

# Questions de Cour Java / C++

## Constructeurs / Destructeurs

Qu'est ce qu'un constructeur ?

- Quel est son utilité ?
- Peut-on l'invoquer par une méthode normale ?
- Quel est son type de retour ?

Un constructeur est une fonction appelée lors de la création d'un objet.

Celui-ci est généralement utilisé pour initialiser les attributs de la classe.

Non car elle n'est pas applicable à un objet.

Le type de retour est celui de la classe (déclaration implicite).

Pour quel raison ne définit-on pas toujours de constructeurs par défaut lorsqu'on a définit d'autres constructeurs ?

Parfois il se peut que l'on veuille forcer la construction de l'objet avec des arguments.

En ne définissant pas de constructeur par défaut et en définissant au moins un constructeur, on bloque ainsi la possibilité d'appeler le constructeur par défaut.

En C++ on est parfois obligé de définir un constructeur par défaut, pour quelle raison ?

Lors d'accès de donnée en mémoire, il est envisageable de construire la classe sous la forme canonique de Coplien. Cette factorisation nécessite l'implémentation d'un constructeur par défaut.

Expliquer les principes de création d'une instance en Java et C++, ainsi que les mécanismes de gestion mémoire associés. Même question pour la destruction d'une instance.

Soit `maClass` une classe possédant un constructeur par défaut.

- Création en Java : `maClass objet = new maClass();`

`maClass objet` crée une référence de type `maClass` dont le nom est `objet`.

`new maClass()` permet d'allouer de la mémoire pour stocker les différents attributs de la classe `maClass` et renvoi une référence vers la zone mémoire allouée.

`objet` est donc une référence vers un nouveau objet de type `maClass`.

- Création en C++ : `maClass objet;`

Cette fois si le constructeur par défaut est directement appelé, ainsi l'allocation mémoire a bien été effectuée. On peut cependant le faire dynamiquement comme en Java en utilisant `new maClass()` dans le cas où on a crée un pointeur vers un `maClass`.

Ex :

```
maClass* objet;           //Création du pointeur
```

```
objet = new maClass();    //Allocation car appel du constructeur par défaut.
```

- Destruction en Java:

En Java l'utilisateur ne se charge pas de la destruction des objets qui ne sont plus utilisés, il n'existe pas d'opérateur pour cela. Java s'occupe lui même de la suppression des objets, ce mécanisme s'appelle le "garbage collector" (en français : ramasse miette).

- Destruction en C++:

En C++ pour libérer la mémoire d'un objet créé par `new`, il suffit de faire appel au mot clé `delete`.

Ex:

```
maClass* objet; //Création d'un pointeur vers une instance maClass
```

```
objet = new maClass(); //création en mémoire d'un objet maClass (appel du constructeur par  
//défaut)
```

```
delete objet; //supprime les données en mémoire
```

On supprime souvent les données dans le destructeur qui sera appelé lors de la sortie d'un bloc.

## Généricité

Quelles sont les deux formes de généricité offertes par Java depuis l'introduction de "Java Generics" et que permet Java Generics par rapport à l'ancienne forme de généricité. "Java Generics" permet de définir un type pour les collections et de rendre des fonctions polymorphes. Auparavant, toutes les collections étaient des collections d'Objects ce qui fait que l'on pouvait mélanger les types d'objets au sein de la collection à l'aide de simple casts.

Quel est la différence majeure entre Java Génériques et les patrons de classes C++ en terme de classe génériques comme de Tabelaux<Compte> par exemple ?

Les fonctions et classes manipulant des types génériques doivent être implémentée dans le header car elle seront réécrite dans le code lors de la compilation avec les types qui auront véritablement servis au sein du code.

Exemples:

---

```
#include<iostream> /*Permet d'avoir les flux std::out et std::endl pour
                    l'affichage console
                    */

using namespace std; /* Permet d'écrire "cout" et "endl" au lieu de std::endl et
                    std::out
                    */

//Header = Emplacement des templates (template = patron)

template<typename Type>                                //On définit un nouveau type Type

Type calculerSomme(Type operande1, Type operande2){    //Type est connu
    Type resultat = operande1 + operande2;
    return resultat;
}

int main(){
    int n = 4, p = 12, q = 0;
    q = calculerSomme(n, p); /* On appelle notre fonction calculerSomme comme
                             une fonction normale, au moment de la
                             précompilation, le mot "Type" sera remplacer
                             par le mot "int"
                             */
    cout << "Le resultat est : " << q << endl;    //Affichage du resultat
    return 0;
}
```

---

```
template<typename Type2,typename Type3>
```

```
class A {  
    public: A();  
    ...  
};
```

```
int main(){  
    A<int, double> obj1;    /* Appel du constructeur par défaut de  
                           A<int,double>  
                           */  
    A<double, int> obj2;    /* Appel du constructeur par défaut de  
                           A<double, int>  
                           */  
    return 0;  
}
```

## Accessibilité

Quels sont les différents modificateurs d'accessibilité (en Java et C++) et à quoi donnent-ils accès ?

Les différents modificateurs d'accessibilités sont les suivants:

- **Private** : accessible uniquement au sein même de la classe.
- **Protected** : accessible par les classes qui héritent et celles du même package
- **Par défaut** : accessible par le package uniquement
- **Public** : accessible de partout

Quels modificateurs applique-t-on habituellement aux différents membres d'une classe (en Java et en C++) et pourquoi ?

