

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	

La mémoire de la JVM s'organise en plusieurs zones :

- **zone des méthodes** : données des classes, dont méthodes (leurs codes-octet) et attributs statiques (leurs valeurs)
- **tas** : zone où sont stockés les objets alloués dynamiquement
- **pile(s)** (une par thread<sup>1</sup>) : là où sont stockées les données temporaires de chaque appel de méthode en cours
- **zone(s) des registres** (une par thread), contenant notamment les registres suivants :
  - l'adresse de la prochaine instruction à exécuter (« program counter ») sur le thread
  - l'adresse du sommet de la pile du thread

1. Fil d'exécution parallèle. Cf. chapitre sur la programmation concurrente.

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (2)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (1)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (2)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (1)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (2)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (1)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (2)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (1)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (2)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (1)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (2)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (1)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (2)
---------------------	-------------------

Introduction	Compléments en 200
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Le système de types	
Sous-objets	
Transfertage	
Polytypiques(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

Zones de la mémoire	La pile/stack (1)
---------------------	-------------------

Introduction	Compl

## Types de données

# Système de types de Java

Caractéristiques principales

- typage qualifié de "fort" (concept plutôt flou : on peut trouver bien plus strict !)
- **typage statique** : le compilateur vérifie le type des expressions du code source
- **typage dynamique** : à l'exécution, les objets connaissent leur type. Il est testable à l'exécution (permet traitement différencié<sup>1</sup> dans code polymorphe).
- sous-typage, permettant le polymorphisme : une méthode déclarée pour argument de type **T** est appellable sur argument pris dans tout sous-type de **T**.
- 2 "sortes" de type : types primitifs (valeurs directes) et types référence (objets)
- **typage nominatif**<sup>2</sup> : 2 types sont égaux ssi ils ont le même nom. En particulier, si **class A { int x; }** et **class B { int x; }** alors **A.x = new B();** ; ne passe pas la compilation bien que **A** et **B** aient la même structure.

1. Via la liaison tardive/dynamique et via mécanismes explicites : **instanceof** et réflexion.
2. Contre : typage structurel (ce qui compte est la structure interne du type, pas le nom donné)



## Types de données en Java

# Types de données en Java

Zoom sur les types primitifs



## Types de données en Java

# Types de données en Java

Zoom sur les types primitifs



## Références et valeurs directes

Interprétations

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

Valeurs calculées stockées, en pile ou dans un champ, sur 1 mot (2 si **long** ou **doublé**)

- types primitifs : directement la valeur intéressante
- types références : une adresse mémoire (pointant vers un objet dans le tas).

**Dans les 2 cas** : ce qui est stocké dans un champ ou dans la pile n'est qu'une suite de 32 bits, indistinguables de ce qui est stocké dans un champ d'un autre type.

L'interprétation fait de cette valeur dépendra uniquement de l'instruction qui l'utilisera, mais la compilation garantit que ce sera la bonne interprétation.

**Cas des types référence** : quel que soit le type, cette valeur est interprétée de la même façon, comme une adresse. Le type décrit alors l'objet référencé seulement.

**Exemple** : une variable de type **String** et une de type **Point2D** contiennent tous deux le même genre de données : un mot représentant une adresse mémoire. Pourtant la première pointera toujours sur une chaîne de caractères alors que la seconde pointera toujours sur la représentation d'un point du plan.

## Références et valeurs directes

Conséquence sur le passage de paramètres à l'appel d'une méthode

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Discussion	Discussion	

**Rappel** : En Java, quand on appelle une méthode, on passe les paramètres par valeur uniquement : une copie de la valeur du paramètre est empilée avant appel.

Ainsi :

- pour les types primitifs<sup>1</sup> → la méthode travaille sur une copie des données réelles
- pour les types référence → c'est l'adresse qui est copiée ; la méthode travaille avec cette copie, qui pointe sur... les mêmes données que l'adresse originale.

**Conséquence** :

- Dans tous les cas, affecter une nouvelle valeur à la variable-paramètre ne sert à rien : la modification serait perdue au retour.
- Mais si le paramètre est une référence, on peut modifier l'objet référencé. Cette modification persiste après le retour de la méthode.

1. = types à valeur directe, pas les types référence

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	
Style	Style	
Objets et classes	Objets et classes	
Types et polymorphisme	Types et polymorphisme Mémoire et JVM	
Le système de types	Le système de types Sous-typage Transfertage Polymorphisme(s)	
Surcharage	Surcharage	
Interfaces	Interfaces	
Héritage	Héritage	
Généricité	Généricité	
Concurrence	Concurrence	
Interfaces graphiques	Interfaces graphiques	
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions	
Révision	Révision	

	Compléments en POO	Alfric Degorre
Introduction	Introduction	
Généralités	Généralités	



<tbl\_r cells="3" ix="3" maxcspan="1"

## Type dynamique et statique

Quand vérifier la cohérence des données ?

- La vérification du bon typage d'un programme peut avoir lieu à différents moments :
  - langages très « bas niveau » (assembleur x86, p. ex.) : jamais ;
  - C, C++, OCaml, ... : dès la compilation (**typage statique**) ;
  - Python, PHP, Javascript, ... : seulement à l'exécution (**typage dynamique**) ;

Remarque : typages statique et dynamique ne sont pas mutuellement exclusifs.

- Les entités auxquelles ont attribué un type ne sont pas les mêmes selon le moment où cette vérification est faite.

Typage statique → concerne les expressions du programme

Typage dynamique → concerne les données existant à l'exécution.

Où se Java se situe-t-il ? Que type-t-on en Java ?

- Il existe même des langages où le programmeur décide ce qui est vérifié à l'exécution ou à la compilation : « typage graduel ».

## Stades de vérification et entités typables en Java

Java → langage à typage statique, mais avec certaines vérifications à l'exécution<sup>1</sup> :

- À la compilation on vérifie le type des expressions<sup>2</sup> (**analyse statique**).  
Toutes les expressions sont vérifiées.
- À l'exécution, la JVM peut vérifier le type des objets<sup>3</sup>.  
Cette vérification a seulement lieu lors d'événements bien précis :
  - quand l'on souhaite différencier le comportement en fonction de l'appartenance ou non à un type (lors d'un test **instanceof**<sup>4</sup> ou d'un appel de méthode d'instance<sup>5</sup>) ;
  - quand on souhaite interrompre le programme sur une exception en cas d'incohérence de typage<sup>6</sup> : notamment lors d'un *downcasting*, ou bien après exécution d'une méthode générique dont le type de retour est une variable de type.

