

## Sessions\_INF224

### INF224 - Partiel Novembre 2022

#### Question 1: (?? pts)

---

Q : Langage réflexif : Explication + Exemples

**Réponse:**

En programmation informatique, la réflexion est la capacité d'un programme à examiner, et éventuellement à modifier, ses propres structures internes de haut niveau lors de son exécution.

On appelle réflexivité le fait pour un langage de programmation de permettre l'écriture de tels programmes. Un tel langage de programmation est dit réflexif.

Par exemple: Lisp, Smalltalk, Java(Partiellement), Python.

---

#### Question 2: (?? pts)

---

Q: Enoncer et Expliquer le théorème de Boehem et Jacopini

**Réponse:**

Tout programme comportant éventuellement des sauts conditionnels peut être réécrit en un programme équivalent n'utilisant que les structures de contrôle while et if-then. Conséquence: La voie est ouverte vers la programmation moderne // diminution d'usage des goto(s).

---

#### Question 3: (?? pts)

---

Q: Swing Events. Explain the methods (advantages + disadvantages).

**Réponse:**

**Objets hybrides**

**Version 1:**

- plus souple : autant de listeners que l'on veut
- mais peu concise : on multiplie les objets et les lignes de code

**Version 2:**

- plus simple mais limitée : on ne peut avoir qu'une seule méthode actionPerformed()
- peu adaptée si beaucoup de commandes

**Classes imbriquées:**

- souplesse sans la lourdeur !
  - La classe imbriquée à accès aux variables d'instance de la classe dans laquelle elle se trouve (attention! pas en c++)
-

## Question 4: (4,5 pts)

---

Q : Un constructeur automobile vous demande de développer le système multimédia de sa nouvelle voiture. Ce système permettra :

- d'appeler un correspondant au téléphone
- de jouer un morceau de musique
- d'écouter une radio.

On suppose que chacun de ces 3 types d'éléments aura :

- un nom permettant de le retrouver,
- une donnée associée qui sera, respectivement, un entier (numéro de téléphone), une string (nom de fichier), un flottant (fréquence).

De plus, tous les éléments seront contenus dans la même base.

Le but est d'écrire les classes adéquates, en C++ et en suivant une approche orientée objet, de telle sorte qu'une méthode de la base prenant en argument le nom d'un élément effectue l'action correspondant à cet élément (appeler, jouer, etc.)

L'implémentation des actions sera en commentaire. Pour simplifier, on ne mettra que ce qui est indispensable et on écrira toutes les classes dans un unique fichier header.

---

## Question 5: (0,75 pts)

---

Q: En Java quelles affirmations sont vraies (plusieurs réponses) :

Veuillez choisir au moins une réponse :

- a. Une interface peut être implémentée par plusieurs classes
- b. Une classe peut implémenter plusieurs interfaces
- c. Une classe peut implémenter plusieurs classes
- d. Une interface peut implémenter plusieurs classes

Réponse: a et b

---

## Question 6: (0,75 pts)

---

Q: Quelle proposition est correcte ?

- a. Le downcasting implicite est permis en C++ mais interdit en Java
- b. Le downcasting implicite est permis en Java mais interdit en C++
- c. Le downcasting implicite est interdit en Java et en C++
- d. Le downcasting implicite est permis en Java et en C++

Réponse: c

---

## Question 7: (0,75 pts)

---

Q: Quelle proposition est correcte ?

- a. En Java les variables sont toujours passées par référence
- b. En Java les variables sont toujours passées par valeur
- c. En Java certaines variables sont toujours passées par valeur et d'autres par référence
- d. En Java les variables peuvent être passées par valeur ou par référence selon le choix du programmeur

Réponse: b

---

### Question 8: (0,75 pts)

---

Q: Quelle proposition est correcte ?

- a. En C++ les variables sont toujours passées par référence
- b. En C++ les variables sont toujours passées par valeur
- c. En C++ certaines variables sont toujours passées par valeur et d'autres par référence
- d. En C++ les variables peuvent être passées par valeur ou par référence selon le choix du programmeur

Réponse: d

---

### Question 9: (0,75 pts)

---

Q: Le polymorphisme de classe se fait grâce à (plusieurs réponses) :

Veuillez choisir au moins une réponse :

- a. La redéfinition de fonction
- b. Les classes virtuelles
- c. Les fonctions virtuelles
- d. Les fonctions friend
- e. Un autre mécanisme

Réponse: a et c

---

### Question 10: (0,75 pts)

---

Q: Après la ligne suivante en C++ :

```
auto value = 3;
```

- a. on pourrait ensuite écrire: value = "toto";
- b. on ne pourrait pas ensuite écrire: value = "toto";
- c. ce code est de toute façon illégal en C++