En règle générale les attributs membres de classe (variables static) sont privés car leurs vue révélerais l'implémentation de la classe et l'utilisateur d'un object se contente de manipuler les objects et non la classe elle même. Il existe cependant des cas où les attributs membres de classes peuvent être visibles, notamment les constantes par exemples.

Les méthodes static sont quant à elles publics car servent de fonctions "générale" pour l'utilisateur : Par exemple `java.lang.Math.sin(double d)`

Quels sont les modes par défaut (en Java et C++) lorsque l'on ne précise aucun modificateur d'accès ?

Les modes par défauts s'appellent les modificateurs "friendly" et rendent les données accessible à l'ensemble du package.

Ces modificateurs sont-ils conservés (en Java et C++) lors de l'héritage ? Et peuvent-ils être modifiés lors de l'héritage ?

En C++:

Il est possible de choisir soi même l'accessibilité des classes en rajoutant le mot clé correspondant dans l'héritage de la classe:

Ex:

```
class A{
    public A();
};

class B : protected A{    //B hérite avec l'accessibilité protégée de A
    public B();
};
```

Pour qu'une classe fille ait les mêmes droits d'accès que sa(ou ses) classes mère (sauf pour les éléments private de la classe mère qui ne peuvent pas exister dans la classe fille), il faut mettre le mot clé **public**.

Pour qu'une classe fille ait les droits protégés de sa classe mère, il faut mettre le mot clé **protected**.

Pour qu'une classe fille ait les droits privés de sa classe même, il faut mettre **private** ou ne rien mettre.

## Polymorphisme (Tombe fréquemment)

Qu'est ce que le polymorphisme et à quoi s'applique la notion de polymorphisme ?

Le polymorphisme consiste à donner divers sens à des méthodes de même symbole (mais pas forcément de même signature) .

Il existe 3 types de polymorphisme.

- Le polymorphisme de surcharge : la signature d'une méthode apparaît dans plusieurs classes sans liens.
- Le polymorphisme par paramètre : on peut définir deux méthodes de même symbole à dans une même classe à condition qu'elle n'ait pas la même signature
- Le polymorphisme d'héritage : la méthode d'une classe est déjà implémentée par celle qu'elle hérite (par défaut : celle d'`Object`)

Sur quels critères distingues-t'on des méthodes surchargées entre elles ?

Expliquez la notion de lien dynamique. Cette notion est-elle disponible dans des langages comme Java et C++ et à quelles conditions ?

Cette notion consiste à définir les méthodes d'un objet en fonction de son instanciation et non en fonction de sa référence.

Cette notion existe en Java mais non en C++.

Ex :

```
public class A {  
    ...  
    public void toto(){  
        System.out.println("totoA");  
    }  
    ...  
}  
  
public class B extends A{  
    ...  
    public void toto(){  
        System.out.println("totoB");  
    }  
  
    public void ok(){  
        System.out.println("OK"); //Méthode non présente dans A  
    }  
}  
  
public static void main(String[ ] args) {  
    A b = new B();  
    b.ok()                // Impossible car ok() n'est pas dans A qui correspond au  
                          // type de b  
    b.toto();             // Fait appel à la méthode B.toto() et non A.toto() donc  
                          // affiche "totoB" et non "totoA"  
}
```

## Membres de classe

Qu'est ce qu'un membre de classe et comment peut-on y accéder ?

Les membres de classes sont les attributs et méthodes d'une classe définits comme static.

Si le nom de la classe concernée est maClasse et qu'une méthode statique est

```
public static void maMethode(){...}; , on accède à cette méthode par  
maClasse.maMethode();
```

Les membres d'instance ont-il accès aux membres de classe ?

Oui, les membres d'instance sont vus sur tous les éléments de la classe.

Les membres de classe ont-il accès aux membres d'instance ?

Non, les méthodes statiques ne peuvent avoir accès qu'aux variables statiques de la classe car les autres variables n'existent que lors de la création d'un objet.

Faites le lien entre la notion de membres de classe et la notion de métaclasse

Une métaclasse est une classe regroupant des informations sur une autre classe.

Au lieu d'utiliser une métaclasse pour regrouper les informations d'une classe, on pourrait définir ces informations comme membres de classe car elles dépendent uniquement de la classe considérée et non des différentes instances.

## Copie par référence ou copie profonde

Quelle est la nuance entre la copie par référence et la copie profonde lorsque l'on veut copier des objets dans une collection par exemple ?

En copiant par référence les éléments un à un d'une collection, toute modification faite sur un objet de la première collection sera visible sur la seconde, ce qui n'est pas le cas de la copie en profondeur puisque celle-ci rends les objets totalement indépendants dans les deux collections.

Quels sont les avantages et/ou les inconvénients de chacune ces méthodes ?

Tout dépend de ce que l'on veut faire, si on considère que les objets avec lesquels on travaillent sont les mêmes, on fait des copies par référence. Si on désire vraiment faire une copie conforme et indépendante du modèle, alors il faut faire une copie profonde.

En mémoire, la copie par référence est moins coûteuse car elle prend la taille d'une adresse mémoire alors que la copie profonde doit réallouer de la mémoire afin de copier chaque élément de chacun des objets.

Exemple de copie profonde :

```
//Les données sont copiées mais on manipule un autre objet
public Point getCopy(Point p) {
    Point p = new Point();
    p.setX(x);
    p.setY(y);
    return p;
}
```

Exemple de copie par référence :

```
//On renvoi le même objet que celui avec lequel on travail
public Point getCopy(Point p) {
    return p;
}
```