- C'est en fait une caractéristique habituelle des langages à typage essentiellement statique mais autorisant le polymorphisme par sous-type.
- Expression = élément syntaxique du programme représentant une valeur calculable.
- Ces entités n'existent pas avant l'exécution, de toute façon !
- Code-octet : **instanceof**.
- Code-octet : **invokeinterface** ou **invokevirtual**.
- Code-octet : **checkcast**.

## Compléments en 200

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Type statique et statique

Quand vérifier la cohérence des données ?

- La vérification du bon typage d'un programme peut avoir lieu à différents moments :
  - langages très « bas niveau » (assembleur x86, p. ex.) : jamais ;
  - C, C++, OCaml, ... : dès la compilation (**typage statique**) ;
  - Python, PHP, Javascript, ... : seulement à l'exécution (**typage dynamique**) ;

Remarque : typages statique et dynamique ne sont pas mutuellement exclusifs.

- Les entités auxquelles ont attribué un type ne sont pas les mêmes selon le moment où cette vérification est faite.

Typage statique → concerne les expressions du programme

Typage dynamique → concerne les données existant à l'exécution.

Où se Java se situe-t-il ? Que type-t-on en Java ?

- Il existe même des langages où le programmeur décide ce qui est vérifié à l'exécution ou à la compilation : « typage statique ».

## Stades de vérification et entités typables en Java

À l'exécution : les objets

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Stades de vérification et entités typables en Java

À la compilation : les expressions

## Type statique et statique

Quand vérifier la cohérence des données ?

- Le compilateur sait que l'expression "bonjour" est de type **String**. (idem pour les types primatifs : 42 est toujours de type **int**).
- Si on déclare **Scanner s**, alors l'expression **s** est de type **Scanner**.
- Le compilateur sait aussi déterminer que **1.0 + 2** est de type **double**.
- (Java ≥ 10) Après **var m = "coucou"**; l'expression **m** est de type **String**.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- int x = 1;** **System.out.println(x/2);** est bien typé.
- en revanche, **Math.cos("bonjour")** est mal typé.

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Stades de vérification et entités typables en Java

À l'exécution : les objets

## Type statique et statique

Quand vérifier la cohérence des données ?

- Le compilateur sait que l'expression "bonjour" est de type **String**. (idem pour les types primatifs : 42 est toujours de type **int**).
- Si on déclare **Scanner s**, alors l'expression **s** est de type **Scanner**.
- Le compilateur sait aussi déterminer que **1.0 + 2** est de type **double**.
- (Java ≥ 10) Après **var m = "coucou"**; l'expression **m** est de type **String**.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- int x = 1;** **System.out.println(x/2);** est bien typé.
- en revanche, **Math.cos("bonjour")** est mal typé.

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Stades de vérification et entités typables en Java

À l'exécution : les objets

## Type statique et statique

Quand vérifier la cohérence des données ?

- Le compilateur sait que l'expression "bonjour" est de type **String**. (idem pour les types primatifs : 42 est toujours de type **int**).
- Si on déclare **Scanner s**, alors l'expression **s** est de type **Scanner**.
- Le compilateur sait aussi déterminer que **1.0 + 2** est de type **double**.
- (Java ≥ 10) Après **var m = "coucou"**; l'expression **m** est de type **String**.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- int x = 1;** **System.out.println(x/2);** est bien typé.
- en revanche, **Math.cos("bonjour")** est mal typé.

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Stades de vérification et entités typables en Java

À l'exécution : les objets

## Type statique et statique

Quand vérifier la cohérence des données ?

- Le compilateur sait que l'expression "bonjour" est de type **String**. (idem pour les types primatifs : 42 est toujours de type **int**).
- Si on déclare **Scanner s**, alors l'expression **s** est de type **Scanner**.
- Le compilateur sait aussi déterminer que **1.0 + 2** est de type **double**.
- (Java ≥ 10) Après **var m = "coucou"**; l'expression **m** est de type **String**.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- int x = 1;** **System.out.println(x/2);** est bien typé.
- en revanche, **Math.cos("bonjour")** est mal typé.

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Stades de vérification et entités typables en Java

À l'exécution : les objets

## Type statique et statique

Quand vérifier la cohérence des données ?

- Le compilateur sait que l'expression "bonjour" est de type **String**. (idem pour les types primatifs : 42 est toujours de type **int**).
- Si on déclare **Scanner s**, alors l'expression **s** est de type **Scanner**.
- Le compilateur sait aussi déterminer que **1.0 + 2** est de type **double**.
- (Java ≥ 10) Après **var m = "coucou"**; l'expression **m** est de type **String**.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- int x = 1;** **System.out.println(x/2);** est bien typé.
- en revanche, **Math.cos("bonjour")** est mal typé.

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Stades de vérification et entités typables en Java

À l'exécution : les objets

## Type statique et statique

Quand vérifier la cohérence des données ?

- Le compilateur sait que l'expression "bonjour" est de type **String**. (idem pour les types primatifs : 42 est toujours de type **int**).
- Si on déclare **Scanner s**, alors l'expression **s** est de type **Scanner**.
- Le compilateur sait aussi déterminer que **1.0 + 2** est de type **double**.
- (Java ≥ 10) Après **var m = "coucou"**; l'expression **m** est de type **String**.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- int x = 1;** **System.out.println(x/2);** est bien typé.
- en revanche, **Math.cos("bonjour")** est mal typé.

Aldric Degorre

### Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Stades de vérification et entités typables en Java

À l'exécution : les objets

## Type statique et statique

Quand vérifier la cohérence des données ?

- Le compilateur sait que l'expression "bonjour" est de type **String**. (idem pour les types primatifs : 42 est toujours de type **int**).
- Si on déclare **Scanner s**, alors l'expression **s** est de type **Scanner**.
- Le compilateur sait aussi déterminer que **1.0 + 2** est de type **double**.
- (Java ≥ 10) Après **var m = "coucou"**; l'expression **m** est de type **String**.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- int x = 1;** **System.out.println(x/2);** est bien typé.
- en revanche, **Math.cos("bonjour")** est mal typé.

## Relation entre type statique et type dynamique

Pour une variable ou expression :

- son **type statique** est son type tel que déduit par le compilateur (pour une variable : c'est le type indiqué dans sa déclaration) ;
- son **type dynamique** est la classe de l'objet référencé (par cette variable ou par le résultat de l'évaluation de cette expression).
- Le type dynamique ne peut pas être déduit à la compilation.
- Le type dynamique change<sup>1,2</sup> au cours de l'exécution.

La vérification statique et les règles d'exécution garantissent la propriété suivante :

**Le type dynamique d'une variable ou expression est toujours un sous-type (cf. juste après) de son type statique.**

1. Pour une variable : après chaque affectation, un objet différent peut être référencé.

Pour une expression : une expression peut être évaluée plusieurs fois lors d'une exécution du programme et donc référencer, tour à tour, des objets différents.  
2. Remarque : le type (la classe) d'un objet donné est, en revanche, fixé(e) dès son instantiation.