Réponse: b

Explication: value est alors un entier et on peut pas lui affecter un string

---

### Question 11: (0,75 pts)

---

Q: Qu'affiche ce programme ?

```
#include <iostream>
class X {
public:
    X() { std::cout << "X"; }
    void foo1() { std::cout << "1"; }
    virtual void foo2() { std::cout << "2"; }
};

class Y : public X {
public:
    Y() { std::cout << "Y"; }
    void foo1() { std::cout << "1"; }
    void foo2() override { std::cout << "2"; }
};

int main() {
    X* obj = new Y();
    obj->foo1();
    obj->foo2();
    return 0;
}
```

- a. XY11
- b. XY12
- c. XY21
- d. XY22

Réponse: b

---

## Question 12: (0,75 pts)

---

Q: Qu'affiche ce programme ?

```
#include <iostream>
class X {
public:
    X() { std::cout << "X"; }
    ~X() { std::cout << "x"; }
};

class Y : public X {
public:
    Y() { std::cout << "Y"; }
    ~Y() { std::cout << "y"; }
};

int main() {
    Y a;
    Y * b = new Y();
    return 0;
}
```

- a. XYXY
- b. XYXYyx
- c. XYXYxyx
- d. XYXYxy
- e. XYXYxyxy

**Réponse: d**

**Explication:**

- car B\* val = new B(); cela appelle d'abord le constructeur de la classe A (la classe de base) puis le constructeur de la classe B (la classe dérivée).
- on crée une instance de Y donc XY
- on crée une 2ème instance de Y donc XY
- automatiquement le compili va erase a, les destructeurs s'appellent dans le sens inverse yx
- On aurait eu yxyx si on avait ajouté delete(b)

### Question 13: (?? pts)

**Q: Qu'affiche ce programme ?**

```
#include <iostream>
class X {
    int i = 0;
public:
    X(int i) : i(i) { std::cout << i; }
    X(const X & x) { x.i = i; std::cout << i; }
};

int main() {
    X a(5);
    X b(a);
}
```

- a. 00
- b. 50
- c. 05
- d. 55
- e. il ne compile pas

**Réponse: e**

**Explication:** vous essayez d'assigner la valeur de i de l'objet actuel (l'objet en cours de copie) à l'objet source x

### Question 14: (1.5 pts)

**Q: La classe X est incomplète, pourquoi ? Complétez-la en faisant bien attention aux plantages que l'exécution de la fonction main() pourrait produire.**

```
#include <iostream>
class X {
    int * p = nullptr;
public:
    X(int i) { p = new int(i); }
};

int main() {
    X a(5);
    X b(a);
}
```

## Réponse:

Il faut ajouter:

```
~X() { delete p; } // pour éviter la fuite mémoire
```

on crée b par copie de a donc on doit ajouter un constructeur de copie !

```
X(const X &other) {
    if (other.p) {
        p = new int(*other.p);
    }
}
```

## Question 15: (?? pts)

Q: Qu'affiche ce programme C++ ?

```
#include <iostream>
class X {
public:
    virtual char foo() { return 'x'; }
};

class Y : public X {
public:
    char foo() override { return 'y'; }
};

int main() {
    Y* p = new Y();
    std::cout << p->foo() << std::endl;
    std::cout << ((X*)p)->foo() << std::endl;
}
```

- A : yy
- B : yx
- C : xx

## Réponse: A (yy)

### Explication:

vous utilisez un cast explicite pour convertir le pointeur de type Y *en un pointeur de type X* avec (X\*)p. Cependant, le fait que foo() soit virtuelle signifie que la résolution de la fonction se fait au moment de l'exécution (liaison dynamique).

Lorsque vous appelez ((X\*)p)->foo(), le système sait qu'il s'agit toujours d'un objet de type Y, car la conversion de pointeur ne change pas le type d'objet auquel il pointe. Par conséquent, la fonction virtuelle foo() de la classe Y

## Question (?? pts)

Q: Qu'affiche ce programme C++ ?

```
#include <iostream>
template <int N> struct Foo {
    static const int value = N * Foo <N-1>::value;
};

template <> struct Foo<0> {
    static const int value = 1;
};

int main() {
    int x = Foo<4>::value;
    int y = Foo<0>::value;
    std::cout << x << " " << y;
}
```

- a. 4 0
- b. 4 1
- c. 12 1
- d. 24 1

Réponse: d

Explication: code récursif qui fait N\*Foo

## Question (?? pts)

Q: Ce programme Java fait-il ce qu'on souhaite ?

On souhaite qu'il affiche "Dupond1". Est-ce le cas ?

