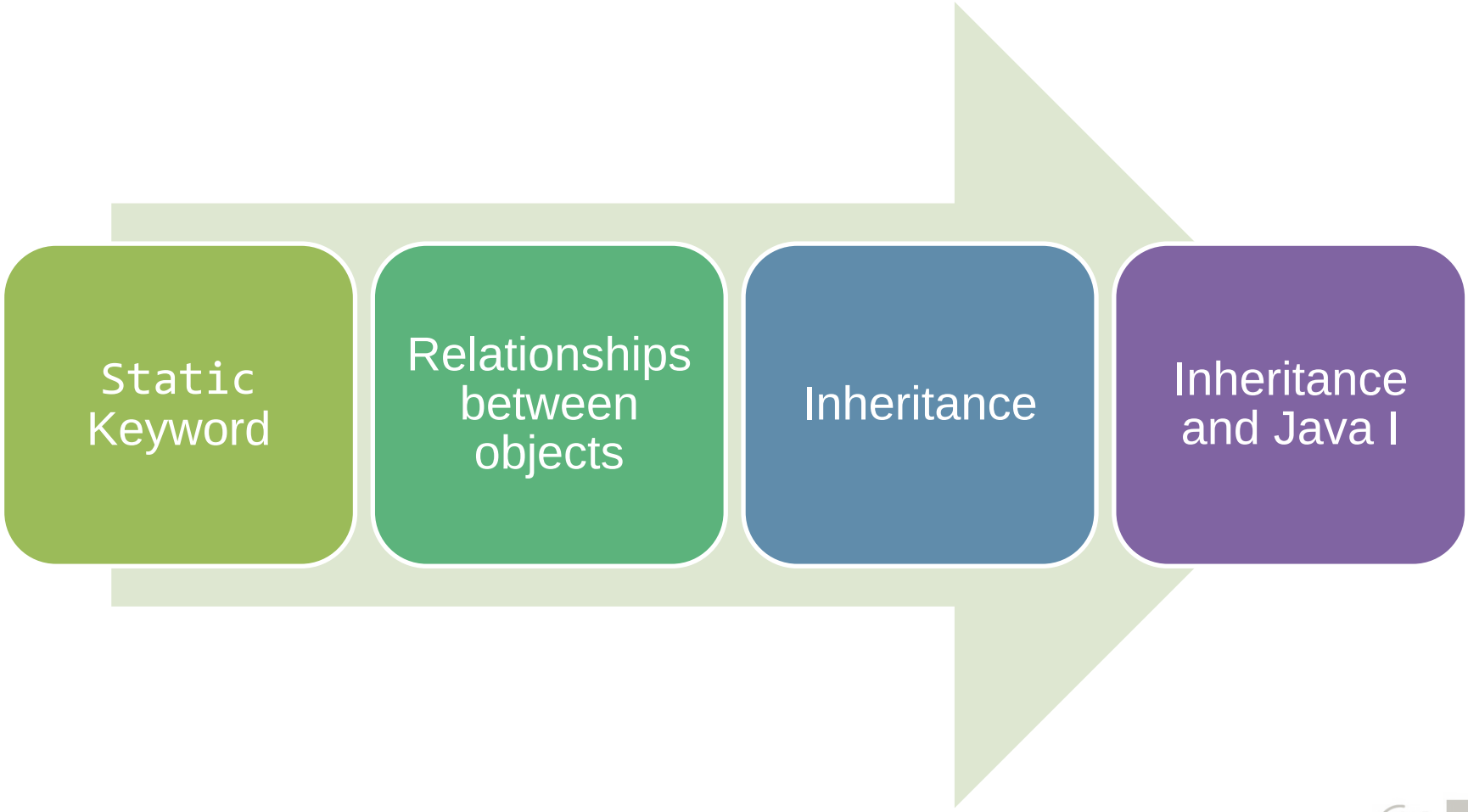
Relationships between objects I

Christian Rodríguez Bustos

Object Oriented Programming

Agenda


Static Keyword
Relationships between objects
Inheritance
Inheritance and Java I

`static` Keyword

`Static` attributes

`Static` methods

Static attributes

Static attributes are **common to all instanced** objects

Static attributes are **class attributes**

```
package lesson;

public class Student {

    private int id;
    private static int numberOfStudents = 0;

    // ...
    public Student() {
        id = ++numberOfStudents;
    }

    public int getID() {
        return id;
    }

    public static int getNumberOfStudents() {
        return numberOfStudents;
    }
    // ...
}
```

Static attributes example

```java
package lesson;

public class StudentTest {

    public static void main(String[] args) {

        Student studentA = new Student();
        System.out.println("ID Student A: " + studentA.getID());

        Student studentB = new Student();
        System.out.println("ID Student B: " + studentB.getID());

        Student student3 = new Student();
        System.out.println("Id Student C: " + student3.getID());

        System.out.println("Number of students: " + Student.getNumberOfStudents());

    }
}
```

Output - Assignment03 (run)

```
run:
ID Student A: 1
ID Student B: 2
Id Student C: 3
Number of students: 3
BUILD SUCCESSFUL (total time: 0 seconds)
```

