



# *Systems programming*

## *Object Oriented Programming*

I. Object ***Based*** Programming

II. Object ***Oriented*** Programming

Telematics Engineering

M. Carmen Fernández Panadero

<mcfp@it.uc3m.es>





# Scenario V:

## Reusing code. Inheritance

- Once you are able to create your own classes, you are ready to work in teams and reuse code developed by your colleagues. Your team will provide you with a set of classes and you are required to create specializations or generalizations of them.
- **Objective:**
  - Be able to create a **derived class** adding some characteristics (attributes) and behavior (methods) to an existing class.
  - Be able to extract all the common code from a set of similar classes in order to group it into a new **parent class** so that it is easier to maintain.
  - Be able to create objects, and reference and access their attributes and methods, depending on their position in the inheritance hierarchy and their modifiers.
- **Work plan:**
  - Memorize the **nomenclature** related to inheritance
  - Memorize the java **syntax** related to inheritance (**extends**) and to reference (**super, this**) and access (**modifiers**) to the different members.
  - Know basic inheritance mechanisms, such as attribute **hiding**, **overriding** of methods and **overloading** of constructors, and know what they are used for and how they are used.





# Contents



- Basic inheritance concepts
- Inheritance hierarchy
- Overriding I: Attribute Hiding
- Overriding II: Method Overriding
- Constructors of derived classes
- Static and Final Modifiers
- Scope and access





# Inheritance

## What is it?



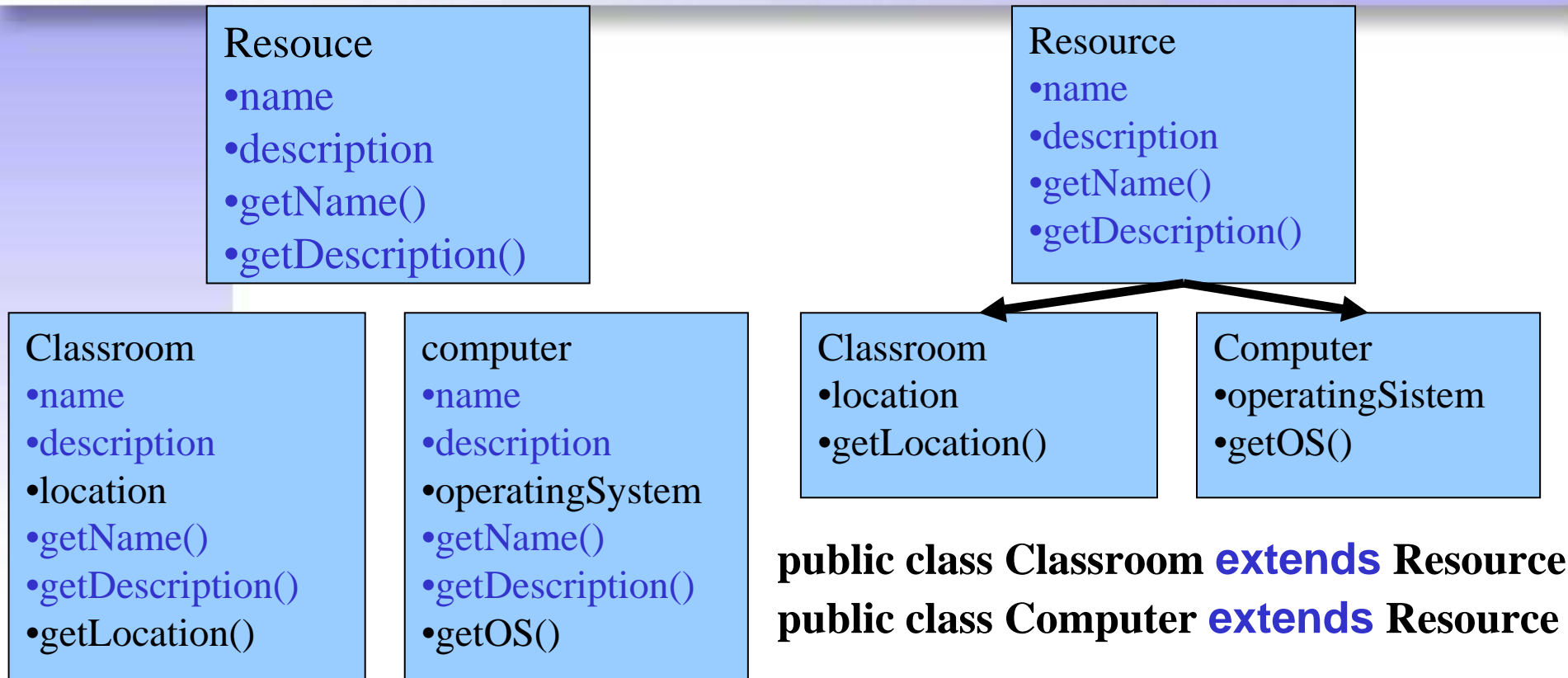
- Mechanism for software reuse
- Allows to define from a class other related classes that can be a:
  - **Specialization** of the given class. (Eg “Car” class is a specialization of the class “Vehicle”)
    - **Scenario:** We have to implement a new class that is very similar to a previous one but it needs additional information (characteristics and behaviour)
    - **Solution:** Create a class derived from the old one and add it new functionality without having to rewrite the common code
  - **Generalization** of the given class. (The “Vehicle” class is a generalization of the “Car” class).
    - **Scenario :** We have a set of similar classes with code that is repeated in every class and thus difficult to update and maintain (eg. a letter should be added to the serial number)
    - **Solution :** We move the code that is repeated to a single site (the parent class)





