

Introduction to OO Programming

With the help of

- the JAVA language
- Object Oriented DESIGN

may the force be with us

Session 2

Objects



OVERVIEW of Session 2

- OOP Principles
- Classes, objects & constructors
- Encapsulation - Access
- Class fields & methods
- Wrapper classes

After, you'll be able to:

- Describe **class** and **instance** concepts
- Create **objects** (instances of classes)
- Differentiate variables and **fields**
- Know how to use **accessors**
- Make the best choice between “public” and “private”
- (Use geek words such as “static”)

1 – OOP Principles

(PARTIALLY BASED ON P.BELLOT – TELECOM PARISTECH)

Class and Objects

BASE CLASS : SUPERHERO



Fields	Methods
<ul style="list-style-type: none">sexheroNamerealNamecitypowerssuperpower	<ul style="list-style-type: none">getNamepowerUpattack...

AN OBJECT : BATMAN



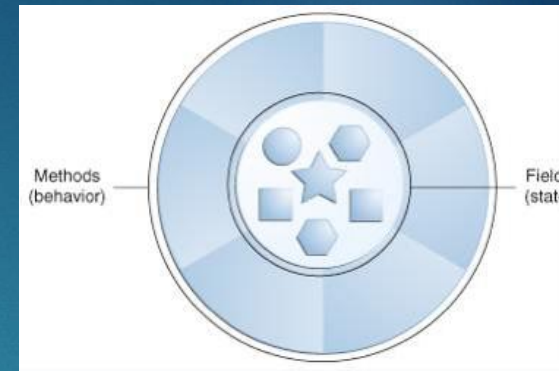
Fields values	Methods
<ul style="list-style-type: none">sex: "male"heroName: "Batman"realName: "B.Wayne"city: "Gotham"powers: {"fly", "boxing"}superpower: {}	<ul style="list-style-type: none">getNamepowerUpattack...

OO programming was created in 1967 – Oslo University

- The problem was : how to simulate a group of robots in a company.
- Trying to solve this problem with imperative programming could give a serious headache to the programmer:
 - Try to imagine a centralized program whose aim is to pilot all robots at the same time, with a simple data structure representing their environment. Not impossible, but very tricky !



THE SOLUTION : OOP !



Problem

In OOP, we describe each part of the problem in a "**class**". It concerns Robots but Control processing as well

A « **Class** » is a template (« shape ») to create « **objects** »

Fields

The class describes the data contained in all such objects

E.g., a robot **has**: ...

These are the "**fields**" of the objects (their data)

Current fields values taken together represent its "state"

Messages

Important difference between imperative programming and OOP:

"Objects can send messages to other objects"

A message received by an object gives an answer after internal processing (exec. "behavior"). These are « **methods** ». E.g.: a robot **can**: ...

Message example:

- A human sends a message to a robot to order it to start or stop
- A robot sends a message to control that its energy level is low

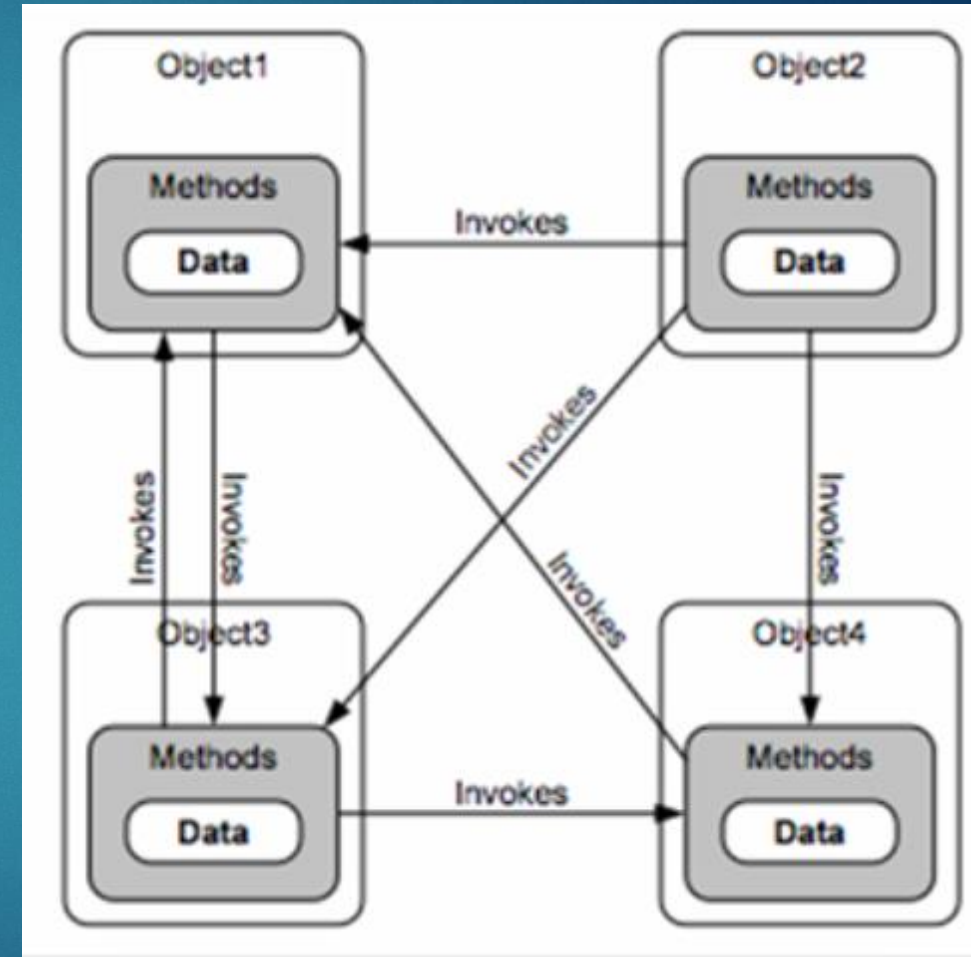
...

