

Ch. 1 - Différences entre Java et C++

Langage C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique



26 octobre 2018

Table des matières

- 1 Introduction
- 2 Généralités
- 3 Gestion mémoire
- 4 Expressions et fonctions
- 5 Conteneurs
- 6 Programmation orientée objet
- 7 Conclusion

Table des matières

1 Introduction

2 Généralités

3 Gestion mémoire

4 Expressions et fonctions

5 Conteneurs

6 Programmation orientée objet

7 Conclusion

Table des matières

- 1 Introduction
- 2 Généralités
- 3 Gestion mémoire
- 4 Expressions et fonctions
- 5 Conteneurs
- 6 Programmation orientée objet
- 7 Conclusion

Table des matières

- 1 Introduction
- 2 Généralités
- 3 Gestion mémoire
- 4 Expressions et fonctions
- 5 Conteneurs
- 6 Programmation orientée objet
- 7 Conclusion

Table des matières

- 1 Introduction
- 2 Généralités
- 3 Gestion mémoire
- 4 Expressions et fonctions
- 5 Conteneurs
- 6 Programmation orientée objet
- 7 Conclusion

Table des matières

- 1 Introduction
- 2 Généralités
- 3 Gestion mémoire
- 4 Expressions et fonctions
- 5 Conteneurs
- 6 Programmation orientée objet
- 7 Conclusion

Table des matières

- 1 Introduction
- 2 Généralités
- 3 Gestion mémoire
- 4 Expressions et fonctions
- 5 Conteneurs
- 6 Programmation orientée objet
- 7 Conclusion

Introduction

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.

■ Syntaxes similaires

Overview (1/2)

- Deux langages très présents dans l'entreprise.
- Très différents, avec leurs avantages et leurs inconvénients.
- Choisir son langage en connaissance de cause.

C++

- Bâti sur le C, fournit principalement la POO.
- Mécanisme de manipulation d'adresse

Java

- Inspiré du C/C++ : certaines fonctionnalités « sujettes aux erreurs » (p. ex., surcharge d'opérateur) sont supprimées.
- Syntaxes similaires

Overview (2/2)

- **Java** : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -o3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Overview (2/2)

- Java : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -o3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Overview (2/2)

- Java : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -o3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Overview (2/2)

- Java : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -O3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Overview (2/2)

- Java : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -O3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Overview (2/2)

- Java : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -O3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Overview (2/2)

- Java : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -O3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Overview (2/2)

- Java : librairie standard très riche (bdd, crypto, xml, etc.), documentation standard (javadoc).
- C++ : librairie standard moins riche, besoin parfois de recourir à des librairies tierces.
- C++ : équivalent javadoc : doxygen (non standard).
- En général, un « bon » code C++ est légèrement plus performant qu'un « bon » code Java.
 - Uniquement si le code C++ est bien écrit, en optimisation -O3.
 - Programmer de « bons » codes requiert une très bonne maîtrise du langage (en Java comme en C++).
- Idée : le compilateur C++ produit du langage machine directement exécuté, Java produit du bytecode interprété par la JVM.
- Pas de compilateur standard en C/C++.

Généralités

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Compilation

■ Principes de compilation très différents.

Java	C++
La compilation produit du bytecode	Compilation en trois temps produisant du langage machine
La JVM interprète le bytecode	Code directement exécuté sur la machine
Le bytecode est portable d'une machine à l'autre	Code non portable
Pas de recompilation nécessaire	Nécessité de recompiler et d'adapter les bibliothèques si nécessaires.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Paradigmes

Java

En quasi-totalité OO

Le programmeur doit placer l'intégralité du code dans des classes

`static` permet « d'émuler » le procédural (augmente les performances au prix de la POO)

Existence de types primitifs au comportement différent

C++

Procédural et OO

Pas d'obligation de créer des classes

POO aux performances identiques au procédural, sauf si usage de polymorphisme

Types de base au comportement identique aux autres types.

