Object Oriented Programming

Digital Urban Visualization. People as Flows.

12.10.2015

iA

zuend@arch.ethz.ch

treyer@arch.ethz.ch

Object Oriented Programming?

Pertaining to a technique or a programming language that supports objects, classes, and inheritance.

Object oriented programming languages enable a software architecture with a special data type:
Objects

Data Types

primitive data types

Name	Definition	Example
byte	8 bits	00010111
short	16 bit full number	23
int	32 bit full number	23
long	64 bit full number	23
float	32 bit floating point number	23.0
double	64 bit floating point number	23.0
char	16 bit Unicode character	23.0
boolean	one bit information, no specified size	true

Data Types

your own data type

It is possible to define your own data types. They can be whatever you can imagine, for example, cars, students, plants, …

More complex data types have states and functionalities. For example a car has some amount of gasoline left, and has the possibility to turn left.

Classes & Objects

In Java your own data type is defined with the *class* keyword.

A class is the blueprint for your data type. All the functionalities, states and parameters for it have to be defined inside its curly brackets.

```
public class Student {
// Class definition



}
```

Classes & Objects

First define the states and parameters the class should have. In the following example, it has only one property; the name of the student.

It is set to **private**, because it should not be changeable from outside the object itself.

```
public class Student {
// Class definition
private String name;




}
```

Classes & Objects

An object is an entity that is constructed according to the blueprint, i.e. the class definition.

When an object is constructed, it needs to know how to set its initial values. This is done by defining a **constructor method** that must have the same name as the class.

```
public class Student {
// Class definition
private String name;
public Student(String name){
this.name = name;
}



}
```

Classes & Objects

It is now possible to use this object in your code. You could, for example, put the following inside the **main** method:

*Student myStudent = **new** Student("Hans");*

This assigned a new object of type **Student** to the variable **myStudent**. The private variable of **myStudent** is set to *Hans*.

```
public class Student {
// Class definition
private String name;
public Student(String name){
this.name = name;
}


}
```

Classes & Objects

It is not very useful to have an object without the possibilities to do something with it. For this, one can define functions inside the class and add functionalities with that.

If the function has to be accessible from outside, it has to be **public**.

```
public class Student {
// Class definition
private String name;
public Student(String name){
this.name = name;
}
public void sayHello(){
System.out.println(this.name + " says: Hello World");
}
}
```

Classes & Objects

functions

```
public void sayTwoThings(String firstThing, String secondThing) {
System.out.println(firstThing);
System.out.println(secondThing);
}
```

A function is defined by the following properties:

Access Modifier: it defines if a function is accessible from outside the object or not. The keywords are *private, public,* and *protected*.
Return Type: defines the data type that is returned when the function is called. It can be any data type. If the function does not have a return value, the keyword is *void*.
Function Name: defines the name of the function.
Input Parameters: defines the input type, it can be a list of as many as desired, belongs inside the brackets, and is defined by data type and parameter value.

Classes & Objects

where to put class definitions

When making a new class definition, the best way is to make a new file. In Eclipse this can be done by right clicking on the package and then
New → *Class*.
In the dialogue put in the name of your class and click *Finish*.

Classes & Objects

accessing functions

Student.java file:

```
public class Student {
private String name;
public Student(String name){
this.name = name;
}
public void sayHello(){
System.out.println(this.name + " says: Hello World");
}
}
```

Classes & Objects

accessing functions

Student.java file:

```
public class Student {
private String name;
public Student(String name){
this.name = name;
}
public void sayHello(){
System.out.println(this.name + " says: Hello World");
}
}
```

main.java file:

```
public class Main {
public static void main(String[] args) {
Student myStudent;
myStudent = new Student("Hans");
myStudent.sayHello();
}
}
```

Classes & Objects

accessing functions

Student.java file:

```
public class Student {
private String name;
public Student(String name){
this.name = name;
}
public void sayHello(){
System.out.println(this.name + " says: Hello World");
}
}
```

main.java file:

```
public class Main {
public static void main(String[] args) {
Student myStudent;
myStudent = new Student("Hans");
myStudent.sayHello();
}
}
```