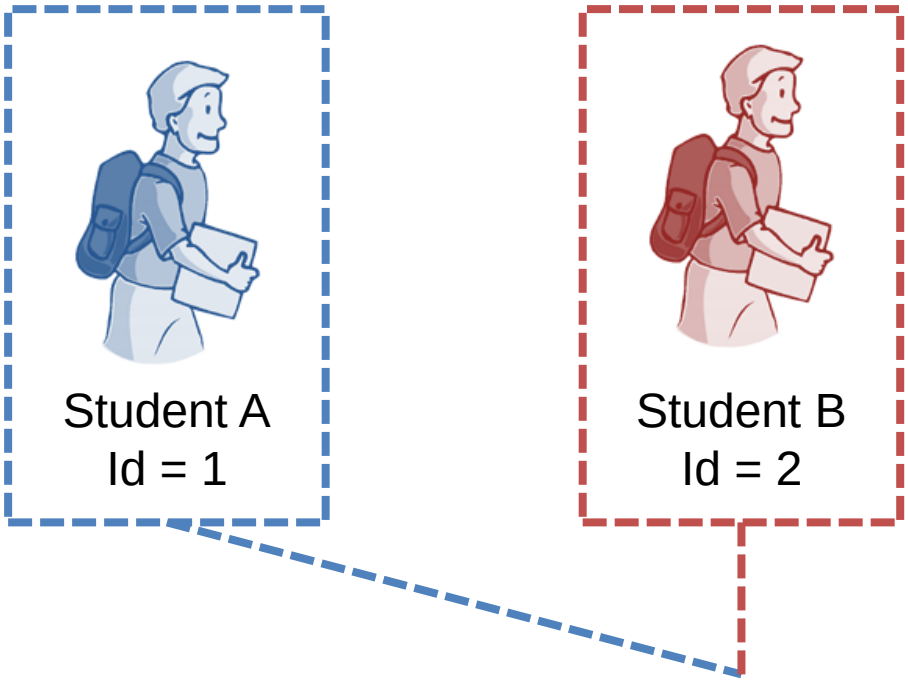
Static attributes example

Student A
Id = 1

Static property Number of students = **1**

```
Student studentA = new Student();
System.out.println("ID Student A: " + studentA.getID());
```

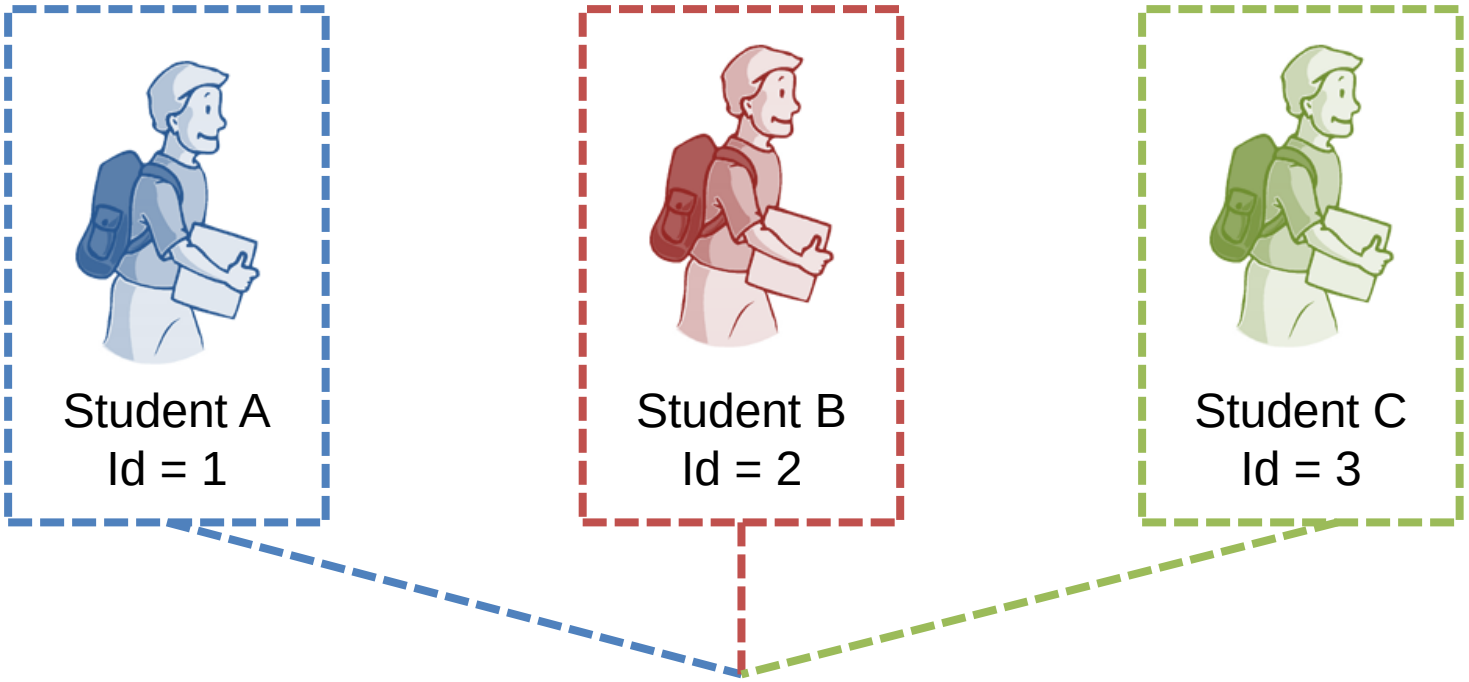
Static attributes example



Static property Number of students = **2**

```
Student studentB = new Student();
System.out.println("ID Student B: " + studentB.getID());
```

Static attributes example



Static property Number of students = **3**

```
Student student3 = new Student();
System.out.println("Id Student C: " + student3.getID());
```

Accessing `static` attributes

```
s) {

;
: " + studentA.getID());

;
: " + studentB.getID());

;
: " + student3.getID());

udents: " + Student.getNumberOfStudents());
```

```
package lesson;

public class Student {

    private int id;
    private static int numberOfStudents = 0;

    // ...
    public Student() {
        id = ++numberOfStudents;
    }

    public int getID() {
        return id;
    }

    public static int getNumberOfStudents() {
        return numberOfStudents;
    }
    // ...
}
```

Accessing `static` methods or attributes

We **do not need to instantiate any object** to access static attributes or methods

```
Student.getNumberOfStudents()
```

Examples

Static methods typically take all they data from parameters and compute something from those parameters.

```
MovementHandler.isWinningMovement(board, current)
```

```
Math.cos(15.6);
```

```
Math.max(4.7,8.9);
```

```
double pi = Math.PI;
```

```
double e = Math.E;
```

```
Math.sin(5.8);
```

TicTacToe Examples

```
MovementHandler.isWinningMovement(board, current)
```

```
UI.printWelcome(player1, player2)
```

```
UI.printWinner(current)
```

```
play()
```

```
TurnController.existFreeSquares(board, player1, player2)
```

```
UI.printTie()
```

Accessing `static` methods or attributes

Be careful!!!