# Inheritance

## What is it?



The attributes and methods that appear in blue in the parent class are repeated in the subclasses. (Left picture)

It is not necessary to repeat the code, you only have to say that a class **extends** the other or **inherits** from it. (Right picture)





# Inheritance

## Nomenclature



- If we define the class car from the class vehicle, it is said that:
  - “car” *inherits* attributes and methods from “vehicle”
  - “car” *extends* “vehicle”
  - “car” is a *subclass* of “vehicle”
    - Derived* class
    - Child* class
  - “vehicle” is a *superclass* of “car”
    - Base* class
    - Parent* class
- Inheritance implements the *is-a* relation
  - A car **is-a** vehicle; a dog **is-a** mammal, etc.





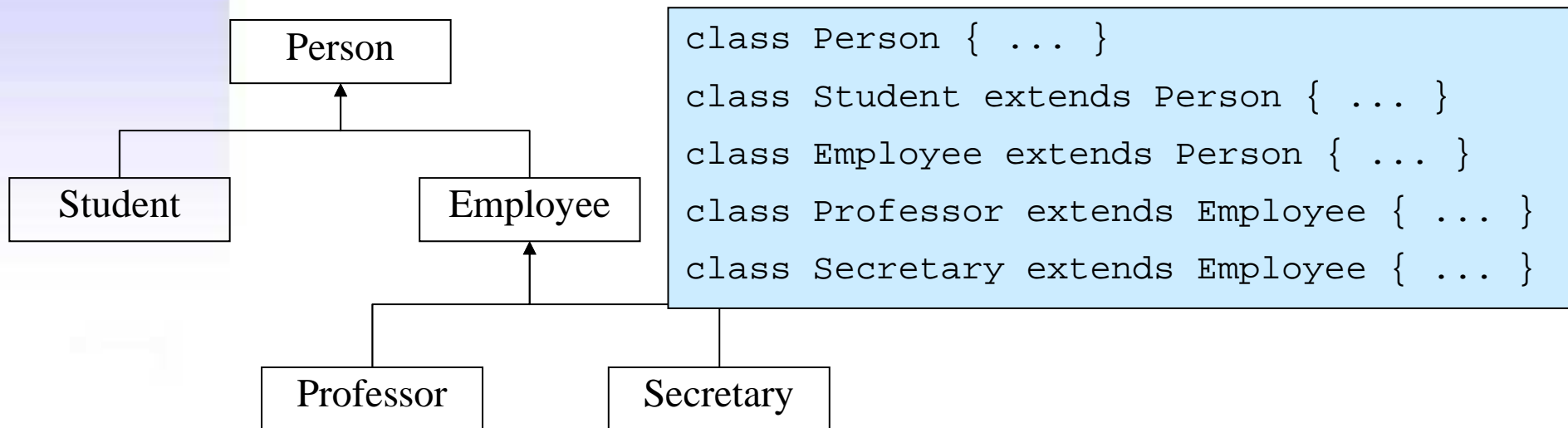
# Inheritance

## Declaration of subclasses



- The syntax for declaring subclasses is:

