

Introduction	Généralités Programmation orientée objet Java Compilation et exécution	Style Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
<ul style="list-style-type: none"> Principe de la POO : des messages¹⁶ s'échangent entre des objets qui les traitent pour faire progresser le programme. → P00 = paradigme centré sur la description de la communication entre objets. Pour faire communiquer un objet a avec un objet b, il est nécessaire et suffisant de connaître les messages que b accepte : l'interface de b. Ainsi objets de même interface interchangeable → polymorphisme. Fonctionnement interne d'un objet¹⁷ caché au monde extérieur → encapsulation. <p>Pour résumer : la POO permet de raisonner sur des abstractions des composants réutilisés, en ignorant leurs détails d'implémentation.</p> <p>16. appels de méthodes</p> <p>17. Notamment son état, représenté par des attributs.</p>								

Introduction	Généralités Programmation orientée objet Java Compilation et exécution	Style Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
<p>La POO permet de découper un programme en <u>composants</u></p> <ul style="list-style-type: none"> peu dépendants les uns des autres (faible couplage) <ul style="list-style-type: none"> → code robuste et évolutif (composants testables et déboguables indépendamment et aisément remplaçables); réutilisables, au sein du même programme, mais aussi dans d'autres; <ul style="list-style-type: none"> → facilite la création de logiciels de grande taille. <p>POO → (discutable) façon de penser naturelle pour un cerveau humain "normal"¹⁸.</p> <p>18. Non "déformé" par des connaissances mathématiques pointues comme la théorie des catégories (cf. programmation fonctionnelle).</p>								

« Java » (Java SE) est en réalité une plateforme de programmation caractérisée par :

- le langage de programmation Java
 - orienté objet à classes,
 - à la syntaxe inspirée de celle du langage C²⁴,
 - au typage statique,
 - à gestion automatique de la mémoire, via son ramasse-miettes (garbage collector).
 - sa machine virtuelle (JVM²⁵), permettant aux programmes Java d'être multi-plateforme (le code source se compile en code-octet pour JVM, laquelle est implémentée pour nombreux types de machines physiques).
 - les bibliothèques officielles du JDK (fournissant l'API²⁶ Java), très nombreuses et bien documentées (+ nombreuses bibliothèques de tierces parties).
24. C sans pointeurs et **struct** \simeq Java sans objet
25. Java Virtual Machine
26. Application Programming Interface

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Aldric Degoire

Alain Degorre

Compléments en POO

Introduction

Généralités

Programmation orientée objet

Java

Compilation et exécution

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Introduction	Généralités Programmation orientée objet Java Compilation et exécution	Style Objets et classes	Types et polymorphisme Héritage Généricité	Concurrence Interfaces graphiques Gestion des erreurs et exceptions	Révision
« JVM » est le standard de machine virtuelle pour Java publié ³⁶ par Oracle.	<ul style="list-style-type: none"> L'implémentation par Oracle est appelée <i>HotSpot</i>. Autres implémentations de la JVM : notamment <i>OpenJ9</i>, par la fondation Eclipse. Autres VMs pour java (incompatibles avec JVM) : notamment <i>Dalvik</i> (sous Android). On peut même compiler Java vers d'autres 'cibles' : code natif (cf GCJ, cf ART dans Android ≥ 5, jaotc/Graal dans Java ≥ 9, ...), voire vers autre langage (ex : Javascript, via GWT). Vice-versa, d'autres langages que Java sont compilés pour la JVM : Scala, Groovy, Clojure, Gosu, Ceylon, Kotlin... 				

36. Spécifié par la "JVM^S" :
<https://docs.oracle.com/javase/specs/jvms/se11/html/index.html>

Introduction	Généralités Programmation orientée objet Java Compilation et exécution	Style Objets et classes	Types et polymorphisme Héritage Généricité	Concurrence Interfaces graphiques Gestion des erreurs et exceptions	Supplément
Est-ce que le bytecode est vraiment juste interprété par la JVM ?					

En réalité, plusieurs stratégies :

- Simple interprétation du code-octet au fur et à mesure de son exécution.
- JIT, « Just In Time Compilation » : pendant l'exécution du programme, la VM traduit (une bonne fois pour toutes) en code natif optimisé les morceaux de code qui s'exécutent souvent
- AOT, « Ahead Of Time Compilation » : "on" compile tout ou partie du code-octet vers des instructions natives avant son exécution

La JVM HotSpot (JVM par défaut depuis Java 3), fait du JIT. Depuis ~ Java 9, Oracle expérimente AOT via GraalVM.

37. Dans IntelliJ IDEA, ces modules correspondent désormais aux modules du projet, subdivision déjà proposée par cet IDE, avant Java 9.
 38. C.-à-d. un ensemble de packages ou modules partageant une configuration commune dans un IDE (comme Eclipse, NetBeans, IntelliJ IDEA, ...) ou dans un moteur de production (make, ant, maven, gradle, ...).
 Dans Eclipse, en plus, un « espace de travail » regroupe les projets apparaissant dans une même fenêtre.

Le code compilé :

- organisé de façon similaire au code source.
- Mais, à chaque un fichier `.java` correspond (au moins) un fichier `.class`.
- Un programme compilé est distribuable via une ou des archives `.jar`³⁹.
- Si on utilise JPMs, il y a exactement un fichier `.jar` par module.

39. C'est en réalité un fichier `.zip` avec quelques métadonnées supplémentaires.

40. Si un code source contient plus de commentaires que de code, c'est en réalité assez "louche".

- lisible par le programmeur d'origine
- lisible par l'équipe qui travaille sur le projet
- lisible par toute personne susceptible de travailler sur le code source (pour le logiciel libre : la Terre entière !)

Les commentaires⁴⁰ et la javadoc peuvent aider, mais rien ne remplace un code source bien écrit.

- le compilateur (la syntaxe de Java⁴¹)
 - le guide⁴² de style qui a été publié par Sun en même temps que le langage Java (→ conventions à vocation universelle pour tout programmeur Java)
 - les directives de son entreprise/organisation
 - les directives propres au projet
- ... et ainsi de suite (il peut y avoir des conventions internes à un package, à une classe, etc.)
- » et enfin... le bon sens!⁴³

Nous parlerons principalement du 2ème point et des conventions les plus communes.

41. L'équivalent du livre de grammaire dans l'analogie avec la langue vivante.

42. À rapprocher des avis émis par l'Académie Française ?

43. Mais le bon sens ne peut être acquis que par l'expérience.

Nommer les entités

(classes, méthodes, variables, ...)

Nommer les entités

Codage, et langue

Introduction	Généralités	Style	Noms	Métrique	Commentaires	Patrons de conception	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
Règles de capitalisation pour les noms (auxquelles on ne déroge pratiquement jamais) :			• ... de classes, interfaces, énumérations et annotations ⁴⁴ → UpperCase[Case										

- ... de classes, interfaces, énumérations et annotations⁴⁴ → **UpperCase[Case**
- ... de variables (locales et attributs), méthodes → **lowerCase[Case**
- ... de constantes (**static final** ou valeur d'**enum**) → **SCREAMING_SNAKE_CASE**
- ... de packages → tout en minuscules sans séparateur de mots⁴⁵. Exemple :

```
com.masociete.bibliotheque.truc46.
```

→ rend possible de reconnaître à la première lecture quel genre d'entité un nom désigne.

44. C.-à-d. tous les types référence
 45. “ ” autorisé si on traduit des caractères invalides, mais pas spécialement encouragé
 46. pour une bibliothèque éditée par une société dont le nom de domaine internet serait masociete.com

Introduction	Généralités	Style	Noms	Métrique	Commentaires	Patrons de conception	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
• Se restreindre aux caractères suivants :			• a-z, A-Z : les lettres minuscules et capitales (non accentuées), • 0-9 : les chiffres, • _ : le caractère soulignement (seulement pour snake_case).										

Explication :

- \$ (dollar) est autorisé mais réservé au code automatiquement généré;
 - les autres caractères ASCII sont réservés (pour la syntaxe du langage);
 - la plupart des caractères unicode non-ASCII sont autorisés (p. ex. caractères accentués), mais aucun standard de codage imposé pour les fichiers .java.⁴⁷
- **Interdits** : commencer par **0-9**; prendre un nom identique à un mot-clé réservé.
- **Recommandé** : Utiliser l'**Anglais américain** (pour les noms utilisés dans le programme **et** les commentaires **et** la javadoc).
47. Si l'environnement d'exécution n'a pas le même réglage par défaut → incompatibilité du code source.

Introduction	Généralités	Style	Noms	Métrique	Commentaires	Patrons de conception	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
Nature grammaticale (1)	Compléments en POO	Aldric Degoire											

Introduction	Généralités	Style	Noms	Métrique	Commentaires	Patrons de conception	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
Nature grammaticale des identifiants :													

Introduction	Généralités	Style	Noms	Métrique	Commentaires	Patrons de conception	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
Les noms de méthodes contiennent généralement un verbe , qui est :													

- **get** si c'est un accesseur en lecture ("getteur") ; ex : **String getName()** ;
 - **is** si c'est un accesseur en lecture d'une propriété booléenne ; ex : **boolean isInitialized()** ;
 - **set** si c'est un accesseur en écriture ("setteur") ; ex : **void getName(String name)** ;
 - tout autre verbe, à l'indicatif, si la méthode retourne un booléen (méthode prédictif) ; à l'impératif⁴⁹, si la méthode sert à effectuer une action avec effet de bord⁵⁰ ;
 - au participe passé si la méthode retourne une version transformée de l'objet, sans Modifier l'objet (ex : **List.sort()**).
49. ou infinitif sans le "to", ce qui revient au même en Anglais
50. c.-à-d. mutation de l'état ou effet physique tel qu'un affichage; cela s'oppose à fonction pure qui effectue juste un calcul et en retourne le résultat

Nommer les entités par ligne

Nommer les entités

Concision versus information	Compléments en POO
<ul style="list-style-type: none">Pour tout identificateur, il faut trouver le bon compromis entre information (plus long) et facilité à l'écrire (plus court).Typiquement, plus l'usage est fréquent et local, plus le nom est court : ex. : variables de boucle <code>for (int idx = 0; idx < anArray.length; idx++) { ... }</code>plus l'usage est lointain de la déclaration, plus le nom doit être informatif (sont particulièrement concernés : classes, membres publics... mais aussi les paramètres des méthodes !) ex. : paramètres de constructeur <code>Rectangle(double centerX, double centerY, double width, double length){ ... }</code> <p>Toute personne lisant le programme s'attend à une telle stratégie → ne pas l'appliquer peut l'induire en erreur.</p>	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Noms</p> <p>Métrique</p> <p>Commentaires</p> <p>Patrons de conception</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrency</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>

Nombre de caractères par ligne

Où couper les lignes	Compléments en POO
<ul style="list-style-type: none">On limite le nombre de caractères par ligne de code. Raisons :<ul style="list-style-type: none">certains programmeurs préfèrent désactiver le retour à la ligne automatique⁵¹ ;même la coupure automatique ne se fait pas forcément au meilleur endroit;longues lignes illisibles pour le cerveau humain (même si entièrement affichées);certains programmeurs aiment pouvoir afficher 2 fenêtres côté à côté.Limite traditionnelle : 70 caractères/ligne (les vieux terminaux ont 80 colonnes⁵²).De nos jours (écrans larges, haute résolution), 100-120 est plus raisonnable⁵³.Arguments contre des lignes trop petites :<ul style="list-style-type: none">découpage trop élémentaire rendant illisible l'intention globale du programme;incitation à utiliser des identifiants plus courts pour pouvoir écrire ce qu'on veut en une ligne (→ identifiants peu informatifs, mauvaise pratique).	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Noms</p> <p>Métrique</p> <p>Commentaires</p> <p>Patrons de conception</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrency</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>

⁵¹. De plus, historiquement, les éditeurs de texte n'avaient pas le retour à la ligne automatique.

⁵². Et d'où vient ce nombre 80 ? C'est le nombre du de colonnes dans le standard de cartes perforées d'IBM inventé en... 1928 ! Et pourquoi ce choix en 1928 ? Parce que les machines à écrire avaient souvent 80 colonnes... bref c'est de l'histoire très ancienne !

⁵³. Selon moi, mais attention, c'est un sujet de débat houleux !

Où couper les lignes

Indentation	Compléments en POO
<p>Indenter = mettre du blanc en tête de ligne pour souligner la structure du programme. Ce blanc est constitué d'un certain nombre d'<u>indentations</u>.</p> <ul style="list-style-type: none">En Java, typiquement, 1 indentation = 4 espaces (ou 1 tabulation).Le nombre d'indentations est égal à la profondeur syntaxique du début de la ligne ≈ nombre de paires de symboles⁵⁴ ouvertes mais pas encore fermées.⁵⁵Tout éditeur raisonnablement évolué sait indenter automatiquement (règles paramétrables dans l'éditeur). Pensez à demander régulièrement l'indentation automatique, afin de vérifier qu'il n'y a pas d'erreur de structure! <p>Exemple :</p> <p>voici un exemple (qui n'est pas du Java ; mais suit ses "conventions d'indentation")</p> <p>54. Parenthèses, crochets, accolades, guillemets, chevrons, ...</p> <p>55. Pas seulement : les règles de priorité des opérations créent aussi de la profondeur syntaxique.</p>	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Noms</p> <p>Métrique</p> <p>Commentaires</p> <p>Patrons de conception</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrency</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>

Taille des classes	Nombre de paramètres des méthodes	Et ainsi de suite	
<p>Quelle est la bonne taille pour une classe ?</p> <ul style="list-style-type: none"> Déjà, plusieurs critères de taille : nombre de lignes, nombre de méthodes, Le découpage en classes est avant tout guidé par l'<u>abstraction objet</u> retenue pour modéliser le problème qu'on veut résoudre. En pratique, une classe trop longue est désagréable à utiliser. Ce désagrément traduit souvent une décomposition insuffisante de l'abstraction.⁵⁶ Conseil : se fixer une limite de taille et décider, au cas par cas, si et comment il faut "réparer" les classes qui dépassent la limite (cela incite à améliorer l'aspect objet du programme). En général, pour un projet en équipe, suivre les directives du projet. <p>56. Le « S » de « SOLID » : <u>single responsibility principle/principe de responsabilité unique</u>.</p>	<p>Autre critère : le nombre de paramètres.</p> <p>Trop de paramètres (>4) implique :</p> <ul style="list-style-type: none"> Une signature longue et illisible. Une utilisation difficile ("ah mais ce paramètre là, il était en 5e ou en 6e position, déjà ?") <p>Il est souvent possible de réduire le nombre de paramètres.</p> <ul style="list-style-type: none"> en utilisant la surcharge, ou bien en séparant la méthode en plusieurs méthodes plus petites (en décomposant la tâche effectuée), ou bien en passant des objets composites en paramètre <p>Voir aussi : patron "monteur" (le constructeur prend pour seul paramètre une instance du <u>Builder</u>).</p>	<p>Compléments en POO Aldric Degorre</p> <p>Introduction Généralités Style Noms Métrique Commentaires Patrons de conception Objets et classes Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions</p> <p>Introduction Généralités Style Noms Métrique Commentaires Patrons de conception Objets et classes Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions</p>	
		<p>Pour une méthode, la taille est le nombre de lignes.</p> <ul style="list-style-type: none"> <u>Principe de responsabilité unique</u>⁵⁷ : une méthode est censée effectuer une tâche précise et compréhensible. <p>→ Un excès de lignes</p> <ul style="list-style-type: none"> nuit à la compréhension; peut traduire le fait que la méthode effectue en réalité plusieurs tâches probablement séparables. <p>Quelle est la bonne longueur ?</p> <ul style="list-style-type: none"> Mon critère⁵⁸ : on ne peut pas bien comprendre une méthode si on ne peut pas la parcourir en un simple coup d'œil → faire en sorte qu'elle tienne en un écran (~ 30-40 lignes max.) En général, suivre les directives du projet. <p>57. Oui, là aussi!</p> <p>58. qui n'engage que moi !</p>	<p>Compléments en POO Aldric Degorre</p> <p>Introduction Généralités Style Noms Métrique Commentaires Patrons de conception Objets et classes Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions</p>

Commentaires

Plusieurs sortes de commentaires (1)

- En ligne :
 - `int length; // length of this or that`
 - Pratique pour un commentaire très court tenant sur une seule ligne (ou ce qu'il en reste...)
 - en bloc :

```
/*
 * Un commentaire un peu plus long.
 * Les "*" intermédiaires ne sont pas obligatoires, mais Eclipse
 * les ajoute automatiquement pour le style. Laissez-les !
 */
```

Commentaires
La lavanda

nataires
à laquelle

- Propos de la JavaDoc :
 - Les commentaires au format JavaDoc sont compilables en document HTML (dans Eclipse : menu "Project", "Generate JavaDoc...")
 - Pour toute déclaration de type (classe, interface, enum...) ou de constructeur, méthode), un squelette de documentation au bon format (balises) peut être généré avec la combinaison **Alt+Shift** dans Eclipse).
 - Il est **indispensable** de documenter tout ce qui est public.
 - Il est **fortement recommandé** de documenter tout ce qui n'est pas utilisable par d'autres programmeurs, qui n'ont pas accès au code.
 - Il est **utile** de documenter ce qui est privé, pour soi-même et les autres de l'équipe.