Static methods cannot access Non-Static attributes

```
non-static variable id cannot be referenced from a static context
--
(Alt-Enter shows hints)
```

```
// ...
public static int getID() {
    return id;
}
// ...
```

Relationships between objects

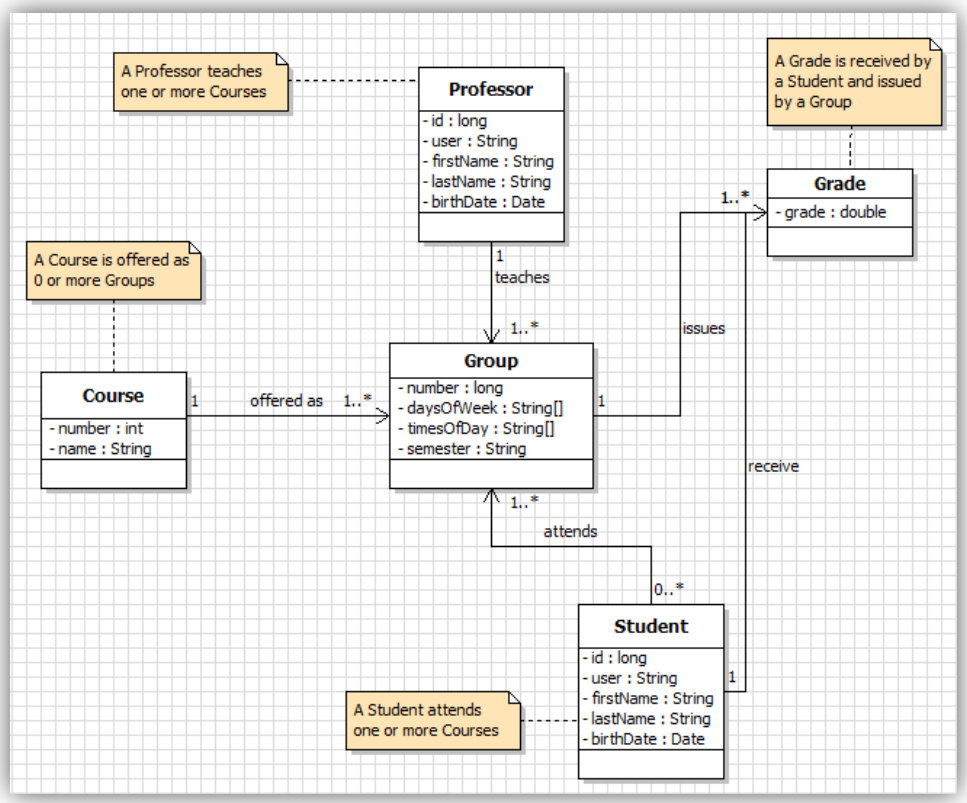
Association and links

Aggregation and composition

UML notation

Associations are structural relationship

Associations are structural relationship that exists between **classes**



- A **Professor** *teaches* one or many **Groups**
- A **Course** *is offered* as one or many **Groups**
- Zero or many **Students** *attends* one or many **Groups**

Links

Are relations between two specifics **objects** (*instances*)

Association: A student *attends at* Any group

Link: Bruce Wayne *attends at* Math 3B

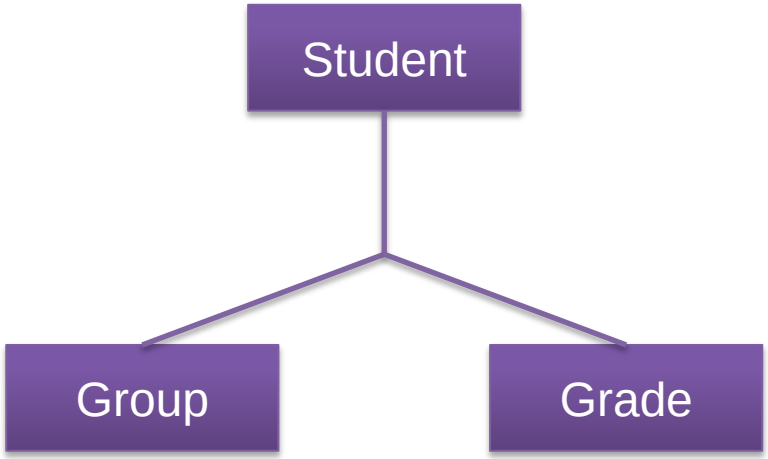
A ***specific*** student — A ***specific*** group

