

Programmation Orientée Objet : Introduction à la POO

Jean-Cédric Chappelier

Faculté I&C

Organisation du travail (semestre)

		MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	19.02.26		0	Intro + compil. séparée	
2	26.02.26	1. Intro POO	0	Intro POO	
3	05.03.26	2. Constructeurs/Des	0	Constructeurs	
4	12.03.26	3. Surcharge des opé	0	Surcharge	
5	19.03.26	4. Héritage	0	Héritage	
6	26.03.26	5. Polymorphisme	0	Polymorphisme 1	
7	02.04.26		1	Polymorphisme 2 / Collections hétérogènes	
-	09.04.26		-	vacances Pâques	
8	16.04.26	6. Héritage multiple	1	Héritage multiple	
9	23.04.26		-	-	Midtem
10	30.04.26	(7. Etude de cas)	-	Templates	
11	07.05.26		-	Structure de données abstraites ; Bibliothèques	
12	14.05.26		-	(Ascension)	
13	21.05.26	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions	
14	28.05.26		-	-	Examen

Objectifs de la leçon d'aujourd'hui

- ▶ Rappel : au sujet de l'IA (LLMs)
- ▶ Concepts fondamentaux
- ▶ Étude de cas

Bien apprendre à programmer / Rôle de l'IA

[Une partie du matériel de ces 4 slides est empruntée à mes collègues J. Sam, C. Salzmann et S. Doeraene]

Pour apprendre à programmer, il faut **pratiquer par soi-même**
(y compris le debugging)

Se confronter aux difficultés, y réfléchir, reformuler, etc.
sont nécessaires pour (bien) apprendre

- 👉 Lire une solution (IA ou autre), sans avoir au préalable **FAIT par soi-même**, ne sert à rien

« *On a l'impression de devenir bête.
On ne réfléchit plus, car on peut demander directement à une IA.* »

Utilisation de GenAI et apprentissage

Certes, les outils à base d'IA générative (ChatGPT, Github-Copilot, Claude, etc.) révolutionnent désormais l'approche au codage et sont déjà considérés, à juste titre, comme des supports indispensables

Mais ils ne sont pas faits pour l'apprentissage !

(cf <https://www.epfl.ch/education/teaching/index-html/ai-teaching/ai-in-student-learning/>)

Utiliser une calculatrice vous apprend-il les bases du calcul ?

Pouvez-vous comprendre pourquoi un calcul (sur calculette) est faux sans connaître les bases de l'arithmétique ?

Derrière l'écriture d'un programme, il y a des enjeux de **modélisation** !

Si l'on ne comprend pas ce qui caractérise une bonne modélisation, on ne saura pas interroger une IA de façon adéquate

Vous n'avez pas encore acquis toutes les bonnes méthodes de travail.
Utiliser des outils genAI pour **croire** de gagner du temps, vous ne développez ni votre efficacité, ni vos méthodes d'apprentissage et d'analyse pertinentes.

👉 On veut former des ingénieur(e)s, pas juste des codeuses/codeurs.

Pourquoi les IA disent des bêtises (surtout aux débutant(e)s)

Étudiant(e)s :

- ▶ question (« *prompt* ») mal posée :
mauvais ou manque de contexte, mauvaise compréhension de l'erreur
- ▶ manque de recul critique pour évaluer la pertinence de la réponse
👉 gardez un **esprit critique**

GenAI :

- ▶ mélange de différents langages de programmation
(en raison de fortes similitudes mais subtiles différences)
- ▶ ne compile pas et n'exécute pas le code, ne fait que le **prédire**
(prédit la suite la plus probable suivant ses données d'entraînement)
- ▶ génération d'erreurs par reformulation (p.ex. troncature de parties de code)

Conseils d'utilisation d'IA dans ce cours

Pour ce cours, du point de vue de l'enseignement, nous allons faire comme si les outils d'IA n'existaient pas

- 👉 **Le but n'est pas de vous apprendre à interroger ces outils, mais de vous enseigner ce qui constitue un bon programme, techniquement et méthodologiquement.**

De votre côté, l'usage de ces outils est toléré pour des tâches simples et comme support à la compréhension

- 👉 Il est indispensable de vous **re-approprier** le matériel ou les explications produites par ces outils en vous montrant capable de les **comprendre**, de les **re-expliquer** par vous même et d'y porter un **regard critique**.