Notion de sous-typage	
Interprétations	<p>Pourquoi le sous-typage structurel est insuffisant ?</p> <p><b>Exemple :</b></p> <ul style="list-style-type: none"> <li>• Dans le cours précédent, les instances de la classe <i>FiboGen</i> génèrent la suite de Fibonacci.</li> <li>• <u>Contrat possible</u><sup>1</sup> : « le rapport de 2 valeurs successives tend vers <math>\varphi = \frac{1+\sqrt{5}}{2}</math> (nombre d'or) ».</li> <li>• (On sait prouver ce contrat pour la méthode <code>next</code> des instances directes de <i>FiboGen</i>.)</li> <li>• Or rien empêche de créer un sous-type <i>BadFib</i> (sous-classe<sup>2</sup>) de <i>Fibogen</i> dont la méthode <code>next</code> renverrait toujours 0.</li> </ul> <p>→ Les instances de <i>BadFib</i> seraient alors des instances de <i>FiboGen</i> violant le contrat.</p> <hr/> <ol style="list-style-type: none"> <li>1. Raisonnable, dans le sens où c'est une propriété mathématique démontée pour la suite de Fibonacci, qui donc doit être vraie dans toute implémentation correcte.</li> <li>2. Une sous-classe est bien un sous-type au sens structurel : les méthodes sont héritées.</li> </ol>
Introduction	<p>Compléments en POO en POO Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Sous-typage Transfertage Polymorphisme(s) Structure Interfaces Héritage Généricité Concurrence Interfaces graphiques Gestion des émetteurs et récepteurs Analyses</p>
Interprétations	<p>Compléments en POO en POO Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Sous-typage Transfertage Polymorphisme(s) Structure Interfaces Héritage Généricité Concurrence Interfaces graphiques Gestion des émetteurs et récepteurs Analyses</p>

- Notion de sous-typepage
- Interprétations
- **Interprétation idéale : Principe de Substitution de Liskov**<sup>1</sup> (LSP). Un sous-type doit respecter tous les contrats du supertype.  
Les propriétés du programme prouvables comme conséquence des contrats du supertype sont alors effectivement vraies quand on utilise le sous-type à sa place.  
*Exemple : les propriétés largeur et hauteur d'un rectangle sont modifiables indépendamment. Un carré ne satisfait pas ce contrat. Donc, selon le LSP, le type carré modifiable n'est pas sous-type de rectangle modifiable.*  
*En revanche, carré non modifiable est sous-type de rectangle non modifiable, selon le LSP.*

## Notion de sous-typage

Interprétations : en pratique

- Pour les 8 types primitifs, il y a une relation de sous-typage pré-définie.
- Pour les types référence, le sous-typage est nominal :  $A$  n'est sous-type de  $B$  que si  $A$  est déclaré comme tel (**implements** ou **extends**).  
Mais la définition de  $A$  ne passe la compilation que si certaines contraintes structurelles<sup>1</sup> sont vérifiées, concernant les redéfinitions de méthodes.
- Types primitifs et types référence forment deux systèmes déconnectés. Aucun type référence n'est sous-type ou supertype d'un type primitif.

---

1. Cf. cours sur les interfaces et sur l'héritage pour voir quelles sont les contraintes exactes.



Notion de sous-typage pour Java	
Compléments en POO Aldric Degorre	Principe fondamental – explications
Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Sous-typage Transtypage Polymorphisme(s) Surcharge Interfaces Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions	<p>Pourquoi ce remplacement ne gène pas l'<u>exécution</u> :</p> <ul style="list-style-type: none"> <li>les objets sont utilisables <u>sans modification</u> comme instances de tous leurs supertypes<sup>1</sup> (<b>sous-typage inclusif</b>). P. ex. : <code>Object o = "toto"</code> fonctionne.</li> <li>Java<sup>2</sup> s'autorise, si nécessaire, à remplacer une valeur primitive par la valeur la plus proche dans le type cible (<b>sous-typage coercitif</b>). P. ex. : après l'affectation <code>float f = 1_000_000_000_123L;</code>, la variable <code>f</code> vaut <code>1.0E12</code> (on a perdu les derniers chiffres).</li> </ul> <p>1. Les contraintes d'implémentation d'interface et d'héritage garantissent que les méthodes des supertypes peuvent être appelées. 2. Si nécessaire, javac convertit les constantes et insère des instructions dans le code-octet pour convertir les valeurs variables à l'exécution.</p>
Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Sous-typage Transtypage Polymorphisme(s) Surcharge Interfaces Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions	<h3>Transstypage (type casting)</h3> <p><b>Corollaires :</b></p> <ul style="list-style-type: none"> <li>on peut affecter à toute variable une expression de son sous-type (ex : <code>double z = 12;</code>);</li> <li>on peut appeler toute méthode avec des arguments d'un sous-type des types déclarés dans sa signature (ex : <code>Math.pow(3, -2);</code>);</li> <li>on peut appeler toute méthode d'une classe T donné sur un récepteur instance d'une sous-classe de T (ex : <code>"toto".hashCode()</code>).</li> </ul> <p>Ces caractéristiques font du sous-typage la base du système de polymorphisme de Java.</p>
Compléments en POO Aldric Degorre	<h3>Transstypage (type casting)</h3> <p><b>Corollaires :</b></p> <ul style="list-style-type: none"> <li>on peut affecter à toute variable une expression de son sous-type (ex : <code>double z = 12;</code>);</li> <li>on peut appeler toute méthode avec des arguments d'un sous-type des types déclarés dans sa signature (ex : <code>Math.pow(3, -2);</code>);</li> <li>on peut appeler toute méthode d'une classe T donné sur un récepteur instance d'une sous-classe de T (ex : <code>"toto".hashCode()</code>).</li> </ul> <p>Ces caractéristiques font du sous-typage la base du système de polymorphisme de Java.</p>
Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Sous-typage Transtypage Polymorphisme(s) Surcharge Interfaces Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions	<h3>Transstypage (type casting)</h3> <p><b>Corollaires :</b></p> <ul style="list-style-type: none"> <li>on peut affecter à toute variable une expression de son sous-type (ex : <code>Double vers Number</code>)</li> <li>d'un type vers un sous-type (ex : <code>Object vers String</code>)</li> <li>d'un type primitif vers sa version "emballée" (ex : <code>int vers Integer</code>)</li> <li>d'un type emballé vers le type primitif correspondant (ex : <code>Boolean vers boolean</code>)</li> <li>conversion en <code>String</code> : de tout type vers <code>String</code> (implicite pour concaténation)</li> <li>parfois combinaison implicite de plusieurs mécanismes.</li> </ul> <p>1. Détaillez dans la JLS, chapitre 5. 2. On ne mentionne pas les mécanismes explicites et évidents tels que l'utilisation de méthodes tenant du A et retournant du B. Si on va par là, tout type est convertible en tout autre type.</p>