A class describes which **kind of messages** it can manage and how it **answers** these messages. These are the "methods" of the object.

E.g., to the start method, a robot may respond "Done" if startup succeeded or "Failed" if not.

Viewed from the programmer

- Describe each message that factory objects can receive
- Create as many variables (instances) of these classes: one control, X robots etc...
- Objects interact by sending messages between each others
- This can happen without a "central program" managing these interactions.
- To change an object state, one simply calls ("invokes") a method on this object.




source: beginnersbook.com/2013/04/oops-concepts/

Another example: Web interface


- Everything seen on the browser's window (buttons, text inputs, etc...) is part of the interface
- Users may interact with the interface via keyboard or mouse, which triggers messages (i.e., method invocations).
- As with robots, one can imagine a data structure to describe a web interface (**DOM : Document Object Model**)
 - It would be tricky to do this with a centralized program !
 - **Why?**
- In OOP, each element on the interface is an object, it manages its own data
- Each object has some common fields:
 - Position
 - Size / shape
 - Depth (which one is in front)
- Each object has specific methods
- ..that depend on its class (kind): button, text input, ...
- Objects may send messages between them upon user action




Subscription plan

**Advanced Plan**
10 keywords, 1 project
\$59.00 / month
Active keywords: 9
[Upgrade plan](#) [Cancel subscription](#)


Payment method



Credit card
**** * 3487
John C. Wayne
[Change payment method](#)

**Basic Plan**
Good for start
3 keywords
1 project
\$49.00 / month
[Choose Basic](#)

- ✓ Sentiment analysis & score
- ✓ Highlights reporting
- ✓ Daily Google rank update
- ✓ Weekly sentiment
- ✓ Data history

**Advanced Plan** MOST POPULAR
More keywords means more data
10 keywords
1 project
\$59.00 / month
[Choose Advanced](#)

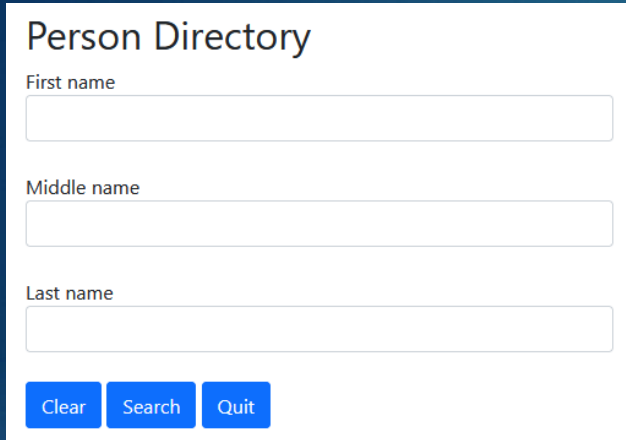
- ✓ Sentiment analysis & score
- ✓ Highlights reporting
- ✓ Daily Google rank update
- ✓ Weekly sentiment
- ✓ Data history

A subscription Web interface with different views

Object architecture

- To give an object the ability to send messages to another, it must "know" its existence.
- Most of the time, it has a "**reference**" (automatically managed pointer, address) to the other object. But in Java the syntax is handled smoothly (**no evil *, & or ->**)

Our user interface



Person Directory

First name

Middle name

Last name

Main interface contains/has 3 text fields,

⇒ Interface class code must have 3 fields referencing these inputs

Main window creates results windows, then on closing, it will close "child" windows as well.

⇒ Window class must have a field listing all references to child windows

```
public class MainWindow {  
    String firstName;  
    String middleName;  
    String lastName;  
  
    Button clearB, searchB, quitB;  
  
    ...  
  
    ArrayList resultWindows;  
  
    ...  
}
```

**This is not how real Web interfaces work
(a fictive example)**

Our GUI

A child interface may be closed with button "Close". Upon closing, child interface should warn Main interface that it has been closed

⇒ Each child interface must have a reference to "parent" interface

Each button triggers actions in interface

Person Directory

First name

Middle name

Last name

Main Interface

Result

First name: Elon
Middle name: ---
Last name: Musk

Site: Tesla Headquarters
Address: 1 Tesla Road Austin, TX 78725
Email address: ceo@tesla.com



