INF224 - Partiel Novembre 2022

Question 1: (?? pts)

Q : Langage réflexif : Explication + Exemples

Question 2: (?? pts)

Q: Enoncer et Expliquer le théorème de Boehem et Jacopini

Question 3: (?? pts)

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

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

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++

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

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

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

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";

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

c. ce code est de toute façon illégal en C++

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;
}
```

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).

a. XY11

b. XY12

c. XY21

d. XY22

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;
}
```

on cree une instance de Y donc XY

on cree une 2eme instance de Y donc XY

automatiquement le compi va erase a, les destructeurs s'appelent dans le sens inverse yx

On aurait eu yxyx si on avait ajoute delete(b)

a. XYXY

b. XYXYyx

c. XYXYyxyx

d. XYXYxy

e. XYXYxyxy

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; }
};
```

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

Qu'est-ce que ce programme C++ affiche ?
A : yy
B : yx
C : xx

```
#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;
}
```

```
int main() {

   X a(5);

   X b(a);

}
```

a. 00

b. 50

c. 05

d. 55

e. il ne compile pas

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 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); }
                                ~X() {delete p;} pour eviter la fuite memoire
};
                                X(const X &other) {
                                   if (other.p) {
                                      p = new int(*other.p);
                                   }
int main() {                    }

   X a(5);

   X b(a);
```

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

```
}
```

Question 15: (?? 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;
}
```

code recurssif qui fait N*Foo<N-1>

a. 4 0

b. 4 1

c. 12 1

d. 24 1

```
public String getName() {
      return name;
   }

   public int getId() {
      return id;
   }

   public static void main(String[] args) {
      User u = new User("Dupond", 1);
      String name = u.getName(); // Obtient le nom
      int id = u.getId(); // Obtient l'identifiant
      System.out.println(name + id); // Affiche "Dupond1"
   }
```

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);
  }
}
```

Le code n'affiche pas ce que l'on veut, il faut mettre dans getNameAndId: this.name = name et this.id = id

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 V
C : aucune des deux

Que signifie le concept d'encapsulation ?

FIN DU PARTIEL

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. V
D : C'est un principe de mise en relation des objets d'une application.

En C++, un objet const d'une classe peut appeler une méthode non-const de cette classe ?
A : Vrai
B : Faux V en c++ un objet const ne peut pas appeler les methodes nn const
C : Un objet ne peut pas être const.
D : Aucune de ces réponses n'est vraie.

Mémoire Automatique: La mémoire automatique est généralement allouée pour les variables locales dans une fonction ou un bloc de code.
ex: int a = 5;

Mémoire Statique: La mémoire statique est allouée pour des variables dont la durée de vie est égale à celle du programme. Cette mémoire est généralement utilisée pour les variables globales, statiques et constantes.
ex:static int count = 0;

Mémoire Dynamique: La mémoire dynamique est allouée au moment de l'exécution et peut être utilisée pour créer des objets ou des tableaux de taille variable. Elle permet une flexibilité dans la gestion de la mémoire.
ex:
int* ptr = new int; // 'ptr' est alloué dynamiquement
*ptr = 5; // Utilisation de la mémoire
delete ptr;

Q1. (2 points) Qu'est le théorème de Boehm et Jacopini ? Quels en ont été les conséquences ?

Q2. (2 points) Que sont les contraintes en Prolog ? A quoi servent-elles ?

(Merci de répondre aux questions suivantes sur une autre feuille)

User* user = new User("Dupond", 1);

Q3. (2 points) Quels sont les types de mémoire que peut l'on peut utiliser dans un programme C++ ou Java ? Y a-t'il des différences entre ces 2 langages ? oui difference seulement en memoire dynamique.

classe/statique: Appelée à l'aide du nom de la classe, n'a pas acces direct au attribut d'instance, Elle peut être appelée sans créer d'instance de la classe.