Commentaires

Plusieurs sortes de commentaires (2)

- en bloc JavaDoc :

```
    * Returns an expression equivalent to current expression, in which  
    * every occurrence of unknown var was substituted by the expression  
    * specified by parameter by.
```

* @param var
* @param by
* @return
* @throws UnknownExprException
*/
expression subst(UnknownExpr var, Expression by);

Patrons de conception (1) au design narratif

ESSAYS IN LITERATURE

- Analogie langage naturel : patron de conception = figure de style
 - Ce sont des stratégies standardisées et éprouvées pour arriver à une fin.
ex : créer des objets, décrire un comportement ou structurer un programme
 - Les utiliser permet d'éviter les erreurs les plus courantes (pour peu qu'on utilise le bon patron !) et de rendre ses intentions plus claires pour les autres programmeurs qui connaissent les patrons employés.
 - Connaître les noms des patrons permet d'en discuter avec d'autres programmeurs.⁵⁹

559. De la même façon qu'apprendre les figures de style en cours de Français, permet de discuter avec d'autres personnes de la structure d'un texte...

Compléments
en POO
Aldric Degorre

Objets

Autre vision bas niveau, plus en phase avec le haut niveau

- **Objet** = ensemble d'informations regroupées en un certain nombre d'enregistrements contigüs, se référençant les uns les autres, tel que tout est accessible depuis un enregistrement principal⁶⁵.
- C'est donc un graphe orienté connexe dont les nœuds sont des enregistrements et les arcs les référencements.

L'enregistrement principal est une origine de ce graphe.

- Les informations stockées dans ce graphe fournissent les services (méthodes) prévus par le type (interface) de l'objet.

Si nécessaire, on peut distinguer cette notion de celle d'**« objet simple »** (= **struct**), en utilisant l'**expression graphe d'objet**.

Introduction
Généralités
Style
Objets et classes
Objets et commentaires
Nombres et constantes
Encapsulation
Trans mutables
Types et polymorphisme
Héritage
Généricité
Concurrence
Interfaces graphiques
Gestion des erreurs et exceptions
Discussion

65. "l'"objet" visible depuis le reste du programme

Classes	Exemple
Pour l'objet juste donné en exemple, la classe Personne pourrait être :	<pre>public class Personne { // attributs private String nom; private int age; private boolean marie; // constructeur public Personne(String nom, int age, boolean marie) { this.nom = nom; this.age = age; this.marie = marie; } // méthodes (ici : accesseurs) public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } public int getAge() { return age; } public void setAge(int age) { this.age = age; } public boolean getMarie() { return marie; } public void setMarie(boolean marie) { this.marie = marie; } }</pre>

Compléments en POO	Aldric Degorre
Introduction	Généralités
Style	Objets et classes
Objets et classes	Membres et contexte
Encapsulation	Types mutables
Types et polymorphisme	Types et polymorphisme
Héritage	Héritage
Généricité	Généricité
Concurrence	Concurrence
Interfaces graphiques	Interfaces graphiques
Gestion des erreurs et exceptions	Gestion des erreurs et exceptions
Exemple	Exemple
<h1>Classes</h1> <p>Langages à prototypes, langages à classes</p>	
<ul style="list-style-type: none">• Besoin : créer de nombreux objets similaires (même interface, même schéma de données).• 2 solutions → 2 familles de LOO :<ul style="list-style-type: none">• LOO à classes (Java et la plupart des LOO) : les objets sont instanciés à partir de la description donnée par une classe;• LOO à prototypes (toutes les variantes d'ECMAScript dont JavaScript; Self, Lисааc, ...) : les objets sont obtenus par extension d'un objet existant (le <u>prototype</u>).	
	→ l' existence de classes n'est pas une nécessité en POO

Classes

Classes

Compléments en POO

Aldric Degorre

Exemple (le même en UML)

Compléments en POO

Aldric Degorre

Points de vue pertinents pour Java

Personne	
- nom : String	
- age : int	
- marie : boolean	
+ << Create >> Personne(nom : String, age : int, marie : boolean) : Personne	
+ getNom() : String	
+ setNom(nom : String)	
+ getAge() : int	
+ setAge(age : int)	
+ getMarie() : boolean	
+ setMarie(marie : boolean)	

Exemple

Classe = patron/modèle/moule/... pour définir des objets similaires ⁶⁷ .	
Autres points de vue :	
Classe =	
• ensemble cohérent de <u>définitions</u> (champs, méthodes, types auxiliaires, ...), en principe relatives à un même type de données ⁶⁸	
• conteneur permettant l' <u>encapsulation</u> (= limite de visibilité des membres privés). ⁶⁸	

- 67. "similaires" = utilisables de la même façon (même type) et aussi structurés de la même façon.
- 68. Remarque : en Java, l'encapsulation se fait par rapport à la classe et au paquetage et non par rapport à l'objet. En Scala, p. ex., un attribut peut avoir une visibilité limitée à l'objet qui le contient.

Instanciation

De la classe à l'objet

Classes	
Autres points de vue (non-OO) :	

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Objets et classes

Membres et constructeurs

Encapsulation

Types imbriqués

Types polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Révision

Classes	
Autres points de vue (non-OO) :	

Introduction

Généralités

Style

Objets et classes

Objets et classes

Membres et constructeurs

Encapsulation

Types imbriqués

Types polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Révision

- Une classe permet de "fabriquer" plusieurs objets selon un même modèle : les instances⁷⁰ de la classe.
- Ces objets ont le même type, dont le nom est celui de la classe.
- La fabrication d'un objet s'appelle l'**instanciation**. Celle-ci consiste à

• réservoir la mémoire ($\simeq \text{malloc}$ en C)

• initialiser les données⁷¹ de l'objet

On instancie la classe Truc via l'expression "**new Truc(params)**", dont la valeur

est une référence vers un objet de type Truc nouvellement créé.⁷²

Les aspects ci-dessus sont pertinents en Java, mais ne retenir que ceux-ci serait manquer l'essentiel : i.e. : **classe = concept de POO**.

Java force à tout définir dans des classes → encourage cet usage détourné de la construction **class**.

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Objets et classes

Membres et constructeurs

Encapsulation

Types imbriqués

Types polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Révision

70. En POO, "instance" et "objet" sont synonymes. Le mot "instance" souligne l'appartenance à un type.

71. En l'occurrence : les attributs d'instance déclarés dans cette classe.

72. Ainsi, on note que le type défini par une classe est un type référence.

Constructeur : fonction⁷³ servant à construire une instance d'une classe.

• Déclaration :

```
MacClasse(* paramètres * / {  
    // instructions ; ici "this" désigne l'objet en construction  
}
```

NB : même nom que la classe, pas de type de retour, ni de **return** dans son corps.

• Typiquement, “// instructions” = initialisation des attributs de l’instance.

• Appel toujours précédé du mot-clé **new**:

```
MacClasse monObjet = new MacClasse(... paramètres...);
```

Cette instruction déclare un objet **monObjet**, crée une instance de **MacClasse** et l'affecte à **monObjet**.

73. En toute rigueur, un constructeur n'est pas une méthode. Notons tout de même les similitudes dans les syntaxes de déclaration et d'appel et dans la sémantique (exécution d'un bloc de code).

Il est possible de :

- définir plusieurs constructeurs (tous le même nom → cf. surcharge);
- définir un constructeur secondaire à l'aide d'un autre constructeur déjà défini : sa première instruction doit alors être **this**(**paramsAutreConstructeur**) ;⁷⁴
- ne pas écrire de constructeur :

- Si on ne le fait pas, le compilateur ajoute un **constructeur par défaut** sans paramètre.⁷⁵
- Si on a écrit un constructeur, alors il n'y a pas de constructeur par défaut⁷⁶.

74. Ou bien **super**(**params**) ; si utilisation d'un constructeur de la superclasse.

75. Les attributs restent à leur valeur par défaut (0, **false** ou **null**), ou bien à celle donnée par leur initialiseur. S'il y en a un.

76. Mais rien n'empêche d'écrire, en plus, à la main, un constructeur sans paramètre.

```
public class Personne {  
    // attributs  
    public static int derNumINSEE = 0;  
    public final NomCompletnom;  
    public final int numINSEE;  
    // constructeur  
    public Personne(String nom, String prenom) {  
        this.nom = new NomCompletnom, prenom);  
        this.numINSEE = ++derNumINSEE;  
    }  
    // méthode  
    public String toString() {  
        return String.format("%s,%d", nom.nom, nom.prenom, numINSEE);  
    }  
}  
// et même... classe imbriquée!  
public static final class NomCompletnom {  
    public final String nom;  
    public final String prenom;  
    private NomCompletnom(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}
```

Le **corps** d'une classe **C** consiste en une séquence de définitions : constructeurs⁷⁷ et **membres** de la classe.

Plusieurs catégories de membres : attributs, méthodes et types membres⁷⁸.

Un membre **m** peut être

- soit non statique ou d'instance (relatif à une instance de **C**)

Utilisable en écrivant « **m** » n'importe où ou un **this** (**récepteur** implicite) de type **C** existe et ailleurs en écrivant « **recepteurDeTypeC.m** ».

- soit statique (relatif à la classe **C**) → mot-clé **static** dans déclaration.

Utilisable sans préfixe dans le corps de **C** et ailleurs en écrivant « **C.m** ».

Les **membres d'un objet** donné sont les membres non statiques de la classe de l'objet.

77. D'après la JLS 8.2, les constructeurs ne sont pas des membres. Néanmoins, sont déclarés à l'intérieur d'une classe et acceptent, comme les membres, les modificateurs de visibilité (**private**, **public**, ...).

78. Souvent abusivement appelés "classes internes".

Accéder à un membre : où et comment ? → notion de contexte

Contexte (associé à tout point du code source) :

- dans une définition⁷⁹ statique : contexte = la classe contenant la définition;
- dans une définition non-statique : contexte = l'objet "courant", le **récepteur**⁸⁰.

Désigner un membre **m** déjà défini quelque part :

- écrire soit juste **m** (**nom simple**), soit **chemin.m** (**nom qualifié**)

Membre non statique = lié à (la durée de vie et au contexte d') une instance.		En bref : Membre non statique = lié à (la durée de vie et au contexte d') une classe ⁸² .	
Membre statique = lié à (la durée de vie et au contexte d') une classe ⁸² .		En bref : Membre statique = lié à (la durée de vie et au contexte d') une classe ⁸² .	
Style	statique (ou "de classe")	non statique (ou "d'instance")	non statique (ou "d'instance")
Attribut	donnée globale ⁸³ , commune à toutes les instances de la classe.	donnée propre ⁸⁴ à chaque instance (nouvel exemplaire de cette variable alloué et initialisé à chaque instanciation).	donnée propre ⁸⁴ à chaque instance (nouvel exemplaire de cette variable alloué et initialisé à chaque instanciation).

82. +permanent et « global ». NB : ça ne veut pas dire visible de partout ; **static private** est possible!

83. Correspond à variable globale dans d'autres langages.

84. Correspond à champ de **struct** en C.

Membres d'une classe

Compléments en POO

Introduction

Style	Objets et classes	statique (ou "de classe")	non statique (ou "d'instance")
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	donnée globale ⁸³ , commune à toutes les instances de la classe.	donnée propre ⁸⁴ à chaque instance (nouvel exemplaire de cette variable alloué et initialisé à chaque instanciation).
Types et polymorphisme	Héritage	méthode	message à instance concernée : le récepteur de la méthode (this).
	Généricité	type membre	comme statique, mais instances contenant une référence vers instance de la classe englobante.
	Concurrence		
	Interfaces graphiques		
	Gestion des erreurs et exceptions		
Détails	Résumé		

Compléments en POO

Introduction

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Compléments en POO

Introduction

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur le cas des attributs	
Style	Objets et classes	Style	Objets et classes
Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types imbriqués	Attribut	Objets et classes objets et classes Membres & constantes Encapsulation Types polymorphisme Héritage

Membres statiques et membres non-statiques

Compléments en POO

Membres statiques et membres non-statiques		Membres statiques et membres non-statiques	
Zoom sur le cas des attributs		Zoom sur	

Fabriques statiques et membres non-statiques

Remarque, on peut réécrire une méthode statique comme non statique de même comportement, et vice-versa :

```
class C { // ici f et g font la même chose
    void f() { instr(this); } // exemple d'appel : x.f()
    static void g(C that) { instr(that); } // exemple d'appel : C.g(x)
}
```

Mais différences essentielles :

- pour que **this** soit de type **C**, doit être déclarée dans **C** alors qu'il n'y a pas de relation entre le lieu de déclaration de **g** et le type de **that**.
→ Conséquences en termes de visibilité/encapsulation.
- Les appels **x.f()** et **C.g(x)** sont équivalents si **x** est instance directe de **C**.
Mais c'est faux si **x** est instance de **D**, sous-classe de **C** redéfinissant **f** (cf. héritage), car la version redéfinie sera appelée : **f** est sujette à la liaison dynamique.