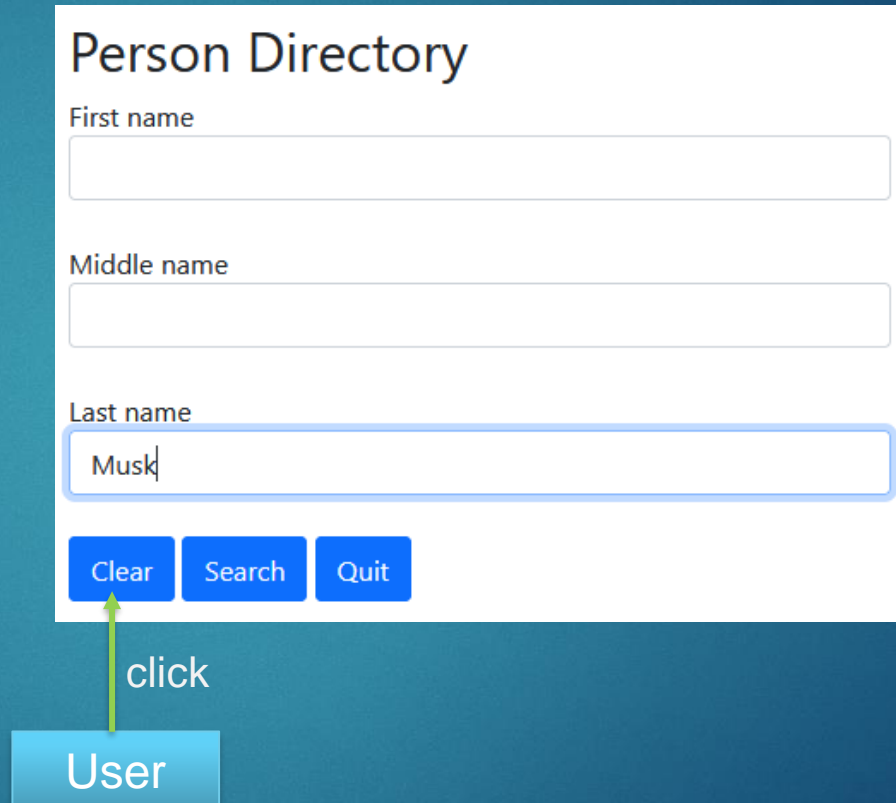
Can you draw all links between components (objects) issued from this user interface?



what does this mean?

Cascading messages

- What happens when the user clicks on the "clear" button ?
- When the user clicks, button receives the "click()" message.
- Click method on clear button simply sends "clearFields()" message to its Interface.



Can you draw all messages and answers exchanged in this context?

Please keep it simple !

- Dev. must create **methods** to setup each receivable message
- These **methods** should generally be very simple, "The simpler the better"
 - If your method is more than 20 lines, then it should be two or more methods.
- Example with search form "click()" method of the clear button:
 - 1) Redraw button as "pressed()" (another method on button class)
 - 2) send message "clearFields()" to its Interface (Form or Bloc)
 - 3) Wait for "ok" response
 - 4) Redraw as initial (unpressed())

Please keep it simple (simpler, simplest)

Non-intrusive principle

- Avoid working on an object state from outside this object...
 - Clear button doesn't clear by its own, it 'asks' Interface to do it
- **Advantage?** The day a new field is added in Interface, you just have to modify the clearFields from Interface class...

Delegation principle

- Some of your work can be done by someone else!
- **Advantage?** Your code is encapsulated and localized in a small number of classes

OOP recap !



- **Objects** are 'entities' communicating thanks to
- **Objects** contain values saved in Among them, one can find other object r..... that allow to send to these objects
- Each an object can receive is implemented within a which is a function (in structured/imperative programming) that executes in the context of the object.

OOP is a philosophy

- Software system =

Data

+

Processing

Data centered programming: data is encapsulated in objects, processings are distributed across methods in objects, according to data on which they apply.

OOP

Always ask : what data is used in my software, before asking what your software actually does (Data before actions)

Processing centered programming: data is stored in data structures accessible by processes; processes are composed of functions

Imperative programming

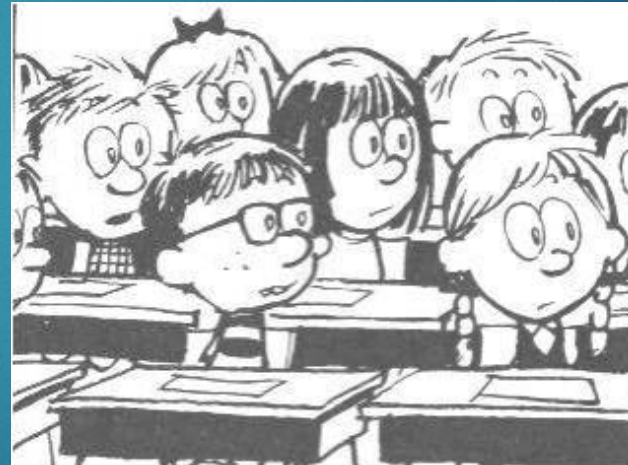


A small quiz now !