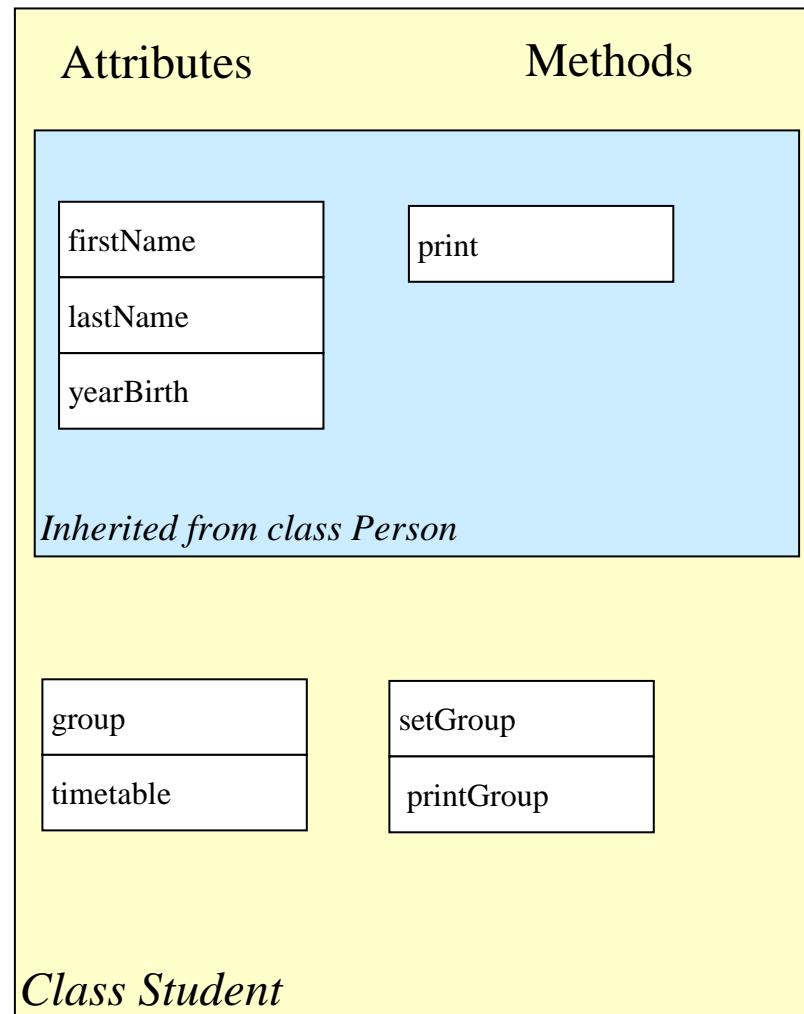
```
class Subclass extends Superclass { ... }
```





# Inheritance

## Subclass







# Inheritance

## How is it used? Eg.: Person.java



```
public class Person {  
    protected String firstName;  
    protected String lastName;  
    protected int yearBirth;  
  
    public Person () {  
    }  
    public Person (String firstName, String lastName,  
                   int yearBirth){  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.yearBirth = yearBirth;  
    }  
    public void print(){  
        System.out.print("Personal data: " + firstName  
                          + " " + lastName + " (" +  
                          + yearBirth + ")");  
    }  
}
```





# Inheritance

## How is it used? Eg.: Person.java



```
public class Student extends Person {
    protected String group;
    protected char timetable;

    public Student() {
    }

    public Student(String firstName, String lastName,
                   int birthYear) {
        super(firstName, lastName, birthYear);
    }

    public void setGroup(String group, char timetable)
        throws Exception {
        if (group == null || group.length() == 0)
            throw new Exception ("Invalid group");
        if (timetable != 'M' && timetable != 'A')
            throw new Exception ("Invalid timetable");

        this.group = group;
        this.timetable = timetable;
    }

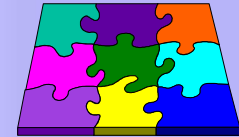
    public void printGroup(){
        System.out.print(" Group " + group + timetable);
    }
}
```





# Inheritance

## How is it used? Eg.: Test.java



```
public class Test {
    public static void main (String[] args) throws Exception{

        Person neighbor = new Person ("Luisa", "Asenjo Martínez", 1978);
        Student aStudent = new Student ("Juan", "Ugarte López", 1985);
        aStudent.setGroup("66", 'M');

        neighbor.print();

        System.out.println();

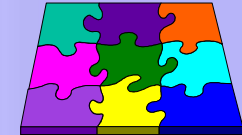
        aStudent.print();
        aStudent.printGroup();
    }
}
```





# Inheritance

## Consequences of extension of classes



- Inheritance of the interface
  - The public part of the subclass contains the public part of the superclass
    - The `student` class contains the method `print()`
- Inheritance of the implementation
  - The implementation of the subclass contains the implementation of the superclass
    - When calling the method of the superclass on an object of the subclass (`aStudent.print()`) the expected behavior takes place