Problème, les limitations des constructeurs :

- même nom pour tous, qui ne renseigne pas sur l'usage fait des paramètres ;
- impossibilité d'avoir 2 constructeurs avec la même signature ;
- si appel à constructeur auxiliaire, nécessairement en première instruction ;
- obligation de retourner une nouvelle instance → pas de contrôle d'instances
- obligation de retourner une instance directe de la classe.

En écrivant une **fabrique statique** on contourne toutes ces limitations :

```
public abstract class C { // ou bien interface
    ... / la fabrique :
    public static C of(D arg) {
        if (arg ...) return new CImpl1(arg);
        else if (arg ...) return ...
        else return ...
    }
}
```

85. I.e.: possibilité de choisir de réutiliser une instance existante au lieu d'en créer une nouvelle.

Encapsulation

Compléments en POO
Aldric Degorre

Introduction
Généralités
Style
Objets et classes
Bijoux et classes
Membres et contextes
É encapsulation
Types imbriqués
Types et polymorphisme
Héritage
Généricité
Concurrence
Interfaces graphiques
Gestion des erreurs et exceptions

L'encapsulation

Encapsulation
Concept essentiel de la POO

Introduction
Généralités
Style
Objets et classes
Bijoux et classes
Membres et contextes
É encapsulation
Types imbriqués
Types et polymorphisme
Héritage
Généricité
Concurrence
Interfaces graphiques
Gestion des erreurs et exceptions

= restriction de l'accès depuis l'extérieur aux choix d'implémentation internes.

• **bonne pratique** favorisant la pérennité d'une classe.
Minimiser la « surface » qu'une classe expose à ses clients⁸⁶ (= en réduisant leur **couplage**) facilite son déboguage et son évolution future.⁸⁷

• empêche les clients d'accéder à un objet de façon incorrecte ou non prévue. Ainsi,
la correction d'un programme est plus facile à vérifier (moins d'interactions à vérifier);
plus généralement, seuls les **invariants de classe**⁸⁸ ne faisant pas intervenir d'attributs non privés peuvent être prouvés.

→ L'encapsulation rend donc aussi la classe plus fiable.

86. **Clients** d'une classe : les classes qui utilisent cette classe.
87. En effet : on peut modifier la classe sans modifier ses clients.
88. Différence avec l'item du dessus : les invariants de classe doivent rester vrais dans tout contexte d'utilisation de la classe, pas seulement dans le programme courant.

Encapsulation

Exemple

Est-il vrai que « le n ème appel à `next` retourne le n ème terme de la suite de Fibonacci » ?

Pas bien :

```
public class Fibogen {
    public int a = 1;
    public int next() {
        int ret = a; a = b; b += ret;
        return ret;
    }
}
```

Toute autre classe peut interférer en modifiant directement les valeurs de `a` ou `b`

→ on ne peut rien prouver !

89. Or il y a un bug, en théorie, si on exécute `next` plusieurs fois simultanément (sur plusieurs threads).

Exemple

Bien : (ou presque)

```
public class Fibogen {
    private int a = 1, b = 1;
    public int next() {
        int ret = a; a = b; b += ret;
        return ret;
    }
}
```

Seule la méthode `next` peut modifier directement les valeurs de `a` ou `b`

→ S'il y a un bug, c'est dans la méthode `next` et pas ailleurs !⁸⁹

Introduction

Généralités

Style

Objets et classes

Méthodes et constructeurs

Types mutables

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Héritage

Généricité

Encapsulation

Niveaux de visibilité : avertissements

- Toute déclaration de membre non **private** est susceptible d'être utilisée par un autre programmeur dès lors que vous publiez votre classe.
- Elle fait partie de l'API⁹² de la classe.
- vous devez donc **la documenter**⁹³ (EJ3 Item 56)
- et vous vous engagez à **ne pas modifier**⁹⁴ sa spécification⁹⁵ dans le futur, sous peine de "casser" tous les clients de votre classe.

Ainsi il faut bien réfléchir à ce que l'on souhaite exposer.⁹⁶

92. Application Programming Interface
93. cf. JavaDoc
94. On peut modifier si ça va dans le sens d'un renforcement compatible.
95. Et, évidemment, à faire en sorte que le comportement réel respecte la spécification!
96. Il faut aussi réfléchir à une stratégie : tout mettre en **private** d'abord, puis relâcher en fonction des besoins ? Ou bien le contraire ? Les opinions divergent !

Encapsulation

Niveaux de visibilité et documentation

Introduction	Généralités
Style	Objets et classes
	Objets et classes Membres et constructeurs
	Encapsulation
	Type intermédiaires
	Types et polymorphisme
	Héritage
	Généricité
	Concurrence
Interfaces graphiques	
Gestion des erreurs et exceptions	
Discussion	

Attention, les niveaux de visibilité ne font pas forcément ce à quoi on s'attend.

- **package-private** → on peut, par inadvertance, créer une classe dans un paquetage déjà existant⁹⁷ → garantie faible.
- **protected** → de même et, en +, toute sous-classe, n'importe où, voit la définition.
- **Aucun niveau ne garantit la confidentialité des données.**
- Constantes : lisibles directement dans le fichier .**Class**.
- Variables : lisibles, via réflexion, par tout programme s'exécutant sur la même JVM.
- Si la sécurité importe : bloquer la réflexion⁹⁸.

L'encapsulation évite les erreurs de programmation mais **n'est pas un outil de sécurité**!⁹⁹

97. Même à une dépendance tierce, même sans recompilation. En tout cas, si on n'utilise pas JPMS.
98. En utilisant un **SecurityManager** ou en configurant **module-info.java** avec les bonnes options.
99. Méditer la différence entre sûreté (*safety*) et sécurité (*security*) en informatique. Attention, cette distinction est souvent faite, mais selon le domaine de métier, la distinction est différente, voire inversée !

Encapsulation

Compléments en POO

Aldric Degorre

Encapsulation

Java Platform Module System (JPMS, Java ≥ 9)

Introduction	Généralités
Style	Objets et classes
	Objets et classes Membres et constructeurs
	Encapsulation
	Type intermédiaires
	Types et polymorphisme
	Héritage
	Généricité
	Concurrence
Interfaces graphiques	
Gestion des erreurs et exceptions	
Discussion	

Encapsulation

Accesseurs (get, set, ...) et propriétés (1)

- Pour les classes publiques, il est recommandé¹⁰¹ de mettre les attributs en **private** et de donner accès aux données de l'objet en définissant des méthodes **public** appelées **accesseurs**.
- Par convention, on leur donne des noms explicites :
 - public T getX()**¹⁰² : retourne la valeur de l'attribut "getteur".
 - public void setX(T nx)** : affecte la valeur nx à l'attribut "setteur".
- Le couple **getX** et **setX** définit la **propriété**¹⁰³ de l'objet qui les contient.
- Il existe des propriétés en lecture seule (si juste getteur) et en lecture/écriture (getteur et setteur).

101. EJ3 Item 16 : "In public classes, use accessor methods, not public fields"
102. Variante : **public boolean isX()**, seulement si **T** est **boolean**.
103. Terminologie utilisée dans la spécification JavaBeans pour le couple getteur+setteur. Dans nombre de LOO (C#, Kotlin, JavaScript, Python, Scala, Swift, ...), les propriétés sont cependant une sorte de membre à part entière supportée par le langage.

Syntaxe du fichier module-info.java :

```
module nom_du_module {
    requires nom_d_un_module_dont_on_depend;
    exports nom_d_un_package_defini_ici;
}
```

Ce sujet sera développé en TP.

100. Et les dépendances sont fermées à la réflexion, mais on peut permettre la réflexion sur un package en le déclarant avec **opens** dans **module-info.java**.

Encapsulation

Accesseurs (get, set, ...) et propriétés (2)

- Une propriété se base souvent sur un attribut (privé), mais d'autres implémentations sont possibles. P. ex. :

```
// propriété "numberOfFingers" :
public int getNumberOfFingers() { return 10; }
```

(accès en lecture seule à une valeur constante → on retourne une expression constante)

- L'utilisation d'acesseurs laisse la possibilité de changer ultérieurement l'implémentation de la propriété, sans changer son mode d'accès public¹⁰⁴. Ainsi, quand cela sera fait, il ne sera pas nécessaire de modifier les autres classes qui accèdent à la propriété.

104. ici, le couple de méthodes `getX()`/`setX()`

Encapsulation

Accesseurs (get, set, ...) et propriétés (3)

Exemple

Compléments
en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Objets et classes

Membres et constantes

Types intérieurs

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des erreurs et exceptions

Exemple : propriété en lecture/écriture avec contrôle validité des données.

```
public final class Person {
    // propriété "age"
    // attribut de base (qui doit rester positif)
    private int age;

    // getteur, accesseur en lecture
    public int getAge() {
        return age;
    }

    // setteur, écriture contrôlée
    public void setAge(int a) {
        if (a >= 0) age = a;
    }
}
```

Exemple

Compléments
en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Membres et constantes

Encapsulation

Types intérieurs

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des erreurs et exceptions

Encapsulation

Accesseurs (get, set, ...) et propriétés (5)

Exemple : propriété en lecture seule avec évaluation paresseuse.

```
public final class Entier {
    public Entier(int valeur) { this.valeur = valeur; }

    private final int valeur;

    // propriété `diviseurs` :
    private List<diviseurs> diviseurs;

    public List<Integer> getDiviseurs() {
        if (diviseurs == null) diviseurs =
            Collections.unmodifiableList(Outils.factorise(valeur)); // <- calcul
        return diviseurs;
    }
}
```

Exemple

Compléments
en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Membres et constantes

Encapsulation

Types intérieurs

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des erreurs et exceptions

Comportements envisageables pour get et set :

- contrôle de validité avant modification;
- initialisation paresseuse : la valeur de la propriété n'est calculée que lors du premier accès (et non dès la construction de l'objet);
- consignation dans un journal pour débogage ou surveillance;
- observabilité : le setteur notifie les objets observateurs lors des modifications;
- vétoabilité : le setteur n'a d'effet que si aucun objet (dans une liste connue de "veto-eurs") ne s'oppose au changement;
- ...

Encapsulation

Accesseurs (get, set, ...) et propriétés (4)

Compléments
en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Membres et constantes

Encapsulation

Types intérieurs

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des erreurs et exceptions

Encapsulation

Aliasing : pourquoi les restrictions de visibilité ne suffisent pas pour garantir l'encapsulation

Introduction

Généralités

Style

Aliasing → objet → ref1

Quand un attribut référence un objet qui est aussi référencé à l'extérieur de cette classe, le bénéfice de l'encapsulation est alors annulé.

À éviter : attribut privé de C → objet → ref2 → reference externe à C

Cela revient¹⁰⁵ à laisser l'attribut en **public**, puisque le détenteur de cette référence peut faire les mêmes manipulations sur cet objet que la classe contenant l'attribut.

105. Quasiment : en effet, si l'attribut est privé, il reste impossible de modifier la valeur de l'attribut, i.e. l'adresse qu'il stocke, depuis l'extérieur.

Discussion

Compléments
en POO

Aliadic Degorre

Introduction

Généralités

Objets et classes

Membres et constantes

É encapsulation

Types imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Compléments
en POO

Aliadic Degorre

Introduction

Généralités

Objets et classes

Membres et constantes

É encapsulation

Types imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Exemple

Lesquelles des classes A, B, C et D garantissent que l'entier contenu dans l'attribut d garde la valeur qu'on y a mise à la construction ou lors du dernier appel à **setData** ?

```
class Data {
    public int x;
    public Data(int x) { this.x = x; }
    public Data copy() { return new Data(x); }
}

class A {
    private final Data d;
    public A(Data d) { this.d = d.copy(); }
}

class B {
    private final Data d;
    // copie défensive (EJ3 Item 50)
    public B(Data d) { this.d = d.copy(); }
    public Data getData() { return d; }
}

class C {
    private final Data d;
    public C(Data d) { this.d = d; }
}

class D {
    private final Data d;
    public void setData(Data d) { this.d = d; }
    public void useData() { Client.use(d); }
}
```

Reviens à répondre à : les attributs de ces classes peuvent-ils avoir des *alias* extérieurs ?

Compléments
en POO

Aliadic Degorre

Introduction

Généralités

Style

Objets et classes

Membres et constantes

É encapsulation

Types imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Détails

Copie défensive

Qu'est-ce que c'est et comment la réalise-t-on ?

- **Copie défensive** = copie profonde réalisée pour éviter des *alias* indésirables.

- **Copie profonde** : technique consistant à obtenir une copie d'un objet « égale »¹⁰⁶ à son original au moment de la copie, mais dont les évolutions futures seront indépendantes.

- **2 cas, en fonction du genre de valeur à copier :**

- Si type primitif ou **immutable**¹⁰⁷, pas d'évolutions futures → une copie directe suffit.
- Si type **mutable** → on crée un nouvel objet dont les attributs contiennent des copies profondes des attributs de l'original (et ainsi de suite, récursivement : on copie le graphe de l'objet¹⁰⁸).

106. La relation d'égalité est celle donnée par la méthode **equals**.

107. Type **immutable** (*immutable*) : type (en fait toujours une classe) dont toutes les instances sont des objets non modifiables.

C'est une propriété souvent recherchée, notamment en programmation concurrente.

Contraire : **mutable** (*mutable*).

108. Il savoir en quoi consiste le graphe de l'objet, sinon la notion de copie profonde reste ambiguë.

Compléments
en POO

Aliadic Degorre

Introduction

Généralités

Style

Objets et classes

Membres et constantes

É encapsulation

Types imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Encapsulation

Aliasing : comment l'empêcher.Compléments
en POO

Aliadic Degorre

Introduction

Généralités

Style

Objets et classes

Membres et constantes

É encapsulation

Types imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Aliasing souvent indésirable (pas toujours !) → il faut savoir l'empêcher. Pour cela :

```
class A {
    // Mettre les attributs sensibles en private :
    private Data data;
    // Et effectuer des copies défensives (EJ3 Item 50)...
    // - de tout objet qu'on souhaite partager,
    //   - qu'il soit retourné par un getteur :
    public Data getData() { return data.copy(); }
    // ou passé en paramètre d'une méthode extérieure :
    public void foo() { X.bar(data.copy()); }
    // - de tout objet passé en argument pour être stocké dans un attribut
    //   - que ce soit dans les méthodes
    public void setData(Data data) { this.data = data.copy(); }
    // - ou dans les constructeurs
    public A(Data data) { this.data = data.copy(); }
}
```

Résumé : ni divulguer ses références, ni conserver une référence qui nous a été donnée.

Immuabilité...