- ⇒ An object contains both data and processing (treatments)
- ⇒ An object doesn't need to know other object methods it uses
- ⇒ An object holds references to other objects
- ⇒ An object doesn't need to know about referenced objects fields
- ⇒ Messages represent an interface which allows the outside to communicate with an object
- ⇒ A class is a variable
- ⇒ An instance is a variable
- ⇒ An object is a variable
- ⇒ Part of the state of an object is stored in a (global) variable (named field)

2 – Classes, objects & constructors



Java syntax

```
public class SuperHero {  
    String realName = "Unknown";  
    String heroName;  
    String gender = "m"; // default value  
    String city;  
    String[] superPowers;  
    SuperHero[] enemies;  
  
    public String toString() {...}  
    int addFriend(SuperHero friend) {...}  
    ...  
}
```

Constraints :

- One File = One public class
- File name is the ClassName : SuperHero.java "same case"

```
// In another class, declare a variable which will contain  
// a reference to an object  
  
SuperHero sp; // The class name is used to type a variable
```


"Constructors"

- A constructor is used to initialize a **new** object/instance. It is responsible to initialize fields (state) of a new object
- A constructor is a simple method having the same name as the class
- If no constructor is defined in a class, the java compiler creates a default empty constructor without any parameter

```
/* Declare AND INIT a class object instance */  
SuperHero batman = new SuperHero();
```

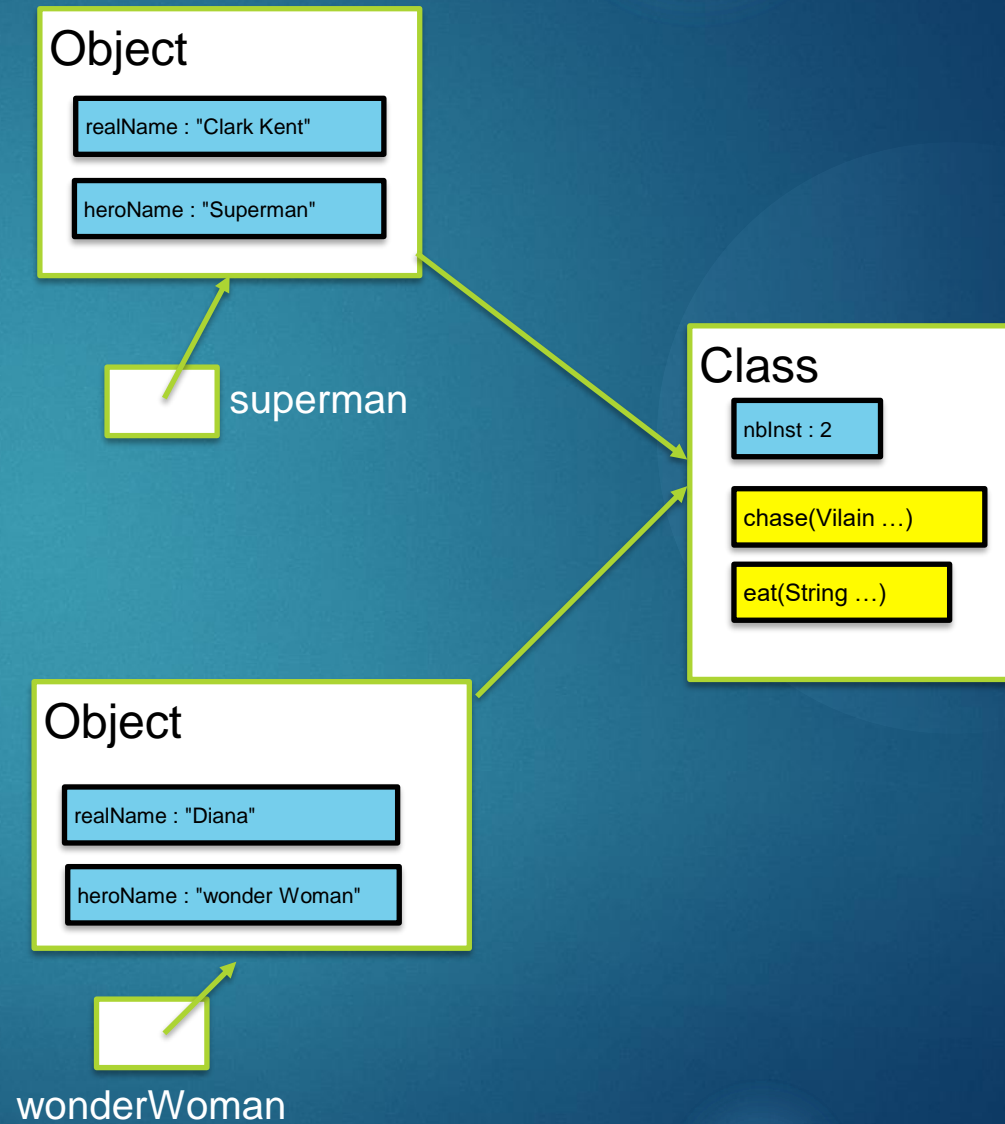
Constructors - 2

- A class may have more than one constructor (they must not have the same parameters – param names are ignored) : « constructor overloading »
- No return type as it's implicit ! (obvious?)
- **this** keyword: reference to the current (under construction) object
- Once a constructor is declared, objects **MUST** be initialized by explicitly calling a constructor (**new** keyword)

```
class SuperHero {  
    public SuperHero(String heroName){  
        this.heroName = heroName;  
    }  
    public SuperHero(String heroName, String realName){  
        this.realName = realName;  
        this.heroName = heroName;  
    }  
  
    public static void main(String[] args){  
        SuperHero batman = new SuperHero("Gotham");  
        SuperHero spiderman = new SuperHero("Spiderman", "P.Parker");  
    }  
}
```