# Inheritance

## Inheritance hierarchy in Java

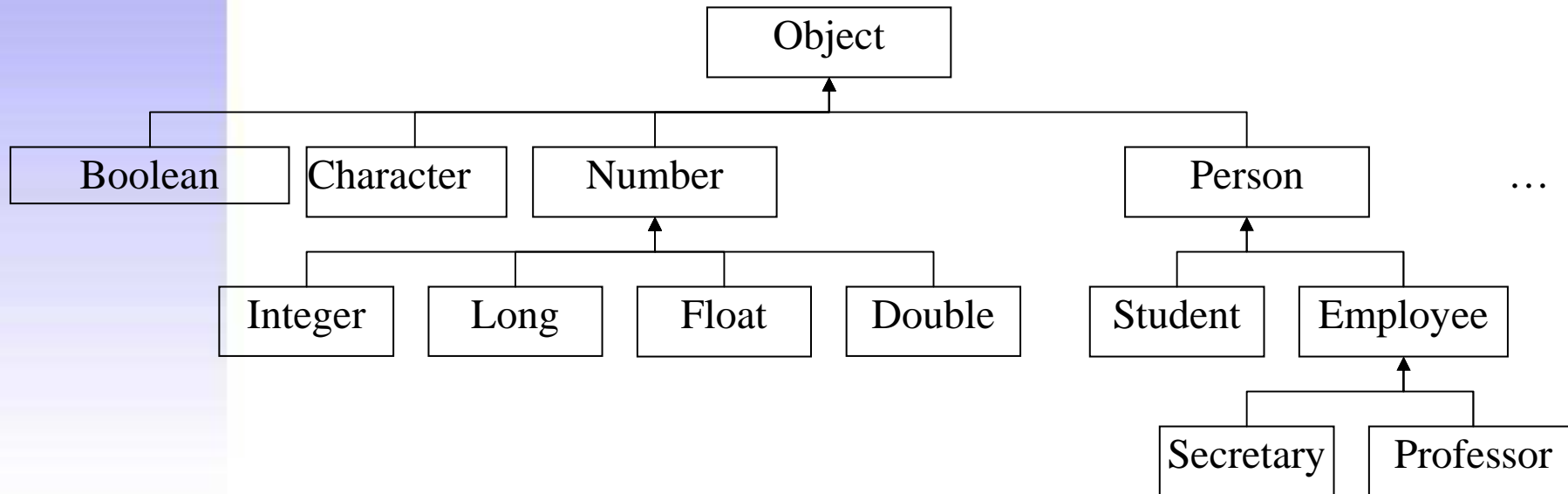


- In Java, all classes are related through a **single inheritance hierarchy**
- A class can:
  - Explicitly inherit from another class, or
  - Implicitly inherit from the class Object (defined in the Java core)
- This is the case both for predefined classes and for user-defined classes





# Inheritance hierarchy





# Inheritance

## Overriding



- Modification of the elements of the base class inside the derived class
- The derived class can define:
  - An attribute with the same name as one of the base class → **attribute hiding**
  - A method with the same signature as one of the base class → **Method overriding**
- The second case is more usual

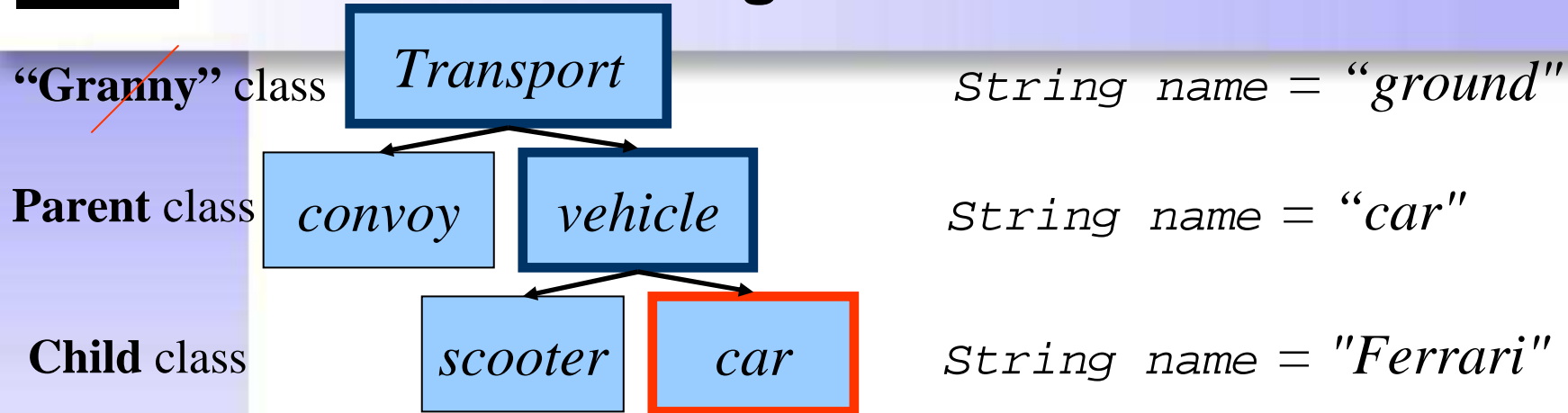




# Overriding I (Shadowing)

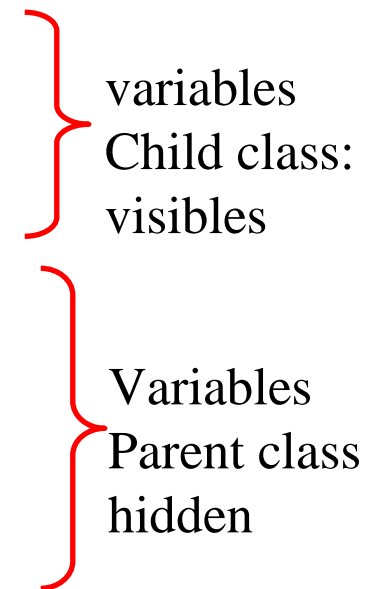


## Attribute hiding



- How can I access to hiding variables?

