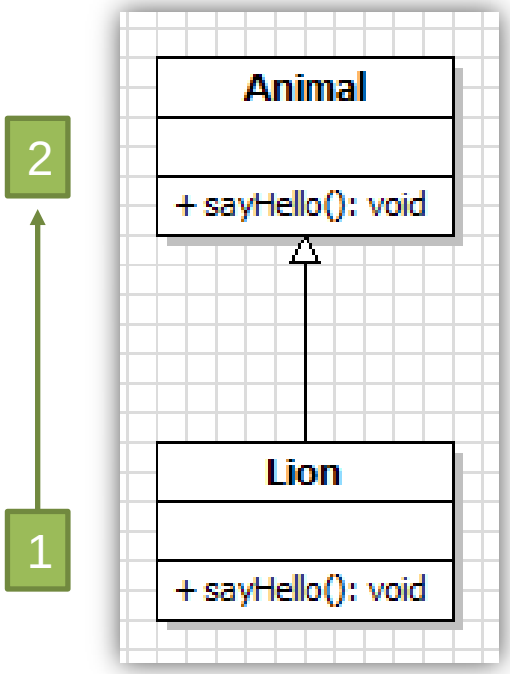
```
public class Lion extends Animal {

    @Override
    public void walk(long metters) {
    }

    // ...
```

```
method does not override or implement a method from a supertype
--
(Alt-Enter shows hints)
```

Execution order

When we override methods, first is called the more specific method

More general

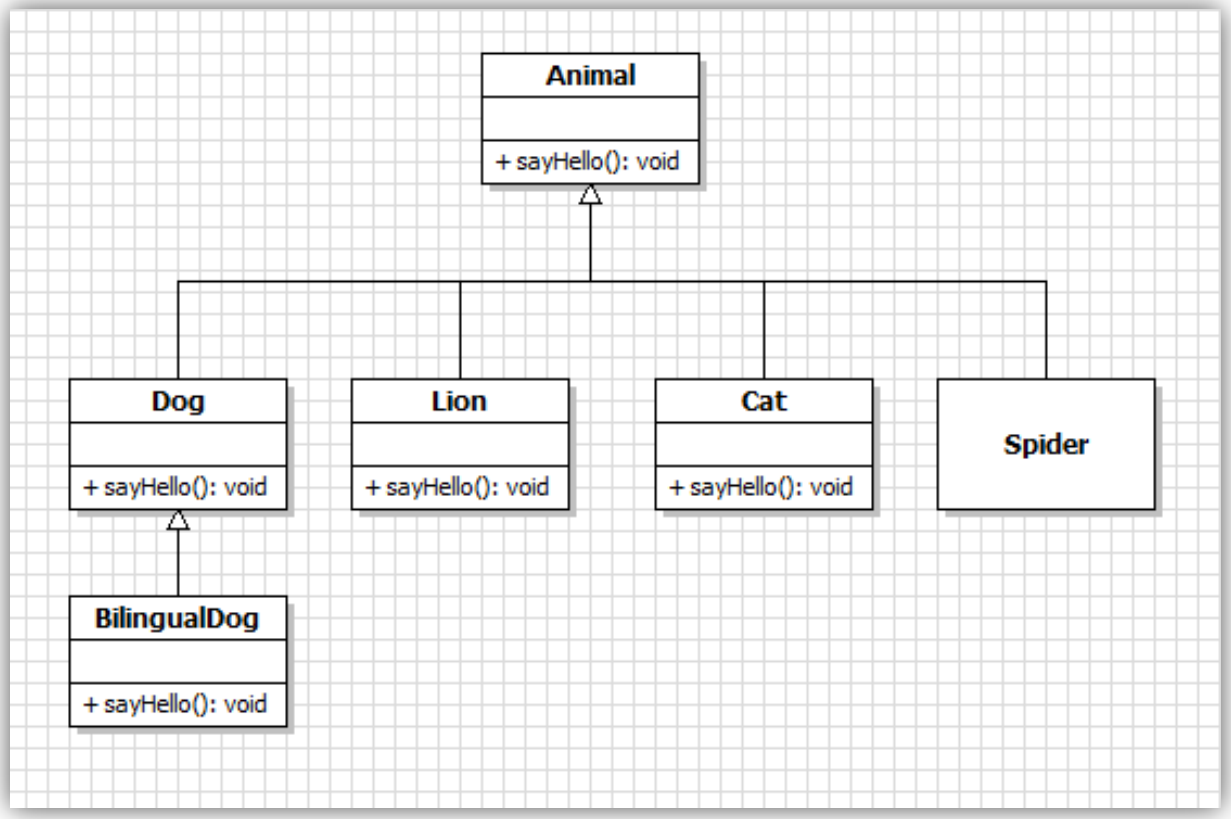


More specific

6. Reusing superclass behaviors

Reusing superclass methods

A Normal Dog can say GUAU!!!



A Bilingual Dog can say GUAU!!!

A Bilingual Dog can <u>also</u> say REGUAUSS!!!

Normal Dog

A Normal Dog can only say GUAU!!!

A Bilingual Dog speaks normal **Dog Language** and **Ancient Dog Language**

```java
public class Dog extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say GUAU!!!");
    }
}
```

We can reuse the superclass methods with **super** keyword

```
public class BilingualDog extends Dog {

    @Override
    public void sayHello() {

        super.sayHello();

        System.out.println("I say REGUAUSS!!!");
    }
}
```

A Bilingual dog can speak as normal dog using the same methods

I am reusing the superclass dog sayHello() method

Reusing superclass methods

```java
public class ZooTest {

    public static void main(String[] args) {

        Animal dog = new Dog();
        Animal cat = new Cat();
        Animal lion = new Lion();
        Animal spider = new Spider();
        Animal biligualDog = new BilingualDog();

        dog.sayHello();
        cat.sayHello();
        lion.sayHello();
        spider.sayHello();

        System.out.println();

        biligualDog.sayHello();

    }
}
```

```java
public class BilingualDog extends Dog {

    @Override
    public void sayHello() {

        super.sayHello();

        System.out.println("I say REGUAUSS!!!");
    }
}
```

```
run:
I say GUAU!!!
I say MEOW!!!
I say GRRRR!!!
I have nothing to say

I say GUAU!!!
I say REGUAUSS!!!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Time to play in the pet store

1. Using UML design a **class hierarchy** (at least 3 levels of inheritance) for a **pet store with** at least 6 different kind of pets
2. Create the **Java classes definitions** (encapsulated) for the pets available on the pet store, each pet must have at least 3 attributes (not inherited).
3. Create a **test class** for your pet store, this class must show a menu with the available pets (previously created).
4. User can select a pet and the **system must show all information available for this pet (including ancestor information).**
5. Program finish when user select the option finish in the main menu
6. **You have to use the keyword super and override annotation.**

References

[Barker] J. Barker, *Beginning Java Objects: From Concepts To Code*, Second Edition, Apress, 2005.

[Oracle] *Understanding Instance and Class Members,* Available: http://download.oracle.com/javase/tutorial/java/javaOO/classvars.html

[Oracle] Java API documentation, *Class Object,* Available: http://download.oracle.com/javase/6/docs/api/java/lang/Object.html