Accessing fields and methods

- Objects of a class all have the same structure (**fields**)
- Objects are dynamically allocated in memory, each containing its proper values, saved in **fields**)
- No pointer needs to be manipulated by the dev: wonderWoman and superman are variables containing **references** to **objects**
- In contrast to pointers, with references de-referencing (& in C) is automatic
- **Fields** of an object can be manipulated by **methods** declared in the class (and stored in the class memory space)



Accessing fields and methods

- Syntax :
`objectName.field`
and
`objectName.method(params)`
- Inside a given class, though **it's not recommended**, a method can reference the current object's `fields` without `this` keyword



Where does this happen in this code?

```
class SuperHero {  
  
    ...  
  
    public SuperHero(String heroName, String realName){  
        this.realName = realName;  
        this.heroName = heroName;  
    }  
  
    public String toString(){  
        return "my name is "+heroName+" (" +realName+")";  
    }  
  
    ...  
  
    public static void main(String[] args){  
        SuperHero superman = new SuperHero("Superman", "Clark Kent");  
        SuperHero wonderwoman = new SuperHero("Wonder Woman", "Diana");  
        Vilain vilain = new Vilain("Lex Luthor");  
        supermman.chase(vilain);  
        System.out.println(wonderwoman);  
    }  
}
```

Lab part 1

Dates & Timespans



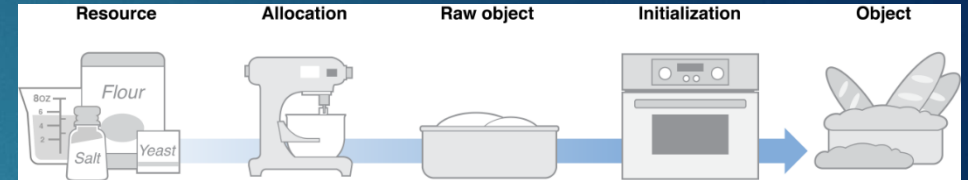
Consider **MyDate** class (with **day**, **month** and **year** int fields) having the following documentation (first, read it carefully!!!):



```
public MyDate() // initializes with the current date (of today)
public MyDate(int day, int month, int year)
//initializes with the date day/month/year without checking parameters
public int getDay() // returns the day of the month
public int getMonth() // returns the month (values between 1 and 12)
public int getYear() // returns the year
public boolean equals(MyDate other) // returns true if the two dates are equal, else it returns false
public int compare(MyDate other)
// returns 0 if the dates are equal
// a strictly positive integer if this precedes strictly other
// a strictly negative integer if other precedes strictly this
public MaDate smallest(MyDate other)
// returns the one of the two dates which precedes the other. If the two dates are equal, it returns this
public String toString()
// returns a String describing the date as "day/month/year" where day, month and year are numbers
public String dateInLetters()
// returns a String describing the date as "day month year" where month is in letters (January, ...)
```

Lab part 1

Dates & Timespans



Question 1. Basic object creation and message passing (method invocation)

Write Java statements for:

1. Creating a date d1 initialized with the current day
2. Creating a date d2 initialized with the date 20/07/1969 (Which event???)
3. Printing the smallest of the two dates or one of the two if they are equal. Do it in two different ways, using the methods **compare()** and **smallest()**

Question 2. *Back to arrays...*

Let's consider N as a constant of type int (<30). Write the statements for:

1. Creating an array of N ints, initialized with value -1
2. Creating an array of N instances of MyDate, each initialized with the current date
3. Creating an array of N instances of MyDate, the instance at index i is initialized with the date "i/04/2069"

Lab part 1

Dates & Timespans



Question 3.

Implement the class *MyDate* whose documentation was given before

Indication.

For getting the day, the month and the year of the current day :

```
java.util.Calendar now = java.util.Calendar.getInstance();  
int day = now.get(java.util.Calendar.DAY_OF_MONTH);  
int month = now.get(java.util.Calendar.MONTH) + 1;  
int year = now.get(java.util.Calendar.YEAR);
```

There are other possible ways:

<https://mkyong.com/java/java-how-to-get-current-date-time-date-and-calender/>

Lab part 1

Dates & Timespans



Question 4.

Write a class **Period** with the following specification:

- A period is a range of two dates (objects of *MyDate*): *startingDate* and *endingDate*
- This class should provide two constructors:
 1. With 2 parameters: starting and ending dates of the period. These dates can be received in any order (you should identify which date precedes the other)
 2. With only one date as a parameter. The period encompasses the date of the current day and the date received as a parameter (here again, you should compare the two dates)
- 🕒 Class *Period* should provide the following methods:
 - A method **equals(...)**: returns true if the two periods are identical, else returns false
 - A method **intersectsWith(...)**: returns true if the two periods overlap, else it returns false
 - A method **isBefore(...)**: returns true if the current period precedes the one received as a parameter, else returns false
 - A **toString()** method: returns a String with format "15/04/1970 - 18/04/1971"

Create several objects of this class and invoke the different methods above

3 - Encapsulation - Access

No headache !