Conseil : Utilisez les IA (Chatbots) pour chercher des réponses à vos questions mais **pas** pour produire du code ni pour déboguer votre code

- 👉 <https://www.epfl.ch/education/teaching/index-html/ai-teaching/ai-in-student-learning/>

Concepts fondamentaux

- ▶ Encapsulation :
regrouper données et traitements d'un même « concept »
- ▶ Abstraction :
se focaliser sur ce qui est caractéristique de ce « concept »
 - 👉 interface
 - cacher les détails

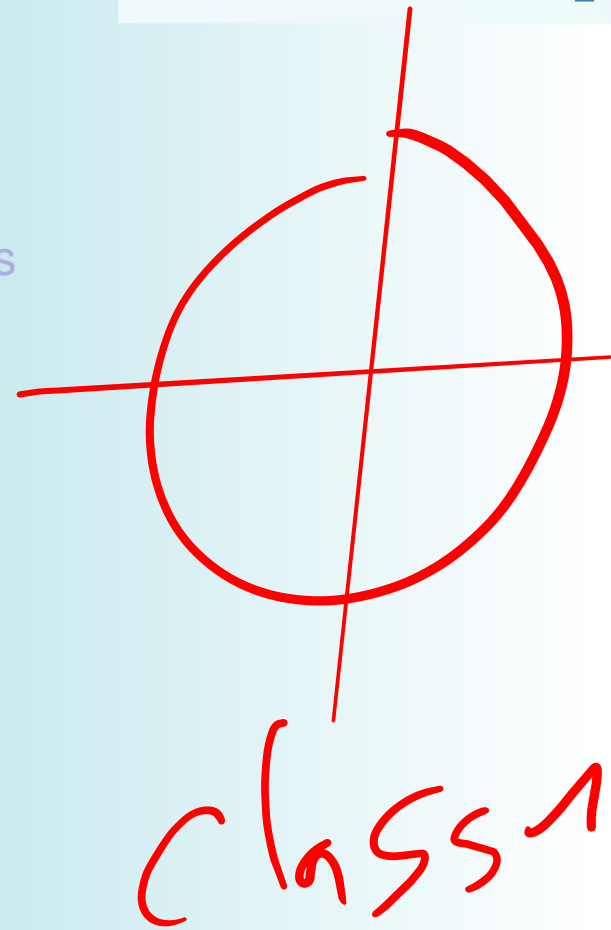
Encapsulation / Abstraction : Résumé

Objectifs

Rappel :
IA/LLMs

Concepts

Étude de cas



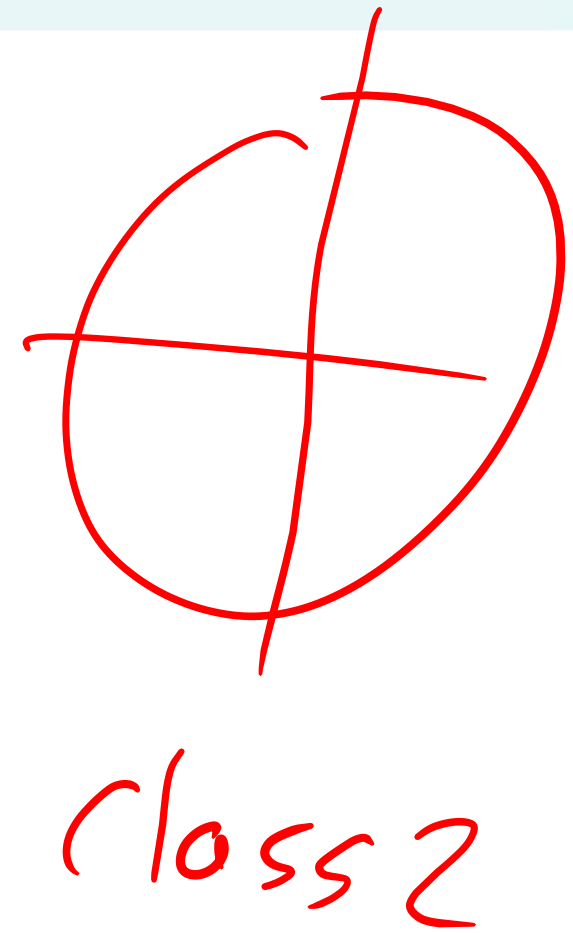
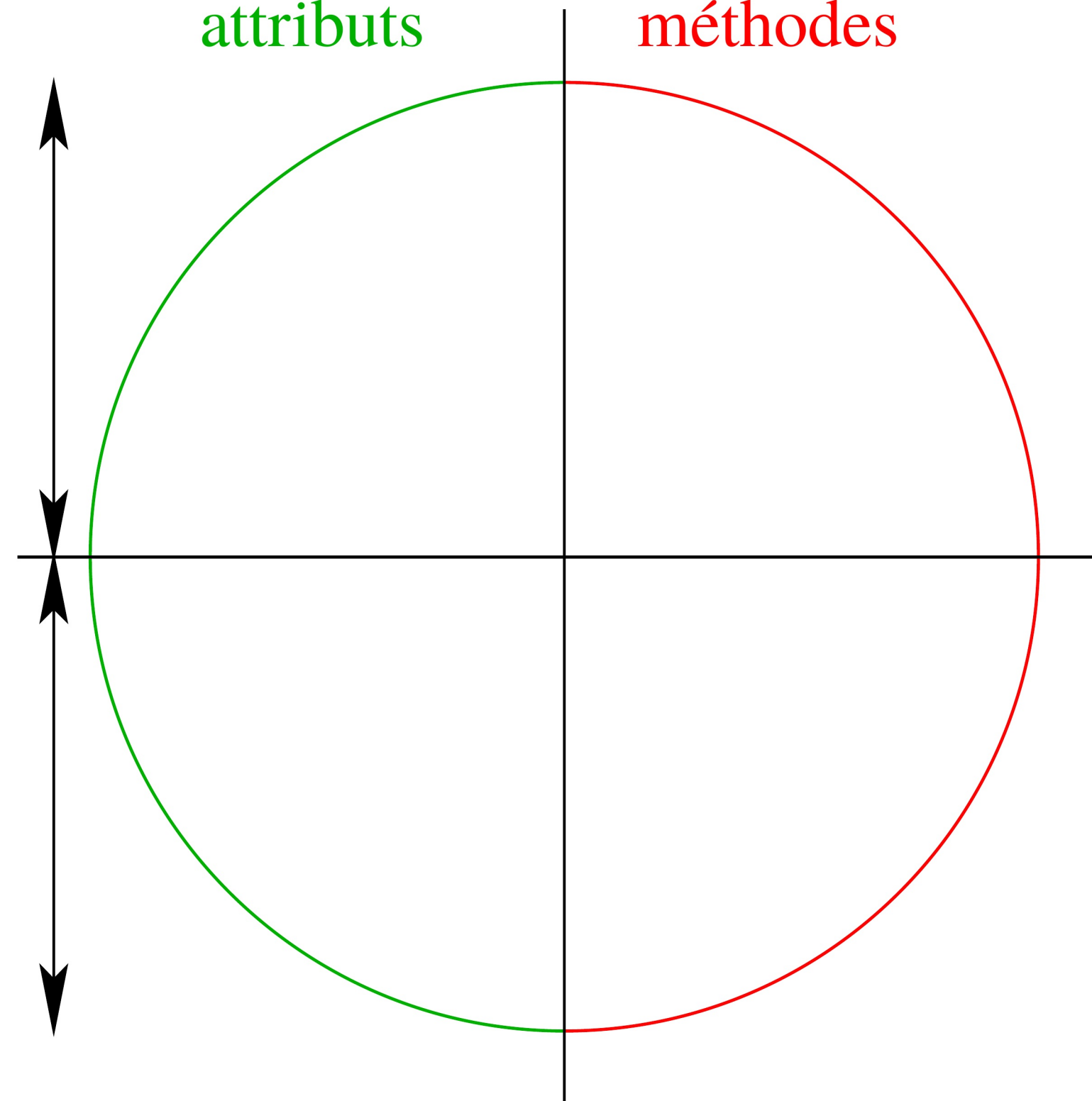
Interface
(partie visible)
public:

Détails d'
Implémentation
(partie interne/cachée)
private:

OBJET

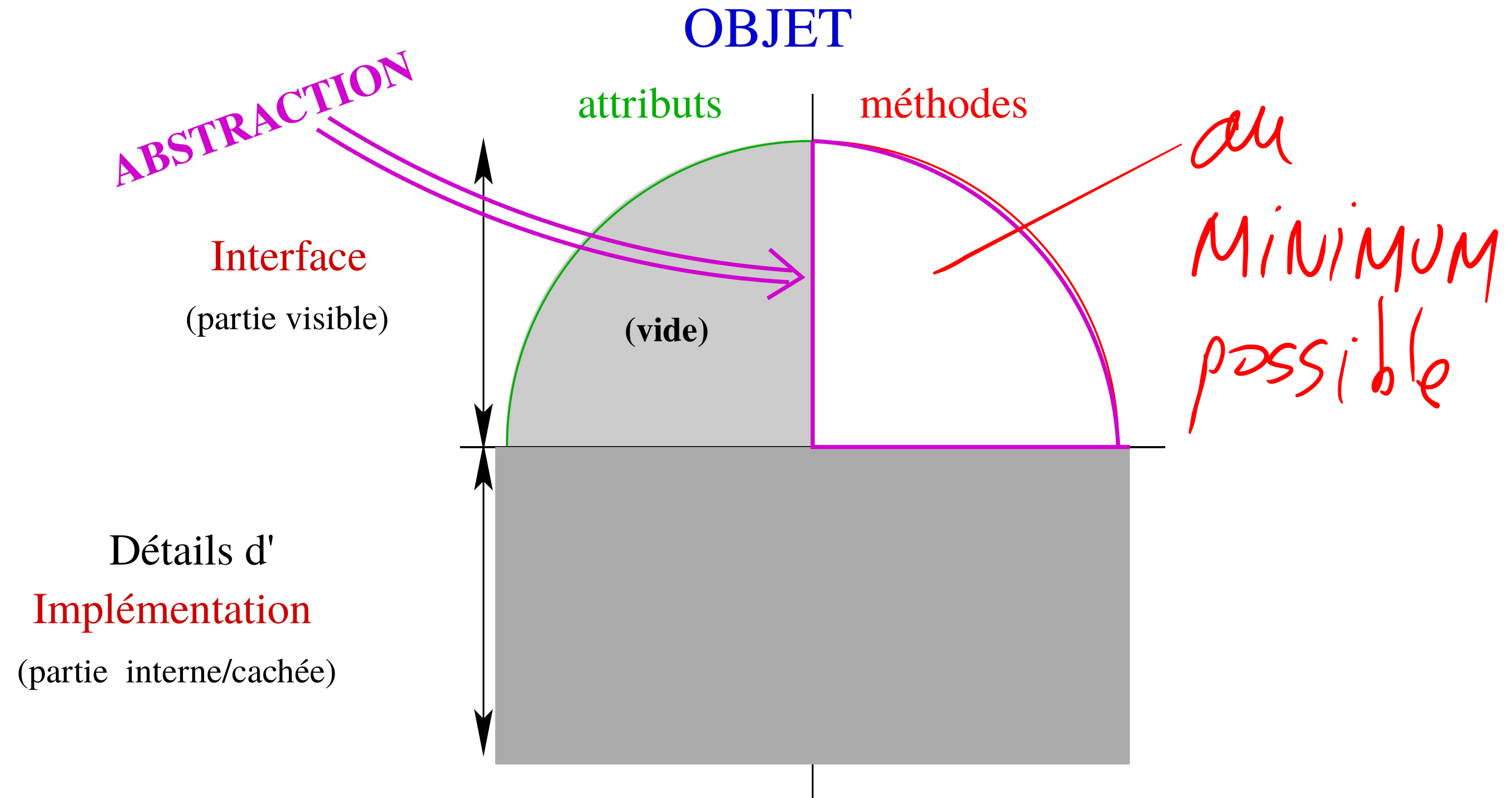
attributs

méthodes



Encapsulation / Abstraction : Résumé

MIEUX :



Encapsulation / Abstraction : vues en C++

```
class Concept {  
public:  
    methodes importantes;  
  
private:  
    attributs;  
  
    methodes secondaires;  
};  
  
Concept une_instance;
```


Étude de cas

Comment représenter des nombres complexes ?

① `struct complexe { double x; double y; };`

② `typedef array<double, 2> complexe;`

utilisation

① `complexe z;` ... `z.x` ... `z.y`

② `complexe z;` ... `z[0]` `z[1]`

③ `struct complexe { double rho; double theta; };`

Représenter des nombres complexes ?

Premières idées (non POO) :

1. `typedef array<double, 2> Complexe;`
2. `struct Complexe { double x; double y; };`

et l'on déclarerait par exemple : `Complexe z;`

Pour l'affecter, avec le premier on écrirait :

```
z[0] = 1.0; z[1] = 2.0;
```

et avec le second :

```
z.x = 1.0; z.y = 2.0;
```

👉 Laquelle vous semble la plus claire/la plus parlante ?

Représenter des nombres complexes ?

Mais pourquoi avoir défini les nombres complexes comme

```
struct Complexe { double x; double y; };
```

et non pas comme

```
struct Complexe { double rho; double theta; };
```

Qui « a raison » ? Laquelle est la meilleure ?

Représenter des nombres complexes ?

Mais pourquoi avoir défini les nombres complexes comme

```
struct Complexe { double x; double y; };
```

et non pas comme

```
struct Complexe { double rho; double theta; };
```

Qui « a raison » ? Laquelle est la meilleure ?

👉 Aucune, elles sont toutes les deux aussi **mauvaises** !

Découplage des codes

Elles sont les deux mauvaises car le code utilisateur
(par exemple `z.x = 1.0`)
est directement dépendant du choix d'implémentation :