Organisation du code

Java	C++
L'intégralité du code d'une classe est placé dans un unique fichier <code>.java</code>	Déclarations dans un <code>.h</code> , définitions dans un <code>.cpp</code> associé
Utilisation package \simeq répertoire	Utilisation de namespaces
Utilisation via le mot-clé package	Utilisation via un bloc namespace
Importation via <code>import</code>	Importation en précompilation via <code>#include</code>
<code>import static</code> possible	<code>using namespace</code> disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Organisation du code

Java

L'intégralité du code d'une classe est placé dans un unique fichier `.java`

Utilisation package \simeq répertoire

Utilisation via le mot-clé package

Importation via `import`

`import static` possible

C++

Déclarations dans un `.h`, définitions dans un `.cpp` associé

Utilisation de namespaces

Utilisation via un bloc namespace

Importation en précompilation via `#include`

`using namespace` disponible

Gestion mémoire

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Classes d'allocation et valeur par défaut

Java	C++
Pas de notion de classe d'allocation	Trois classes d'allocation principales
En général, les variables sont locales	La mémoire allouée sur la pile est locale
Durée de vie « d'un bloc »	Durée de vie dépendant de la classe
Toutes les variables ont une valeur par défaut	Pointeurs constants possibles (références constantes)
Pas de valeur par défaut aux paramètres	Possibilité de valeur par défaut aux paramètres

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Toute type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Tout type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Toute type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Toute type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Tout type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Tout type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Tout type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Toute type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

Création / destruction

Java

La mémoire est allouée à l'affectation / instantiation

Toute type de base est détruit en fin de bloc

Destruction asynchrone d'objets

Pas de mécanisme de destructeur

C++

L'allocation mémoire dépend de la classe d'allocation

Toute variable automatique est détruite en fin de bloc

Destruction synchrone explicite par l'utilisateur

Mécanisme de destructeur

En C++ : résumé

■ Classes d'allocation :

- 1 Statique (segment de données)
- 2 Automatique (pile)
- 3 Dynamique (tas) : `new`

Hygiène C++

- Hygiène C++ : Pas de `new`
- Utilisation de « pointeurs intelligents » en C++ pour gestion « transparente » de la mémoire

En C++ : résumé

■ Classes d'allocation :

- 1 Statique (segment de données)
- 2 Automatique (pile)
- 3 Dynamique (tas) : `new`

Hygiène C++

- Hygiène C++ : Pas de `new`
- Utilisation de « pointeurs intelligents » en C++ pour gestion « transparente » de la mémoire

En C++ : résumé

■ Classes d'allocation :

- 1 Statique (segment de données)
- 2 Automatique (pile)
- 3 Dynamique (tas) : `new`

Hygiène C++

- Hygiène C++ : Pas de `new`
- Utilisation de « pointeurs intelligents » en C++ pour gestion « transparente » de la mémoire

En C++ : résumé

■ Classes d'allocation :

- 1 Statique (segment de données)
- 2 Automatique (pile)
- 3 Dynamique (tas) : `new`

Hygiène C++

- Hygiène C++ : Pas de `new`
- Utilisation de « pointeurs intelligents » en C++ pour gestion « transparente » de la mémoire

En C++ : résumé

- Classes d'allocation :
 - 1 Statique (segment de données)
 - 2 Automatique (pile)
 - 3 Dynamique (tas) : `new`

Hygiène C++

- Hygiène C++ : Pas de `new`
- Utilisation de « pointeurs intelligents » en C++ pour gestion « transparente » de la mémoire

En C++ : résumé

- Classes d'allocation :
 - 1 Statique (segment de données)
 - 2 Automatique (pile)
 - 3 Dynamique (tas) : `new`

Hygiène C++

- Hygiène C++ : Pas de `new`
- Utilisation de « pointeurs intelligents » en C++ pour gestion « transparente » de la mémoire

En C++ : résumé

- Classes d'allocation :
 - 1 Statique (segment de données)
 - 2 Automatique (pile)
 - 3 Dynamique (tas) : `new`

Hygiène C++

- Hygiène C++ : Pas de `new`
- Utilisation de « pointeurs intelligents » en C++ pour gestion « transparente » de la mémoire

Pointeurs et références

Java

- Uniquement des pointeurs (= adresses) d'objets

C++

- Pointeurs et références (= alias) pour tout type
- Un pointeur est manipulé comme une variable (accès au champs via \rightarrow)
 - Arithmétique de pointeur
- Une référence est manipulée comme un pointeur constant