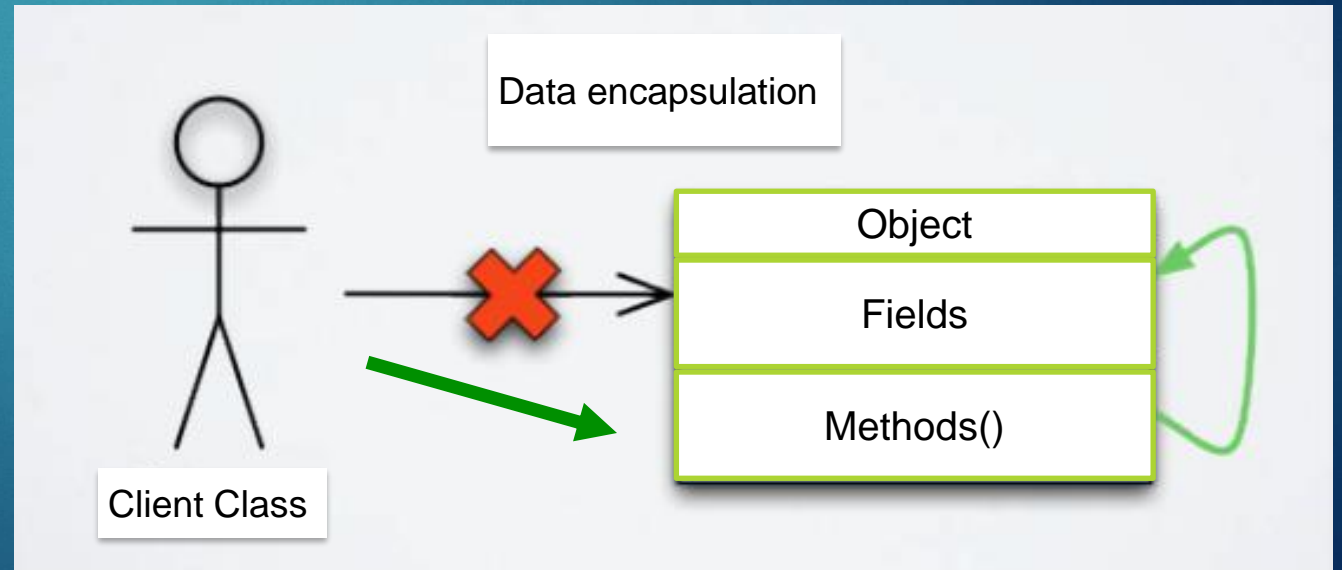


ENCAPSULATION

This is **THE fundamental principle** in OOP!

It consist in **hiding** internal fields to other classes by adding "**accessors**" methods


Instead of directly accessing the class fields, we should invoke special "accessor" methods on the objects



ENCAPSULATION WITH JAVA

- **Consistency**: protect your class fields, ensure data integrity, no external class can lead to an inconsistent state.
- **Abstraction**: using classes doesn't really need to know how these classes are implemented (their internal fields, their type, ...)
- **Maintenance**: the day you want to change internal behavior of your class, assuming your accessors don't change, all client classes will still work as expected!
- **Keep it private**: 'private' keyword hide access to your class fields (in other classes)
- **Accessors**: for each field, you may declare a "getter" and a "setter", both "public" to access your data
- Accessors can check input values, and react if inconsistent data!

```
public class Species {  
    private String name;  
    private int population;  
    private double growthRate;  
    // ACCESSEURS pour population :  
    public void setPopulation(int newPopulation) {  
        if (newPopulation >= 0)  
            population = newPopulation;  
        else {  
            System.out.println("ERROR: using a negative population.");  
            System.exit(0);  
        }  
    }  
    public int getPopulation() {  
        return population;  
    }  
}
```



Lab(yrinth)

1. Design a **Cell** class that can tell whether a given side (North, South, East, West) of the cell is opened or closed. The cell can be freely modified so that one of its side is opened or closed. By default a cell is closed on all sides.
2. Design a **Labyrinth** class that contains cells on different rows and cols (but same number of each). Access to the lab cells must occur only through accessors
3. A labyrinth does not allow a direct access to its cells for consistency, but can accept orders to **open(x,y,dir)** a certain wall (including an edge of the labyrinth) of a given cell and requests to know whether a given wall **isOpened(x,y,dir)**

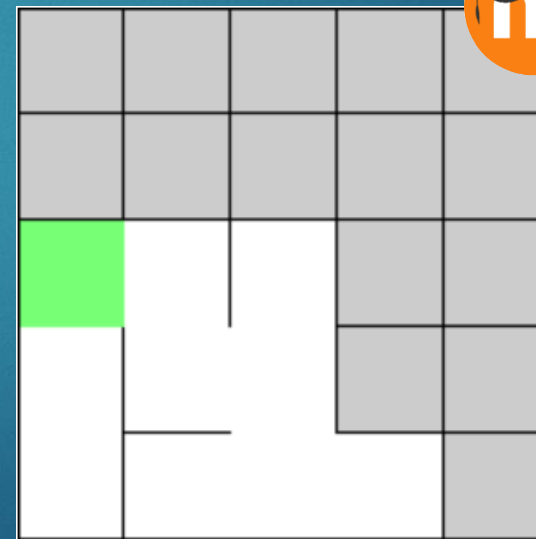


<https://codeboard.io/projects/229418>

Lab(yrinth)