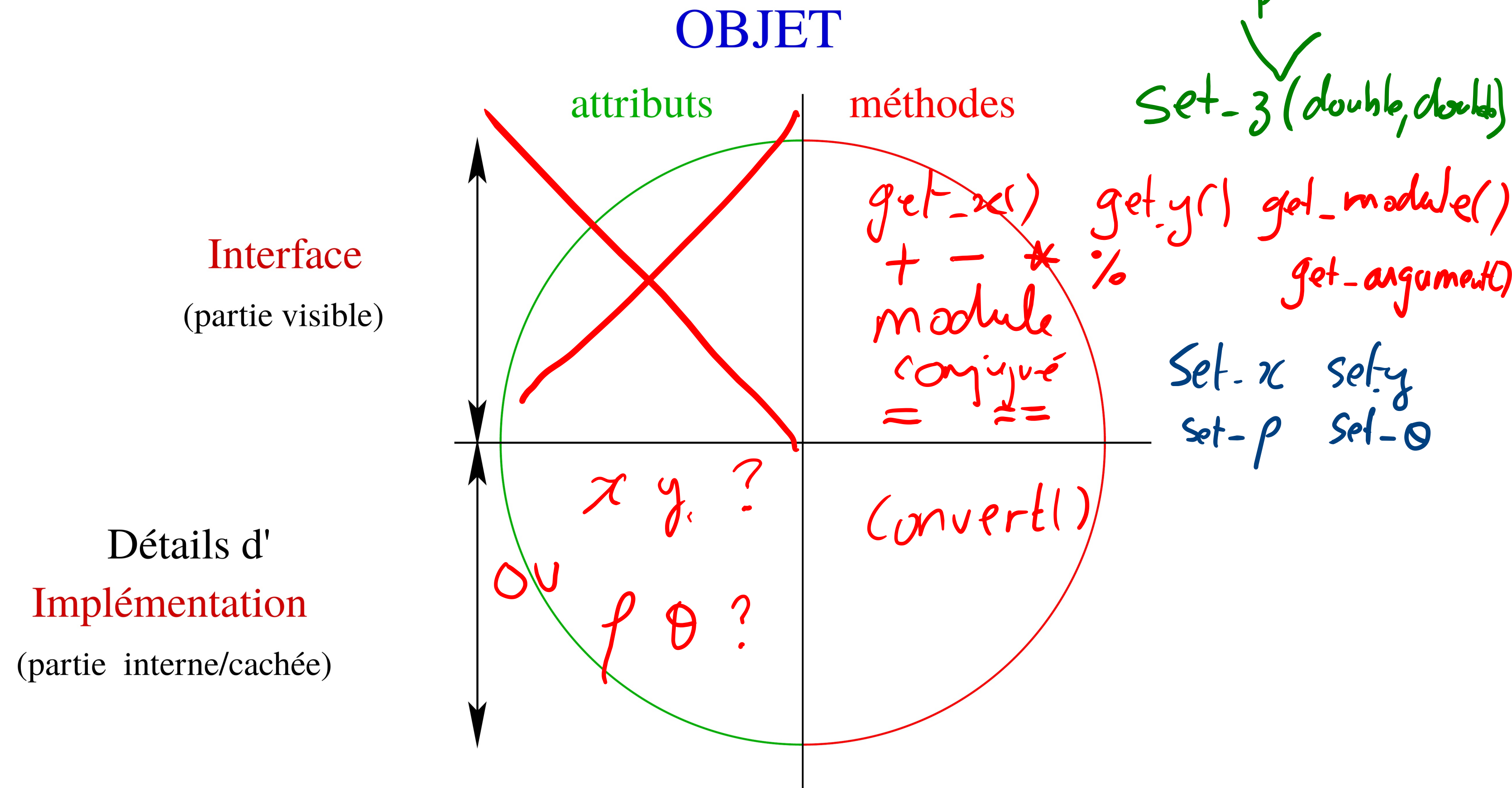
si l'on change la *représentation interne* des Complexes
on est obligé de changer **tout** le code qui l'utilise : – (

- 👉 le *couplage* entre le code « producteur » et le code « utilisateur »
est *trop fort*!

Découpler ces codes, réduire les dépendances est la raison profonde
des principes d'encapsulation et d'abstraction en POO.

Représentation POO des nombres complexes

La première question à se poser est :
quelles sont les caractéristiques attendues des nombres complexes ?



Représentation POO des nombres complexes

La première question à se poser est :
quelles sont les caractéristiques attendues des nombres complexes ?