Pointeurs et références

Java

- Uniquement des pointeurs (= adresses) d'objets

C++

- Pointeurs et références (= alias) pour tout type
- Un pointeur est manipulé comme une variable (accès au champs via \rightarrow)
 - Arithmétique de pointeur
- Une référence est manipulée comme un pointeur constant

Pointeurs et références

Java

- Uniquement des pointeurs (= adresses) d'objets

C++

- Pointeurs et références (= alias) pour tout type
- Un pointeur est manipulé comme une variable (accès au champs via \rightarrow)
 - Arithmétique de pointeur
- Une référence est manipulée comme un pointeur constant

Pointeurs et références

Java

- Uniquement des pointeurs (= adresses) d'objets

C++

- Pointeurs et références (= alias) pour tout type
- Un pointeur est manipulé comme une variable (accès au champs via `->`)
 - Arithmétique de pointeur
- Une référence est manipulée comme un pointeur constant

Pointeurs et références

Java

- Uniquement des pointeurs (= adresses) d'objets

C++

- Pointeurs et références (= alias) pour tout type
- Un pointeur est manipulé comme une variable (accès au champs via \rightarrow)
 - Arithmétique de pointeur
- Une référence est manipulée comme un pointeur constant

Pointeurs et références

Java

- Uniquement des pointeurs (= adresses) d'objets

C++

- Pointeurs et références (= alias) pour tout type
- Un pointeur est manipulé comme une variable (accès au champs via \rightarrow)
 - Arithmétique de pointeur
- Une référence est manipulée comme un pointeur constant

Pointeurs et références

Java

- Uniquement des pointeurs (= adresses) d'objets

C++

- Pointeurs et références (= alias) pour tout type
- Un pointeur est manipulé comme une variable (accès au champs via \rightarrow)
 - Arithmétique de pointeur
- Une référence est manipulée comme un pointeur constant

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Sorties en mémoire

Java

- Exception lancée automatiquement
- Fin brutale du programme

C++

- Pas de comportement par défaut
- Pas de stacktrace
- En C++, une sortie en mémoire peut
 - causer une erreur de segmentation
 - écraser la valeur d'une variable contiguë
 - écraser l'adresse de retour d'une fonction
 - écraser des adresses dans la pile d'exécution, etc.

Expressions et fonctions

Conversions

Java

Conversions non dégradantes implicites acceptées

Pas de tronquage

Possibilité de convertir une fille en mère

Pas d'autre conversions en relation d'héritage

Pas d'autres conversions entre classes possibles

C++

Des conversions dégradantes implicites sont autorisées

Tronquage possible

Possibilité de conversions de fille en mère et inversement

`dynamic_cast` : « conversions d'héritage »

Conversions entre classes et type de base via « constructeurs de conversion » et surcharge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Conversions

Java

Conversions non dégradantes implicites acceptées

Pas de tronquage

Possibilité de convertir une fille en mère

Pas d'autre conversions en relation d'héritage

Pas d'autres conversions entre classes possibles

C++

Des conversions dégradantes implicites sont autorisées

Tronquage possible

Possibilité de conversions de fille en mère et inversement

`dynamic_cast` : « conversions d'héritage »

Conversions entre classes et type de base via « constructeurs de conversion » et surcharge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Conversions

Java

Conversions non dégradantes implicites acceptées

Pas de tronquage

Possibilité de convertir une fille en mère

Pas d'autre conversions en relation d'héritage

Pas d'autres conversions entre classes possibles

C++

Des conversions dégradantes implicites sont autorisées

Tronquage possible

Possibilité de conversions de fille en mère et inversement

`dynamic_cast` : « conversions d'héritage »

Conversions entre classes et type de base via « constructeurs de conversion » et surcharge d'opérateur de cast

Conversions

Java

Conversions non dégradantes
implicites acceptées

Pas de tronquage

Possibilité de convertir une fille
en mère

Pas d'autre conversions en rela-
tion d'héritage

Pas d'autres conversions entre
classes possibles

C++

Des conversions dégradantes
implicites sont autorisées

Tronquage possible

Possibilité de conversions de
fille en mère et inversement

`dynamic_cast` : « conversions
d'héritage »