Q4. (2 points) Qu'est-ce qu'une méthode d'instance, une méthode de classe, une fonction non-membre? Quelles sont leurs propriétés respectives ? Y a-t'il des différences entre Java et C++? oui Les fonctions non-membre pas possible en java

fonction qui n'est pas définie à l'intérieur d'une classe. N'a pas accès aux attributs d'instance ou statiques d'une classe.

Q5. (2 point) Qu'est-ce que la copie profonde et la copie superficielle ? Dans quel cas la copie profonde est-elle utile ? Y a-t'il des differences entre Java et C++ à cet égard ?

superficielle: copie champ à champ , probleme s il y a des poiteur
profonde: copie les pointés récursivement donc les modifications apportées à l'objet original n'affecteront pas la copie.

en java meme probleme si l'objet contient des références (comme poiteur), il faut utiliser des fonction comme clone et copy

Q1. 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).

tout programme peut s'écrire en n'utilisant que while et if-then (donc sans goto)

Q2. 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 résolveurs de contraintes, travaillant sur des données qui peuvent être des entiers, des booléens, des domaines finis d'entiers, des rationnels, des intervalles de rationnels.

specifient des relations et des conditions que les solutions doivent respecter.

Q3. 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.

Q4. 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)

Q5. Copie superficielle: copie champ à champ => copie les pointeurs
Problématique: si l'objet contient des pointeurs.

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.
-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.

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)

Q1. (2 points) Quand dit-on qu'un langage est réflexif ? Connaissez- vous des langages réflexifs ?

Q2. (2 points) En programmation logique, qu'appelle-ton la négation par l'échec (negation by failure) ?

(Merci de répondre aux questions suivantes sur une autre feuille)

Q3. (1 point) Quelle est la différence entre une méthode d'instance et une méthode de classe ? Y a-t'il des différences entre Java et C++ ?

Q4. (1 point) A quoi servent les destructeurs en C++ et dans quels cas sont-ils nécessaires ? Y a t'il des destructeurs en Java ?

Q5. (2 points) Qu'est-ce que la copie profonde et la copie superficielle ? Dans quel cas la copie profonde est-elle utile ? Y a-t'il des différences entre Java et C++ à cet égard ?

Q1. 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. son comportement.

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.

Q2. Négation par l'échec : ce qui n'est pas vrai est faux (hypothèse du monde clos). Ce n'est pas une négation logique.

Q3. Repondu ci-dessus

Q4. Le destructeur est appelé automatiquement quand l'objet est détruit. Il sert à signaler la destruction de l'objet. En Java, la méthode finalise() joue le même rôle.
les destructeur permettent de liberer de la memoire/ des ressoureces lorsque l'objet est detruit.
necessaire si on utilise l'allocation dynamique. en java on a pas besoin car il y a le ramasse miette.

On n'utilise pas "finalise()" 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

Q5. Copie superficielle: copie champ à champ => copie les pointeurs

Problématique: si l'objet contient des pointeurs.

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.
-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.

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)

Q6. (2 points) On souhaite écrire une méthode qui permette de récupérer l'heure et les minutes sous la forme de 2 entiers passés en arguments. On suppose que foo() et calcul() sont des méthodes de la même classe et que les ... sont des appels système adéquats. Est-ce que ce qui suit est correct en **C++ et** en **Java** ? Corrigez si nécessaire et expliquez.

```
void getTime(int heure, int minutes) {
    heure = ...;
    minutes = ...;
}
void calcul() {
   int heure, minutes;
   getTime(heure, minute);
}
```

Q6. 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);

Q6. (3 points) Soit la classe *Point* qui represente un point à l'écran. Ecrire, en C++, le constructeur et les méthodes *setXY()* et *getXY()*. Le *constructeur* et *setXY()* prendront deux entiers en argument qui changeront les valeurs des variables d'instance x et y. Inversement, *getXY()* permettra de récuper **à la fois** les valeurs de ces deux variables. Ecrire un exemple où l'on crée un point et où on appelle ensuite *getXY()*.