Higher order associations

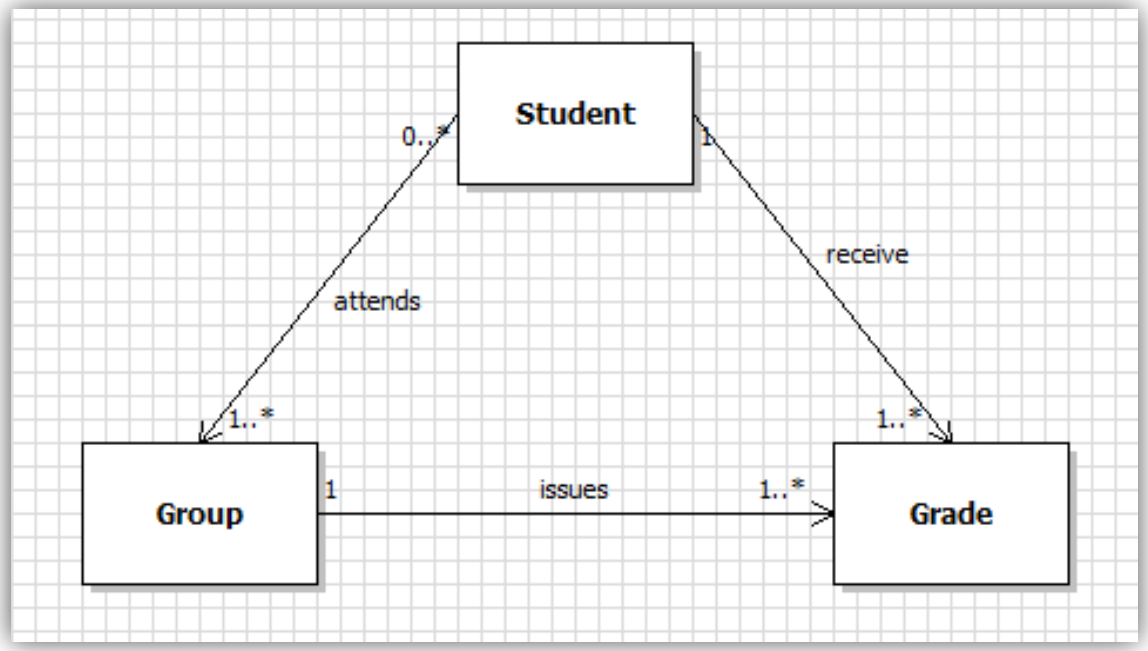
One or more **Student** *attends* one or more **Groups**

One or more **Students** *receive* one or more **Grades**

One **Group** *issues* one or more **Grades**



Higher order associations are represented by two classes associations



Aggregation and Composition

Aggregation: Is a specific type of association, is represented typically by "*consists of*", "*is composed of*" and "*has a*"

A Team ***is composed by*** one or more Students

A Department ***is composed of*** one or more Professors

A Club ***has*** Members

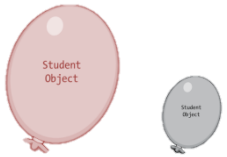
Composition: Is a strong form of aggregation, in which **the "parts" cannot exist without the "whole."**

A Building ***is composed by*** one or more Rooms

A University ***is composed of*** Departments

A Board ***is composed of*** Squares

Aggregation code example




```
public class Team {

    private Student[] students;

    public Team(Student[] students) {
        this.setStudents(students);
    }

    // ...
```

Class definition

A Team ***is composed by*** one or more Students

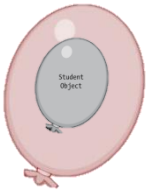
```
Student[] students = new Student[3];
Team team = new Team(students);
```

Test code

If Team is destroyed, the Students **still** exist

Composition code example

A Board ***is composed of*** Squares



```
public class Board {

    private Square[][] board;

    public Board() {

        board = new Square[3][3];
        // ...
    }

    // ...
```

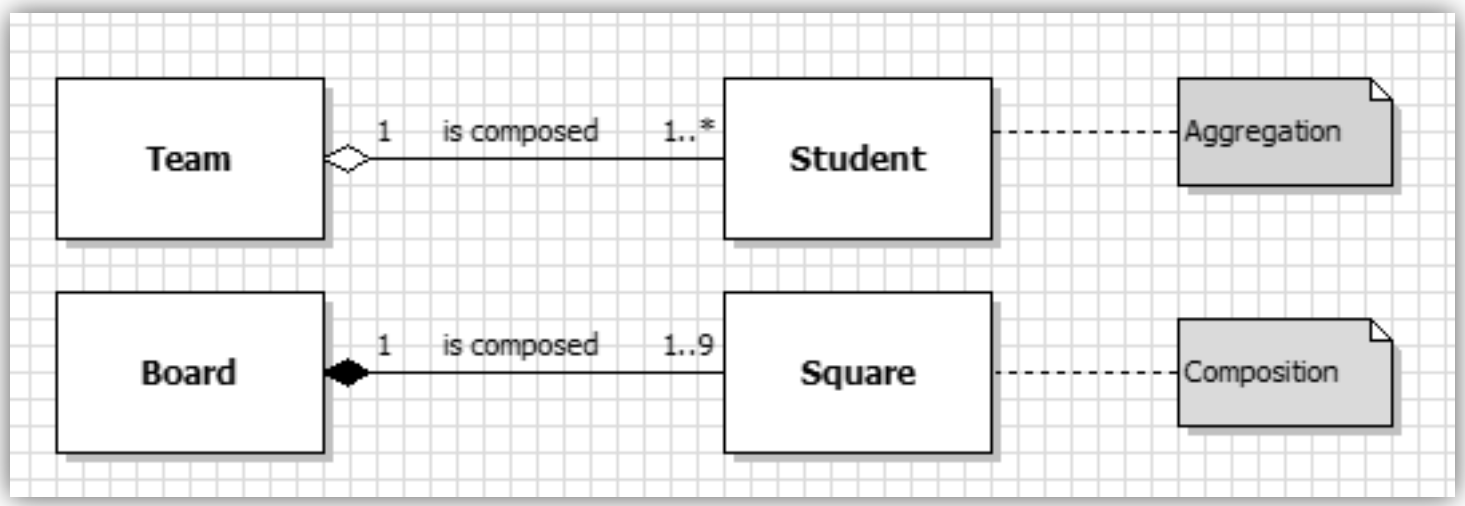
Class definition

```
Board board = new Board();
```

Test code

If Board is destroyed, the squares are destroyed too

UML notation



Aggregation is depicted as an **unfilled diamond**

Composition is depicted as a **filled diamond** and a solid line.