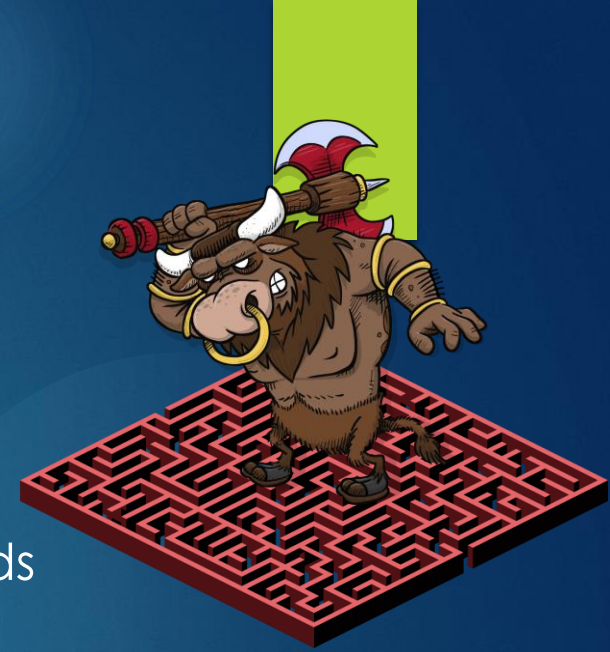
Next we look at labyrinth generation: we want to open walls so that there is a path between any two cells in the labyrinth

4. Implement a **Minotaur** class, where the beast has a position (x and y) and add one of its instances in the labyrinth class, then change the **toString()** method to show its current position
5. During the labyrinth construction, the minotaur must remove walls so that he can travel from any cell to any other cell. He proceeds as follows:
 - He starts at a random cell **c**
 - While some cells are not **visited**:
 - He chooses a neighboring cell **nc** at random
 - If **nc** has not yet been visited:
 - He breaks the wall between **c** and **nc**
 - **nc** is marked as visited
 - **nc** becomes the current cell

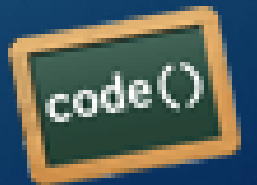


Lab(yrinth)

Next we look at Theseus trying to find the Minotaur in the labyrinth, while using the Ariane thread to keep him linked to the labyrinth entrance.



6. Implement a **Human** class that has **xH=0,yH=0** starting coordinates as fields and a **traverse(xCurrent,yCurrent)** method allowing a human to explore a labyrinth in the following way:
 - While not on the cell holding the minotaur do:
 - The current cell is marked as visited
 - While there is an accessible neighboring cell **c** do:
 - **traverse(xc,yc)**
 - return as not finding the Minotaur from here
 - The human found the Minotaur, return the cell where the beast was hiding
7. Test the method from the main file





5 – Class methods & fields

Sharing is the way of happiness

PURPOSE

Sharing fields and methods
between all class instances

Use the "**static**" keyword

- Each object has its own values for fields, these are "instance" variables
- Objects instances from same class don't share anything except the class def (method definitions)
- It could be useful to share other values
 - Ex: bicycle have only 2 wheels, whatever instance
 - So storing this value for each instance would result in memory waste, nope?

Another example:

store the number of instances

```
class SuperHero {  
    private static int numberOfHeroes = 0;  
    private String heroName;  
    private int heroId;  
}
```

Access:

SuperHero.numberOfHeroes

batman.numberOfHeroes

Another use case example

store the number of instances

```
class SuperHero {
    private static int numberOfHeroes = 0;
    private String heroName;
    private int id;

    public SuperHero (String heroName){
        this.heroName = heroName;
        // increment and store assigned ID
        this.id = ++ SuperHero.numberOfHeroes;
    }

    // return the object generated Id
    public int getId(){
        return this.id;
    }
}
```


Class Methods

A class method is run in the context of the class, not in the one of its instances (compilation error if you try to !)

Such a method can't therefore access any instance field (it knows no instance, no "this" inside)

```
class SuperHero {  
    private static int numberOfHeroes = 0;  
    private String heroName;  
    private int id;  
  
    public SuperHero (String heroName){  
        this.heroName = heroName;  
        // increment and store assigned ID  
        this.id = ++ SuperHero.numberOfHeroes;  
    }  
  
    // return the object generated Id  
    public int getId(){  
        return id;  
    }  
  
    public static int getNumberOfHeroes() {  
        return numberOfHeroes;  
    }  
}
```

Class Methods

A class method is run in the context of the class, not in the one of its instances (compilation error if you try to !)

Such a method can't therefore access any instance field (it knows no instance, no "this" inside)

You already know a static method, which is ... ??

```
class SuperHero {  
    private static int numberOfHeroes = 0;  
    private String heroName;  
    private int heroId;  
  
    public SuperHero (String heroName){  
        this.heroName = heroName;  
        // increment and store assigned ID  
        this.id = ++ SuperHero.numberOfHeroes;  
    }  
  
    // return the object generated Id  
    public int getID(){  
        return id;  
    }  
  
    public static int getNumberOfHeroes() {  
        return numberOfHeroes;  
    }  
}
```

DID YOU GET IT ?