Conversions entre classes et
type de base via « construc-
teurs de conversion » et sur-
charge d'opérateur de cast

Passage d'arguments

Java	C++
Les types primitifs sont passés par valeur	Tous les types sont passés par valeur (par défaut)
Les objets sont passés par adresse	Possibilité de passage par adresse et par référence (C++)
On ne peut pas changer ce comportement	Possibilité de changer le comportement si demande explicite
Aucune conversion dégradante n'est effectuée	Des conversions sont possibles
Les paramètres sont évalués de la gauche vers la droite	Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Passage d'arguments

Java

Les types primitifs sont passés par valeur

Les objets sont passés par adresse

On ne peut pas changer ce comportement

Aucune conversion dégradante n'est effectuée

Les paramètres sont évalués de la gauche vers la droite

C++

Tous les types sont passés par valeur (par défaut)

Possibilité de passage par adresse et par référence (C++)

Possibilité de changer le comportement si demande explicite

Des conversions sont possibles

Pas de priorité d'évaluation en général

Constantes

Java	C++
Variables déclarées via <code>final</code>	Variables déclarées via <code>const</code>
Pas de notion de pointeur constant	Pointeurs constants possibles (références constantes)
Pas de notion fonctions constantes	Possibilité de fonctions constantes
Les objets sont passés par adresse	Possibilité de passage par adresse et par référence (C++)
Les énumérations déclarent des membres constants	

Constantes

Java	C++
Variables déclarées via <code>final</code>	Variables déclarées via <code>const</code>
Pas de notion de pointeur constant	Pointeurs constants possibles (références constantes)
Pas de notion fonctions constantes	Possibilité de fonctions constantes
Les objets sont passés par adresse	Possibilité de passage par adresse et par référence (C++)
Les énumérations déclarent des membres constants	

Constantes

Java	C++
Variables déclarées via <code>final</code>	Variables déclarées via <code>const</code>
Pas de notion de pointeur constant	Pointeurs constants possibles (références constantes)
Pas de notion fonctions constantes	Possibilité de fonctions constantes
Les objets sont passés par adresse	Possibilité de passage par adresse et par référence (C++)
Les énumérations déclarent des membres constants	

Constantes

Java

Variables déclarées via `final`

Pas de notion de pointeur constant

Pas de notion fonctions constantes

Les objets sont passés par adresse

C++

Variables déclarées via `const`

Pointeurs constants possibles (références constantes)

Possibilité de fonctions constantes

Possibilité de passage par adresse et par référence (C++)

Les énumérations déclarent des membres constants

Constantes

Java

Variables déclarées via `final`

Pas de notion de pointeur constant

Pas de notion fonctions constantes

Les objets sont passés par adresse

C++

Variables déclarées via `const`

Pointeurs constants possibles (références constantes)

Possibilité de fonctions constantes

Possibilité de passage par adresse et par référence (C++)

Les énumérations déclarent des membres constants

Constantes

Java

Variables déclarées via `final`

Pas de notion de pointeur constant

Pas de notion fonctions constantes

Les objets sont passés par adresse

C++

Variables déclarées via `const`

Pointeurs constants possibles (références constantes)

Possibilité de fonctions constantes

Possibilité de passage par adresse et par référence (C++)

Les énumérations déclarent des membres constants

Constantes

Java

Variables déclarées via `final`

Pas de notion de pointeur constant

Pas de notion fonctions constantes

Les objets sont passés par adresse

C++

Variables déclarées via `const`

Pointeurs constants possibles (références constantes)

Possibilité de fonctions constantes

Possibilité de passage par adresse et par référence (C++)

Les énumérations déclarent des membres constants

Constantes

Java	C++
Variables déclarées via <code>final</code>	Variables déclarées via <code>const</code>
Pas de notion de pointeur constant	Pointeurs constants possibles (références constantes)
Pas de notion fonctions constantes	Possibilité de fonctions constantes
Les objets sont passés par adresse	Possibilité de passage par adresse et par référence (C++)
Les énumérations déclarent des membres constants	