Inheritance

Hierarchy of classes

UML Notation

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 have 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 to 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 one or another?

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() + '}';

    }

    // ...
```

Gradate Student **inherit all 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
```

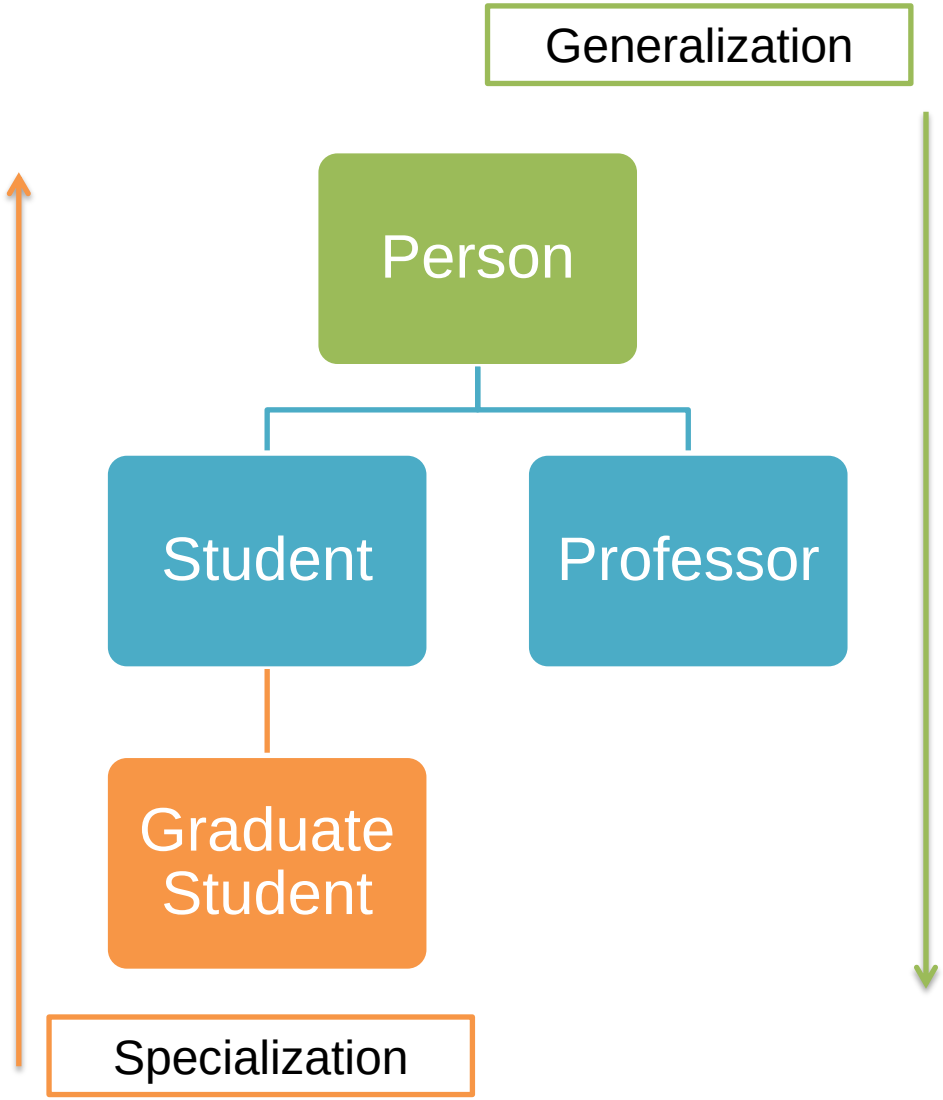
A _____ *Is a* _____

subclass superclass

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

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

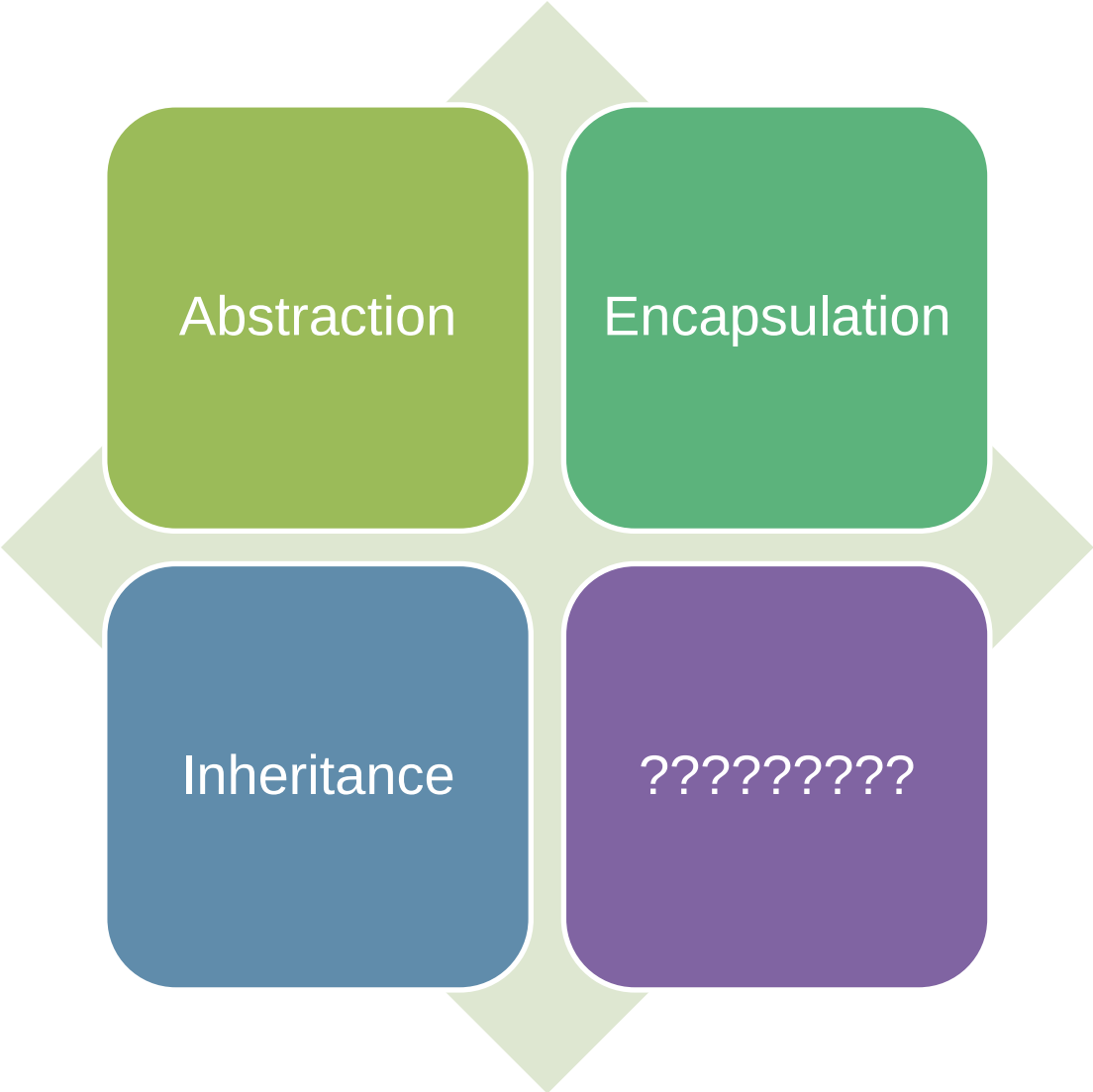
Class Hierarchies



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”