Copie défensive

Qu'est-ce que c'est et comment la réalise-t-on? (exemple)

Introduction	Généralités	Style	Objets et classes	Objets et classes	Objets et classes	Membres et constantes	Types imbriqués	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Détails

- ... ah et comment savoir si un type est immuable? Nous y reviendrons.
- Sont notamment immuables :
- la classe `String`;
 - toutes les primitive wrapper classes : `Boolean`, `Char`, `Byte`, `Short`, `Integer`, `Long`, `Float` et `Double`;
 - d'autres sous-classes de `Number` : `BigInteger` et `BigDecimal`;
 - plus généralement, toute classe ¹¹⁰ dont la documentation dit qu'elle l'est.
- En pratique, les 8 types primitifs (`boolean`, `byte`, `short`, `int`, `long`, `float`) se comportent aussi ¹¹¹ comme des types immuables.
- ¹¹⁰. Les types définis par les interfaces ne peuvent pas être garantis immuables.
- ¹¹¹. Fonctionnellement. Pour d'autres aspects, comme la performance, le comportement est différent.
- ¹¹². Mais cette distinction n'a pas de sens pour des valeurs directes.

Encapsulation

Aliasing : remarque sans rapport avec l'encapsulation

Introduction	Généralités	Style	Objets et classes	Objets et classes	Objets et classes	Membres et constantes	Types imbriqués	Types et polymorphisme	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Supplément

En cas d'`alias` extérieur d'un attribut `a` de type mutable dans une classe `C` :

- on ne peut pas prouver d'invariant de `C` faisant intervenir `a`, notamment, la classe `C` n'est pas immuable (certaines instances pourraient être modifiées par un tiers);
- on ne peut empêcher les modifications concurrentes ¹¹³ de l'objet `aliasé`, dont le résultat est notoirement imprévisible. ¹¹⁴

Il reste possible néanmoins de prouver des invariants de `C` ne faisant pas intervenir `a`; cela peut être suffisant dans bien des cas (y compris dans un contexte concurrent).

L'impossibilité d'`alias` extérieur au frame ¹¹⁵ d'une méthode est aussi intéressante, car elle autorise la JVM à optimiser en allouant l'objet directement en pile plutôt que dans le tas.

En effet : comme l'objet n'est pas référencé en dehors de l'appel courant, il peut être détruit sans risque au retour de la méthode.

La recherche de la possibilité qu'une exécution créée des `alias` externes (à une classe ou une méthode) s'appelle l'`escape analysis` ¹¹⁶.

- ¹¹³. Faisant intervenir un autre `thread`, cf. chapitre sur la programmation concurrente.
- ¹¹⁴. Plus généralement, ce problème se pose dès qu'un objet peut être partagé par des méthodes de classes différentes.
- Si la référence vers cet objet ne sort pas de la classe, il est possible de synchroniser les accès à cet objet.
- ¹¹⁵. `frame` = zone de mémoire dans la pile, dédiée au stockage des informations locales pour un appel de méthode donné.
- ¹¹⁶. Traduction proposée : `analyse d'échappement` ?

Encapsulation

Aliasing, est-ce toujours « mal » ?

Pour conclure sur l'*aliasing*.

Il n'y a pas que des inconvénients à partager des références :

- ➊ *Aliaser* permet d'éviter le surcoût (en mémoire, en temps) d'une copie défensive.
- ➋ *Optimisation* à considérer si les performances sont critiques.
- ➌ *Aliaser* permet de simplifier la maintenance d'un état cohérent dans le programme (vu qu'il n'y a plus de copies à synchroniser).

Mais dans tous les cas il faut être conscient des risques :

- dans 1., mauvaise idée si plusieurs des contextes partageant la référence pensent être les seuls à pouvoir modifier l'objet référencé ;
- dans 2., risque de modifications concurrentes dans un programme *multi-thread* → précautions à prendre.

Discussion

Compléments
en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

objets et classes

Membres et constantes

Encapsulation

Types imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

<p>Types imbriqués</p> <p>Pour quoi faire ?</p> <ul style="list-style-type: none"> • définitions du contexte englobant incluses dans contexte imbriqué (sans chemin). • Type englobant et types membres peuvent accéder aux membres private des uns des autres → utile pour <u>partage de définitions privées entre classes "amies"</u>. 	<p>L'exemple ci-dessous compile :</p> <pre> class TE { static class TIA { private static void fIA() { fE(); } // pas besoin de donner le chemin de fE } static class TIB { private static void fIB() { } } private static void fE() { TIB.fIB(); } // TIB.fIB visible malgré private } friend class en C++ (quoique de façon plus grossière...).</pre> <p>119. La notion de classe imbriquée peut effectivement, en outre, satisfaire le même besoin que la notion de</p>
<p>compléments en POO Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Membres et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions</p>	<p>compléments en POO Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Membres et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions</p> <p>On peut créer une instance de MonIterateur depuis n'importe quel contexte (même statique) dans MaListe avec juste "new MonIterateur< -> (this);"</p>

Plusieurs sortes de types imbriqués

Classification des types imbriqués/nested types¹²⁰

- **types membres statiques/static member classes**¹²¹ : types définis directement dans la classe englobante, définition précédée de **static**
- **classes internes/inner classes** : les autres types imbriqués (toujours des classes)
 - **classes membres non statiques/non-static member classes**¹²² : définies directement dans le type englobant
 - **classes locales/local classes** : définies dans une méthode avec la syntaxe habituelle
(class NomClasseSeLocale { /*contenu */ })
 - **classes anonymes/anonymous classes** : définies "à la volée" à l'intérieur d'une expression, afin d'instancier un objet unique de cette classe :
new NomSuperTypeDirect() { /*contenu */ }.

120. J'essaye de suivre la terminologie de la JLS... traduite, puis complétée par la logique et le bon sens.

121. La JLS les appelle *static nested classes*... oubliant que les interfaces membres existent aussi!

122. parfois appelées juste *inner classes* ; pas de nom particulier donné dans la JLS.

Accès à l'instance imbriquée et à l'instance englobante : <code>this</code> et <code>Outer.this</code>	<p>Soit <code>TI</code> un type imbriqué dans <code>TE</code>, type englobant. Alors, dans <code>TI</code> :</p> <ul style="list-style-type: none"> • <code>this</code> désigne toujours (quand elle existe) l'instance courante de <code>TI</code> ; • <code>TE.this</code> désigne toujours (quand elle existe) l'instance englobante, c.-à-d. l'instance courante de <code>TE</code>, c.-à-d. : <ul style="list-style-type: none"> • si <code>TI</code> classe membre non statique, la valeur de <code>this</code> dans le contexte où l'instance courante de <code>TI</code> a été créée. Exemple : <pre data-bbox="393 1295 570 2007"> class CE { int x = 1; class CI { int y = 2; void f() { System.out.println(CE.this.x + " " + this.y); } } } // alors new CE().new CI().f(); affichera "1 2" </pre> • si <code>TI</code> classe locale, la valeur de <code>this</code> dans le bloc dans lequel <code>TI</code> a été déclaré. <p>La référence <code>TE.this</code> est en fait stockée dans toute instance de <code>TI</code> (attribut caché).</p>
Compléments en POO	Introduction Généralités Style Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Aldrich Degorre	Aldrich Degorre
	Exemple

Types imbriqués	La définition de classe se place comme une instruction dans un bloc (gén. une méthode) :	Exemple de classe locale
	<pre>class MaListe<T> implements List<T> { ... public Iterator<T> iterator() { class MonIterator<T> implements Iterator<T> { public boolean hasNext() {...} public T next() {...} public void remove() {...} } return new MonIterator<T>() } }</pre>	

Classes anonymes

Classe anonyme =

- cas particulier de classe locale avec yntaxe allégée
→ comme classes locales, accès aux déclarations du bloc¹²⁵,
- déclaration "en ligne" : c'est syntaxiquement une expression, qui s'évalue comme une instance de la classe déclarée;
- déclaration de classe sans donner de nom ⇒ instanciable une seule fois
 → c'est une classe singleton;
- autre restriction : un seul supertype direct¹²⁶ (dans l'exemple : `Iterator`).

Question : comment exécuter des instructions à l'initialisation d'une classe anonyme alors qu'il n'y a pas de constructeur?
Réponse : utiliser un "bloc d'initialisation" (Au besoin, cherchez ce que c'est!)

Syntaxe encore plus concise : lambda-expressions (cf. chapitre dédié), par ex.

`x -> System.out.println(x).`

Types imbriqués Exemple de classe anonyme	<p><u>La définition de classe est une expression dont la valeur est une instance¹²⁴ de la classe.</u></p> <pre style="background-color: #FFFFCC; border: 1px solid black; padding: 10px;"> class Mailiste<T> implements List<T> { ... public Iterator<T> iterator() { return /* de là */ new Iterator<T>() { public boolean hasNext() {...} public T next() {...} public void remove() {...} } /* à là */ } }</pre>	<u>La seule instance.</u>
Compléments en POO Aldric Degorre	Introduction Généralités Style Objets et classes Membres et constructeurs Encapsulation Types indépendants Types et polymorphisme Héritage Généricité Concurrente Interfaces graphiques Gestion des erreurs et exceptions	

Classes anonymes

Astuce avec Java ≥ 10

Types imbriqués

Accessibilité du contexte englobant

Introduction	Compléments en POO Aldric Degorre
Généralités	Introduction Généralités Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Style	Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Objets et classes	Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Objets et classes	Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions

Le mot-clé var¹²⁷ permet de faire des choses sympas avec les classes anonymes :

```
// Créeation d'objet singleton utilisable sans déclarer de classe nommée ou
d'interface :
var plop = new Object() { int x = 23; };
System.out.println(plop.x);
```

Sans **var** il aurait fallu écrire le type de **plop**. En l'occurrence le plus petit type dénotable connu ici est **Object**.

Or la classe **Object** n'a pas de champ **x**, donc **plop.x** ne compilerait pas.

127. Remplaçant un type dans une déclaration, pour demander d'inférer le type automatiquement.

Introduction	Compléments en POO Aldric Degorre
Généralités	Introduction Généralités Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Style	Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Objets et classes	Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Objets et classes	Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions

Le mot-clé var¹²⁷ permet de faire des choses sympas avec les classes anonymes :

```
// Créeation d'objet singleton utilisable sans déclarer de classe nommée ou
d'interface :
var plop = new Object() { int x = 23; };
System.out.println(plop.x);
```

Sans **var** il aurait fallu écrire le type de **plop**. En l'occurrence le plus petit type dénotable connu ici est **Object**.

Or la classe **Object** n'a pas de champ **x**, donc **plop.x** ne compilerait pas.

127. Remplaçant un type dans une déclaration, pour demander d'inférer le type automatiquement.

Introduction	Compléments en POO Aldric Degorre
Généralités	Introduction Généralités Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Style	Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Objets et classes	Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
Objets et classes	Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions

Types imbriqués

Remarques et limitations diverses

public et **static**.

Dans une interface englobante, les types membres sont **public** et **static**.
Dans les classes locales (et anonymes), on peut utiliser les variables locales du bloc seulement si elles sont **effectivement finales** (c.-à-d. déclarées **final**, ou bien jamais modifiées).

Explication : l'instance de la classe locale peut "survivre" à l'exécution du bloc. Donc elle doit contenir une copie des variables locales utilisées. Or les 2 copies doivent rester cohérentes → modifications interdites.

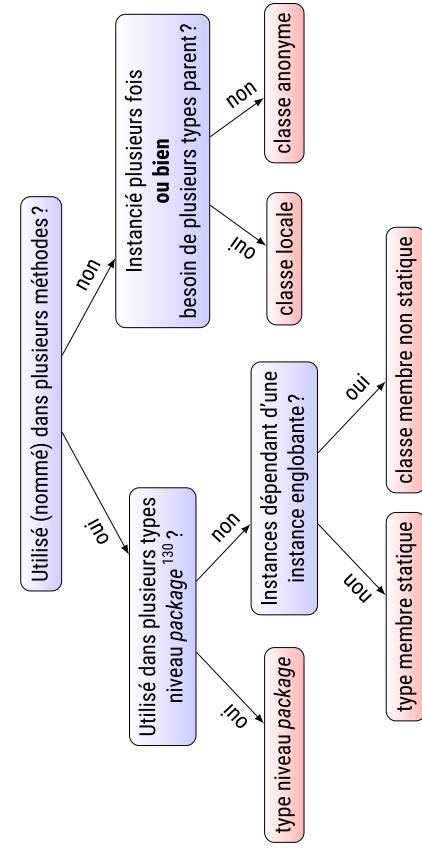
Une alternative non retenue : stocker les variables partagées dans des objets dans le tas, dont les références seraient dans la pile. On pourrait aisément programmer ce genre de comportement au besoin.

Les classes internes¹³² ne peuvent pas contenir de membres statiques (à part attributs **final**). La raison est le décalage entre ce qu'est censé être une classe interne (prétendue dépendance à un certain contexte dynamique) et son implémentation (classe statique toute bête ; ce sont en réalité les instances de la classe interne qui contiennent une référence vers, par exemple, l'instance englobante).

Une méthode statique ne pourrait donc pas accéder à ce contexte dynamique, rompant l'illusion recherchée.

- 132. Nécessairement et implicitement.
- 133. Tous les types imbriqués sauf les classes membres statiques

Plusieurs sortes de types imbriqués

D'accord, mais lequel choisir ?¹³¹

- 130. C'est à dire non imbriqués, définis directement dans les fichiers .java.
- 131. Cf. Effective Java 3rd edition, Item 24 : Favor static member classes over nonstatic.

Types de données

Système de types de Java

Caractéristiques principales

Compléments en POO

Aldric Degorre

Introduction	Généralités	Introduction	Généralités
Style	Style	Style	Style
Objets et classes	Objets et classes	Objets et classes	Objets et classes
Types et polymorphisme	Types et polymorphisme	Types et polymorphisme	Types et polymorphisme
Le système de types	Sous-typeage	Le système de types	Sous-typeage
	Polytypisme(s)		Polytypisme(s)
	Interfases		Interfases
Héritage	Héritage	Héritage	Héritage
Généricité	Généricité	Généricité	Généricité
Concurrency	Concurrency	Concurrency	Concurrency
Interfaces graphiques	Interfaces graphiques	Interfaces graphiques	Interfaces graphiques
Gestion des erreurs et exceptions			
Résumé	Résumé	Résumé	Résumé

● **type de données** = ensemble d'éléments représentant des données de forme similaire, traitables de la même façon par un même programme.

- Chaque langage de programmation a sa propre idée de ce à quoi il faut donner un type, de quand il faut le faire, de quels types il faut distinguer et comment, etc.

On parle de différents **systèmes de types**.

• typage qualifié de "fert" (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 ¹³⁴ 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 ¹³⁵ : 2 types sont égaux si 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.
			134. Via la liaison tardive/dynamique et via mécanismes explicites : instanceof et réflexion.
			135. Contraire : 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 primatifs

Compléments en POO

Aldric Degorre

Introduction	Généralités	Introduction	Généralités
Style	Style	Style	Style
Objets et classes	Objets et classes	Objets et classes	Objets et classes
Types et polymorphisme	Types et polymorphisme	Types et polymorphisme	Types et polymorphisme
Le système de types	Sous-typeage	Le système de types	Sous-typeage
	Polytypisme(s)		Polytypisme(s)
	Interfases		Interfases
Héritage	Héritage	Héritage	Héritage
Généricité	Généricité	Généricité	Généricité
Concurrency	Concurrency	Concurrency	Concurrency
Interfaces graphiques	Interfaces graphiques	Interfaces graphiques	Interfaces graphiques
Gestion des erreurs et exceptions			
Résumé	Résumé	Résumé	Résumé

● Pour des raisons liées à la mémoire et au polymorphisme, 2 catégories¹³⁶ de types :

	types primatifs	types référence
données représentées	données simples	données complexes (objets)
valeur ¹³⁷ (« directe »)	donnée directement	adresse d'un objet ou null
espace occupé	32 ou 64 bits	32 bits (adresse)
nombre de types	8 (fixé, cf. page suivante)	nombre fournis dans le JDK et on peut en programmer
casse du nom	minuscules	majuscule initiale (par convention)

Il existe quelques autres différences, abordées dans ce cours.