Quelles seraient les différences si l'on écrivait cette classe en Java?

```
class Point {
  int x, y;
public:
  Point(...);
  void setXY(...);
  void getXY(...);
};
```

Q7. (3 points) Quelles sont les techniques de programmation typiquement utilisées pour ajouter des listeners aux composants graphiques Java Swing. Expliquez en vous aidant de schémas. Les mêmes techniques peuvent-elles être utilisées en C++ ? Expliquez.

Q6.

```
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;
}
```

Q7.

```
#include <iostream>
using namespace std;

class Point {
  int x, y;

  public:
  Point(int x_val, int y_val) : x(x_val), y(y_val) {}

  void setXY(int x_val, int y_val) {
    x = x_val;
    y = y_val;
  }

  void getXY(int &x_out, int &y_out) const {
    x_out = x;
    y_out = y;
  }

};

int main(int argc, char const *argv[])
{
    Point p(1,2);
    int x, y;
    p.getXY(x, y);
    cout << "x= " << x << " y= " << y << endl;
    return 0;
}
```

Q1. (2 points) Qu'est-ce qui caractérise la programmation fonctionnelle ? (en moins de 5 lignes)

Q2. (2 points) En PROLOG, qu'appelle-t-on la négation par l'échec ? (en moins de 5 lignes)

(Merci de répondre aux questions suivantes sur une autre feuille)

Q3. (2 points) Quels sont les types de mémoire que peut l'on peut utiliser dans un programme C++ ou Java ? Y a-t'il des différences entre ces 2 langages ?

Q4. (2 points) Qu'est-ce qu'une méthode d'instance, une méthode de classe, une fonction non-membre? Quelles sont leurs propriétés respectives ? Y a-t'il des différences entre Java et C++?

Q5. (2 point) A quoi servent les destructeurs en C++ et dans quels cas sont-ils nécessaires ? Y a t'il des destructeurs en Java ?

Q1. L'approche fonctionnelle de la programmation se concentre sur la notion de fonctions. Dans un programme fonctionnel, tous les éléments peuvent être compris

La programmation fonctionnelle est un paradigme de programmation qui se concentre sur l'utilisation de fonctions comme éléments principaux pour construire des programmes.
plus robuste, plus facile à vérifier et à maintenir

comme des fonctions et le code peut être exécuté par des appels successifs de fonctions.

Q2. Négation par l'échec : ce qui n'est pas vrai est faux (hypothèse du monde clos). Ce n'est pas la négation logique.

Q3. 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 Mp
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.

Q4.

Q5. Le destructeur est appelé automatiquement quand l'objet est détruit. Il sert à signaler la destruction de l'objet. En Java, la méthode finalise() joue le même rôle.

A1. (2 points) Citer le théorème de Boehm et Jacopini. Quels ont été les conséquences de ce théorème ?

A2. (2 points) Pourquoi dit-on que la négation de Prolog n'est pas réellement la négation logique ?

Merci de répondre aux questions suivantes sur une autre feuille
(pensez à mettre votre NOM !)

B1. (2 points) Quels sont les types de mémoire que peut l'on peut utiliser dans un programme C++ ou Java ? Y a-t'il des différences ?

B2. (2 points) Qu'appelle-t'on "liaison tardive" (ou liaison dynamique ou polymorphisme d'héritage) et quel est son but ? Y a t'il des différences entre Java et C++ ? Lesquelles ?

B3. (2 points) Dans quels cas la copie d'objet peut-elle être dangereuse ? Que convient-il alors de faire? Y a-t'il des différences entre Java et C++ ?

A1. 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).

A2. Parce que en Prolog on utilise la notion de négation par 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.

B1. 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.

B2. Avec le polymorphisme d'héritage, un objet peut être vu sous plusieurs formes, par exemple: un Square est aussi un Rect.

Cela repose sur le principe de l'héritage, où une classe dérivée hérite des propriétés et méthodes d'une classe de base.

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!)

B3. 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
Différence => Java ne permet de copier que les références (pas les objets pointés)

A1. (2 points) Citer le théorème de Boehm et Jacopini. Quels ont été les conséquences de ce théorème ?

