Les principales différences entre C++ et Java

Hélène Toussaint, mai 2012

Ce document a pour but de synthétiser les différences "*les plus*" (à mon sens...) importantes entre C++ et Java. Il n'est donc pas exhaustif et ne référence pas les différences dans les bibliothèques auxiliaires (STL / collections ...). Il s'adresse à des développeurs C++ ou Java déjà à l'aise avec les concepts de la POO (Programmation Orienté Objet).

Pour une étude approfondie du langage Java (resp. C++) je conseille Programmer en Java (resp. Programmer en langage C++) de Claude Delannoy aux Editions Eyrolles.

Thème	**C++**	**Java**
Compilation et portabilité	La compilation (qui passe par plusieurs étapes : préprocessing, création des fichiers *objet*, éditions des liens) fournit un fichier binaire (ou exécutable) directement utilisable sur la machine sur laquelle il a été créé. Pour l'utiliser sur une autre machine il faut bien souvent recompilé le source...	La compilation d'un code source produit un code intermédiaire (et non un code binaire) qui a besoin de la JVM (Java Virtual Machine) pour être interprété. Ce code est portable sur toute machine disposant de la JVM (attention quand même aux différences entre les versions de la JVM).
Conception	Permet la programmation orienté objet (POO) et la prog. procédurale : aucune obligation de créer des classes	Java est presque pure POO : les classes sont obligatoires MAIS - il existe des types primitifs (*int*, *double*, *boolean*, ...) - on peut créer des méthodes de classes statiques (*static*) qui sont utilisables sans instancier d'objet
Variable Constantes	Variables déclarées avec le mot-clé *const*	Variables déclarées avec le mot-clé *final*
Méthodes constantes	Le mot-clé *const* est placé après la déclaration de la méthode. Seules les méthodes constantes peuvent agir sur une objet constant	N'existe pas. (*final* appliqué à une méthode a un sens complètement différent -> voir plus bas)
Conversion de type	Toutes les conversions numériques de type par affectation sont acceptées quitte à dégrader fortement l'information	Seules les conversions qui relèvent de promotion numérique sont autorisées par affectation (les autres doivent être demandées explicitement)
Définition / déclaration de classes	Généralement la déclaration d'une classe est dans un fichier "*.h*" distinct du fichier d'implémentation (ou définition) "*.cpp*"	Les déclarations et définitions sont dans le même fichier "*.java*"
Pointeur	Un pointeur représente l'adresse de l'objet pointé, on le manipule comme une variable	Il n'existe pas de pointeur en Java

Instanciation d'un objet	Deux solutions : - par sa déclaration -> objet automatique - par *new* -> objet dynamique (pointeur)	Une seule solution : - par *new* et on dispose alors d'une référence sur l'objet (et non d'un pointeur comme en C++)

Instanciation - constructeur par défaut	parenthèses non obligatoires :
	`A a;`
	`A * a = new A;`
	`A * a = new A();`
		parenthèses obligatoires, seule déclaration possible :
		`A * a = new A();`

Gestion mémoire	- Toutes les variables allouées par *new* doivent être explicitement détruites par *delete* - La déclaration d'un objet entraîne la réservation d'un emplacement mémoire pour l'objet	- Java dispose d'un *ramasse miettes :* le ramasse miettes détecte les objets qui ne sont plus référencés et les détruit -> *delete* n'existe pas en Java - la déclaration d'un objet entraîne la réservation d'un emplacement mémoire pour une référence à l'objet (et non pour l'objet) la mémoire dédiée à l'objet sera effectivement allouée avec *new*
Passage d'arguments	3 possibilités : - par valeur (les modifications restent locales à la fonction) - par adresse *ou pointeur* (les modifications sont effectives hors de la fonction) - par référence (les modifications sont effectives hors de la fonction)	1 seule possibilité : par valeur MAIS - pour les types primitifs les modifications sont donc locales à la fonction, - pour les objets (le nom de l'objet étant une référence) les modifications sont effectives hors de la fonction
Init. par défaut des champs d'un objet	Non	Oui (à 0, *false* ou NULL)
Affectation d'objet `a = b`	Pour les objets automatiques copie superficielle (recopie des valeurs des différents champs)	Copie de la référence uniquement
Valeurs d'argument par défaut	Oui : la déclaration d'une fonction peut contenir des arguments par défaut : `void f ( int n = 0);`	N'existe pas