Nom	description	t. contenu	t. utilisée	exemples
byte	entier très court	8 bits	1 mot ¹³⁸	127,-19
short	entier court	16 bits	1 mot	-32_768 , 15_903
int	entier normal	32 bits	1 mot	23_411_431
long	entier long	64 bits	2 mots	3_411_431_434L
float	réel à virgule flottante	32 bits	1 mot	3_214.991 f
double	idem, double précision	64 bits	2 mots	-223.12 , 4.324E12
char	caractère unicode	16 bits	1 mot	'a' 'Q' '\0'
boolean	valeur de vérité	1 bit	1 mot	true, false

Cette liste est exhaustive : le programmeur ne peut pas définir de types primatifs.

Chaque type primitif a un nom **commençant par une minuscule**, qui est un mot-clé réservé de Java (**String, int = "true"** ne compile pas).

138. 1 **mot** = 32 bits

136. Les distinctions primitif/objet et valeur directe/référence coïncident en Java, mais c'est juste en Java. Ex : C++ possède à la fois des objets valeur directe et des objets accessibles par pointeur ! En Java (< 14), on peut donc remplacer "type référence" par "type objet" et "type primitif" par "type à valeur directe" sans changer le sens d'une phrase... mais il est question que ça change (projet Valhalla) !

137. Les 32 bits stockés dans un champs d'objet ou empilés comme résultat d'un calcul.

Pile :

- les opérations courantes prennent/retirent leurs opérandes du sommet de la **pile** et écrivent leurs résultats au sommet de cette même pile (ordre LIFO) ;
- lors de l'appel d'une méthode, ses paramètres effectifs sont empilés; lors de son retour, toute la mémoire qui a servi à son exécution (paramètres, variables locales, résultats intermédiaires) est libérée/dépilée, et le résultat empilé ;
- dans la pile, on ne stocke que des blocs d'1 ou 2 mots → on ne peut stocker que des valeurs primitives et des adresses¹³⁹ d'objets. ¹⁴⁰

¹³⁹ Des pseudo-adresses, pour être exact.
¹⁴⁰ En réalité, la JVM peut optimiser en mettant les objets locaux en pile. Mais ceci est invisible.

Tas :

- Objets (tailles diverses) stockés dans le **tas**.
- Tas géré de façon automatique : quand on crée un objet, l'espace est réservé automatiquement et quand on ne l'utilise plus, la JVM le détecte et libère l'espace (**ramasse-miettes/garbage-collector**).
- L'intérieur de la zone réservée à un objet est constitué de champs, contenant chacun une valeur primitive ou bien une adresse d'objet.

¹⁴¹ Des registres¹⁴², contient notamment les registres suivants :

141. Chaque fil d'exécution parallèle. Cf. chapitre sur la programmation concurrente.

Autres zones de mémoire dans la JVM :

- zone des méthodes : classes, dont méthodes (code octet) et attributs statiques
- zone des registres¹⁴², contient notamment les registres suivants :
 - l'adresse de la prochaine instruction à exécuter
 - l'adresse du sommet de la pile

¹⁴² Comme la pile, les registres sont dupliqués pour chaque *thread*.

Références et valeurs directes

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

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 faite 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

ConSEQUENCE sur l'affectation et l'égalité

Références et valeurs directes

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

Compléments en POO
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polytypisme(s)

Surchage

Interfaces

héritage

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Révision

La distinction référence/valeur directe a plusieurs conséquences à l'exécution.

Pour `x` et `y` variables de types références :

- Après l'affectation `x = y`, les deux variables désignent le même emplacement mémoire (**aliasing**).

Si ensuite on exécute l'affectation `x.a = 12`, alors après `y.a` vaudra aussi 12.

- Si les variables `x` et `z` désignent des emplacements différents, le test d'identité `x == z` vaut **false**, même si le contenu des deux objets référencés est identique.

143. Pour les primitives, identité et égalité sont la même chose. Pour les objets, le test d'égalité est la méthode `public boolean equals(Object other)`.

Références et valeurs directes

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

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 144 → la méthode travaille sur une copie des données réelles
- pour les types références → 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 méthode.

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

Compléments en POO
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polytypisme(s)

Surchage

Interfaces

héritage

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Révision

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. 147

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

145. Les types primaires et les types immuables se comportent de la même façon pour de nombreux critères.

146. Passage par référence : quand on passe une variable `v` (plus généralement, une `value`) en paramètre, le paramètre formel (à l'intérieur de la méthode) est un alias de `v` (un pointeur "déguisé" vers l'adresse de `v`, mais utilisable comme si c'était `v`).

Toute modification de l'alias modifie la valeur de `v`. En outre, le pointeur sous-jacent peut pointer vers la pile (si `v` variable locale), ce qui n'est jamais le cas des "réferences" de Java.

Compléments en POO
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polytypisme(s)

Surchage

Interfaces

héritage

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Discussion

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. 147

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

145. Les types primaires et les types immuables se comportent de la même façon pour de nombreux critères.

146. Passage par référence : quand on passe une variable `v` (plus généralement, une `value`) en paramètre, le paramètre formel (à l'intérieur de la méthode) est un alias de `v` (un pointeur "déguisé" vers l'adresse de `v`, mais utilisable comme si c'était `v`).

Toute modification de l'alias modifie la valeur de `v`. En outre, le pointeur sous-jacent peut pointer vers la pile (si `v` variable locale), ce qui n'est jamais le cas des "réferences" de Java.

Compléments en POO
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polytypisme(s)

Surchage

Interfaces

héritage

Héritage

Généricité

Concurrence

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Discussion

Stades de vérification et entités typables en Java

Stades de vérification et entités typables en Java

Compléments en P00

Aldric Degoire

Java → langage à typage statique, mais avec certaines vérifications à l'exécution 148 :	
• À la compilation on vérifie le type des expressions ¹⁴⁹ (analyse statique).	
Toutes les expressions sont vérifiées.	
• À l'exécution, la JVM peut vérifier le type des objets ¹⁵⁰ .	
Ce type de vérification n'est pas systématique. Il a lieu seulement lors d'événements bien précis : p. ex. : lors d'un <i>downcasting</i> ou d'un test <code>instanceof</code> .	

148. C'est en fait une caractéristique habituelle des langages à typage essentiellement statique mais autorisant le polymorphisme par sous-type.
149. morceaux de texte évaluables du programme
150. Ces entités n'existent pas avant l'exécution, de toute façon!

À la compilation : les expressions

Introduction	Compléments en P00
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Le système de types	
Sous-typage	
Transfertage	
Polymporphisme(s)	
Surchage	
Inferencage	
Héritage	
Généricité	
Concurrente	
Interfaces graphiques	
Gestion des erreurs et exceptions	

Introduction	Compléments en P00
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Le système de types	
Sous-typage	
Transfertage	
Polymporphisme(s)	
Surchage	
Inferencage	
Héritage	
Généricité	
Concurrente	
Interfaces graphiques	
Gestion des erreurs et exceptions	

Stades de vérification et entités typables en Java

À l'exécution : les objets

Introduction	Compléments en P00
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Le système de types	
Sous-typage	
Transfertage	
Polymporphisme(s)	
Surchage	
Inferencage	
Héritage	
Généricité	
Concurrente	
Interfaces graphiques	
Gestion des erreurs et exceptions	

Lors de l'instanciation d'un objet, le nom de sa classe (= son type dynamique exact), y est inscrit (**définitivement**). Ceci permet :

- d'exécuter des tests demandés par le programmeur (comme `instanceof`);
- à la méthode `getClass()` de retourner un résultat;
- de faire fonctionner la liaison dynamique (dans `x.f()`), la JVM regarde le type de l'objet référencé par `x` avant de savoir quel `f()` exécuter);
- de vérifier la cohérence de certaines conversions de type :
`Object o; ... ; String s = (String)o;`

Ceci ne concerne pas les valeurs primitives/directes : pas de place pour coder le type dans les 32 bits de la valeur directe! (et c'est en fait inutile)

La différence entre objets et valeurs directes provient du traitement différent du polymorphisme et des conversions de type (casts, voir plus loin).

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 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-type¹⁵⁴ de son type statique.

- 152. 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.
- 153. Remarque : le type (la classe) d'un objet donné est, en revanche, fixé(e) dès son instantiation.
- 154. Nous expliquons le sous-typage juste après.

Notion de sous-typage

Notion de sous-typage

Interprétations

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types	Sous-typage	Transfertage	Polytypisme(s)	Surcharge	Interfaces	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Analyse

- **Définition :** le type *A* est **sous-type** de *B* ($A <: B$) (ou bien *B supertype* de *A* ($B :> A$)) si toute entité¹⁵⁵ de type *A*

- est de type *B*
- (alt.) « peut remplacer » une entité de type *B*.

- plusieurs interprétations possibles (mais contraintes par notre définition de « type »).

155. Pour Java, entité = soit expression, soit objet.

Notion de sous-typage

Interprétations

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types	Sous-typage	Transfertage	Polytypisme(s)	Surcharge	Interfaces	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Analyse

- **Interprétation faible : ensembliste.** Tout sous-ensemble d'un type donné forme un sous-type de celui-ci.
Exemple : tout carré est un rectangle, donc le type carré est sous-type de rectangle.
→ insuffisant car un type n'est pas un simple ensemble¹⁵⁶ : il est aussi muni d'opérations, d'une structure, de contrats¹⁵⁷, ...
- **Contrat :** propriété que les implémentations d'un type s'engagent à respecter. Un type honore un tel contrat si et seulement si **toutes ses instances** ont cette propriété.

156. Pour les algébristes, on peut faire l'analogie avec les groupes, par exemple : un sous-ensemble d'un groupe n'est pas forcément un groupe (il faut aussi qu'il soit stable par les opérations de groupe, afin que la structure soit préservée).
157. Formels ou informels (javadoc).

Notion de sous-typage

Interprétations

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types	Sous-typage	Transfertage	Polytypisme(s)	Surcharge	Interfaces	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Analyse

- Pourquoi le sous-typage structurel est insuffisant ?

Exemple :

- **Dans le cours précédent, les instances de la classe FiboGen génèrent la suite de Fibonacci.**
- **Contrat possible**¹⁵⁹ : « le rapport de 2 valeurs successives tend vers $\varphi = \frac{1+\sqrt{5}}{2}$ (nombre d'or) ».
(On sait prouver ce contrat pour la méthode next des instances directes de FiboGen.)
- Or rien empêche de créer un sous-type BadFib (sous-classe¹⁶⁰) de Fibogen dont la méthode next retournerait toujours 0.
→ Les instances de BadFib seraient alors des instances de FiboGen violant le contrat.

159. Raisonnable, dans le sens où c'est une propriété mathématique démontrée pour la suite de Fibonacci, qui donc doit être vraie dans toute implémentation correcte.
160. Une sous-classe est bien un sous-type au sens structurel : les méthodes sont héritées.

158. Contravariance des paramètres et covariance du type de retour.

Notion de sous-typage

Notion de sous-typage

Interprétations

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types	Sous-typage	Transfertage	Polytypisme(s)	Surchage	Interfaces	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Analyse

- **Interprétation idéale : Principe de Substitution de Liskov**¹⁶¹ (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.

161. C'est le « L » de la méthodologie SOLID (Design Principles and Design Patterns. Robert C. Martin.).

Notion de sous-typage

Notion de sous-typage pour Java

Interprétations de sous-typage en Java (1)

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types	Sous-typage	Transfertage	Polytypisme(s)	Surchage	Interfaces	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Résumé

Les grandes lignes du sous-typage selon Java : (détails dans JLS 4.10 et ci-après)

- Pour les 8 types primaires, 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 structurelles¹⁶² sont vérifiées, concernant les redéfinitions de méthodes.
- Types primaires et types référence forment deux systèmes déconnectés. Aucun type référence n'est sous-type ou supertype d'un type primaire.

163. **float** (1 mot) n'est pas plus précis ou plus « large » que **long** (2 mots), mais il existe néanmoins une conversion automatique du second vers le premier.

164. Via la valeur unique du caractère.

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

Notion de sous-typage pour Java

Relation de sous-typage en Java¹⁶⁸ (2)

Types référence :

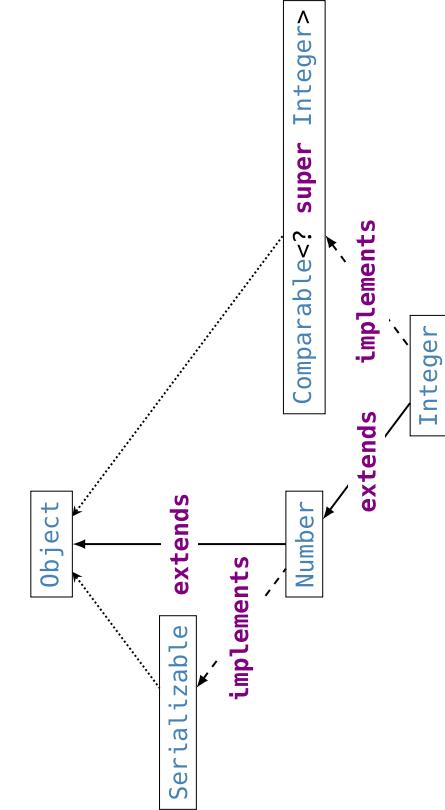
$A <: B$ si **B** est **Object**¹⁶⁵ ou s'il existe des types $A_0 (= A), A_1, \dots, A_n (= B)$ tels que pour tout $i \in 1..n$, une des règles suivantes s'applique :¹⁶⁶

- (**implémentation d'interface**) A_{i-1} est une classe, A_i est une interface et A_{i-1} implémente A_i ;
- (**héritage de classe**) A_{i-1} et A_i sont des classes et A_{i-1} étend A_i ;
- (**héritage d'interface**) A_{i-1} et A_i sont des interfaces et A_{i-1} étend A_i ;
- (**covariance des types tableau**)¹⁶⁷ A_{i-1} et A_i resp. de forme $a[]$ et $b[]$, avec $a <: b$;¹⁶⁸
- 165. C'est vrai même si A est une interface n'hérite de **Object**.
- 166. Pour être exhaustif, il manque les règles de sous-typage pour les types génériques.
- 167. Les types tableau sont des classes (très) particulières, implémentant les interfaces **Cloneable** et **Serializable**. Donc tout type tableau est aussi sous-type de **Object**, **Cloneable** et **Serializable**.
- 168. Cf. JLS 4.10.