A2. (2 points) Pourquoi dit-on que la négation de Prolog n'est pas réellement la négation logique ?

(Merci de répondre aux questions suivantes sur une autre feuille)

B1. (2 points) Quels sont les types de mémoire que peut l'on peut utiliser dans un programme C++ ou Java ? Y a-t'il des différences ?

Multimedia m = new Photo(); ?

B2. (2 points) Qu'appelle-t'on "liaison tardive" (ou liaison dynamique ou polymorphisme d'héritage) et quel est son but ? Y a t'il des différences entre Java et C++ ? Lesquelles ?

B3. (2 points) Dans quels cas la copie d'objet peut-elle être dangereuse ? Que convient-il alors de faire? Y a-t'il des différences entre Java et C++ ?

↝ copie de classes

B4. (2 points) Quels sont les principaux problèmes que l'héritage multiple peut poser ? Quelles sont les solutions ? Comparer C++ et Java.

B5. (2 points) Quelles sont les techniques de programmation typiquement utilisées pour ajouter des listeners aux composants graphiques Java Swing ? Quels sont leurs avantages ou inconvénients respectifs ? Faire les schémas correspondants.

classe abstraite : interface

A1. 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).

A2. Parce qu'en Prolog on utilise la notion de négation par 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.

B1. 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.

B2. 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!)

B3. 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
Différence => Java ne permet de copier que les références (pas les objets pointés)

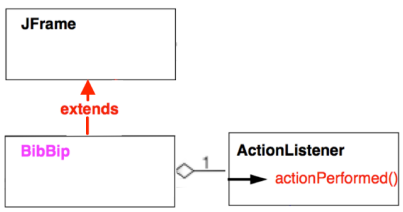
Shape est inclue (donc définie) 2 fois !
=> erreur de compilation !

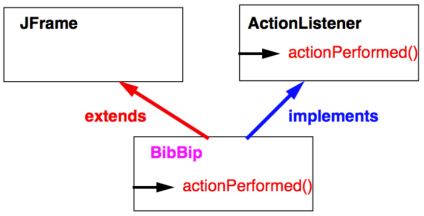
B4. Le problème du diamant = > Cela peut poser poser problème à la instanciation de la classe fille car son constructeur va faire appel aux constructeurs de Mere 1 , 2 qui eux même vont faire appel à celui de la classe GrandMere. Le compilateur va cracher. En effet, ce dernier ne saura pas par qui passer (Mere 1 ou Mere 2) pour accéder au constructeur de GrandMere. ~~Solution: le command #ifndef évite les inclusions multiples.~~

solutions au probleme de diamant:
- pas de variable dans la classe grand mere
- heritage virtuel

autre probleme:
collisions des noms => solution: redefinir les fct ou utiliser using
heritage en diamant => heritage imbrique

B5.
Objets hybrides



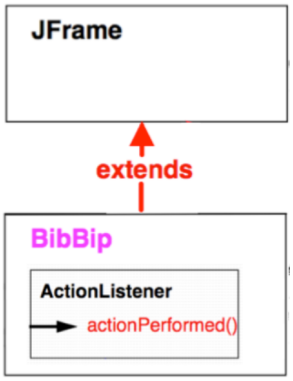


Version 1 Version 2

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 imbriquée



souplesse sans la lourdeur !

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

Q1. (2 points) Donner et définir les 4 grands paradigmes en programmation.

Q2. (2 points) Citer le théorème de Boehm et Jacopini. A quoi a-t-il servi ?

(Merci de répondre aux questions suivantes sur une autre feuille)

Q3. (2 points) Qu'est-ce que l'encapsulation en programmation objet ? Quels sont les buts recherchés ? Concrètement, comment est-ce implémenté en C++ et y a t'il des différences avec Java ?

Q4. (2 points) Qu'est-ce que la programmation générique et quels sont ses avantages ? Y a-t'il des différences entre C++ et Java ? Citez des exemples typiques de cas où c'est utile.