- ▶ partie réelle (en lecture, en écriture ?), partie imaginaire
 - ▶ module, argument
 - ▶ conjugué
 - ▶ addition
 - ▶ ...
- 👉 pour garantir le *découplage* de code dont on parlait précédemment, **toutes** ces caractéristiques doivent être des *méthodes* (et ceci *indépendamment* du choix d'implémentation, c.-à-d. indépendamment des attributs choisis)

Un exemple possible (1/3)

```
#include <iostream>
#include <cmath>    // pour abs, sqrt, atan, cos et sin
#include <numbers>  // pour numbers::pi, >= C++20
using namespace std;

class Complexe {
public:

    // accesseurs
    double x()      const { return x_; }
    double y()      const { return y_; }
    double rho()    const { return sqrt(x_ * x_ + y_ * y_); }
    double theta()  const {
        const double module(rho());
        constexpr double precision(1e-15);
        if (abs(module) < precision)
            { return 0.0; }
        else if ((abs(y_) < precision) and (x_ < 0.0))
            { return numbers::pi; }
        else
            { return 2.0 * atan(y_ / (x_ + module )); }
    }
}
```

(fonction
outil
privée)

$$z = 4 + 3i$$

utilisation!

$z.x()$

$z.rho()$

Complexe z;

Un exemple possible (2/3)

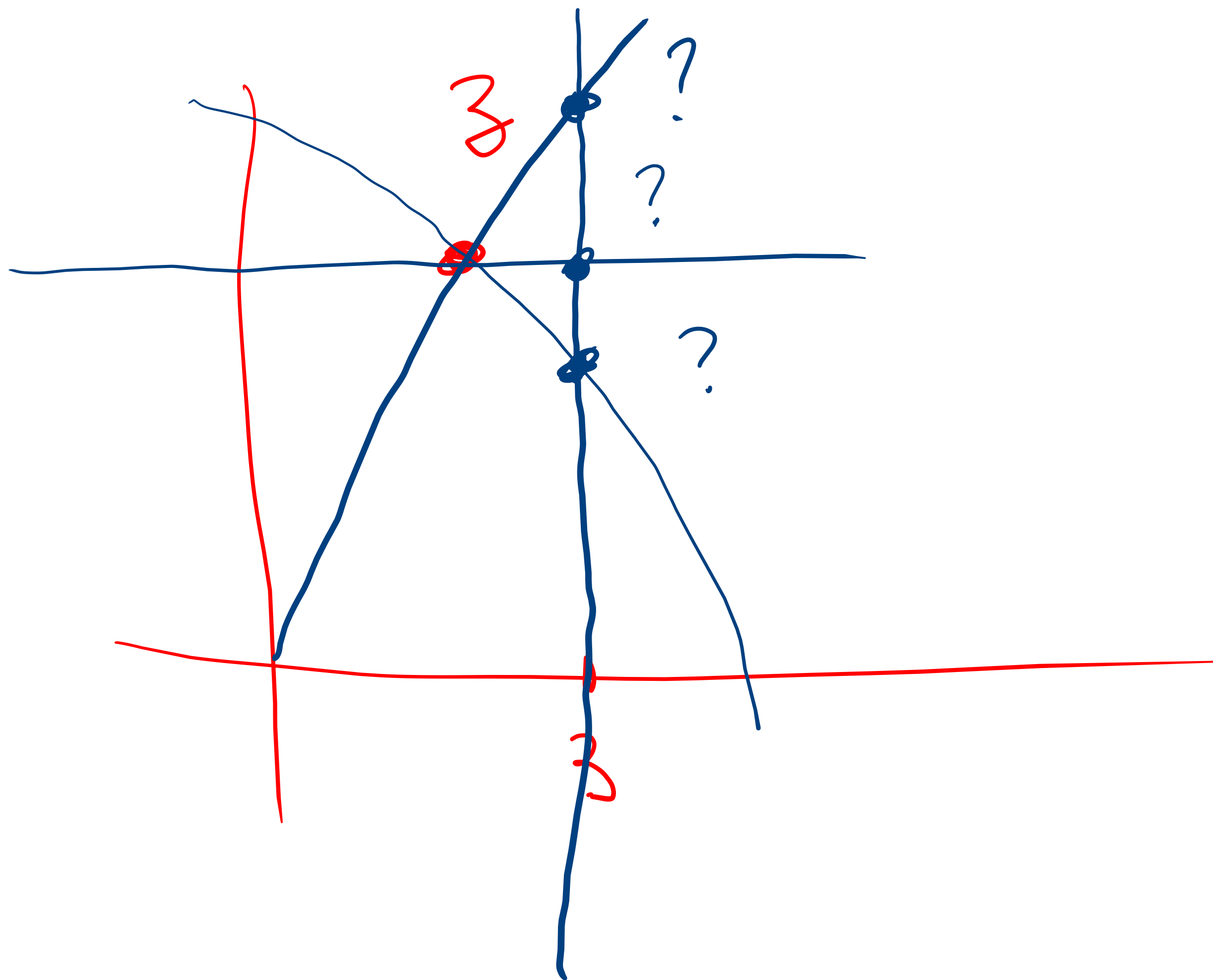
```
// manipulateurs
void cartesiennes(double abscisse, double ordonnee)
{ x_ = abscisse ; y_ = ordonnee ; }
void polaires(double module, double argument)
{ x_ = module * cos(argument) ;
  y_ = module * sin(argument) ;
}

void set_x(double abscisse)           //
{ cartesiennes(abscisse, y()); }      // Ces quatre là sont
void set_y(double ordonnee)          // TRÈS discutables !
{ cartesiennes(x(), ordonnee); }      //
void set_rho(double module)          // (et sont discutés
{ polaires(module, theta()); }        // en cours)
void set_theta(double argument)      //
{ polaires(rho(), argument); }       //

// autres opérations
Complexe conjugue() const {
  // sera plus simple à écrire quand nous aurons les constructeurs
  Complexe c;
  c.cartesiennes(x_, -y_);
  return c;
}
```