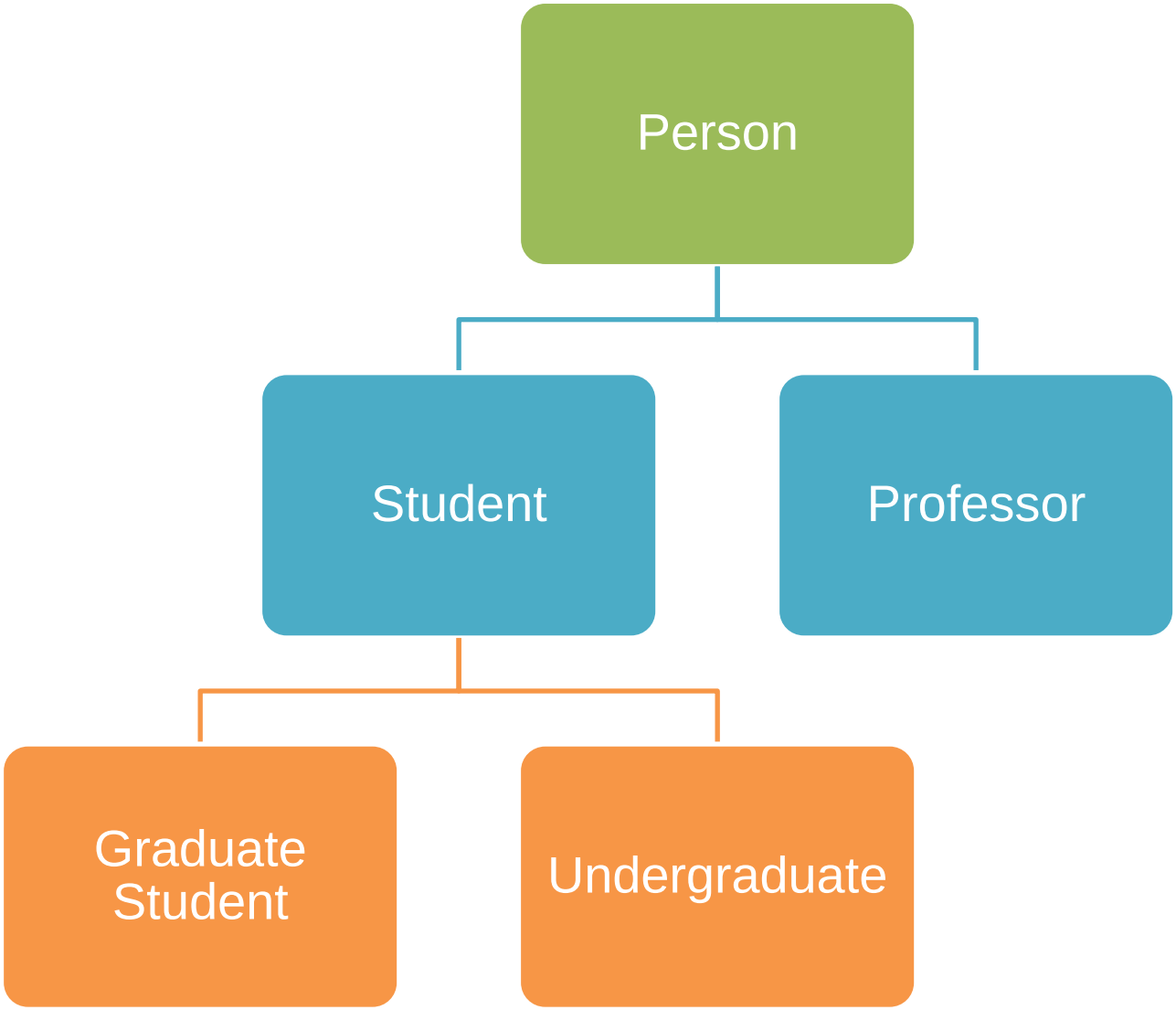
Subclasses are more **concise**

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