Q5. (2 points) On souhaite écrire une fonction swap() qui échange la valeur de deux entiers qui lui sont passés en argument. Ecrire cette fonction en C++ et en Java. Quelles sont les différences? Pourquoi ?

Q1. Orienté objet, fonctionnelle, logique et impérative

Q2. 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).

Q3. 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.

Q4. ?
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.
exemple: algorithme de tri et de recherche, programmation fonctionnelle. colections et conteneurs(liste, map, files d attente)

Q5. C++

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

On peut pas faire ça en java car les entiers sont passe par valeur. pour contouner cela on peut utiliser un tableau qui contient les deux valeurs.

Q6. (2 points) Quelles sont les techniques de programmation typiquement utilisées pour ajouter des listeners aux composants graphiques Java Swing. Expliquez en vous aidant éventuellement de schémas.

Q7. (2 points) Quels sont les cas d'utilisation du mot-clé `const` en C++ ? En prenant un exemple, (par exemple le TP) expliquez pourquoi il est important de préciser ce qui est constant. Quelles sont les alternatives, par exemple en Java ?

Q8. (1 point) Qu'est-ce qui caractérise les fonctions lambda ? Pourquoi permettent-elles de simplifier le code ?

Q. Quels sont les trois principaux types de l'Orienté objet? Y a t'il de différences notables entre c++ et java?
Méthodes d'instance, Héritage et Polymorphisme. Il n'y a pas de différences notables entre les deux langages.

Q7 - Le mot clé const en C++ est utilisé pour définir des constantes qui ne peuvent pas être modifiées après leur initialisation. Son utilisation est importante pour la sécurité du code, la clarté et la gestion de la mémoire.
exemple:
Déclaration de variables constantes : const int max_value = 100;
Paramètres de fonction : void display(const int &value) { ... } =>Lorsqu'un paramètre de fonction est passé par référence ou par pointeur
Méthodes de classe :void display() const { ... } => pour indiquer qu'elle ne modifiera pas l'état de l'objet.

en java: il n'existe pas de mot clé const, mais on peut utiliser final pour des objectifs similaires.
exemple: final int MAX_VALUE = 100;

Q8 - Lambda : fonction anonyme qui capture les variables, elle possède une copie des variables locales.
elles simplifient le code car elles permettent de définir des fonctions de manière concise et inline.
=> Syntaxe plus courte que les fonctions classiques
=> Les fonctions lambda sont souvent utilisées comme arguments pour des fonctions d'ordre supérieur

Q7. (4 points) On veut écrire un distributeur de boissons en **Java Swing**. Celui-ci sera constitué de **deux** classes:

- une classe **interface graphique** comprenant un bouton **Café**, un bouton **Thé** et un label affichant un message textuel
- une classe **mécanisme** qui aura deux méthodes:
  - void **start(**type_boisson**)** : lance la distribution de la boisson
  - bool **done()** : est appelée après la distribution de la boisson. Elle renverra **true** si la distribution a été effectuée et **false** si elle a échoué.
  - le contenu de ces deux fonctions sera vide (commentaire) et on supposera que la méthode **done()** est appelée automatiquement.

Ecrire le code de ce programme. Quand on appuiera sur un des boutons, la distribution de la boisson correspondante commencera et un message adéquat s'affichera sur le label. Après la distribution, un autre message s'affichera et indiquera si la distribution de la boisson a été effectuée ou si elle a échoué.

Proposer une **autre manière** d'ecrire ce code (en utilisant une possibilité différente du langage **Java**). Ne mettre que l'essentiel et commentez vos choix.

Q8. (5 points) On veut écrire en langage **C++** un logiciel de gestion des clients pour la cantine. Ce logiciel comportera deux classes : une classe représentant un *client* et une classe représentant la *base des clients*.