Constantes

Java	C++
Variables déclarées via <code>final</code>	Variables déclarées via <code>const</code>
Pas de notion de pointeur constant	Pointeurs constants possibles (références constantes)
Pas de notion fonctions constantes	Possibilité de fonctions constantes
Les objets sont passés par adresse	Possibilité de passage par adresse et par référence (C++)
Les énumérations déclarent des membres constants	

Constantes

Java	C++
Variables déclarées via <code>final</code>	Variables déclarées via <code>const</code>
Pas de notion de pointeur constant	Pointeurs constants possibles (références constantes)
Pas de notion fonctions constantes	Possibilité de fonctions constantes
Les objets sont passés par adresse	Possibilité de passage par adresse et par référence (C++)
Les énumérations déclarent des membres constants	

Conteneurs

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uni-
quement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dyna-
mique

```
int t[10];
```

```
int t[] = new int[10];
```

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uniquement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dynamique

```
int t[10];
```

```
int t[] = new int[10];
```

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uniquement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dynamique

```
int t[10];
```

```
int t[] = new int[10];
```

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uniquement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dynamique

```
int t[10];
```

```
int t[] = new int[10];
```

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uniquement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dynamique

```
int t[10];
```

```
int t[] = new int[10];
```

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uniquement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dynamique

```
int t[10];
```

```
int t[] = new int[10];
```

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uniquement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dynamique

```
int t[10];
```

```
int t[] = new int[10];
```

Tableaux

Java

Pas d'obligation de taille

```
int[] t; ok
```

Tracé de taille via l'attribut
length

Instanciation dynamique uniquement

```
int[] t = new int[10];
```

C++

Spécification de taille obligatoire

```
int k[] : ko
```

```
int k[] = new int[] : ko
```

Pas de tracé de taille en général

Instanciation statique ou dynamique

```
int t[10];
```

```
int t[] = new int[10];
```


Conteneurs

Java

API Collection

ArrayList, LinkedList, etc.

Héritage entre conteneurs

C++

Conteneurs standards

vector, list, etc.

Pas d'héritage entre conteneurs

Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java

API Collection

ArrayList, LinkedList, etc.

Héritage entre conteneurs

C++

Conteneurs standards

vector, list, etc.

Pas d'héritage entre conteneurs

Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java

API Collection

ArrayList, LinkedList, **etc.**

Héritage entre conteneurs

C++

Conteneurs standards

vector, list, etc.

Pas d'héritage entre conteneurs

Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java

API `Collection`

`ArrayList`, `LinkedList`, **etc.**

Héritage entre conteneurs

C++

Conteneurs standards

`vector`, `list`, **etc.**

Pas d'héritage entre conteneurs

Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java

API `Collection`

`ArrayList`, `LinkedList`, **etc.**

Héritage entre conteneurs

C++

Conteneurs standards

`vector`, `list`, **etc.**

Pas d'héritage entre conteneurs

Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java

API `Collection`

`ArrayList`, `LinkedList`, **etc.**

Héritage entre conteneurs

C++

Conteneurs standards

`vector`, `list`, **etc.**

Pas d'héritage entre conteneurs

Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java

API Collection

ArrayList, LinkedList, **etc.**

Héritage entre conteneurs

C++

Conteneurs standards

vector, list, **etc.**

Pas d'héritage entre conteneurs

Les templates permettent
d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java	C++
API <code>Collection</code>	Conteneurs standards
<code>ArrayList</code> , <code>LinkedList</code> , etc.	<code>vector</code> , <code>list</code> , etc.
Héritage entre conteneurs	Pas d'héritage entre conteneurs
	Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java	C++
API <code>Collection</code>	Conteneurs standards
<code>ArrayList</code> , <code>LinkedList</code> , etc.	<code>vector</code> , <code>list</code> , etc.
Héritage entre conteneurs	Pas d'héritage entre conteneurs
	Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java	C++
API <code>Collection</code>	Conteneurs standards
<code>ArrayList</code> , <code>LinkedList</code> , etc.	<code>vector</code> , <code>list</code> , etc.
Héritage entre conteneurs	Pas d'héritage entre conteneurs
	Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