- *name* (car name)
- *this.name* (car name)
- *super.name* (vehicle name)
- *((vehicle)this).name* (vehicle name)
- ~~*super.super.name* (bad)~~
- *((transport)this).name* (transport name)







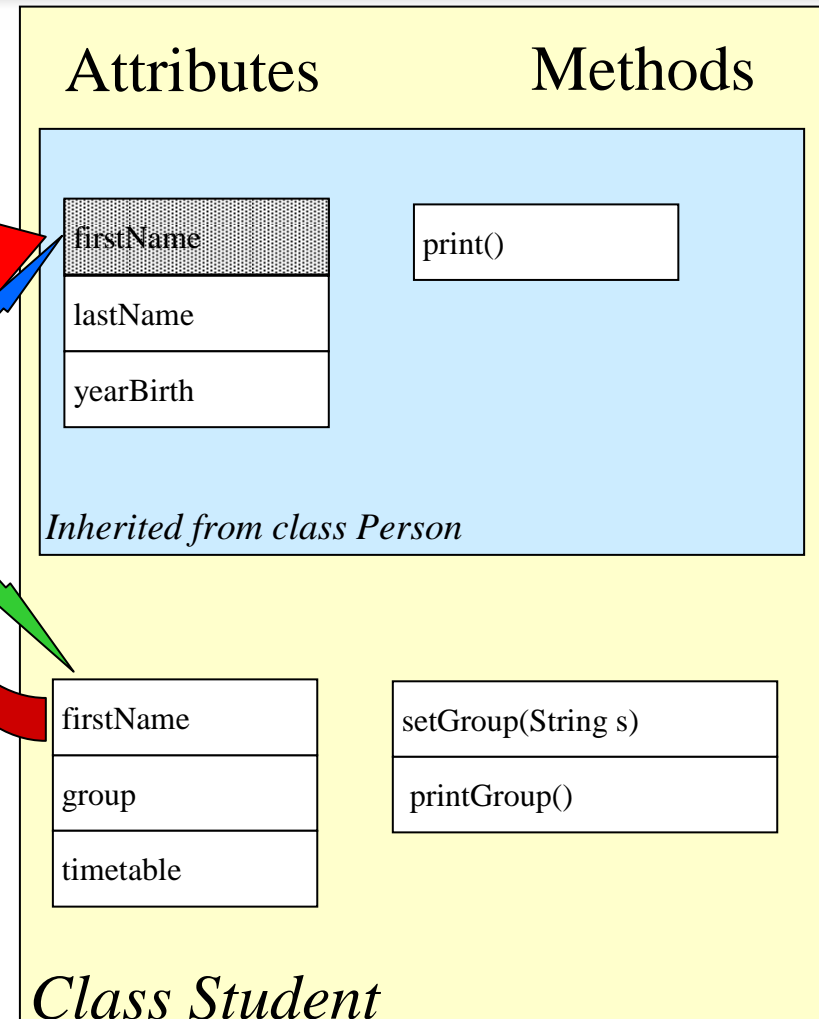
# Overriding I (Shadowing)

## Attribute hiding



```
Student a = new Student(...);  
System.out.println(a.firstName);
```

```
Person p = a;  
System.out.println(p.firstName);
```





# Overriding I (Shadowing)

## Attribute hiding



- When accessing an attribute, the type of the reference is used for deciding which value to access
- The attribute of the subclass needs to have the same name as the one of the superclass
  - But not necessarily the same type
- Not very useful in practice
  - Allows the superclasses to define new attributes without affecting the subclasse





# Overriding I (Shadowing)

## Attribute hiding. Example



```
class SuperShow {  
    public String str = "SuperStr";  
}
```

```
class ExtendShow extends SuperShow {  
    public int str = 7;  
}
```

```
class Show {  
    public static void main (String[] args)  
    {  
        ExtendShow ext = new ExtendShow();  
        SuperShow sup = ext;  
        System.out.println(sup.str);  
        System.out.println(ext.str);  
    }  
}
```

prints SuperStr

prints 7





# Overriding I (Shadowing)

## Attribute hiding



- If we define an *attribute* (*variable*) in a subclass with the same name and type that other one in the superclass. Variable in the superclass remains hidden.
- We can access one variable or the other using *this* and *super*.
  - Eg: “Car” *extends* from “Vehicle” and “Vehicle” *extends* from “Transport”.
  - We define the variable String name in the three classes.
  - How can we know if we are referring to the name of transport, the name of the vehicle or the name of the car?