Les *clients* doivent avoir :

- un **nom** (on omet le prénom pour simplifier),
- un **identifiant** numérique **propre à chaque client** (deux clients ne doivent pas avoir le même identifiant) qui ne **pourra pas être modifié** une fois qu'il a été attribué
- une **valeur** courante (le nombre d'Euros restant sur le compte du client)

La *base* doit permettre :

- d'**ajouter** un nouveau client.
- de **rechercher** un client.
- d'**enlever** un client.

Ces 3 fonctions auront pour **seul argument** le nom du client. La fonction ajoutant un client fera en sorte que l'identifiant soit **automatiquement calculé**.

Ecrire le code de ce programme. N'écrire **que les méthodes indispensables** (pas la peine d'écrire tous les accesseurs et modifieurs). Pour simplifier vous pouvez tout écrire dans un seul header (pas de .cpp).

N'hesitez pas à mettre des commentaires pour expliquer vos choix (ou si vous avez oublié la syntaxe exacte). Attention à gérer correctement la **mémoire.**

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

```
#include <iostream>
using namespace std;

#include <string>
#include <map>
#include <memory>

class Client {
   protected:
   string name;
   int id;
   float val;
   public:
   Client(string const &name_client, int id_client, float val_client = 0.0)
   : name(name_client), id(id_client), val(val_client) {}

   string getName() {
      return name;
   }

   int getId() {
      return id;
   }

   float getVal() {
      return val;
   } };
```

```
class BaseClient {
private:
    map<string, shared_ptr<Client>> cantine;
    int uuid;

public:
    BaseClient() : uuid(1) {}

    // Ajoute un nouveau client avec un identifiant unique
    void addClient(string const &name) {
        auto it = cantine.find(name);
        if (it == cantine.end()) {
            auto newClient = make_shared<Client>(name, uuid++, 0);
            cantine[name] = newClient;
            cout << "Ajout du client: " << name << endl;
        } else {
            cout << "Le client existe déjà." << endl;
        }
    }

    // Recherche un client par nom
    shared_ptr<Client> searchClient(string const &name) {
        auto it = cantine.find(name);
        if (it != cantine.end()) {
            return it->second;
        }
        return nullptr;
    }

    // Supprime un client par nom
    void removeClient(string const &name) {
        auto it = cantine.find(name);
        if (it != cantine.end()) {
            cantine.erase(it);
            cout << "Client supprimé : " << name << endl;
        } else {
            cout << "Client n'existe pas." << endl;
        }
    }
};
```

```
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
pointer

Q7. (3 points) Est-ce que le code **C++** ci-après 1) compile, 2) est fonctionnellement correct, 3) est efficace ? Corrigez.

On supposera que la classe *Graphics* est déclarée dans le header *Graphics.h*, qu'elle appartient au namespace *graphics* et qu'elle a une méthode *drawLine()* prenant en argument deux *Point*(s) (ainsi que des méthodes pour l'initialiser, etc.)

---

compile: non car pas de constructeur et draw doit etre public
fonctionnellement correct: non
efficace: non car il y a des fuite de memoire, on doit utiliser des reference const ds les fcts pour eviter les copies inutiles

```
class Point {
  float x, y;
};

class Line {
   Point p1, p2;
   void draw(Graphics g) {g.drawLine(p1, p2);}
};

void drawLines(Graphics g, list<Line*> lines) {
   for (auto it : lines) it->draw(g);
}

// affiche un triangle equilateral de largeur/hauteur "size" centré autour du point x,y
void drawTriangle(Graphics g, float x, float y, float size) {
   Point p1(x - size/2, y - size/2);
   Point p2(x + size/2, y - size/2);
   Point p3(x, y + size/2);

   list<Line *> lines;
   lines.push_back(new Line(p1, p2));
   lines.push_back(new Line(p2, p3));
   lines.push_back(new Line(p3, p1));
   drawLines(g, lines);
}