Introduction	Compléments en '00 Aldric Degoire
Généralités	
Style	
Objets et classes	
Types et polymorphisme	Mémoire et JVM Le système de types Système Polymorphisme Transtypage Polymorphisme (\$) Surcharge Interfaces Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions Détails

- Élargissement/widening et rétrécissement/narrowing : dans la JLS (5.1), synonymes respectifs de **upcasting** et **downcasting**.
- Inconvénient** : le sens étymologique (= réécriture sur resp. + de bits ou - de bits), ne représente pas la réalité en Java (cf. la suite).
- Promotion** : synonyme de **upcasting**. Utilisé dans la JLS (5.6) seulement pour les conversions implicites des paramètres des opérateurs arithmétiques.<sup>1</sup>

- Alors qu'on pourrait expliquer ce mécanisme de la même façon que la résolution de la surcharge.

Introduction	Compléments en '00 Aldric Degoire
Généralités	
Style	
Objets et classes	Mémoire et JVM Le système de types Système Polymorphisme Transtypage Polymorphisme (\$) Surcharge Interfaces Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions Révision

## Transtypage, cas 1 : *upcasting*

Introduction	Compléments en '00 Aldric Degoire
Généralités	
Style	
Objets et classes	Mémoire et JVM Le système de types Système Polymorphisme Transtypage Polymorphisme (\$) Surcharge Interfaces Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions Révision

- Cas d'application : on souhaite obtenir une expression d'un supertype à partir d'une expression d'un sous-type.
- L'**Upcasting** est en général implicite (pas de marque syntaxique).
- Exemple :

```
double z = 3; // upcasting ( implicite ) de int vers double
```

- Utilité, **polymorphisme par sous-typage** : partout où une expression de type **T** est autorisée, toute expression de type **T'** est aussi autorisée si **T' <: T**.
- Exemple : si **class B extends A {}**, **void f(A a)** et **B b**, alors l'appel **f(b)** est accepté.
- L'**Upcasting** implicite permet de faire du polymorphisme de façon transparente.
- On peut aussi demander explicitement l'**upcasting**, ex : **(double)****4**
- L'**Upcasting** explicite sera rarement, mais permet parfois de guider la résolution de la surcharge : remarquez la différence entre **3/4** et **((double)3)/4**.

## Transtypage, cas 2 : *downcasting*

Introduction	Compléments en '00 Aldric Degoire
Généralités	
Style	
Objets et classes	Mémoire et JVM Le système de types Système Polymorphisme Transtypage Polymorphisme (\$) Surcharge Interfaces Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions Révision

- Downcasting :**
- Cas d'application : on veut écrire du code spécifique pour un sous-type de celui qui nous est fourni.
- Dans ce cas, il faut demander une conversion explicite.
- Exemple : **int x = (int)143.32**.
- Utilité :

  - (pour les objets) dans un code polymorphe, généraliste, on peut vouloir écrire une partie qui travaille seulement sur un certain sous-type, dans ce cas, on teste la classe de l'objet manipulé et on **downcast** l'expression qui le référence :

```
tf (x instanceof String) { String xs = (String) x; ... ; }
```

- Pour les nombres primitifs, on peut souhaiter travailler sur des valeurs moins précises : **int partieEntiere = (int)unReel;**

Implémentations d'objets différents	<p><u>Transtypage</u>, cas 2 : downcasting</p> <p>Avertissement</p> <p>Le code ci-dessous est probablement symptomatique d'une conception non orientée objet :</p> <hr/> <pre>// Anti-patron : <b>void</b> g(<b>Object</b> x) { // Object ou bien autre supertype commun à C1 et C2     <b>if</b> (<b>x instanceof</b> C1) { C1 y = (C1) x; f1(y); }     <b>else if</b> (<b>x instanceof</b> C2) { C2 y = (C2) x; f2(y); }     <b>else</b> { /* quoi en fait ? on générera une erreur ? */ }</pre> <hr/> <p>Quand c'est possible, on préfère utiliser la liaison dynamique :</p> <hr/> <pre><b>public interface</b> I { <b>void</b> f(); } <b>public void</b> g(I x) { x.f(); } // déjà, programmons à l'interface</pre> <hr/> <pre>// puis dans d'autres fichiers (voire autres packages) <b>public class</b> C1 implements I { <b>public void</b> f() { f1(this); } } <b>public class</b> C2 implements I { <b>public void</b> f() { f2(this); } }</pre>																										
Implémentations d'objets différents	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Mémoire et threads</p> <p>Système d'exploitation</p> <p>Transparence</p> <p>Polymporphisme</p> <p>Surcharge</p> <p>Interfaces</p> <p>Héritage</p> <p>Bénéficitez</p> <p>Concurrent</p> <p>Interfaces graphiques</p> <p>Gestion de fichiers et de répertoires</p> <p>Création et destruction d'objets</p> <p>Révision</p>																										

# Transypage, cas 3 : auto-boxing/unboxing

Types "emballés"

Transtypage, cas 2 : <i>downcasting</i>	Le code ci-dessous est probablement symptôme d'une conception non orientée objet :	<pre>// Anti-patron : <b>void</b> g(<b>Object</b> x) { // Object ou bien autre supertype commun à C1 et C2     <b>if</b> (<b>x instanceof</b> C1) {C1 y = (C1) x; f1(y);}     <b>else if</b> (<b>x instanceof</b> C2) { C2 y = (C2) x; f2(y);}     <b>else</b> { /* quoi en fait ? on génère une erreur ? */} }</pre>	Quand c'est possible, on préfère utiliser la liaison dynamique :	<pre><b>public interface</b> I { <b>void</b> f(); } <b>public</b> g(I x) { x.f(); } // déjà , programmons à l'interface</pre>	<pre>// puis dans d'autres fichiers (voire autres packages) <b>public class</b> C1 implements I { <b>public void</b> f() { f1(this); } } <b>public class</b> C2 implements I { <b>public void</b> f() { f2(this); } }</pre>	<p><b>Avantage :</b> les types concrets manipulés ne sont <u>jamais nommés</u> dans g, qui pourra donc fonctionner avec de nouvelles implémentations de I sans modification.</p>
Compléments en POO	Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Mémoire et JVM
Aldric Degorre	Avertissement	Le système de types	Sous-type	Transtypage	Polymorphisme(s)	Générité
Transtypage	Avantages et inconvénients	Surcharge	Imperféc.	Héritage	Concurrence	Interfaces graphiques
Transtypage	Incompatibilité	Inéf.	Gestion des erreurs et exceptions	Révision		