$z^1 = z^2;$

5?



$f, \text{set} - x(3);$

Un exemple possible (3/3)

```
private: // un choix d'implémentation parmi d'autres
    double x_;
    double y_;
};

// =====
int main()
{
    // exemple d'utilisation
    Complexe a;
    a.cartésiennes( 1.0, 2.0);

    cout << a.x() << "+" << a.y() << "i = "
         << a.rho() << "e^(i*" << a.theta() << ")"
         << endl;

    return 0;
}
```

Modularisation de l'exemple : .h

```
#pragma once
// pas besoin d'include ici
class Complexe {
public:
    // accesseurs
    double x() const { return x_; }
    double y() const { return y_; }
    double rho() const;
    double theta() const;

    // manipulateurs
    void cartesiennes(double abscisse, double ordonnee);
    void polaires(double module, double argument);
    // ...

    // autres opérations
    Complexe conjugue() const;

private:
    // un choix d'implémentation parmi d'autres
    double x_;
    double y_;
};
```

1 seule ligne

Modularisation de l'exemple : .cc

```
#include "complexe.h"
#include <cmath>    // pour abs, sqrt, atan, cos et sin
#include <numbers>  // pour numbers::pi, >= C++20
using namespace std;

double Complexe::rho() const { return sqrt(x_ * x_ + y_ * y_); }

double Complexe::theta() const {
    const double module(rho());
    constexpr double precision(1e-15);
    if (abs(module) < precision)
        { return 0.0; }
    else if ((abs(y_) < precision) and (x_ < 0.0))
        { return numbers::pi; }
    else
        { return 2.0 * atan(y_ / (x_ + module )); }
}

void Complexe::cartesiennes(double abscisse, double ordonnee)
{ x_ = abscisse ; y_ = ordonnee ; }

void Complexe::polaires(double module, double argument)
{ // ... etc.
```