int main() {
  Graphics g(500, 500);              // crée une zone de dessin de 500x500 pixels
  drawTriangle(g, 250, 250, 200);
  return g->show();                  // affiche la zone de dessin à l'écran
}
```

Q8. (2 points) On veut maintenant réaliser une bibliothèque de formes géométriques. Ecrire *Shape* la classe de base, ainsi que la classe *Triangle*. La classe de base devra avoir des méthodes pour fixer le point de référence et la taille des objets ainsi qu'une méthode pour les dessiner.

Q9. (4 points) On souhaite maintenant pouvoir associer des *Listeners* aux formes géométriques, de telle sorte que lorsque l'on clique dessus (ou dedans), la methode clicked() du *Listener* soit appelée :

a) Ecrire la classe *Listener* (en C++). Elle devra avoir une unique methode *clicked()*.

b) On veut utiliser ces listeners pour qu'une action soit réalisée lorsqu'on clique sur les formes géometriques (par exemple les selectionner et les changer de couleur). Quelles sont les stratégies typiques pour ajouter des Listeners à des objets graphiques en Java **et** en C++ ?

c) Implémenter une de ces stratégies en C++ : écrire le code pour qu'un triangle change de couleur lorsque l'on clique dessus. On supposera que la méthode *clicked()* est appelée automatiquement par le système et que la classe *Graphics* a une methode *changeColor()* qui permet de spécifier la couleur avant d'appeler *drawLine()*. **Attention**, il ne doit y avoir qu'un seul objet *Graphics* (la zone de dessin) obligatoirement créé dans la fonction *main()*.

Q10. (2 points) On suppose maintenant qu'il existe une classe *Drawing* qui contient un ensemble de *Shape*(s). Cette classe *Drawing* implémente la méthode *search()* qui sert à trouver une *Shape*(s) du *Drawing* correspondant à un certain critère (par exemple : son point de réference est à un certain endroit, elle a une certaine taille, etc.).

Cette fonction est *générique* (elle ne fait pas d'a priori sur le critère) et elle renvoie la première forme correspondant au critère sinon *nullptr*. Pour mémoire, `function<>` désigne un pointeur de fonction générique.

```
class Drawing {
public:
    Shape * search(function< bool(const Shape &) >);
    ...
};
```

Ecrire le code adéquat pour appeler *search()* pour trouver une forme ayant une certaine position (i.e. son point de référence est à un certain endroit, indiqué par une variable). Plusieurs solutions sont possibles, choisissez celle que vous préférez et commentez (si vous ne vous rappelez plus la syntaxe, expliquez le principe).

Quelle devrait être la signature de la fonction *search()* pour qu'elle puisse renvoyer toutes les *Shape*(s) répondant au critère (et pas seulement la première trouvée) ?

Q8. (4 points) Soit la classe *BaseEleves* qui contient la base des données des *Eleve*(s) (*Eleve* est aussi une classe). *BaseEleves* implémente la méthode *search()* qui sert à trouver un *Eleve* à partir de certains critères (par exemple son nom).

---

search() prend en argument un pointeur de fonction générique *test* et elle renvoie le premier *Eleve* pour lequel *test* renvoie *true* (sinon *nullptr* si aucun *Eleve* ne correspond aux critères de *test*).

```
class BaseEleves {
public:
    Eleve * search(std::function< bool(const Eleve &) > test);
    ...
 };
```

Pour mémoire, std::function<> *test* est un pointeur de fonction générique qui peut pointer sur à peu près n'importe quelle fonction (ou lambda) ayant la même signature. Si *test* pointe sur une fonction ayant un `argument`, l'appel de cette fonction se fait comme suit : `(*test)(argument)`

a) Ecrire la classe *Eleve* (on se limitera au nom et à l'année de promotion) et une implémentation possible de la classe *BaseEleves* et de sa méthode *search()*.

b) Ecrire une fonction *foo()* qui prend en argument la base d'élèves et qui fait appel à sa méthode *search()* pour trouver l'élève *"Dupont"*.

B7. (1 point) On souhaite créer une librairie de formes géométriques en **C++** que l'on puisse afficher à l'écran via leur méthode *draw(Graphics&)*. Pour ce faire, on déclare une classe abstraite **Shape** dont dériveront plusieurs sous-classes. On suppose que **Shape** hérite d'une classe **Object** déclarée dans une librairie standard que l'on **ne peut pas modifier** et qu'il y a les *#include* adéquats.

La déclaration ci-après est-elle satisfaisante ? Expliquer et corriger si nécessaire.