# Transypage, cas 3 : auto-boxing/unboxing

Types "emballés"

Subtilités du transystpage	
Compléments en 200	Cas des types primits : possible perte d'information (2)
Introduction	
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Sous-ytage	
Transystpage	
Polympiphane(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

- Cas avec perte d'information possible :
    - tous les **downcastings** primits;
    - **upcastings de `int` vers `float`**, **`long` vers `float`** ou **`long` vers `double`**;
    - **upcasting de `float` vers `double`** hors contexte **`strictfp`**.
  - Cas sans perte d'information : (= les autres cas = les "vrais" upcastings)
    - **upcasting d'entier vers entier plus long**;
    - **upcasting d'entier  $\leq 24$  bits vers `float` et `double`**;
    - **upcasting d'entier  $\leq 53$  bits vers `double`**;
    - **upcasting de `float` vers `double`** sous contexte **`strictfp`**.
1. Par exemple, **`int`** utilise 32 bits, alors que la **mantisse** de **`float`** n'en a que 24 (+ 8 bits pour la position de la virgule)  $\rightarrow$  certains **`int`** ne sont pas représentables en **`float`**.
2. Selon implémentation, mais pas de garantie. Cherchez à quoi sert ce mot-clé!

Subtilités du transystpage	
Compléments en 200	Cas des types primits : exemples
Introduction	
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Sous-ytage	
Transystpage	
Polympiphane(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Détails	

1. C'est du sous-ytage inclusif, comme pour les types référence!
2. Littéral numérique = nombre écrit en chiffres dans le code source.
- ## Subtilités du transystpage
- Cas des types primits : exemples
- **`int i = 42; short s = i;`** ; pour copier un **`int`** dans un **`short`**, on doit le rétrécir. La valeur à convertir est inconnue à la compilation  $\rightarrow$  ce sera fait à l'exécution. Ainsi le compilateur insère l'instruction **`i2s`** dans le code-octet.
  - **`short s = 42; i = s;`** ; comme un **`short`** est représenté comme un **`int`**, il n'y a pas de conversion à faire (**`s2i`** n'existe pas).
  - **`float x = 9; int i = x;`** ; ici on convertit une constante littérale entière en flottant. Le compilateur fait lui-même la conversion et met dans le code-octet la même chose que si on avait écrit **`float x = 9.0f;`**
  - Mais si on écrit **`int i = 9; float x = i;`**, c'est différent. Le compilateur ne pouvant pas convertir lui-même, il insère **`i2f`** avant l'instruction qui va copier le sommet de la pile dans **`x`**.
1. Pour plus d'explications : chercher "représentation des entiers en complément à 2".
2. Ainsi, la valeur d'origine est interprétée modulo 2<sup>n</sup> sur un intervalle centré en 0.

**Types références : exécuter un transtypage ne modifie pas l'objet référencé<sup>1</sup>**

- **downcasting** : le compilateur ajoute une instruction **checkcast** dans le code-octet.  
À l'exécution, **checkcast** lance une **ClassCastException** si l'objet référencé par le sommet de pile (= valeur de l'expression "castée") n'est pas du type cible.

```
// Compile et s'exécute sans erreur :
Comestible x = new Fruit(); Fruit y = (Fruit) x;
// Compile mais ClassCastException à l'exécution :
Comestible x = new Viande(); Fruit y = (Fruit) x;
```

- **upcasting** : invisible dans le code-octet, aucune instruction ajoutée → pas de conversion réelle à l'exécution. car l'inclusion des sous-types garantit, dès la compilation, que le cast est correct (**sous-type inclusif**).

1. en particulier, pas son type : on a déjà vu que la classe d'un objet était définitive

- **polymorphisme ad hoc** (via la surcharge) : le même code recompilé dans différents contextes peut fonctionner pour des types différents.
- Attention : résolution à la compilation → après celle-ci, type concret fixé.
- Donc pas de réutilisation du code compilé → forme très faible de polymorphisme.
- **polymorphisme par sous-type** : le code peut être exécuté sur des données de différents sous-types d'un même type (souvent une **interface**) sans recompilation.
- forme classique et privilégiée du polymorphisme en P00

#### • polymorphisme paramétré :

Concerne le code utilisant les **type génériques** (ou paramétrés, cf. généralité).

Le même code peut fonctionner, sans recompilation<sup>1</sup>, quelle que soit la concrétisation des paramètres.

Ce polymorphisme permet d'exprimer des relations fines entre les types.

1. Dans d'autres langages, comme le C++, le polymorphisme paramètre s'obtient en spécialisant automatiquement le code source d'un template et en l'intégrant au code qui l'utilise lors de sa compilation.

#### Surcharge = situation où existent plusieurs définitions (au choix)

- dans un contexte donné d'un programme, de plusieurs méthodes de même nom ;
- dans une même classe, plusieurs constructeurs ;
- d'opérateurs arithmétiques dénotés avec le même symbole. <sup>1</sup>.

**Signature d'une méthode** =  $n$ -uplet des types de ses paramètres formels.

#### Remarques :

- Interdiction de définir dans une même classe<sup>2</sup> 2 méthodes ayant même nom et même signature (ou 2 constructeurs de même signature).
- 2 entités surchargées ont forcément une signature différente<sup>3</sup>.
  - 1. P. ex. : "/" est défini pour **int** mais aussi pour **double**
  - 2. Les méthodes héritées comptent aussi pour la surcharge. Mais en cas de signature identique, il y a masquage et non surcharge. Donc ce qui est dit ici reste vrai.
  - 3. Nombre ou type des paramètres différent; le type de retour ne fait pas partie de la signature et n'a rien à voir avec la surcharge !

#### Surcharge = situation où existent plusieurs définitions (au choix)

- dans un contexte donné d'un programme, de plusieurs méthodes de même nom ;
- dans une même classe, plusieurs constructeurs ;
- d'opérateurs arithmétiques dénotés avec le même symbole. <sup>1</sup>.

**Signature d'une méthode** =  $n$ -uplet des types de ses paramètres formels.

#### Remarques :

- Interdiction de définir dans une même classe<sup>2</sup> 2 méthodes ayant même nom et même signature (ou 2 constructeurs de même signature).
- 2 entités surchargées ont forcément une signature différente<sup>3</sup>.
  - 1. P. ex. : "/" est défini pour **int** mais aussi pour **double**
  - 2. Les méthodes héritées comptent aussi pour la surcharge. Mais en cas de signature identique, il y a masquage et non surcharge. Donc ce qui est dit ici reste vrai.
  - 3. Nombre ou type des paramètres différent; le type de retour ne fait pas partie de la signature et n'a rien à voir avec la surcharge !