Conteneurs

Java	C++
API <code>Collection</code>	Conteneurs standards
<code>ArrayList</code> , <code>LinkedList</code> , etc.	<code>vector</code> , <code>list</code> , etc.
Héritage entre conteneurs	Pas d'héritage entre conteneurs
	Les templates permettent d'émuler le comportement

Hygiène C++

- Ne pas utiliser les tableaux « à la mode C »
- Utiliser les conteneurs standards

POO

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
 - Affectation
 - Via `()` : doit correspondre à un prototype
 - Via `()` : pas de conversion dégradante
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
 - Affectation
 - Via `()` : doit correspondre à un prototype
 - Via `{}` : pas de conversion dégradante
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
 - Affectation
 - Via `()` : doit correspondre à un prototype
 - Via `{}` : pas de conversion dégradante
 - Dynamique : via `new`
 - Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
 - Affectation
 - Via `()` : doit correspondre à un prototype
 - Via `{}` : pas de conversion dégradante
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
 - Affectation
 - Via `()` : doit correspondre à un prototype
 - Via `{ }` : pas de conversion dégradante
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
 - Affectation
 - Via `()` : doit correspondre à un prototype
 - Via `{ }` : pas de conversion dégradante
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Instanciation

Java

- Types primitifs : affectation
- Objets : `new`

C++

- Automatique et statique
 - Affectation
 - Via `()` : doit correspondre à un prototype
 - Via `{}` : pas de conversion dégradante
- Dynamique : via `new`
- Comportements différents selon le type d'instanciation

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
- Possibilité de réimplémenter le mécanisme via constructeurs de copie
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
- Possibilité de réimplémenter le mécanisme via constructeurs de copie
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
- Possibilité de réimplémenter le mécanisme via constructeurs de copie
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
- Possibilité de réimplémenter le mécanisme via constructeurs de recopie
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
 - Pas les champs statiques et dynamiques
- Possibilité de réimplémenter le mécanisme via constructeurs de recopie
 - Parfois indispensable
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
 - Pas les champs statiques et dynamiques
- Possibilité de réimplémenter le mécanisme via constructeurs de recopie
 - Parfois indispensable
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
 - Pas les champs statiques et dynamiques
- Possibilité de réimplémenter le mécanisme via constructeurs de recopie
 - Parfois indispensable
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
 - Pas les champs statiques et dynamiques
- Possibilité de réimplémenter le mécanisme via constructeurs de recopie
 - Parfois indispensable
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
 - Pas les champs statiques et dynamiques
- Possibilité de réimplémenter le mécanisme via constructeurs de recopie
 - Parfois indispensable
- Possibilité de désactiver la copie implicite d'objets

Copie d'objets

Java

- `a = b` copie l'adresse de l'objet
- Copies des données via l'API `Cloneable`
 - Précautions à prendre

C++

- `a = b` copie tous les champs automatiques de l'objet
 - Pas les champs statiques et dynamiques
- Possibilité de réimplémenter le mécanisme via constructeurs de recopie
 - Parfois indispensable
- Possibilité de désactiver la copie implicite d'objets

Héritage et polymorphisme

Java	C++
Classes et interfaces	Héritage multiple (classes)
	Précautions à prendre
Polymorphisme natif	Polymorphisme activé explicitement par les prototypes et les classes d'allocation
	Motivation : performances
Accès aux superclasses via <code>super</code>	Accès aux superclasses via <code>::</code> (opérateur de résolution de portée)

Héritage et polymorphisme

Java	C++
Classes et interfaces	Héritage multiple (classes)
	Précautions à prendre
Polymorphisme natif	Polymorphisme activé explicitement par les prototypes et les classes d'allocation
	Motivation : performances
Accès aux superclasses via <code>super</code>	Accès aux superclasses via <code>::</code> (opérateur de résolution de portée)

Héritage et polymorphisme

Java	C++
Classes et interfaces	Héritage multiple (classes) Précautions à prendre
Polymorphisme natif	Polymorphisme activé explicitement par les prototypes et les classes d'allocation Motivation : performances
Accès aux superclasses via <code>super</code>	Accès aux superclasses via <code>::</code> (opérateur de résolution de portée)