```
class Shape : public Object {
public:
   void  draw(Graphics&);
   float getX();
   float getY();
   float getWidth();
   float getHeight();

   void  setX(float);
   void  setY(float);
   void  setWidth(float);
   void  setHeight(float);
};
```

il faut declarer les attributs en tant que membre prive: protected: float x, y, width, height;

shape est une classe abstraite donc: virtual void draw(Graphics&) = 0;

les get devrait avoir const car elle ne ,odifient pas l'etat de l'objet.

```
class Shape : public Object {
protected:
    float x, y;        // Coordonnées
    float width, height; // Dimensions de la forme

public:
    // Méthode virtuelle pure pour dessiner la forme
    virtual void draw(Graphics&) = 0;

    // Getters
    float getX() const { return x; }
    float getY() const { return y; }
    float getWidth() const { return width; }
    float getHeight() const { return height; }

    // Setters
    void setX(float newX) { x = newX; }
    void setY(float newY) { y = newY; }
    void setWidth(float newWidth) { width = newWidth; }
    void setHeight(float newHeight) { height = newHeight; }

    // Virtual destructor for proper cleanup of derived classes
    virtual ~Shape() = default;
};
```

B8. (2 points) On veut maintenant définir une classe **Container** qui possède une liste d'**Objects**, certains étant des **Shapes**, d'autres pas. Cette classe aura une méthode *add()* pour y ajouter un **Object** et une méthode *remove()* pour enlever un **Object**. La methode *add()* ne doit pas pouvoir ajouter le même objet plusieurs fois. Ecrire la classe **Container** et ses deux méthodes.

B9. (2 points) On veut rajouter une methode *draw()* à la classe **Container**. Cette méthode affichera les élements qui sont des **Shape** (via leur méthode *draw(`Graphics&`)*), mais **pas** les autres. De plus elle n'a pas d'argument car l'objet **Graphics** sera crée (une seule fois) par **Container** (on suppose que le constructeur de **Graphics** n'a pas d'argument).

a) Modifier la classe **Container** en conséquence,

b) Ecrire la méthode *draw()* (on rappelle qu'on ne peut pas modifier la classe **Object**).

Q9. Exercice (5 points) On souhaite réaliser un carnet d'adresses. Ecrire **en C++** les deux classes suivantes :

- une classe **Adresse** correspondant à un **nom** (on suppose qu'à un nom correspond **une seule** adresse). On se limitera à quelques champs et on définira les fonctions adéquates (inutile de toutes les écrire : lorsqu'elles sont similaires un commentaire suffira).
- une classe **Carnet** regroupant l'ensemble des adresses. Cette classe devra (au moins) avoir les méthodes suivantes :
  - une méthode pour ajouter un objet de type **Adresse** au carnet (on suppose l'objet déjà créé et correctement initialisé)
  - une méthode pour rechercher une **Adresse** dans le carnet à partir du **nom**. Cette méthode devra renvoyer l'objet.
  - une méthode pour supprimer une **Adresse** du carnet, en spécifiant le **nom** correspondant à l'adresse.

Remarques :

- ces méthodes ne doivent **pas** faire d'entrées-sorties (ni lire depuis, ni écrire sur le Terminal).
- si vous ne vous rappelez plus de la syntaxe exacte mettez des commentaires **détaillés** afin qu'il n'y ait pas d'ambiguité.

Q10. Questions bonus (1 point)

Comment faire pour rendre ce programme plus général :

- en permettant d'avoir plusieurs adresses avec le même nom (typiquement, les membres d'une même famille)
- en permettant à l'utilisateur de rajouter autant de champs qu'il le souhaite à une adresse ?