Allocation de tableaux	2 possibilités :
	- tableaux statiques :
	`int t [10];`
	- tableaux dynamiques :
	`int * t = new int [10];`
		Toujours dynamique -> avec new :
		`int t [] = new int [10];`
		ou
		`int [] t = new int [10];`

Taille d'un tableau	Il n'existe pas de champ propre à un tableau, on doit stocker la taille dans une autre variable	Les tableaux se manipulent comme des objets. Il existe un champ par défaut "*length*" auquel on accède par `t.length` (où t est le nom du tableau)
Tableaux en mémoire	En C++ les tableaux à plusieurs indices déclarés de manière statique sont contigus en mémoire Ex : `int t [10][25]` crée 250 cases contigües	En Java les tableaux à plusieurs indices fractionnent la mémoire : `int t [][] = new int [10][25]` non contigus

Héritage : dérivation	Trois possibilités :
	`class B : public A`
	`class B : private A`
	`class B : protected A`
		Une seule possibilité :
		`class B extends A` qui correspond à la dérivation publique de C++

Héritage multiple	Oui	Non (mais il existe la notion d'interface)

Constructeur d'une classe fille	- Doit prendre en charge la partie spécifique de la classe fille : le constructeur de la classe mère est automatiquement appelé par C++
	- si le constructeur de la classe mère nécessite des arguments on peut les préciser dans l'entête *via* ":"
	`B::B( int arg1, double arg2) : A (arg1)`
		- Doit prendre en charge l'intégralité de la construction de l'objets
		- Peut appeler le constructeur de la classe mère via le mot-clé *super*(), dans ce cas ce doit être la première instruction
		`B( int arg1, double arg2){`
		`  super(arg1);`
		`  //...`
		`}`

Dérivations successives	oui	oui (dans ce cas *super* désigne le constructeur la classe directement supérieure)
Membres *protected*	Accessibles seulement aux classes dérivées	Accessibles aux classes dérivées et aussi aux classes du même paquetage

Redéfinition de méthode	Appel à la méthode de la classe mère via l'opérateur de résolution de portée "::"
	`void B::affiche (){`
	`  A::affiche();`
	`  //...`
	`}`
		Appel à la méthode de la classe mère *via* le mot clé super
		`class B extends A {`
		`  public void affiche() {`
		`  super.affiche();`
		`  System.out.println("classe B");`
		`  }`
		`}`

Polymorphisme	Mis en œuvre uniquement si : - ligature dynamique (utilisation de pointeurs sur les objets) - méthodes virtuelles (utilisation du mot clé *virtual*)	Toujours effectif (la ligature est toujours dynamique)
Classes abstraites	Une classe est abstraite (ne peut être instanciée) si elle contient une méthode virtuelle pure	Mot-clé *abstract :* - soit placé devant la classe - soit appliqué à une méthode (ce qui rend immédiatement la classe abstraite)
Arguments de la ligne de commande	`int main (int argc, char * argv[])` `argc` donne le nombre d'élément (nom du programme compris) et `argv` contient les arguments	`public static void main (String[] args)` On accède aux nb d'éléments par args.length, le nom du pgr n'est pas considéré comme un argument
Gestion des flux	Très simple grâce aux opérateurs **<<** et **>>**	Beaucoup plus complexe : plusieurs classes, pas de fonctionnalité de lecture formatée

Quelques fonctionnalités qui n'existent pas en C++

Super-classe **`Object`**	Toutes les classes dérivent implicitement de cette super classe -> elles en possèdent toutes les fonctionnalités (par ex. toString() fournit le nom de la classe et son @)
Classes et méthodes *final*	- Une méthode *final* ne peut pas être redéfinie dans une classe fille - Une classe *final* ne peut plus être dérivée

Interface	Une interface ne définit que des en-tête de méthodes et des constantes
	`public interface I {`
	`  // ... en-tête des méthodes`
	`}`
	Une classe peut implémenter une ou plusieurs interfaces
	`class A implements I  {`
	`  //...`
	`}`