Notion de sous-typage pour Java

Relation pour les types références, diagramme

Notion de sous-typage : le type **Integer** et ses supertypes.



Notion de sous-typage : le type **Integer** et ses supertypes.

Notion de sous-typage pour Java

Principe fondamental

Notion de sous-typage pour Java

Principe fondamental – explications

Pourquoi ce remplacement ne gène pas l'exécution :

- les objets sont utilisables sans modification comme instances de tous leurs supertypes¹⁷¹ (**sous-typage inclusif**). P. ex. : **Object o = "toto"** fonctionne.
- Java¹⁷² s'autorise, si nécessaire, à remplacer une valeur primitive par la valeur plus proche dans le type cible (**sous-typage coercitif**). P. ex. : après l'affectation **float f = 1_000_000_000_123L**; la variable **f** vaut **1.0E12** (on a perdu les derniers chiffres).

171. Les contraintes d'implémentation d'interface et d'héritage garantissent que les méthodes des supertypes peuvent être appelées.
 172. Si nécessaire, javac convertit les constantes et insère des instructions dans le code-octet pour convertir les valeurs variables à l'exécution.

Notion de sous-typage pour Java

Principe fondamental

Notion de sous-typage pour Java

Principe fondamental – explications

Principe fondamental

Dans un programme qui compile, remplacer une expression de type **A** par une de type **B**¹⁶⁹, avec **B <: A**, donne un programme qui compile encore (à moins que sa compilation échoue pour cause de surcharge ambiguë¹⁷⁰).

Remarque : seule la compilation est garantie, ainsi que le fait que le résultat de la compilation est exécutable.

La correction du programme résultant n'est pas garantie !

(pour cela, il faudrait au moins que java impose le respect du LSP, ce qui est impossible)

169. Syntaxiquement correcte ; avec identifiants tous définis dans leur contexte ; et bien typée.
 170. En effet, le type statique des arguments d'une méthode surchargée influe sur la résolution de la surcharge et peut créer des ambiguïtés. Cf. chapitre sur le sujet.

Notion de sous-typage pour Java

Transtypage (type casting)

Compléments en POO

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typage

Transtypage

Polytypisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Résumé

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typage

Transtypage

Polytypisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Résumé

Compléments en POO

Aldric Degorre

Corollaires :

- on peut affecter à toute variable une expression de son sous-type (ex : **double z = 12;**);
- on peut appeler toute méthode avec des arguments d'un sous-type des types déclarés dans sa signature (ex : **Math.pow(3, 'z');**);
- on peut appeler toute méthode d'une classe T donné sur un récepteur instance d'une sous-classe de T (ex : **"toto".hashCode();**).

Ainsi, le sous-typage est la base du système de polymorphisme de Java.

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typage

Transtypage

Polytypisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Transtypage = type casting = conversion de type d'une expression.

Plusieurs mécanismes 173 174 :

- **upcasting** : d'un type vers un supertype (ex : **Double vers Number**)
- **downcasting** : d'un type vers un sous-type (ex : **Object vers String**)
- **boxing** : d'un type primitif vers sa version "emballée" (ex : **int vers Integer**)
- **unboxing** : d'un type emballé vers le type primitif correspondant (ex : **Boolean vers boolean**)
- conversion en **String** : de tout type vers **String** (implicite pour concaténation)
- parfois combinaison implicite de plusieurs mécanismes.

173. Détailles dans la JLS, chapitre 5.
174. 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.

Transtypage (type casting)

Autres termes employés (notamment dans la JLS)

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typage

Transtypage

Polytypisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Transtypage (type casting)

Autres termes employés (notamment dans la JLS)

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typage

Transtypage

Polytypisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Tous ces mécanismes sont des règles permettant vérifier, à la compilation, si une expression peut être placée là où elle l'est.

Parfois, conséquences à l'exécution :

- vraie modification des données (types primitifs),
 - ou juste vérification de la classe d'un objet (downcasting de référence).
- ouverte à la vérification de la classe d'un objet (downcasting de référence).

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

- 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.

→ **On ne prononcera plus étargissement, retrécissement, promotion ou coercition !**

176. Au sens mathématique du terme. Pas forcément une méthode.

Le code ci-dessous est probablement symptomé d'une conception non orientée objet :

```
// Anti-patron :
void g(Object x) { // Object ou bien autre supertype commun à C1 et C2
    if (x instanceof C1) { C1 y = (C1) x; f1(y); }
    else if (x instanceof C2) { C2 y = (C2) x; f2(y); }
    else { /* quoi en fait ? on génère une erreur ? */ }
}
```

Quand c'est possible, on préfère utiliser la liaison dynamique :

```
public interface I { void f(); }
void g(I x) { x.f(); } // déjà , programmons à l'interface
// puis dans d'autres fichiers (voire autres packages)
public class C1 implements I { public void f() { f1(this); } }
public class C2 implements I { public void f() { f2(this); } }
```

Avantage : les types concrets manipulés ne sont jamais nommés dans **g**, qui pourra donc fonctionner avec de nouvelles implémentations de **I** sans modification.

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 :

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

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

Transtypage, cas 3 : auto-boxing/unboxing

Types "emballés"

Transtypage, cas 3 : auto-boxing/unboxing

Conversions automatiques, depuis Java 5

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types Sous-typage Transtypage Polymorphisme(s) Surcharge Interfaces Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Révision
<ul style="list-style-type: none"> Pour tout type primitif, il existe un type référence "emballé" ou "mis en boîte" (<i>wrapper type</i> ou <i>boxed type</i>) équivalent : int↔Integer, double↔Double, ... Attention, contrairement à leurs équivalents primitifs, les différents types emballés ne sont pas sous-types les uns des autres ! Ils ne peuvent donc pas être transtypés de l'un dans l'autre. 				<pre>Double d = new Integer(5); → Double d = new Integer(5).doubleValue(); ou encore Double d = 0. + (new Integer(5));</pre>						
<p>177. Spoiler : cet exemple utilise des conversions automatiques. Voyez-vous lesquelles ?</p>										

- Cette conversion automatique permet, dans de nombreux usages, de confondre un type primitif et le type emballé correspondant.

178. La différence entre **Integer.valueOf(5)** et **new Integer(5)** c'est que la fabrique statique **valueOf()** réutilise les instances déjà créées, pour les petites valeurs (mise en cache).

Subtilités du transtypage

Cas des types primitifs : possible perte d'information (2)

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types Sous-typage Transtypage Polymorphisme(s) Surcharge Interfaces Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Détails

Transtypage : à propos du "cast"

Compléments en POO
Aldric Degorre

i.e., la notation (Type)expr

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types Sous-typage Transtypage Polymorphisme(s) Surcharge Interfaces Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions	Détails

- Particulièrement utile pour le downcasting, mais sert à toute conversion.
- Soient **A** et **B** des types et **e** une expression de type **A**, alors l'expression "(B)e"
- est de type statique **B**
- et a pour valeur (à l'exécution), autant que possible, la "même" que **e**.
- L'expression "(B)e" passe la compilation à condition que (au choix) :
 - A** et **B** soient des types référence avec **A <: B** ou **B <: A**;
 - que **A** et **B** soient des types primitifs tous deux différents de **boolean**¹⁷⁹ ;
 - que **A** soit un type primitif et **B** la version emballée de **A** ;
 - ou que **B** soit un type primitif et **A** la version emballée d'un sous-type de **B** (comparaison implicite d'*unboxing* et *upcasting*).
- Même quand le programme compile, effets indésirables possibles à l'exécution :
 - perte d'information quand on convertit une valeur primitive ;
 - ClassCastException** quand on tente d'utiliser un objet en tant qu'objet d'un type qu'il n'a pas.

^{179.} NB : (**char**) (**byte**)⁰) est égal, alors qu'il n'y a pas de sous-typage dans un sens ou l'autre.

^{180.} 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) → certains **int** ne sont pas représentables en **float**.

^{181.} Selon implémentation, mais pas de garantie. Cherchez à quoi sert ce mot-clé!

Subtilités du transtypage

Cas des types primitifs : points d'implémentation

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Transtypage

Polymorphisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrency

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

- Pour **upcasting** d'entier de ≤ 32 bits vers ≤ 32 bits : dans code-octet et JVM, granularité de 32 bits \rightarrow tous les "petits" entiers codés de la même façon \rightarrow aucune modification nécessaire.¹⁸²
- Pour une conversion de littéral¹⁸³, le compilateur fait lui-même la conversion et remplace la valeur originale par le résultat dans le code-octet.
- Dans les autres cas, conversions à l'exécution dénotées, dans le code-octet, par des instructions dédiées :
 - downcasting** : **d2i**, **d2l**, **d2f**, **f2i**, **f2l**, **i2b**, **i2c**, **i2s**, **l2i**
 - upcasting** avec perte : **i2d** (sans **strictfp**), **i2f**, **l2d**, **l2f**
 - upcasting** sans perte : **i2d** (avec **strictfp**), **i2l**, **i2d**

182. C'est du sous-typage inclusif, comme pour les types référence!
 183. Littéral numérique = nombre écrit en chiffres dans le code source.

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Transtypage

Polymorphisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrency

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

- Et concrètement, que font les conversions ?
- Upcasting d'entier ≤ 32 bits vers **Long** (**i2l**)** : on complète la valeur en recopiant le bit de gauche 32 fois.¹⁸⁴
- Downcasting d'entier vers entier n bits (**i2b**, **i2c**, **i2s**, **l2i**)** : on garde les n bits de droite et on remplit à gauche en recopiant $32 - n$ fois le bit le plus à gauche restant.¹⁸⁵

184. Pour plus d'explications : chercher "représentation des entiers en complément à 2".
 185. Ainsi, la valeur d'origine est interprétée modulo 2^n sur un intervalle centré en 0.

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Transtypage

Polymorphisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrency

Interfaces graphiques

Gestion des erreurs et exceptions

Exemple

- Types références : exécuter un transtypage ne modifie pas l'objet référencé**¹⁸⁶

• **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.

```
// Comestible x = new Fruit(); Fruit y = (Fruit) x;
// Compte mais ClassCastException à l'exécution :
Comestible x = new Viande(); Fruit y = (Fruit) x;
// Ne compile pas !
// Viande 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**).

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

Subtilités du transtypage

Cas des types référence

Polymorphisme

Un principe fondamental de la POO

- Ainsi, après le **cast**, Java sait que l'objet « converti » est une instance du type cible¹⁸⁷.
- Les méthodes exécutées sur un objet donné (avec ou sans cast), sont toujours celles de sa classe, peu importe le type statique de l'expression.¹⁸⁸

Le cast change juste le type statique de l'expression et donc les méthodes qu'on a le droit d'appeler dessus (indépendamment de son type dynamique).

Dans l'exemple ci-dessous, c'est bien la méthode `f()` de la classe B qui est appelée sur la variable `a` de type A :

```
class A { public void f() { System.out.println("A"); } }
class B extends A { @Override public void f() { System.out.println("B"); } }

A a = new B(); // upcasting B -> A
// ici, a: type statique A, type dynamique B
a.f(); // affichera bien "B"
```

187. Sans que l'objet n'ait jamais été modifié par le cast !
 188. Principe de la **liaison dynamique**.

Définition (Polymorphisme)

Une instruction/une méthode/une classe/... est dite **polymorphe** si elle peut travailler sur des données de types concrets différents, qui se comportent de façon similaire.

Le cast change juste le type statique de l'expression et donc les méthodes qu'on a le droit d'appeler dessus (indépendamment de son type dynamique).

Le fait pour un même morceau de programme de pouvoir fonctionner sur des types concrets différents favorise de façon évidente la réutilisation.

Or tout code réutilisé, plutôt que dupliqué quasi à l'identique, n'a besoin d'être corrigé qu'une seule fois par bug détecté.

Donc le polymorphisme aide à « bien programmer », ainsi la POO en a fait un de ses « piliers ».

Il y a en fait plusieurs formes de polymorphisme en Java...

Formes de polymorphisme

Les 3 formes de polymorphisme en Java

Formes de polymorphisme

Les 3 formes de polymorphisme en Java

• **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 POO

• **polymorphisme paramétré** :

Concerne le code utilisant les **type génériques** (ou paramétrés, cf. généricité).
 Le même code peut fonctionner, sans recompilation, quelle que soit la concrétisation des paramètres.
 Ce polymorphisme permet d'exprimer des relations fines entre les types.

Compléments

en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typage

Transfertage

Polymorphisme(s)

surchage

Interfaces

Inéficiences

Héritage

Généricité

Concurrency

Interfaces graphiques

Gestion des erreurs et exceptions

Compléments

en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typage

Transfertage

Polymorphisme(s)

surchage

Interfaces

Généricité

Concurrency

Interfaces graphiques

Gestion des erreurs et exceptions

Résumé

Polymorphisme

Exemples et mécanismes divers

Polymorphisme

Les 3 formes de polymorphisme en Java

• L'opérateur + fonctionne avec différents types de nombres. C'est une forme de polymorphisme résolue à la compilation¹⁸⁹.

```
static void f(Schtroumpfable s, int n) {
    for(int i = 0; i < n; i++) s.schtroumpf();
}
```

`f` est polymorphe : toute instance directe ou indirecte de Schtroumpfable peut être passée à cette méthode, sans recompilation !

L'appel `s.schtroumpf()` dans la classe de `s` (liaison dynamique), ou le sous-typage garantit qu'une telle implémentation existe.

• Et dans l'exemple suivant : `System.out.println(z);` `z` peut être de n'importe quel type. Quel(s) mécanisme(s) intervient(en)-t-il(s) ?

189. En fonction du type des opérande, javac traduit "+" par une instruction parmi `dadd`, `fadd` et `ladd`.

Surcharge

Définition

Compléments
en POO

Aldric Degorre

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. 190 .

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

- Interdiction de définir dans une même classe 191 2 méthodes ayant même nom et même signature (ou 2 constructeurs de même signature).
- \rightarrow 2 entités surchargées ont forcément une signature différente 192 .

190. P.ex. `"/"` est défini pour **int** mais aussi pour **double**

191. 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.

192. Nombre ou type des paramètres différents; le type de retour ne fait pas partie de la signature et n'a rien à voir avec la surcharge !

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Surcharge

Résolution (simplifiée...)

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Surcharge

Exemples

Surcharge

Résolution (simplifiée...)

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Surcharge

Exemples

Surcharge

Résolution : définitions préférables

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surcharge

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

```
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"
        public static void g(0d, 0); // ne compile pas
        g(0d, 0); // affiche "double int"
    }
}
```

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

194. 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.

Surchage

Pourquoi elle ne permet que du polymorphisme "faible"

Compléments en P00

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

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(Object o) { // méthode "réellement" polymorphe
    if (o instanceof String) f((String) o);
    else if (o instanceof Integer) f((Integer) o);
    else { /* gérer l'erreur */ }
}
```

`g()` doit être remplacée en remplaçant les ??? par `String` ou `Integer` pour accepter l'un ou l'autre de ces types (mais pas les 2 dans une même version du programme).

Interfaces

Pour réaliser le polymorphisme via le sous-typeage, de préférence, on définit une **interface**, puis on la fait implémenter par plusieurs classes.