# Overriding II

## Method overriding. What is it?



- If the subclass defines a *method* with the same **signature** (name + number and type of the parameters) the method in the superclass is hidden.
- If the *final* modifier is used in a method, this method can not be overridden
- How can we access hidden methods?:
  - *start()* (run the start method of the car )
  - *this.start()* (run the start method of the car )
  - *super.start()* (run the start method of the vehicle )
  - ~~*super.super.start()*~~ (Bad)

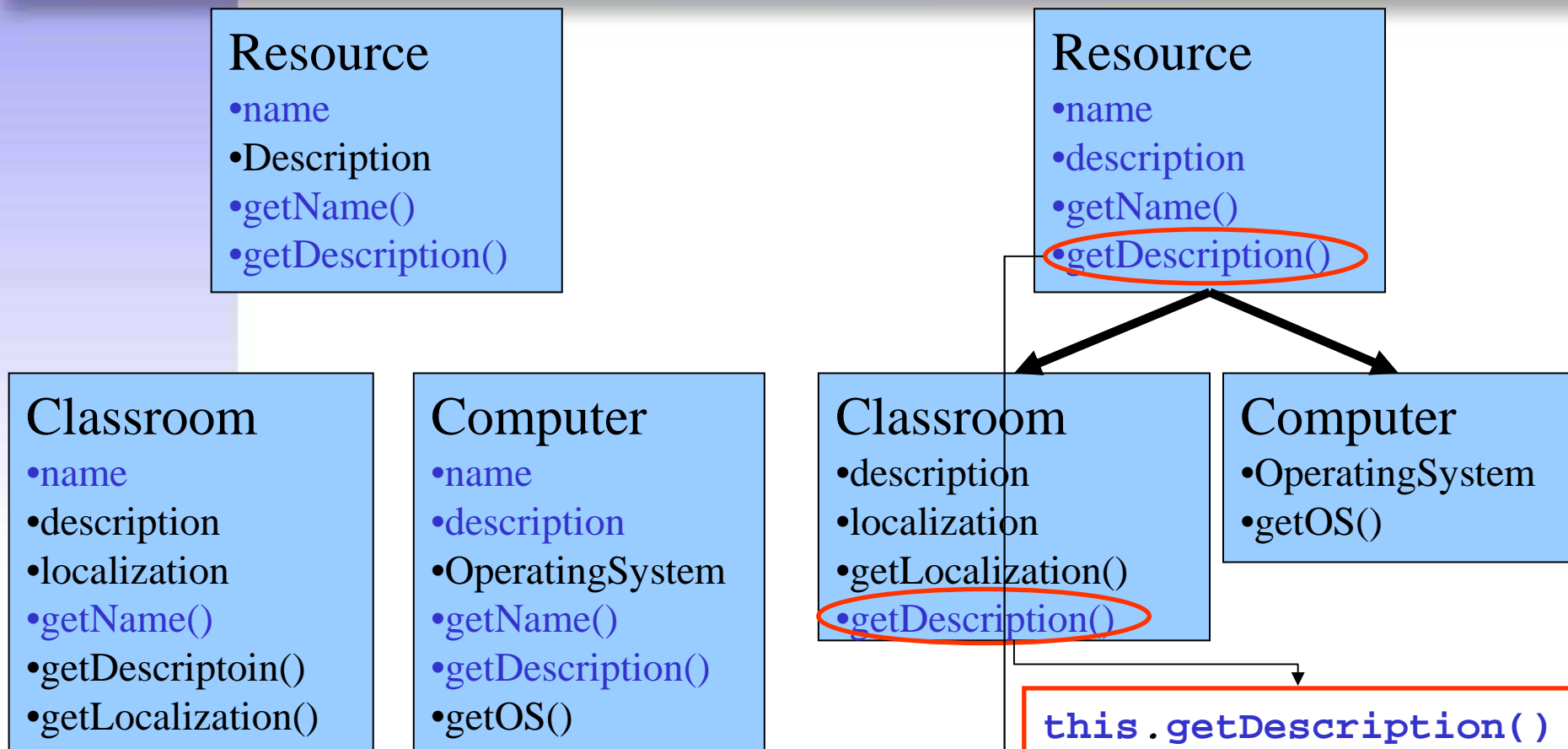
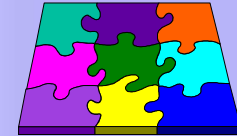
} Methods of the child class: visible  
} Methods of the parent class: hidden





# Overriding II

## Method overriding. What is it?



```
public class Classroom extends Resource
public class Computer extends Resource
```