Héritage et polymorphisme

Java	C++
Classes et interfaces	Héritage multiple (classes) Précautions à prendre
Polymorphisme natif	Polymorphisme activé explicitement par les prototypes et les classes d'allocation Motivation : performances
Accès aux superclasses via <code>super</code>	Accès aux superclasses via <code>::</code> (opérateur de résolution de portée)

Héritage et polymorphisme

Java

Classes et interfaces

Polymorphisme natif

Accès aux superclasses via
`super`

C++

Héritage multiple (classes)

Précautions à prendre

Polymorphisme activé explicitement par les prototypes et les classes d'allocation

Motivation : performances

Accès aux superclasses via `::`
(opérateur de résolution de portée)

Héritage et polymorphisme

Java

Classes et interfaces

Polymorphisme natif

Accès aux superclasses via
`super`

C++

Héritage multiple (classes)

Précautions à prendre

Polymorphisme activé explicitement par les prototypes et les classes d'allocation

Motivation : performances

Accès aux superclasses via `::` (opérateur de résolution de portée)

Héritage et polymorphisme

Java

Classes et interfaces

Polymorphisme natif

Accès aux superclasses via
`super`

C++

Héritage multiple (classes)

Précautions à prendre

Polymorphisme activé explicitement par les prototypes et les classes d'allocation

Motivation : performances

Accès aux superclasses via `::`
(opérateur de résolution de portée)

Héritage et polymorphisme

Java	C++
Classes et interfaces	Héritage multiple (classes) Précautions à prendre
Polymorphisme natif	Polymorphisme activé explicitement par les prototypes et les classes d'allocation Motivation : performances
Accès aux superclasses via <code>super</code>	Accès aux superclasses via <code>::</code> (opérateur de résolution de portée)

Héritage et polymorphisme

Java	C++
Classes et interfaces	Héritage multiple (classes) Précautions à prendre
Polymorphisme natif	Polymorphisme activé explicitement par les prototypes et les classes d'allocation Motivation : performances
Accès aux superclasses via <code>super</code>	Accès aux superclasses via <code>::</code> (opérateur de résolution de portée)

Classes abstraites

Java

- Une classe abstraite est déclarée comme `abstract`
- Les interfaces sont abstraits

C++

- Une classe abstraite contient au moins une méthode virtuelle pure
- `virtual retour nom(params) = 0;`
- Non instanciable en Java comme en C++

Classes abstraites

Java

- Une classe abstraite est déclarée comme `abstract`
- Les interfaces sont abstraits

C++

- Une classe abstraite contient au moins une méthode virtuelle pure
- `virtual retour nom(params) = 0;`
- Non instanciable en Java comme en C++

Classes abstraites

Java

- Une classe abstraite est déclarée comme `abstract`
- Les interfaces sont abstraits

C++

- Une classe abstraite contient au moins une méthode virtuelle pure
- `virtual retour nom(params) = 0;`
- Non instanciable en Java comme en C++

Classes abstraites

Java

- Une classe abstraite est déclarée comme `abstract`
- Les interfaces sont abstraits

C++

- Une classe abstraite contient au moins une méthode virtuelle pure
- `virtual retour nom(params) = 0;`
- Non instanciable en Java comme en C++

Classes abstraites

Java

- Une classe abstraite est déclarée comme `abstract`
- Les interfaces sont abstraits

C++

- Une classe abstraite contient au moins une méthode virtuelle pure
- `virtual retour nom(params) = 0;`
- Non instanciable en Java comme en C++

Classes abstraites

Java

- Une classe abstraite est déclarée comme `abstract`
- Les interfaces sont abstraits

C++

- Une classe abstraite contient au moins une méthode virtuelle pure
- `virtual retour nom(params) = 0;`
- Non instanciable en Java comme en C++

Classes abstraites

Java

- Une classe abstraite est déclarée comme `abstract`
- Les interfaces sont abstraits

C++

- Une classe abstraite contient au moins une méthode virtuelle pure
- `virtual retour nom(params) = 0;`
- Non instanciable en Java comme en C++

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
 - Pas de gestion automatique de la mémoire
- Possibilité de fragmentation de la mémoire

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
- Possibilité de fragmentation de la mémoire