Compléments en P00

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Discussion

Interfaces

Compléments en P00

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Interfaces : syntaxe de la déclaration

```
public interface Comparable { int compareTo(Object other); }
```

Déclaration comme une classe, en remplaçant `class` par `interface`, mais : 196

- constructeurs interdits,
- tous les membres implicitement `public`,
- attributs implicitement `static final` (= constantes),
- types membres nécessairement et implicitement `static`;
- méthodes d'instance implicitement `abstract` (simple déclaration sans corps);
- méthodes d'instance non-abstraites signalées par mot-clé `default`;
- les méthodes `private` sont autorisées (annule `public` et `abstract`), autres membres obligatoirement `public`;
- méthodes `final` interdites.

Rappel : type de données ↔ ce qu'il est possible de faire avec les données de ce type.

En P00 → messages qu'un objet de ce type peut recevoir (méthodes appelables)

Interfaces

Compléments en P00

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transfertage

Polymorphisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Analysé

Interfaces : syntaxe de la déclaration

```
public interface Comparable { int compareTo(Object other); }
```

Déclaration comme une classe, en remplaçant `class` par `interface`, mais : 196

Plusieurs façons de voir la notion d'**interface** :

• supertype de toutes les classes qui l'implémentent :

```
Comparable x = new Fruitt();
```

(parce qu'alors `Fruit` <: `Comparable`)
Si la classe `Fruit` implémente l'interface `Comparable`, alors on a le droit d'écrire :

• contrat qu'une classe qui l'implémente doit respecter

(ce contrat n'est pas entièrement écrit en Java, cf. *Liskov substitution principle*).
• type de tous les objets qui respectent le contrat.

• mode d'emploi pour utiliser les objets des classes qui l'implémentent.

- méthodes `final` interdites.

196. Méthodes `static` et `default` depuis Java 8, `private` depuis Java 9.
197. Ce qui est implicite n'a pas à être écrit dans le code, mais peut être écrit tout de même.

Interfaces : syntaxe de la déclaration

Pourquoi ces contraintes ?

Limites dues plus à l'idéologie (qui s'est assouplie) qu'à la technique.¹⁹⁸

- **À la base** : interface = juste description de la communication avec ses instances.
- **Mais, dès le début**, quelques « entorses » : constantes statiques, types membres.
- **Java 8** permet qu'une interface contienne des implémentations → la construction **interface** va au delà du concept d'« interface » de POO.
- **Java 9** ajoute l'idée que ce qui n'appartient pas à l'« interface » (selon POO) peut bien être privé (pour l'instant seulement méthodes).

Ligne rouge pas encore franchie : une interface ne peut pas imposer une implémentation à ses sous-types (**interdits** : constructeurs, attributs d'instance et méthodes **final**).

Conséquence : une interface n'est pas non plus directement¹⁹⁹ instantiable.

198. Il y a cependant des vraies contraintes techniques, notamment liées à l'héritage multiple.
199. Mais très facile via classe anonyme : **new** UneInterface(){ ... }.

Introduction
Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types
Sous-typeage
Transfertage
Polymorphisme(s)
Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Révision

```
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();
}
```

- Mettre **implements** I dans l'en-tête de la classe A pour implémenter l'interface I.

- Les méthodes de I sont définissables dans A. Ne pas oublier d'écrire **public**.

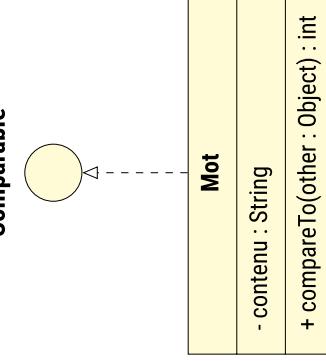
- Pour obtenir une « vraie » classe (non abstraite, i.e. instantiable) : nécessaire de définir toutes les méthodes abstraites promises dans l'interface implémentée.

- Si toutes les méthodes promises ne sont pas définies dans A, il faut précéder la déclaration de A du mot-clé **abstract** (classe abstraite, non instantiable)

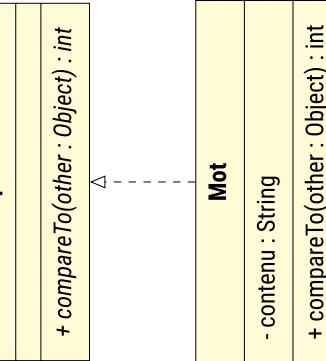
- Une classe peut implémenter plusieurs interfaces :

```
class A implements I, J, K { ... }.
```

ou version "abrégiée" :
Comparable



ou version "abrégiée" :
Comparable



```
public class Tri {
    static void tri(Comparable[] tab) {
        /* ... algorithme de tri
           utilisant tab[i].compareTo(tab[j]) */
        ...
    }
}
```

```
public static void main(String [] args) {
    Mot [] tableau = creeTableauMotsAuHasard();
    // on suppose que creeTableauMotsAuHasard existe
    tri(tableau);
    // Mot [] est compatible avec Comparable []
}
```

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

Exemple

- **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).

Détails
Généralités
Style
Objets et classes
Types et polymorphisme
Le système de types
Sous-typeage
Transfertage
Polymorphisme(s)
Surcharge
Interfaces
Héritage
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et exceptions
Exemple

Compléments
en POO

Aldric Degorre

Introduction
Généralités

Généralités
Style
Objets et classes
Types et polymorphisme
Le système de types
Sous-typeage
Transfertage
Polymorphisme(s)
Surcharge
Interfaces
Héritage
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et exceptions
Détails

```
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;
    }
}
```

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.

Compléments
en POO

Aldric Degorre

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

Héritage d'implémentations multiples

À cause des méthodes par défaut des interfaces

Compléments
en POO

Aldric Degorre

Introduction
Généralités

Généralités
Style
Objets et classes
Types et polymorphisme
Le système de types
Sous-typeage
Transfertage
Polymorphisme(s)
Surcharge
Interfaces
Héritage
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et exceptions
Détails

Héritage d'implémentations multiples

À cause des méthodes par défaut des interfaces

Introduction
Généralités

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 I { default void f() { System.out.println("I"); } }
interface J { default void f() { System.out.println("J"); } }
class A implements I, J {}
```

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(); }
}
```