#### Surcharge = situation où existent plusieurs définitions (au choix)

- dans un contexte donné d'un programme, de plusieurs méthodes de même nom ;
- dans une même classe, plusieurs constructeurs ;
- d'opérateurs arithmétiques dénotés avec le même symbole. <sup>1</sup>.

Une signature  $(\rho_1, \dots, \rho_n)$  **subsume** une signature  $(q_1, \dots, q_m)$  si  $n = m$  et  $\forall i \in [1, n], \rho_i > q_i$ .

Dit autrement : une signature subsumant une autre accepte tous les arguments acceptés par cette dernière.

- Pour chaque appel de méthode  $f(e_1, e_2, \dots, e_n)$  dans un code source, la **signature d'appel** est le  $n$ -uplet de types  $(t_1, t_2, \dots, t_n)$  tel que  $t_i$  est le type de l'expression  $e_i$  (tel que détecté par le compilateur).

C'est celle-ci qui sera appelée à l'exécution.<sup>2</sup>

1. Notamment liées à l'héritage, nous ne détaillerons pas.

2. Exactement celle-ci pour les méthodes statiques. Pour les méthodes d'instance, on a juste déterminé que la méthode qui sera choisie à l'exécution aura cette signature-là. Voir liaison dynamique.

# Surcharge

Compléments en POO  
Aldric Degorre

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Mémoire et JVM  
Le système de types Java  
Sous-espace  
Transfertage  
Polymorphisme(s)  
Surcharge  
Interfaces  
Héritage  
Généricité  
Concurrency  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Discussion

```

public class Surcharge {
    public static void f(double z) { System.out.println("double"); }
    public static void f(int x) { System.out.println("int"); }
    public static void g(int x, double z) { System.out.println("int double"); }
    public static void g(double x, int z) { System.out.println("double int"); }

    public static void main(String[] args) {
        f(0); // affiche "int"
        f(0d); // affiche "double"
        // g(0, 0); ne compile pas
        g(0d, 0); // affiche "double int"
    }
}

```

## Exemples

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Mémoire et JVM  
Le système de types Java  
Sous-espace  
Transfertage  
Polymorphisme(s)  
Surcharge  
Interfaces  
Héritage  
Généricité  
Concurrency  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Discussion

# Surcharge

Compléments en POO  
Aldric Degorre

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Mémoire et JVM  
Le système de types Java  
Sous-espace  
Transfertage  
Polymorphisme(s)  
Surcharge  
Interfaces  
Héritage  
Généricité  
Concurrency  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Discussion

**Surcharge**  
Pourquoi elle ne permet que du polymorphisme "faible"

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Mémoire et JVM  
Le système de types Java  
Sous-espace  
Transfertage  
Polymorphisme(s)  
Surcharge  
Interfaces  
Héritage  
Généricité  
Concurrency  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Discussion

**Alternative**, écrire la méthode, une bonne fois pour toutes, de la façon suivante :

```

public static void g(Objet o) { // méthode "réellement" polymorphe
    if (o instanceof String) f((String) o);
    else if (o instanceof Integer) f((Integer) o);
    else { /* gérer l'erreur */ }
}

```

Mais ici, c'est en réalité du polymorphisme par sous-type<sup>1</sup> (de Object).

1. En fait, une forme bricolée, maladroite de celui-ci : il faut, autant que possible, éviter **instanceof** au profit de la liaison dynamique.

**Surcharge**  
Pourquoi elle ne permet que du polymorphisme "faible"

<p><b>Interfaces : syntaxe de la déclaration</b></p> <pre>public interface Comparable { int compareTo(Object other); }</pre> <p>Déclaration comme une classe, en remplaçant <b>class</b> par <b>interface</b>, mais :</p> <ul style="list-style-type: none"> <li>constructeurs interdits;</li> <li>tous les membres implicitement <b>public</b>;</li> <li>attributs implicitement <b>static final</b> (= constantes);</li> <li>types membres nécessairement et implicitement <b>static</b>;</li> <li>méthodes d'instance implicitement <b>abstract</b> (simple déclaration sans corps);</li> <li>méthodes d'instance non-abstraites signalées par mot-clé <b>default</b>;</li> <li>les méthodes <b>private</b> sont autorisées (annule <b>public</b> et <b>abstract</b>), autres membres obligatoirement <b>public</b>;</li> <li><b>méthodes final</b> interdites.</li> </ul> <ol style="list-style-type: none"> <li>Méthodes <b>static</b> et <b>default</b> depuis Java 8, <b>private</b> depuis Java 9.</li> <li>Ce qui est implicite n'a pas à être écrit dans le code, mais peut être écrit tout de même.</li> </ol>	<p>Compléments en POO Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Surcharge Transposition Polymorphisme(s) Interfaces Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions <b>Discussion</b></p>
<p><b>Interfaces : usage (1)</b></p> <pre>public interface Comparable { int compareTo(Object other); }  class Mot implements Comparable {     private String contenu;      public int compareTo(Object other) {         return ((Mot) autreMot).contenu.length() - contenu.length();     } }</pre>	<p>Compléments en POO Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Surcharge Transposition Polymorphisme(s) Interfaces Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions <b>Exemple</b></p>
<p><b>Mettre implements I dans l'en-tête de la classe A pour implémenter l'interface I.</b></p> <ul style="list-style-type: none"> <li>Les méthodes de I sont définissables dans A. Ne pas oublier d'écrire <b>public</b> I.</li> <li>Pour obtenir une « vraie » classe (non abstraite, i.e. instanciable) : nécessaire de définir toutes les méthodes abstraites promises dans l'interface implementée.</li> <li>Si toutes les méthodes promises ne sont pas définies dans A, il faut précéder la déclaration de A du mot-clé <b>abstract</b> (classe abstraite, non instanciable)</li> </ul> <p>Une classe peut implémenter plusieurs interfaces :</p> <pre>class A implements I, J, K { ... }.</pre>	<p>Compléments en POO Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Mémoire et JVM Le système de types Surcharge Transposition Polymorphisme(s) Interfaces Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions <b>Exemple</b></p>

Introduction  
Aldric Degorre

Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Mémoire et JVM  
Le système de types  
Surcharge  
Transfertage  
Polymorphisme(s)  
Interfaces  
Héritage  
Généricité  
Concurrence  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

Compléments en 200 Aldric Degorre

**Comparable**

```

classDiagram
    class Comparable {
        + compareTo(other : Object) : int
    }
    class Comparable <|-- Object
  
```

**Mot**

<b>Mot</b>
- contenu : String
+ compareTo(other : Object) : int

**Mot**

<b>Mot</b>
- contenu : String
+ compareTo(other : Object) : int

**Notez l'italique pour la méthode abstraite et la flèche utilisée (pointillés et tête en triangle côté interface) pour signifier "implémenté".**

ou version "abégée" :  
**Comparable**



**Comparable**

+ compareTo(other : Object) : int

- Méthode par **défaut** : méthode d'instance, non abstraite, définie dans une interface.