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
- Possibilité de fragmentation de la mémoire

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
- Possibilité de fragmentation de la mémoire

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
 - S'il n'en reste plus, le système met en marche les mécanismes de pagination / swap
- Possibilité de fragmentation de la mémoire

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
 - S'il n'en reste plus, le système met en marche les mécanismes de pagination / swap
- Possibilité de fragmentation de la mémoire

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
 - S'il n'en reste plus, le système met en marche les mécanismes de pagination / swap
- Possibilité de fragmentation de la mémoire

Instanciation de masse

Java

- Instanciation possible tant qu'il reste de la mémoire à la JVM
- Pertes de performance drastiques si instanciation de plusieurs millions d'objets
 - Indépendamment de leur taille et des ressources disponibles

C++

- Instanciation possible tant qu'il reste de la mémoire en machine
 - S'il n'en reste plus, le système met en marche les mécanismes de pagination / swap
- Possibilité de fragmentation de la mémoire

Conclusion

Conclusion

- Deux langages très différents en mécanique
- Pas de « meilleur » langage
 - Choisir son langage en fonction des avantages et inconvénients
 - Par exemple : C++ est « rapide », Java est « portable »
- C++ tend à laisser plus de fonctionnalités disponibles au programmeur
 - Une mauvaise utilisation de ces fonctionnalités peut engendrer des erreurs
- C++ tend à désactiver par défaut certains mécanismes pour des raisons de performances

Conclusion

- Deux langages très différents en mécanique
- Pas de « meilleur » langage
 - Choisir son langage en fonction des avantages et inconvénients
 - Par exemple : C++ est « rapide », Java est « portable »
- C++ tend à laisser plus de fonctionnalités disponibles au programmeur
 - Une mauvaise utilisation de ces fonctionnalités peut engendrer des erreurs
- C++ tend à désactiver par défaut certains mécanismes pour des raisons de performances

Conclusion

- Deux langages très différents en mécanique
- Pas de « meilleur » langage
 - Choisir son langage en fonction des avantages et inconvénients
 - Par exemple : C++ est « rapide », Java est « portable »
- C++ tend à laisser plus de fonctionnalités disponibles au programmeur
 - Une mauvaise utilisation de ces fonctionnalités peut engendrer des erreurs
- C++ tend à désactiver par défaut certains mécanismes pour des raisons de performances

Conclusion

- Deux langages très différents en mécanique
- Pas de « meilleur » langage
 - Choisir son langage en fonction des avantages et inconvénients
 - Par exemple : C++ est « rapide », Java est « portable »
- C++ tend à laisser plus de fonctionnalités disponibles au programmeur
 - Une mauvaise utilisation de ces fonctionnalités peut engendrer des erreurs
- C++ tend à désactiver par défaut certains mécanismes pour des raisons de performances

Conclusion

- Deux langages très différents en mécanique
- Pas de « meilleur » langage
 - Choisir son langage en fonction des avantages et inconvénients
 - Par exemple : C++ est « rapide », Java est « portable »
- C++ tend à laisser plus de fonctionnalités disponibles au programmeur
 - Une mauvaise utilisation de ces fonctionnalités peut engendrer des erreurs
- C++ tend à désactiver par défaut certains mécanismes pour des raisons de performances

Conclusion

- Deux langages très différents en mécanique
- Pas de « meilleur » langage
 - Choisir son langage en fonction des avantages et inconvénients
 - Par exemple : C++ est « rapide », Java est « portable »
- C++ tend à laisser plus de fonctionnalités disponibles au programmeur
 - Une mauvaise utilisation de ces fonctionnalités peut engendrer des erreurs
- C++ tend à désactiver par défaut certains mécanismes pour des raisons de performances

Conclusion

- Deux langages très différents en mécanique
- Pas de « meilleur » langage
 - Choisir son langage en fonction des avantages et inconvénients
 - Par exemple : C++ est « rapide », Java est « portable »
- C++ tend à laisser plus de fonctionnalités disponibles au programmeur
 - Une mauvaise utilisation de ces fonctionnalités peut engendrer des erreurs
- C++ tend à désactiver par défaut certains mécanismes pour des raisons de performances