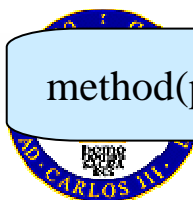
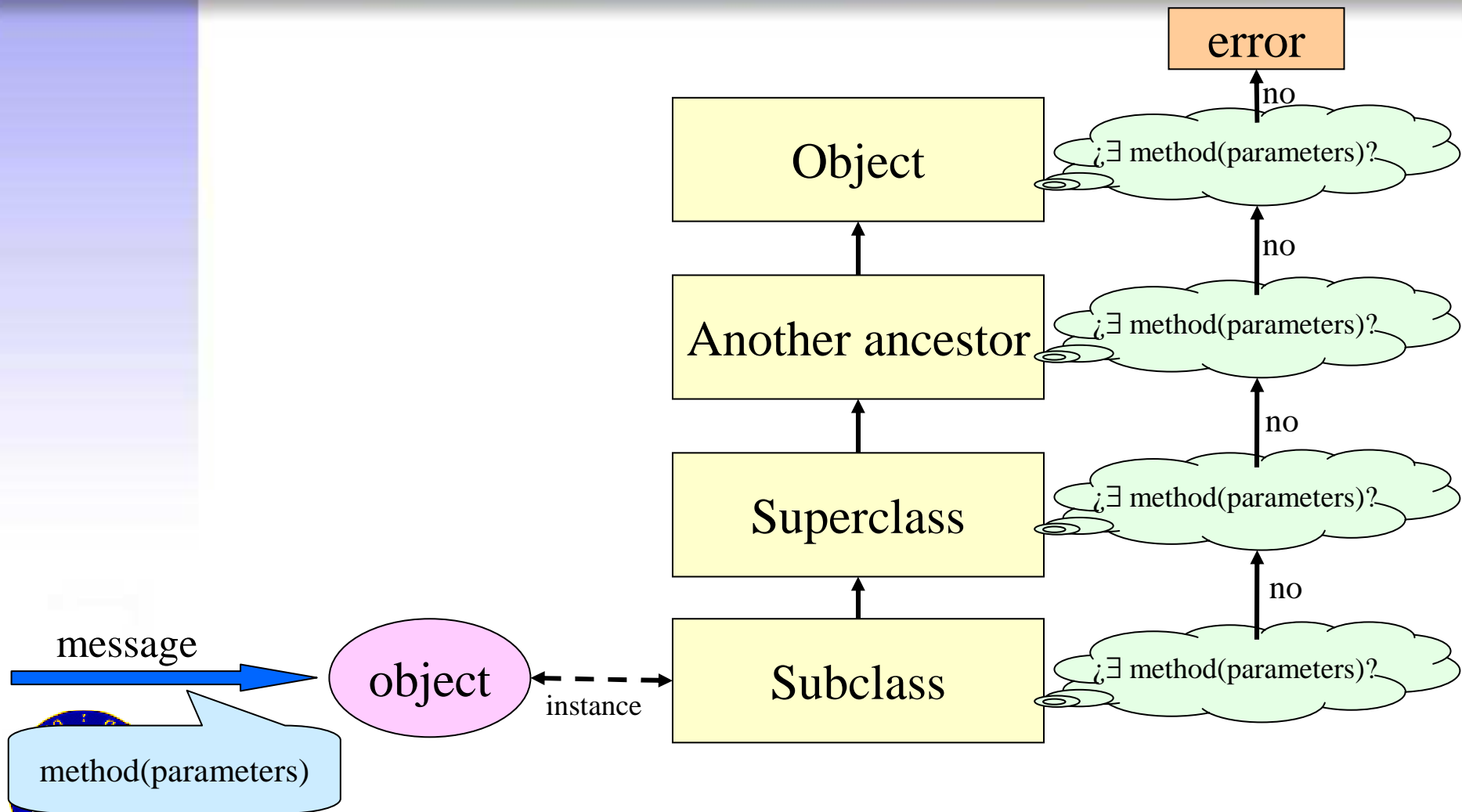
```
this.getDescription()
```

```
super.getDescripcion
```



# Overriding II

## Method overriding.





# Overriding II

## Method overriding.



- When sending a message to an object, the selected method:
  - It depends on the class which the object is an instance of
  - It does not depend on the reference class to which it is assigned, as in the case of attributes







# Overriding II

## Method overriding. Example



```
class SuperShow {
    public String str = "SuperStr";
    public void show() {
        System.out.println("Super.show: " + str);
    }
}
class ExtendShow extends SuperShow {
    public String str = "ExtendStr";
    public void show() {
        System.out.println("Extend.show: " + str);
    }
}
class Show2 {
    public static void main (String[] args) {
        ExtendShow ext = new ExtendShow();
        SuperShow sup = ext;
        sup.show();
        ext.show();
    }
}
```

Both print:  
"Extend.show: ExtendStr"