- A : oui
- B : non. Si vous avez répondu non, corrigez le code.

```
public class User {
    String name;
    int id;

    public User(String name, int id) { this.name = name; this.id = id; }

    public void getNameAndId(String name, int id) {
        name = this.name;
```

```

    id = this.id;
}

public static void main(String[] args) {
    String name = null;
    int id = 0;
    User u = new User("Dupond", 1);
    u.getNameAndId(name, id);
    System.out.println(name + id);
}
}

```

**Réponse: B (non)**

**Correction:**

Le code n'affiche pas ce que l'on veut, il faut mettre dans getNameAndId:

```

this.name = name
this.id = id

```

## Question (?? pts)

**Q: Quelle proposition est correcte ?**

- A : L'héritage permet la généricité et les templates permettent le polymorphisme
- B : L'héritage permet le polymorphisme et les templates permettent la généricité
- C : aucune des deux

**Réponse: B**

## Question (?? pts)

**Q: Que signifie le concept d'encapsulation ?**

- A : C'est un mécanisme de gestion de la mémoire propre à l'OO.
- B : C'est le regroupement d'une collection d'objets dans un nouvel objet.
- C : C'est le principe consistant à différencier les propriétés internes et les propriétés externes d'un objet.
- D : C'est un principe de mise en relation des objets d'une application.

**Réponse: C**

**Développement:**

But : séparer la spécification de l'implémentation; on ne peut interagir avec l'objet que via ses méthodes; seul l'objet peut accéder à ses variables. Permet d'abstraire (exhiber les concepts, cacher les détails d'implémentation); de modulariser (limiter les dépendances entre composants logiciels) et de protéger l'intégrité de l'objet (ne peut pas être modifié à son insu, assure la validité de ses données).

En C++ 4 type de droit d'accès: private : pour les objets de cette classe (défaut si on ne met rien) ; protected : également pour les sous-classes; public : pour tout le monde; friend : pour certaines classes ou certaines fonctions.

En java 3 types: private, protected, public. Par défaut toutes les classes du package y ont accès.

## Question (?? pts)

---

**Q: En C++, un objet const d'une classe peut appeler une méthode non-const de cette classe ?**

- A : Vrai
- B : Faux
- C : Un objet ne peut pas être const.
- D : Aucune de ces réponses n'est vraie.

**Réponse: B**

**Explication:** en c++ un objet const ne peut pas appeler les méthodes non const

---

## Questions Complémentaires et Réponses

**Q: Quels sont les quatre principaux paradigmes de programmation?**

---

**Réponse:** Orienté objet, fonctionnelle, logique et impérative

---

**Q: Qu'est-ce que l'approche fonctionnelle de la programmation?**

---

**Réponse:** L'approche fonctionnelle de la programmation se concentre sur la notion de fonctions. Dans un programme fonctionnel, tous les éléments peuvent être compris comme des fonctions et le code peut être exécuté par des appels successifs de fonctions.

---

**Q: Négation par l'échec en Prolog**

---

**Réponse:** Négation par l'échec : ce qui n'est pas vrai est faux (hypothèse du monde clos). Ce n'est pas une négation logique.

Parce qu'en Prolog on utilise la notion de négation par l'échec, c'est à dire, ce qui n'est pas vrai est faux (hypothèse du monde clos). Ce n'est pas une négation logique.

---

**Q: Contraintes en Prolog**

---

**Réponse:** Une contrainte c'est une relation entre des variables, chacune prenant une valeur dans un domaine donné. Une contrainte restreint les valeurs possibles que les variables peuvent prendre. Elles ont comme objectif d'être le plus logique possible en gardant la résolution SLDNF et fournir un outil puissant. Prolog et les contraintes ouvre le nouveau domaine de la programmation logique avec contraintes (PLC) : Prolog est étendu avec des solveurs de contraintes, travaillant sur des données qui peuvent être des entiers, des booléens, des domaines finis d'entiers, des rationnels, des intervalles de rationnels.

---

**Q: Types de mémoire en C++ et Java**

---

**Mémoire automatique:**

- variables locales et paramètres
- créées à l'appel de la fonction détruites à la sortie de la fonction
- la variable contient la donnée
- Différences => C++ : types de base et objets // Java : que types de base ou références

#### Mémoire globale/static:

- variables globales ou static dont variables de classe
- créées au lancement du programme détruites à la fin du programme
- la variable contient la donnée
- Différences => C++ : types de base et objets // Java : que variables de classe qui sont des types de base ou références.

#### Mémoire dynamique:

- pointés créés par new détruits par delete
- la variable pointe sur la donnée
- Différences => C++ : objets et types de base penser à détruire les pointés via delete ! // Java : que pour les objets.