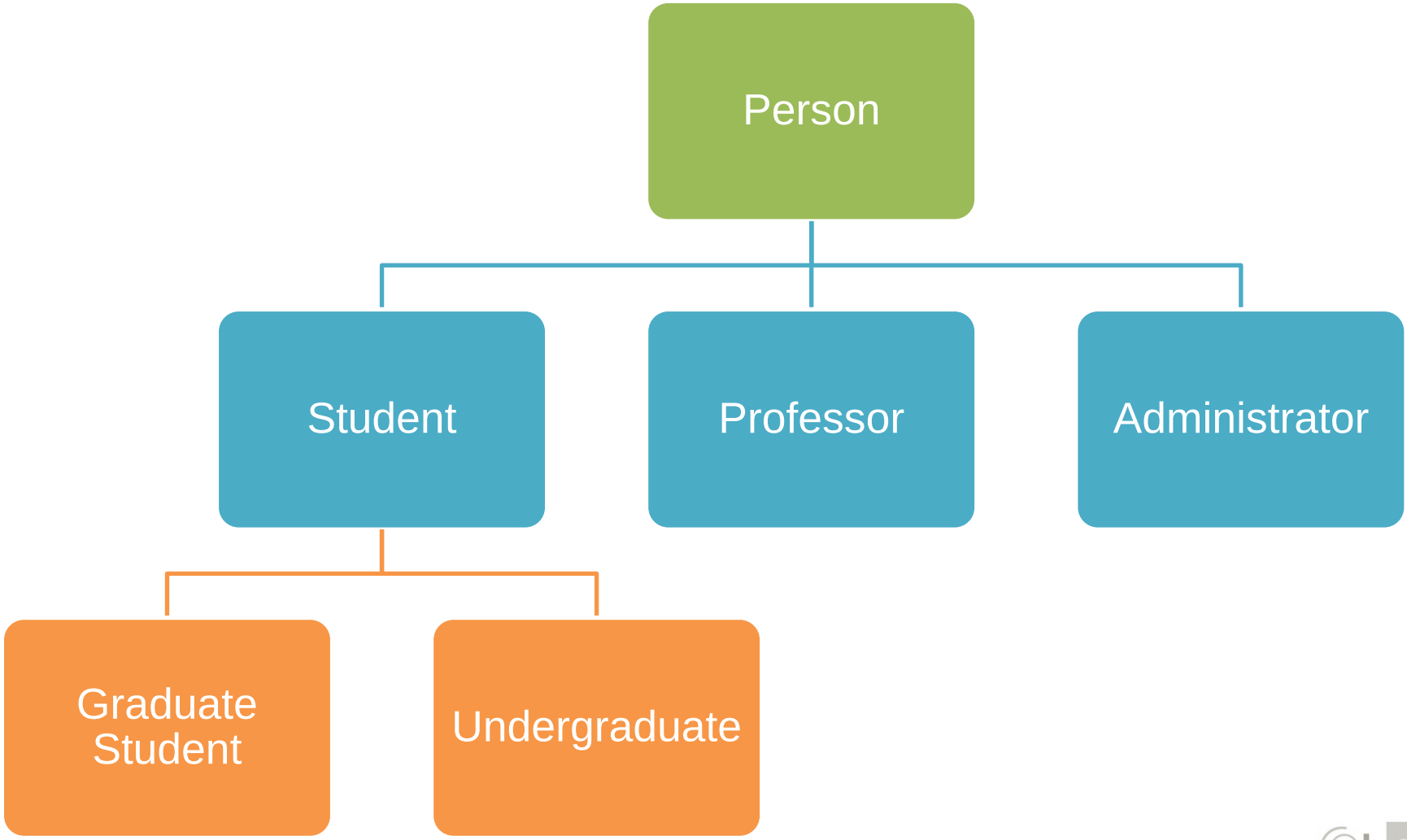
We can **derive** a new class from an existing class