- Sa déclaration est précédée du mot-clé **default**.

- N'utilise pas les attributs de l'objet, encore inconnus, mais peut appeler les autres méthodes déclarées, même abstraites.

- Utilité : implémentation par défaut de cette méthode, héritée par les classes qui implémentent l'interface → moins de réécriture.

- Possibilité d'une forme (faible) d'héritage multiple (via superclasse + interface(s) implémentée(s)).

- **Avertissement** : héritage possible de plusieurs définitions pour une même méthode par plusieurs chemins.  
Il sera parfois nécessaire de « désambiguier » (on en reparlera).

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Mémoire et JVM  
Le système de types  
Surcharge  
Transfertage  
Polymorphisme(s)  
Interfaces  
Héritage  
Généricité  
Concurrence  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

Une classe peut hériter de plusieurs implémentations d'une même méthode, via les interfaces qu'elle implémente (méthodes **default**, Java ≥ 8).

Cela peut créer des ambiguïtés qu'il faut lever. Par exemple, le programme ci-dessous est ambigu et ne compile pas (quel sens donner à **new A().f()**?).

```

interface ArbreBinaire {
    ArbreBinaire gauche();
    ArbreBinaire droite();
    default int hauteur() {
        ArbreBinaire g = gauche();
        int hg = (g == null)?0:g.hauteur();
        ArbreBinaire d = droite();
        int hd = (d == null)?0:d.hauteur();
        return 1 + (hg>hd)?hg:hd;
    }
}
  
```

Pour le corriger, il faut redéfinir **f()** dans **A**, par exemple comme suit :

```

class A implements I,J {
    @Override public void f() { I.super.f(); J.super.f(); }
}
  
```

**Remarque** : on ne peut pas (re)définir par défaut des méthodes de la classe **Object** (comme **toString** et **equals**).

**Raison** : une méthode par défaut n'est là que... par défaut. Toute méthode de même nom héritée d'une classe est prioritaire. Ainsi, une implémentation par défaut de **toString** serait tout le temps ignorée.

Cette construction **NonInterface.super.nomMéthode()** permet de choisir quelle version appeler dans le cas où une même méthode serait héritée de plusieurs façons.

## Héritage d'implémentations multiples

À cause des méthodes par défaut des interfaces

## Du bon usage des interfaces

Programmez à l'interface (1)

Quand une implémentation de méthode est héritée à la fois d'une superclasse et d'une interface, c'est la version héritée de la classe qui prend le dessus.

Java n'oblige pas à lever l'ambiguité dans ce cas.

```
interface I {  
    default void f() { System.out.println("I"); }  
}  
  
class B {  
    public void f() { System.out.println("B"); }  
}  
  
class A extends B implements I {}
```

Ce programme compile et **new A()** ; affiche B.

## Du bon usage des interfaces

Programmez à l'interface

Évitez d'écrire, dans votre programme, le nom <sup>1</sup> des classes des objets qu'il utilise.

Cela veut dire, évitez :



et préférez :



Cela s'appelle « **programmer à l'interface** ».

1. On parle alors de dépendance statique, c'est-à-dire le fait de citer nommément une entité externe (p.e. une autre classe) dans un code source.  
Le fait de référencer un objet d'une autre classe à un moment de l'exécution ne compte pas.

## Compléments en 200

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-Page

Transfert

Polympolymérisation

SurchARGE

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Héritage d'implémentations multiples

Programmez à l'interface

Quand une implémentation de méthode est héritée à la fois d'une superclasse et d'une interface, c'est la version héritée de la classe qui prend le dessus.

Java n'oblige pas à lever l'ambiguité dans ce cas.

```
interface I {  
    default void f() { System.out.println("I"); }  
}  
  
class B {  
    public void f() { System.out.println("B"); }  
}  
  
class A extends B implements I {}
```

Ce programme compile et **new A()** ; affiche B.

## Du bon usage des interfaces

Programmez à l'interface (2)

plutôt facile quand le nom de classe est utilisé en tant que type (notamment dans déclarations de variables et de méthodes)

→ remplacer par des noms d'interfaces (ex : **List** à la place de **ArrayList**)

pour instancier ces types, il faut bien que des constructeurs soient appellés, mais :

- si vous codez une bibliothèque, laissez vos clients vous fournir vos dépendances (p. ex. : en les passant au constructeur de votre classe) → **injection de dépendance**<sup>1</sup>

```
public class MyLib {  
    private final SomeInterface aDependency;  
    public MyLib(SomeInterface aDependency) { this.aDependency = aDependency; }  
}
```

- sinon, circonscrire le problème en utilisant des **fabriques**<sup>2</sup> définies ailleurs (par vous ou par un tiers) : **List<Integer> l = List.of(4, 5, 6);**

## Du bon usage des interfaces

Programmez à l'interface (3)

Pourquoi programmer à l'interface :

Une classe qui mentionne par son nom une autre classe contient une dépendance statique<sup>1</sup> à cette dernière. Cela entraîne des rigidités.

Au contraire, une classe **A** programmée « à l'interface », est

- **polymorphe** : on peut affecter à ses attributs et passer à ses méthodes tout objet implémentant la bonne interface, pas seulement des instances d'une certaine classe fixée « en dur ». → gain en adaptabilité

- **évolutive** : il n'y a pas d'engagement quant à la classe concrète des objets retournés par ses méthodes.

Il est donc possible de changer leur implémentation sans « casser » les clients de **A**.

## Du bon usage des interfaces

Programmez à l'interface

1. Ici, injection via paramètre du constructeur. Mais il existe des frameworks d'injection de dépendance.

2. Plusieurs variantes du patron « fabrique », cf. GOF. Variante la plus aboutie : fabrique abstraite (*abstract factory*). Le client ne dépend que de la fabrique abstraite, la fabrique concrète est elle-même injectée !

## Compléments en 200

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Mémoire et JVM

Sous-Page

Transfert

Polympolymérisation

SurchARGE

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

## Du bon usage des interfaces

Fabriques abstraites

### Du bon usage des interfaces

Fabriques abstraites à l'âge des lambdas

**Besoin :** dans `MyClass`, créer des instances d'une interface `Dep` connue, mais d'implémentation inconnue à l'avance.