---

## Q: Méthode d'instance vs méthode de classe

**Réponse:** Une méthode d'instance est une méthode qui dépend de l'instanciation d'un objet, alors son résultat peut varier en fonction de chaque objet. Une méthode de classe est toujours statique et ne dépend pas des objets. Une fonction non-membre sont les fonctions qui ne sont associées à aucune classe. Il y a pas de différences avec Java (je crois)

---

## Q: Copie superficielle vs Copie profonde

#### Copie superficielle:

- copie champ à champ => copie les pointeurs
- Problématique: si l'objet contient des pointeurs.
- If you have object which consist of simple data type members(value type), In that case shallow copy will be a good option as it is fast and less expensive in terms of memory.

#### Copie profonde:

- copie les pointés (récursivement)
- If Object has been consist of more reference type members. In that case shallow copy will be not effective and deep copy will be a good option.

#### Différence:

- Java ne permet de copier que les références (pas les objets pointés)
- Même problème en Java et C++ => si l'objet contient des références (qui se comportent comme des pointeurs)

#### Cas dangereux:

Même problème en Java et C++ => si l'objet contient des références (qui se comportent comme des pointeurs)

**Solution:** Interdire les opérateurs de copie

---

## Q: Destructeur en C++ et Java

---

**Réponse:** Le destructeur est appelé automatiquement quand l'objet est détruit. Il sert à signaler la destruction de l'objet. En Java, la méthode `finalize()` joue le même rôle.

On n'utilise pas "`finalize()`" en java, effectivement il y a pas besoin du destructeur a cause du Garbage collector. (GabM)

WRONG: `finalize()` is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. However, due to the uncertainty of when to GC is going to collect the object, we don't rely on it much

---

## Q: Polymorphisme d'héritage

---

Avec le polymorphisme d'héritage, un objet peut être vu sous plusieurs formes, par exemple: un Square est aussi un Rect.

**Buts:**

- Pouvoir choisir le point de vue le plus approprié selon les besoins
- Pouvoir traiter un ensemble de classes liées entre elles de manière uniforme sans considérer leurs détails

**Java:**

- liaison dynamique / tardive
- choix de la méthode à l'exécution => appelle la méthode du pointé

**C++:**

- avec `virtual` : liaison dynamique / tardive => méthode du pointé
  - sans `virtual` : liaison statique => méthode du pointeur (le carré devient un rectangle!)
- 

## Q: Problème du diamant

---

**Réponse:** Shape est incluse (donc définie) 2 fois ! => erreur de compilation !

Le problème du diamant => Cela peut poser problème à la instanciation de la classe fille car son constructeur va faire appel aux constructeurs de Mère 1, 2 qui eux même vont faire appel à celui de la classe GrandMère. Le compilateur va cracher. En effet, ce dernier ne saura pas par qui passer (Mère 1 ou Mère 2) pour accéder au constructeur de GrandMère.

**Solution:** le command `#ifndef` évite les inclusions multiples.

---

## Q: Généricité

---

**Réponse:** Il s'agit de l'utilisation de templates (en C++) ou de la classe Generics (Java). Ceci est utile quand une même fonction s'applique à différents types et qu'on ne veut pas écrire la même fonction avec seulement des types différents, ou que l'upcasting n'est pas possible.

---

## Q: Exemple de fonction swap en C++

---

```
void swap(int &i, int &j){  
    int aux;  
    aux = i;  
    i = j;  
    j = aux;  
}
```

On peut pas faire ça en java

---

## Q: Quels sont les trois principaux types de l'Orienté objet? Y a t'il de différences notables entre c++ et java?

---

**Réponse:** Méthodes d'instance, Héritage et Polymorphisme. Il n'y a pas de différences notables entre les deux langages.

---

## Q6: Correction méthode getXY

---

Il manque foo ?

C'est faux en c++ car les variables sont passées par copie. Il faut les passer par référence.

```
void getTime(int& heure, int& minutes);
```

```
void Point::setXY(int x, int y){  
    this->x = x;  
    this->y = y;  
}
```

(je suis pas certain mais je crois que le get est correct)(gabM)

```
void Point::getXY(int &x, int &y){  
    x = this->x;  
    y = this->y;  
}
```

---

## Exemples de code

### Exemple classe Clients et Base

---

```
class Clients{  
private:  
    string nom;  
    const int id;  
    float valeur;  
public:  
    Clients(string nom, int id_, float valeur):id(id_);  
    //get... set...  
    ~Clients(){}  
}
```

```
class Base{
private:
    std::map<int, shared_ptr<Clients>> base;
public:
    void addClient(Client *client){
        this->base[client->getId()] = client;
    }
    Client searchClient(int id){
        it = this->base.find(id)
    }
    void deleteClient(int id){
        this->base.erase(id)
    }
    ~Base(){}
}
```

**WRONG:** You're adding a Client\* to a map of shared pointers