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

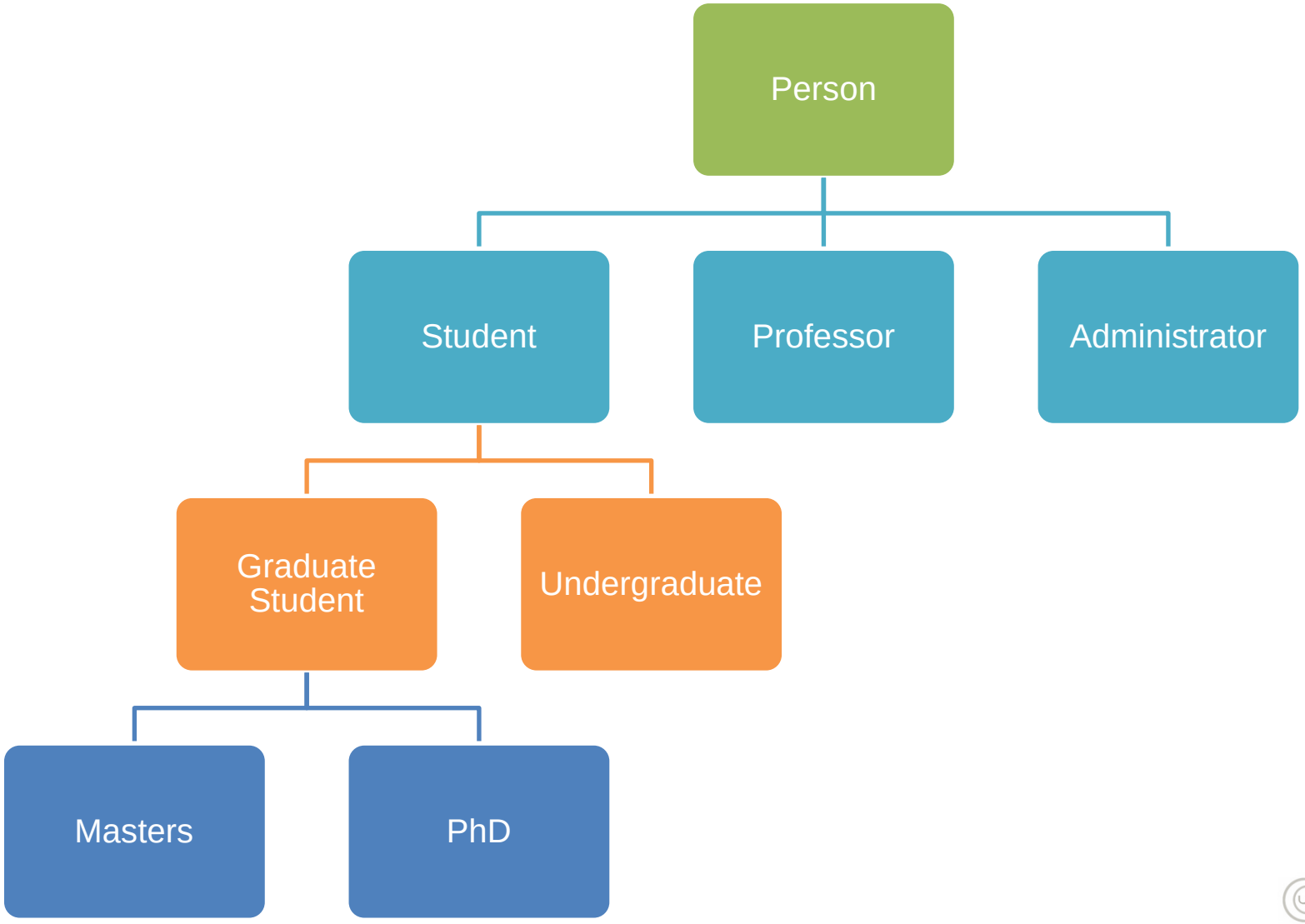
Class Hierarchies inevitably expand over time



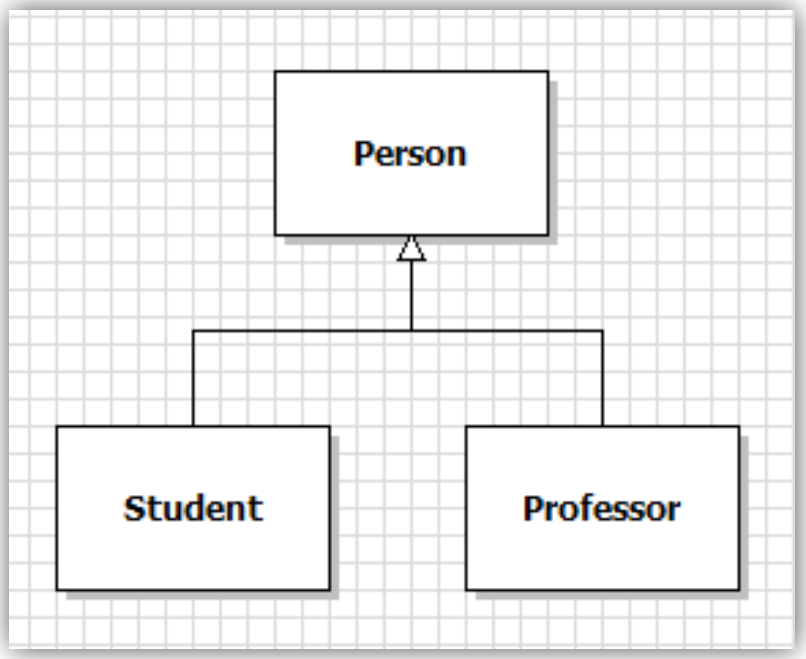
Class Hierarchies inevitably expand over time



Class Hierarchies inevitably expand over time



UML notation



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

Java Inheritance I

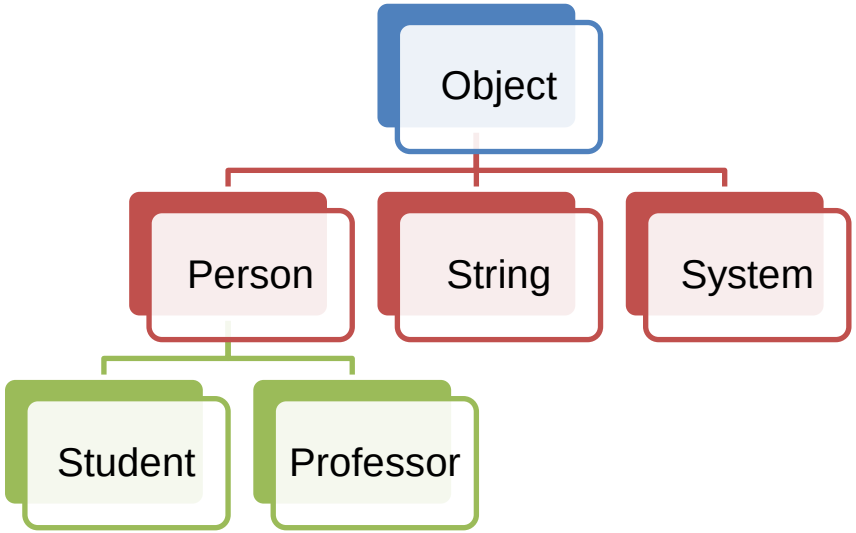
The object class

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

Time to play

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).

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