**Réponse classique :** on écrit une interface `DepAbstractFactory` et on ajoute au constructeur de `MyClass` un argument `DepAbstractFactory factory`. Pour créer une instance de `Dep` on fait juste `factory.create()`.

```
public interface DepAbstractFactory { Dep create(); }

public class MyClass {
    private final DepAbstractFactory factory;
    public MyClass(DepAbstractFactory factory) { this.factory = factory; }
    /* plus loin */ Dep uneInstanceDeDep = factory.create();

    // programme client
    public class DepImpl implements Dep { ... }
    public class DepConcreteFactory implements DepAbstractFactory {
        @Override public Dep create() { return new DepImpl(...); }
        /* plus loin */ MyClass x = new MyClass(new MyDepFactory());
```

### Du bon usage des interfaces

Fabriques abstraites à l'âge des lambdas

**Version moderne : remplacer `DepFactory` par `java.util.function.Supplier`** :

```
public class MyClass {
    private final Supplier<Dep> factory;
    public MyClass(Supplier<Dep> factory) { this.factory = factory; }
    /* plus loin */ Dep uneInstanceDeDep = factory.get();

    // programme client
    public class DepImpl implements Dep { ... }
    /* plus loin */ MyClass x = new MyClass(() -> new DepImpl(...));
```

## Du bon usage des interfaces

Principe d'inversion de dépendance (DIP) (1)

### Du bon usage des interfaces

Principe d'inversion de dépendance (2), sous forme de diagramme UML

### Du bon usage des interfaces

Principe d'inversion de dépendance (2), sous forme de diagramme UML

### Du bon usage des interfaces

Principe d'inversion de dépendance (2)



Principe d'inversion de dépendance (2)

sous forme de diagramme UML

- **Quand ?** quand on programme une bibliothèque dépendant d'un certain composant et qu'il n'existe pas d'interface « standard » décrivant exactement les fonctionnalités de celui-ci.<sup>1</sup>
- **Quoi ?** → on définit alors une interface idéale que la dépendance devrait implémenter et on la joint au joint au package<sup>2</sup> de la bibliothèque.

Les utilisateurs de la bibliothèque auront alors charge d'implémenter cette interface<sup>3</sup> (ou de choisir une implémentation existante) pour fournir la dépendance.

1. Ou simplement parce que vous voulez avoir le contrôle de l'évolution de cette interface.
2. Si on utilise JMS : ce sera un des packages exportés.
3. Typiquement, les utilisateurs emploieront le patron « adaptateur » pour implémenter l'interface fournie à partir de diverses classes existantes.
4. Le « D » de SOLID (Michael Feathers & Robert C. Martin)

(remarquer le sens des flèches entre les 2 packages)

### Pourquoi faire cela ?

- l'interface écrite est idéale et facile à utiliser pour programmer la bibliothèque
- ses évolutions restent sous le contrôle de l'auteur de la bibliothèque, qui ne peut donc plus être « cassée » du fait de quelqu'un d'autre
- la bibliothèque étant « programmée à l'interface », elle sera donc polymorphe.

### Pourquoi dit-on « inversion » ?

Parce que le code source de la bibliothèque qui dépend, à l'exécution, d'un composant supposé plus « concret »<sup>1</sup>, ne dépend pas statiquement de la classe implémentant ce dernier. Selon le DIP, c'est le contraire qui se produit (dépendance à l'interface).

En termes plus savants :

- « Depend upon Abstractions. Do not depend upon concretions. »<sup>2</sup>.
- et donc d'implémentation susceptible de changer plus souvent (justification du DIP par son inventeur)
  - Robert C. Martin (2000), dans "Design Principles and Design Patterns".

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Mémoire et JVM	
Sous-Objets	
Transfertage	
Polymporphisme(s)	
Surcharge	
Interfaces	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	

### Pourquoi faire cela ?

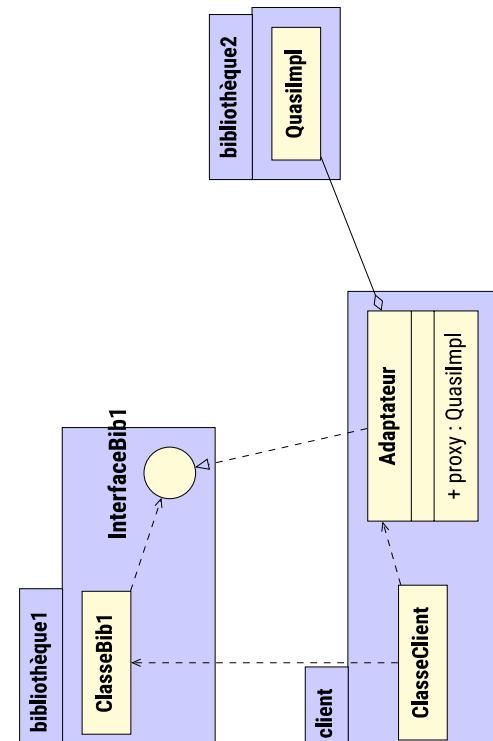
- vous voulez utiliser une bibliothèque dont les méthodes ont des paramètres typés par une certaine interface **I**.
- mais vous ne disposez pas de classe implémentant **I**.
- cependant, une autre bibliothèque vous fournit une classe **C** contenant la même fonctionnalité que **I** (ou presque)

### Quand ?

- vous voulez utiliser une classe de la forme suivante :
- ```
public class CToIAdapter implements I {
    private final C proxy;
    public CToIAdapter(C proxy) { this.proxy = proxy; }
    ...
}
```
- et dans laquelle les méthodes de **I** sont implémentées<sup>1</sup> par des appels de méthodes sur **proxy**.

- De préférence très simplement et brièvement...

| Introduction                      | Compléments en 200 |
|-----------------------------------|--------------------|
| Généralités                       | Aldric Degorre     |
| Style                             |                    |
| Objets et classes                 |                    |
| Types et polymorphisme            |                    |
| Mémoire et JVM                    |                    |
| Sous-Objets                       |                    |
| Transfertage                      |                    |
| Polymporphisme(s)                 |                    |
| Surcharge                         |                    |
| Interfaces                        |                    |
| Héritage                          |                    |
| Généricité                        |                    |
| Concurrence                       |                    |
| Interfaces graphiques             |                    |
| Gestion des erreurs et exceptions |                    |



| Introduction                      | Compléments en 200 |
|-----------------------------------|--------------------|
| Généralités                       | Aldric Degorre     |
| Style                             |                    |
| Objets et classes                 |                    |
| Types et polymorphisme            |                    |
| Mémoire et JVM                    |                    |
| Sous-Objets                       |                    |
| Transfertage                      |                    |
| Polymporphisme(s)                 |                    |
| Surcharge                         |                    |
| Interfaces                        |                    |
| Héritage                          |                    |
| Généricité                        |                    |
| Concurrence                       |                    |
| Interfaces graphiques             |                    |
| Gestion des erreurs et exceptions |                    |