# Overriding II

## Final methods



- Method overriding is useful for
  - Extending the functionality of a method
  - Particularizing the functionality of a method to the derived class
- If it is not desired that subclasses are able to modify a method or an attribute of the base class, the reserved word **final** should be applied to the method or attribute





# this and super references



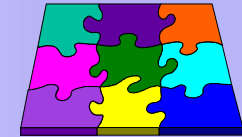
- **this** references to the current class object
- **super**
  - references the current object casted as if it was an instance of its superclass
  - With the **super** reference, the methods of the base class can be explicitly accessed
  - **super** is useful when overriding methods

```
public class Student extends Person {
    // the rest remains the same
    public void print(){
        super.print();
        System.out.print("Group:" + group+ schedule);
    }
}
```





# Overriding vs. overloading



- **Overriding:** The subclass substitutes the implementation of a method of the superclass
  - Both methods need to have the same signature
- **Overloading:** There is more than one method with the same name but different signature
  - The overloaded methods can be declared in the same class or in different classes in the inheritance hierarchy





# Constructors and inheritance

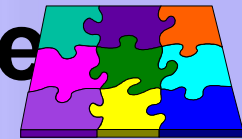


- To create an object, the following steps are done:
  1. The base part is created
  2. The derived part is added
    - If the base class of the object inherits from another class, step 1 is applied in the order of the inheritance chain, until we reach **Object**
- For example, when creating a **Student** object, that extends **Person**, the steps would be:
  1. The part corresponding to **Person** is created. To do so:
    1. The part corresponding to **Object** is created.
    2. The **Person** elements are added
  2. The **Student** elements are added





# Constructors and inheritance



- A call to the constructor of the base class is always done in the constructor of the derived class.
- This is the first action of the constructor (always in the first line)
- Two possibilities:
  - Not explicitly indicate it
  - Explicitly indicate it (mandatory in the first line)





# Constructors and inheritance



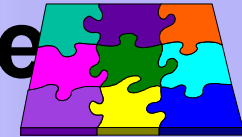
1. If it is not explicit, Java automatically inserts a call to `super()` in the first line of the constructor of the derived class.

```
public Student (String firstName, String lastName,
    int yearBirth, String group, char timetable) {
    // Java inserts here a call to super()
    this.firstName = firstName;
    this.lastName = lastName;
    this.yearBirth = yearBirth;
    this.group = group;
    this.timetable = timetable;
}
```





# Constructors and inheritance



## 2. Explicitly coded

```
public Student (String firstName, String lastName,  
                int birthYear, String group, char  
                timetable) {  
    super(firstName, lastName, birthYear);  
    this.group = group;  
    this.timetable = timetable;  
}
```







# Modifiers and access

## Final



- The **final** modifier can be applied to:
  - **Parameters:** Means that the value of such parameter cannot be changed inside the method

```
public void myMethod(final int p1, int p2){} //p1 value cannot be changed
```

- **Methods:** Means that it cannot be overridden in derived classes

```
public final void myMethod(){ } //myMethod cannot be overridden
```

- **Classes:** Avoid extending the class. It cannot be inherited.

```
public final class myClass(){ } //myClass cannot be extended
```





# Modifiers and access

## Static (static members)



- *static* modifier
- Static members exist only *once per class*, independently of the number of instances (objects) of the class that have been created or even if none (instances) exists.
- Static members can be accessed using the *class name*.
- An static method *cannot* access non-static members directly, it must first create an object.





# Modifiers and access

## Static. Some rules



- Static members are invoked with:

```
ClassName.staticMethod();  
ClassName.staticAttribute;
```

- Non static members require an instance (object) in order to be accessed.

```
ClassName objectName = new ClassName();
```

- Non static members are invoked with:

```
objectName.normalMethod();  
objectName.normalAttribute;
```



- When a static member is invoked (called) from inside the same class , the class name can be deleted. I.e. it can be written:

```
staticMethod();  
staticAttribute;
```

instead  
of:

```
ClassName.staticMethod();  
ClassName.staticAttribute;
```



MODIFIERS		<i>class</i>	<i>method</i>	<i>attribute</i>
access	public	Accessible to any other class		
	(friendly)	Accessible only to classes in the same package		
	protected		Accessible to the class and its subclasses	
	private	Applied to inner classes	Accessible only inside the class	
others	abstract	Cannot be instantiated For <b>inheriting</b> from them At least 1 abstract method	Has no code It is implemented in the subclasses or child classes	
	final	Cannot be extended. It is a leave in the inheritance tree.	Cannot be overridden. It is <b>constant</b> and cannot be modified in the child classes.	Its value cannot be changed, it is <b>constant</b> . It is normally used together with static.
	static	Maximum level class. Applied to inner classes	It is the same for all of the class objects. Use: ClassName.method ();	It is the same for all of the class objects.