Why this statement is impossible in main ?

```
String description = this.toString();
```

```
public String toString() {  
    return "Voiture de "+proprietaire+" possède "+nbRoues+" roues";  
}  
public static void main(String [] args) {  
    //Voiture v;  
    Voiture v = new Voiture("Antoine");  
    Voiture voitureEtrange = new Voiture("Mark",5);  
    System.out.println(v.toString());  
    System.out.println(voitureEtrange);  
    System.out.println("Bizarre : "+voitureEtrange.nbRoues+ "roues");  
}
```


Another quiz !



- ⇒ Methods can exist outside any particular class
- ⇒ Methods can exist without being attached to any particular instance
- ⇒ A static variable is only stored in one place in memory

6 – Wrappers classes

To be or not to bean object



Used to wrap primitive types

Why ?

- With Java, some processing needs to work on objects (instances) and doesn't work with primitive types
- Package `java.lang` gives one wrapper for each primitive type (see course 1)
- Keep in mind that instances from these classes are "immutable" (their state cannot be changed) such as `String` objects.

For What ?

- To manipulate, convert class instance to primitive, and to `String` class, providing mostly `static` methods
- Ex: `Integer.parseInt(String s)`
- They provide constants such as **`MAX_VALUE`, `MIN_VALUE`**

Convert Integers...

String → int (Integer class)

```
static int parseInt (String ch,[int base])
```

Int → String (String class)

```
static String valueOf (int i)
```

Easier to concatenate an int with a
"". (ex: "" + 3 → "3")

Class Integer

Int → Integer

```
new Integer(int i)
```

Integer → int:

```
int intValue()
```

String → Integer:

```
static Integer valueOf(String ch [,int base])
```

Integer → String:

```
String toString()
```

Boxing/unboxing

java automated wrapping

- "Autoboxing" automates translation between primitive types and their wrapper class
- Inverse operation is called "unboxing"
- Example - ArrayList only accepts Objects in list:

```
ArrayList<Integer> list = new ...  
list.add(89); // boxing!  
int i = list.get(n); // unboxing
```

- Other examples:
- ```
Integer a = 89;
a++; // boxing
int i = a; // unboxing
```
- ```
Integer b = new Integer(1);  
b = b + 2; // unboxing + boxing
```
- ```
Double d = 56.9;
d = d / 56.9;
```
- Simple cast:  

```
Long l = (long) i;
```

# Caveats!

Conversion is still done, by compiler... performance loss if boxing / unboxing is frequently used.

Some curious behavior can occur with == comparison

- **Identity** for Integer
- **Equality** for int

```
public class Wrappers {
 public static void main(String [] args) {
 Integer a = new Integer(1);
 int b = 1;
 Integer c = new Integer(1);
 if (a == b)
 System.out.println("a = b");
 else System.out.println("a != b");
 if (b == c)
 System.out.println("b = c");
 else System.out.println("b != c");
 if (a == c)
 System.out.println("a = c");
 else System.out.println("a != c");
 }
}
```

What is the resulting output ?



# OUF – finished !

Your turn to work now !



How the customer explained it



How the project leader understood it



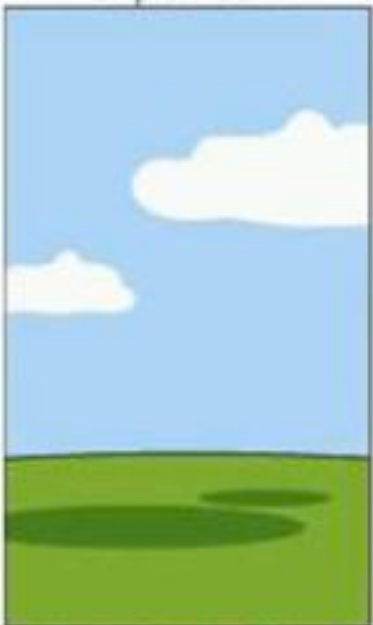
How the engineer designed it



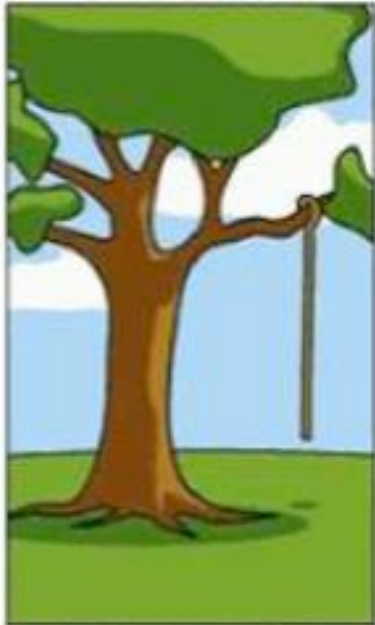
How the programmer wrote it



How the sales executive described it



How the project was documented



What operations installed



How the customer was billed



How the helpdesk supported it



What the customer really needed