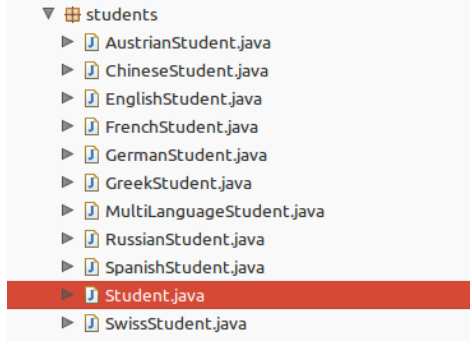
Output:
Hans says: Hello World

Inheritance

Inheritance is very helpful to have a clean software architecture.

It is especially useful when many similar data types are needed, which share a common super-type.

One way to implement this would be to program all students as single classes.

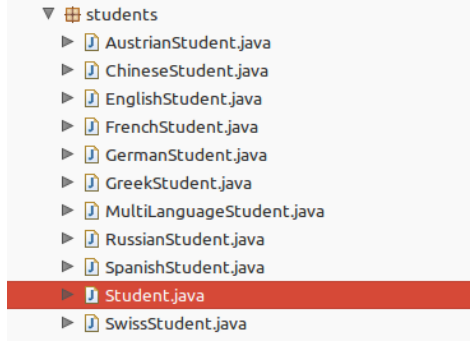


Inheritance

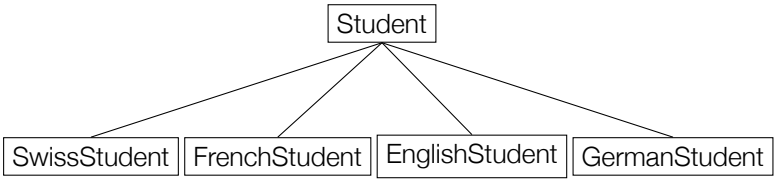
But this is not so clean and may include a lot of work!

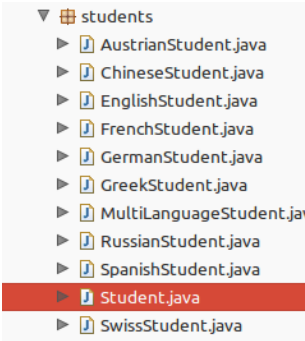
Better is to define a common super-type and then inherit from it and only change the values and functions that are different.

This is also where the *protected* keyword comes into play.



Inheritance





```
public class Student {
        private String name;

        public Student(String name){
                this.name = name;
        }

        public void sayHello(){
                System.out.println(this.name  + " says: " + this.getHello());
        }

        public String getHello(){
                return "Hello World";
        }
}
```

```
public class SwissStudent extends Student {

        public SwissStudent(String name) {
                super(name);
        }

        @Override
        public String getHello(){
                return "Hoi Wält!";
        }
}
```

```
public class RussianStudent extends Student {

        public RussianStudent(String name) {
                super(name);
        }

        @Override
        public String getHello(){
                return "Привет Мир!";
        }
}
```

```
public class EnglishStudent extends Student {

        public EnglishStudent(String name) {
                super(name);
        }

}
```

Polymorphism

An additional property of using inheritance in the software architecture is Polymorphism.

This concept allows to treat every child object as if it was the root type, but get the child's functionalities.

```
public static void main(String[] args) {
    Student lukas = new SwissStudent("Lukas");
    Student danielle = new EnglishStudent("Danielle");
    Student artem = new RussianStudent("Artem");

    lukas.sayHello();
    danielle.sayHello();
    artem.sayHello();
}
```

```
public static void main(String[] args) {
    ArrayList<Student> students = new ArrayList<Student>();
    students.add(new SwissStudent("Lukas"));
    students.add(new EnglishStudent("Danielle"));
    students.add(new RussianStudent("Artem"));

    for (Student studi : students) {
        studi.sayHello();
    }
}
```

Exercise

In the first part of the exercise one thing you have to do, is to implement the super class of **PrintPoint** called **Point**.