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

Compléments
en POO

Aldric Degorre

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

```
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().f()** affiche **B**.

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types	Sous-typeage	Transtypage	Polytypisme(s)	Surcharge	Interfaces	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
Introduction	Généralités	Aldric Degoire													
Évitez d'écrire, dans votre programme, le nom des classes des objets qu'il utilise.															
Cela veut dire, évitez :															
Dépendance															
et préférez :															
Dépendance															
Cela s'appelle « programmer à l'interface ».															
DépendanceImpl															

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Le système de types	Sous-typeage	Transtypage	Polytypisme(s)	Surcharge	Interfaces	Héritage	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et exceptions
Introduction	Généralités	Aldric Degoire													
Évitez d'écrire, dans votre programme, le nom des classes des objets qu'il utilise.															
Cela veut dire, évitez :															
Dépendance															
et préférez :															
Dépendance															
Cela s'appelle « programmer à l'interface ».															
Dépendance															
DépendanceImpl															

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transtypage

Polytypisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Compléments en POO

Aldric Degoire

- **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. ²⁰³.

- **Quoi?** → on définit alors une interface idéale que la dépendance devrait implémenter et on la joint au package²⁰⁴ de la bibliothèque.

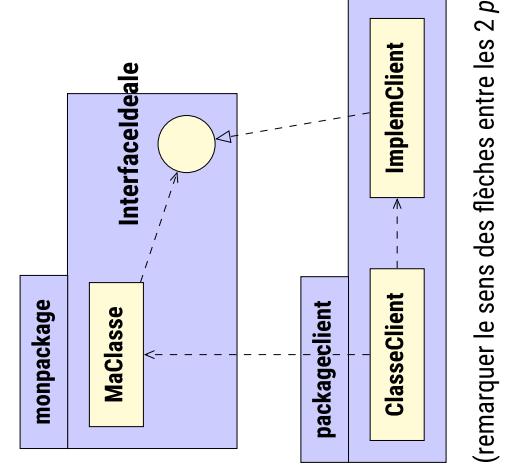
Les utilisateurs de la bibliothèque auront alors charge d'implémenter cette interface²⁰⁵ (ou de choisir une implémentation existante) pour fournir la dépendance.

203. Ou simplement parce que vous voulez avoir le contrôle de l'évolution de cette interface.

204. Si on utilise JPMIS : ce sera un des packages exportés.

205. Typiquement, les utilisateurs emploieront le patron « adaptateur » pour implémenter l'interface fournie à partir de diverses classes existantes.

206. Le « D » de SOLID (Michael Feathers & Robert C. Martin)



(remarquer le sens des flèches entre les 2 packages)

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Le système de types

Sous-typeage

Transtypage

Polytypisme(s)

Surchage

Interfaces

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

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 »²⁰⁷, ne dépend pas de la classe implémentant ce dernier. Selon le DIP, c'est le contraire qui se produit (dépendance à l'interface).

En des termes plus savants :

- *Depend upon Abstractions. Do not depend upon concrections.* ²⁰⁸

^{207.} et donc d'implémentation susceptible de changer plus souvent (justification du DIP par son inventeur)

^{208.} Robert C. Martin (2000), dans "Design Principles and Design Patterns".

Quand ?

- 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)

Quoi ? On crée alors 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 implémentées²⁰⁹ par des appels de méthodes sur **proxy**.

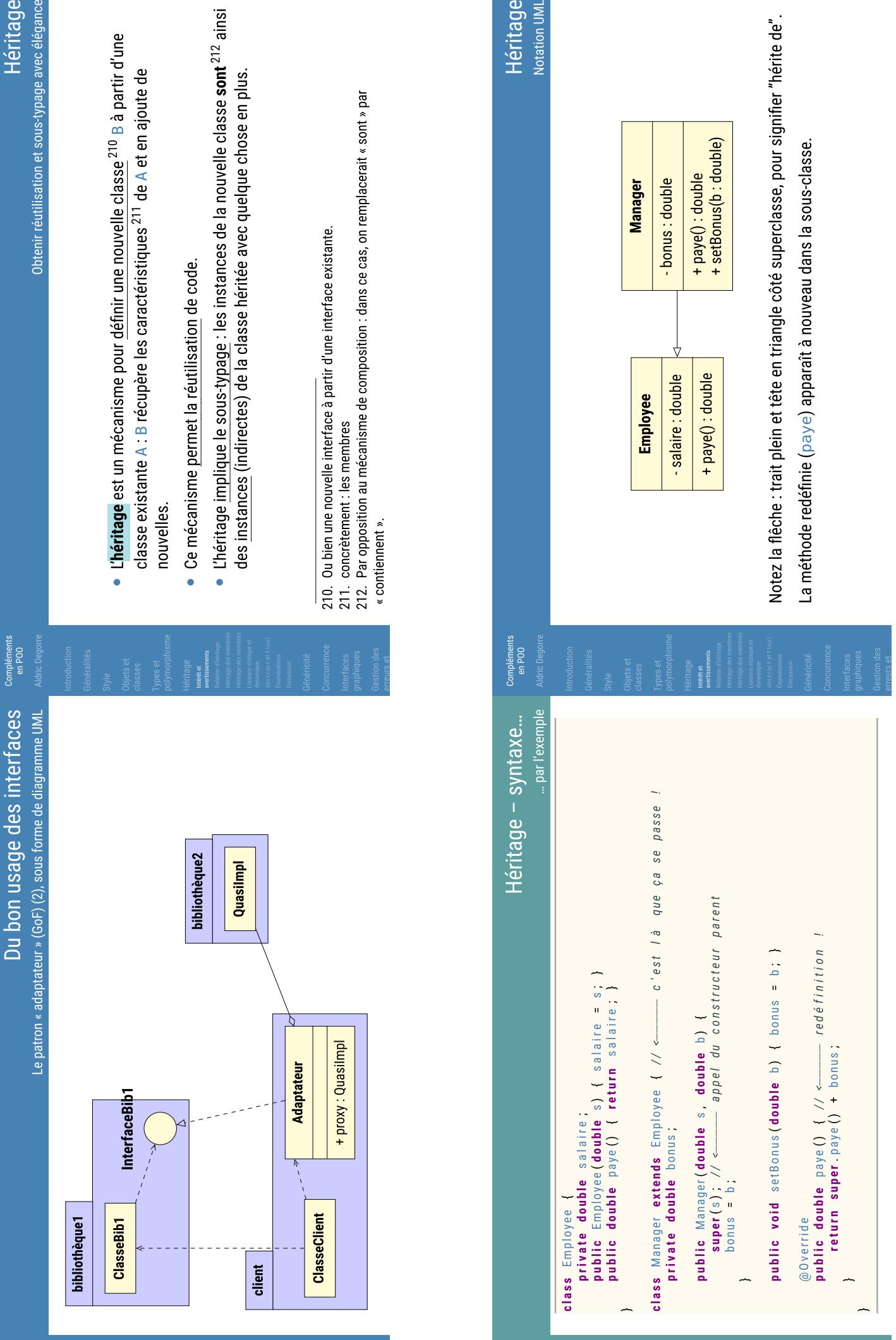
^{209.} De préférence très simplement et brièvement...

Héritage

Obtenir réutilisation et sous-typage avec élégance

Du bon usage des interfaces

Le patron « adaptateur » (GoF) (2), sous forme de diagramme UML



Héritage

Notation UML

Héritage – syntaxe...

... par l'exemple



L'héritage est un mécanisme pour définir une nouvelle classe²¹⁰ B à partir d'une classe existante A : B récupère les caractéristiques²¹¹ de A et en ajoute de nouvelles.

Ce mécanisme permet la réutilisation de code.

L'héritage implique le sous-typage : les instances de la nouvelle classe sont²¹² ainsi des instances (indirectes) de la classe héritée avec quelque chose en plus.

Ou bien une nouvelle interface à partir d'une interface existante.

Concrètement : les membres

Par opposition au mécanisme de composition : dans ce cas, on remplaceait « sont » par « contiennent ».

Notez la flèche : trait plein et tête en triangle côté superclasse, pour signifier "hérite de".

La méthode redéfinie (paye) apparaît à nouveau dans la sous-classe.

Folklore et « piliers » de la P00, avertissement

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Intérêt et avénements	héritage des membres	héritage des méthodes	héritage statique et dynamique	héritage abstrait et final	Relations d'héritage	Énumérations	Discussion	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et	
Compléments en P00	Aldric Degoire	Introduction																
D'après le folklore : « piliers » de la P00 = <u>encapsulation</u> , <u>polymorphisme</u> et héritage.																		
Mais ce dernier principe ne définit pas du tout la P00 ! ²¹³																		
Héritage : mécanisme de réutilisation de code très pratique, mais non fondamental.																		
∃ LOO sans héritage : les premières versions de Smalltalk; le langage Go. La P00 moderne incite aussi à préférer la composition à l'héritage. ²¹⁴																		
Avertissement : l'héritage, mal utilisé, est souvent source de rigidité ou de fragilité ²¹⁵ . Il faudra, suivant les cas, lui préférer l'implémentation d'interface ou la composition.																		
Faiblesses de l'héritage et alternatives possibles seront discutées à la fin de ce chapitre.																		
213. Rappel : P00 = programmation faisant communiquer des objets.																		
214. Ce qui n'empêche que l'héritage soit évidemment au programme de ce cours.																		
215. Ej3 19 : « Design and document for inheritance or else prohibit it »																		

Héritage

De classe ou d'interface ?

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Intérêt et avénements	héritage des membres	héritage des méthodes	héritage statique et dynamique	héritage abstrait et final	Relations d'héritage	Énumérations	Discussion	Généricité	Concurrence	Interfaces graphiques	Gestion des erreurs et	
Compléments en P00	Aldric Degoire	Introduction																
En Java, la notion d'héritage concerne à la fois les classes et les interfaces.																		
L'héritage n'a pas la même structure dans les 2 cas :																		
• Une classe peut hériter directement d'une (et seulement une) classe (« héritage simple »).																		
Par ailleurs, toutes les classes héritent de la classe Object .																		
• Une interface peut hériter directement d'une ou de plusieurs interfaces.																		
Elle peut aussi n'hériter d'aucune interface (pas d'ancêtre commun).																		

Remarque : avec l'héritage, on reste dans une même catégorie, classe ou interface, par opposition à la relation d'implémentation.

mot-clé parent enfant nb. parents graphique racine(s)	héritage de classe classe 1 ²¹⁶ arbre classe(s)	héritage d'interface classe classe ≥ 0 DAG	impléments interface interface ≥ 0 DAG	sous-typage de types référence (tout ça) type type ≥ 1 ²¹⁷ DAG type Object

216. 0 pour classe **Object**
217. 0 pour type **Object**

- Une classe²¹⁸ A peut « **étendre** »/« hériter directement de »/« être dérivée de/être une sous-classe directe d'une autre classe B. Nous noterons A ↘ B.
 - (Par opposition, B est appelée **superclasse directe** ou classe mère de A : B ↘ A).
 - Alors, tout se passe comme si les membres visibles de B étaient aussi définis dans A. On dit qu'ils sont **hérités** Conséquences :
 - 1** toute instance de A peut être utilisée comme²¹⁹ instance de B;
 - 2** donc une expression de type A peut être substituée à une expression de type B.
 - Le système de types en tient compte : A ↘ B ⇒ A <: B (A sous-type de B).
 - Dans le code de A, le mot-clé **super** est synonyme de B. Il sert à accéder aux membres de B, même masqués par une définition dans A (ex : **super . f() ;**).
218. Pour l'héritage d'interfaces : remplacer partout « classe » par interface.
219. Parce qu'on peut demander à l'instance de A les mêmes opérations qu'à une instance de B (critère bien plus faible que le principe de substitution de Liskov!).

Héritage (sous-entendu : « généralisé ») :	
• Une classe A hérite de/est une sous-classe d'une autre classe B s'il existe une séquence d'extensions de la forme : A ↘ A1 ↘ ... ↘ An ↘ B ²²⁰ .	• Une classe A hérite de/est une sous-classe d'une autre classe B s'il existe une séquence d'extensions de la forme : A ↘ A1 ↘ ... ↘ An ↘ B ²²⁰ .
Notation : A ⊑ B (remarques : A ↘ B ⇒ A ⊑ B, de plus A ⊑ A).	Notation : A ⊑ B (remarques : A ↘ B ⇒ A ⊑ B, de plus A ⊑ A).
• Par opposition, B est appelée <u>superclasse</u> (ou ancêtre) de A. On notera B ⊓ A.	• Par opposition, B est appelée <u>superclasse</u> (ou ancêtre) de A. On notera B ⊓ A.
• Héritage implique ²²¹ sous-type : A ⊑ B ⇒ A <: B.	• Héritage implique ²²¹ sous-type : A ⊑ B ⇒ A <: B.
• Pourtant, une classe n'hérite pas de tous les membres visibles de tous ses ancêtres, car certains ont pu être masqués par un ancêtre plus proche. ²²²	• Pourtant, une classe n'hérite pas de tous les membres visibles de tous ses ancêtres, car certains ont pu être masqués par un ancêtre plus proche. ²²²
• super . super n'existe pas ! Une classe est <u>isolée</u> de ses ancêtres indirects.	• super . super n'existe pas ! Une classe est <u>isolée</u> de ses ancêtres indirects.

220. L'héritage généralisé est la fermeture transitive de la relation d'héritage direct.
221. Héritage ⇒_{def.} chaîne d'héritages directs ⇒ <: <: : chaîne de sous-type ⇒_{transitivité de <:} sous-type.
222. C'est pourquoi je distingue héritage direct et généralisé. **Attention** : une instance d'une classe contient physiquement, tous les attributs d'instances définis dans ses superclasses, même masqués ou non visibles.

Héritage : la racine	
	La classe Object
Compléments en POO	Compléments en POO
Aldric Degoire	Aldric Degoire
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Intérêt et avertissements Relation d'héritage Héritage des membres Héritage des méthodes Listes statique et dynamique abstrait et final Enumérations Discussion Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Intérêt et avertissements Relation d'héritage Héritage des membres Héritage des méthodes Listes statique et dynamique abstrait et final Enumérations Discussion Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
La classe Object possède les méthodes suivantes :	Object possède les méthodes suivantes :
• boolean equals(Object other) : teste l'égalité de this et other	• boolean equals(Object other) : teste l'égalité de this et other
• String toString() : retourne la représentation en String de l'objet ²²⁴	• String toString() : retourne la représentation en String de l'objet ²²⁴
• int hashCode() : retourne le « hash code » de l'objet ²²⁵	• int hashCode() : retourne le « hash code » de l'objet ²²⁵
• Class <?> getClass() : retourne l'objet-classe de l'objet.	• Class <?> getClass() : retourne l'objet-classe de l'objet.
• protected Object clone() : retourne un « clone » ²²⁶ de l'objet si celui-ci est Cloneable, sinon quitte sur exception CloneNotSupportedException .	• protected Object clone() : retourne un « clone » ²²⁶ de l'objet si celui-ci est Cloneable, sinon quitte sur exception CloneNotSupportedException .
• protected void finalize() : appelée lors de la destruction de l'objet.	• protected void finalize() : appelée lors de la destruction de l'objet.
• et puis wait , notify et notifyAll que nous verrons plus tard (cf. threads).	• et puis wait , notify et notifyAll que nous verrons plus tard (cf. threads).
224. Utilisé en particulier par println et pour les conversions implicites vers String dans l'opérateur « + ».	224. Utilisé en particulier par println et pour les conversions implicites vers String dans l'opérateur « + ».
225. Entier calculé de façon déterministe depuis les champs d'un objet, satisfaisant, par contrat, a.equals(b) ⇒ a.hashCode() == b.hashCode() .	225. Entier calculé de façon déterministe depuis les champs d'un objet, satisfaisant, par contrat, a.equals(b) ⇒ a.hashCode() == b.hashCode() .
226. Attention : le rapport entre clone et Cloneable est plus compliqué qu'il en a l'air, cf. Ej3 item 13.	226. Attention : le rapport entre clone et Cloneable est plus compliqué qu'il en a l'air, cf. Ej3 item 13.

Héritage : la racine	
	La classe Object
Compléments en POO	Compléments en POO
Aldric Degoire	Aldric Degoire
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Intérêt et avertissements Relation d'héritage Héritage des membres Héritage des méthodes Listes statique et dynamique abstrait et final Enumérations Discussion Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Intérêt et avertissements Relation d'héritage Héritage des membres Héritage des méthodes Listes statique et dynamique abstrait et final Enumérations Discussion Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions
La classe Object est :	La classe Object est :
• superclasse de toutes les classes;	• superclasse de toutes les classes;
• superclasse directe de toutes les classes sans clause extends (dans ce cas, « extends Object » est implicite);	• superclasse directe de toutes les classes sans clause extends (dans ce cas, « extends Object » est implicite);
• racine de l'arbre d'héritage des classes ²²³ .	• racine de l'arbre d'héritage des classes ²²³ .
Et le type Object qu'elle définit est :	Et le type Object qu'elle définit est :
• supertype de tous les types références (y compris interfaces);	• supertype de tous les types références (y compris interfaces);
• supertype direct des classes sans clause ni extends ni implements et des interfaces sans clause extends ;	• supertype direct des classes sans clause ni extends ni implements et des interfaces sans clause extends ;
• unique source du graphe de sous-type des types références .	• unique source du graphe de sous-type des types références .
223. Ce graphe a un degré d'incidence de 1 (héritage simple) et une source unique, c'est donc un arbre. Notez que le graphe d'héritage des interfaces n'est pas un arbre mais un DAG (héritage multiple) à plusieurs sources et que le graphe de sous-type des types références est un DAG à source unique.	223. Ce graphe a un degré d'incidence de 1 (héritage simple) et une source unique, c'est donc un arbre. Notez que le graphe d'héritage des interfaces n'est pas un arbre mais un DAG (héritage multiple) à plusieurs sources et que le graphe de sous-type des types références est un DAG à source unique.

Héritage : la racine

La classe Object : ses méthodes (2)

Héritage : ajout, redéfinition, masquage

Le panorama... (1)

Conséquences :

- Grâce au sous-typage, tous les types référence ont ces méthodes, on peut donc les appeler sur toute expression de type référence.
 - Grâce à l'héritage, tous les objets disposent d'une implémentation de ces méthodes...
 - ... mais leur implémentation dans Object est souvent inutile :
 - equals : teste l'identité (égalité des adresses, comme ==);
 - toString : retourne une chaîne composée du nom de la classe et du hashCode.
- il faut généralement redéfinir equals (et donc 227 hashCode) et toString (cf. EJ3 Items 10, 11, 12).

227. Rappel du transparent précédent : si a.equals(b) alors il faut a.hashCode() == b.hashCode().

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Rélation d'héritage

Héritage des membres

Liasons statique et dynamique

abstrait et final

Énumérations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Compléments en POO

Aldric Degoire

Dans une sous-classe :

- On hérite des membres visibles²²⁸ de la superclasse directe²²⁹.
- Visible = public, protected, voire package-private, si superclasse dans même package.
- On peut masquer (to hide) n'importe quel membre hérité :
 - méthodes : par une définition de même signature
 - ... dans ce cas, le type de retour doit être identique²³⁰, sinon erreur de syntaxe!
 - autres membres : par une définition de même nom
- Les autres membres de la sous-classe sont dits ajoutés.

- ...
228. Il faut en fait visibles et non-private. En effet : private est parfois visible (cf. classes imbriquées).
229. que ceux-ci y aient été directement définis, ou bien qu'elle les aie elle-même hérités
230. En fait, si le type de retour est un type référencé, on peut retourner un sous-type. Par ailleurs il y a des subtilités dans le cas des types paramétrés, cf généricité.

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Rélation d'héritage

Héritage des membres

Liasons statique et dynamique

abstrait et final

Énumérations

Discussion

Compléments en POO

Aldric Degoire

Dans une sous-classe :

- On hérite des membres visibles²²⁸ de la superclasse directe²²⁹.
- Visible = public, protected, voire package-private, si superclasse dans même package.
- On peut masquer (to hide) n'importe quel membre hérité :
 - méthodes : par une définition de même signature
 - ... dans ce cas, le type de retour doit être identique²³⁰, sinon erreur de syntaxe!
 - autres membres : par une définition de même nom
- Les autres membres de la sous-classe sont dits ajoutés.

- ...
228. Il faut en fait visibles et non-private. En effet : private est parfois visible (cf. classes imbriquées).
229. que ceux-ci y aient été directement définis, ou bien qu'elle les aie elle-même hérités
230. En fait, si le type de retour est un type référencé, on peut retourner un sous-type. Par ailleurs il y a des subtilités dans le cas des types paramétrés, cf généricité.

- De même, les définitions non visibles des superclasses restent « portées » par les instances de la sous-classe, même si elles ne sont pas accessibles directement.

class A { private int x; }
class B extends A { int y; }
...
B b = new B(); // <- cet objet contient deux int

231. Mon parti pris : redéfinition = cas particulier du masquage. D'autres sources restreignent, au contraire, la définition de « masquage » aux cas où il n'y a pas de redéfinition (« masquage simple »). La JLS dit que les méthodes d'instance sont redéfinies et jamais qu'elles sont masquées... mais ne dit pas non plus que le terme est inapproprié.

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Rélation d'héritage

Héritage des membres

Liasons statique et dynamique

abstrait et final

Énumérations

Discussion

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Rélation d'héritage

Héritage des membres

Liasons statique et dynamique

abstrait et final

Énumérations

Discussion

Dans une sous-classe :

- Une méthode d'instance (non statique) masquée est dite redéfinie²³¹ (overridden).
- Dans le cas d'une redéfinition, il est interdit de :
 - redéfinir une méthode final,
 - réduire la visibilité (e.g. redéfinir une méthode public par une méthode private),
 - ajouter une exception dans la clause throws (cf. cours sur les exceptions).

- La notion de redéfinition est importante en POO (cf. liaison dynamique).

- De même, les définitions non visibles des superclasses restent « portées » par les instances de la sous-classe, même si elles ne sont pas accessibles directement.

class A { private int x; }
class B extends A { int y; }
...
B b = new B(); // <- cet objet contient aussi deux int

Héritage : ajout, redéfinition, masquage

Héritage : ajout, redéfinition, masquage
Cas tordu

Compléments en POO
Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Héritage des membres

Relations d'héritage

Héritage des méthodes

Liasons statique et dynamique

abstrait et final

Énumérations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Compléments en POO

Aldric Degoire

Remarques diverses (2)

- Les membres non hérités ne peuvent pas être masqués ou redéfinis, mais rien n'empêche de définir à nouveau un membre de même nom (= ajout).

```
class A { private int x; }
class B extends A { int x; } // autorisé !
```

Un ajout simple dans un contexte peut provoquer un masquage 232 dans un autre :

```
package bbb;
public class B extends aaa.A {
    public void f() { System.out.println("B"); } // avec @Override, ça ne compile pas
}
```

```
package aaa;
public class A {
    void f() { System.out.println("A"); } // méthode package-private, invisible dans bbb
    public static void main(String[] args) {
        bbb.B b = new bbb.B();
        b.f(); // contexte : instance de classe B, ici f de bbb.B masque f de aaa.A
        ((A) b).f(); // contexte : instance de classe A, f de aaa.A ni masquée ni redéfinie
    }
}
```

232. Ici, des méthodes d'instance se masquent l'une et l'autre sans se redéfinir.
Cela a l'air de contredire ce qu'on vient de dire, mais en réalité masquage implique redéfinition seulement s'il y a masquage dans le contexte où est définie la méthode masquante.

Héritage : ajout, redéfinition, masquage

Exemple

Non, mais sérieusement, pourquoi distinguer la redéfinition du masquage simple ?

Compléments en POO

Aldric Degoire

```
class GrandParent {
    protected static int a, b; // visibilité protected, assure que l'héritage se fait bien
    protected static void q() {}
    protected void f() {}
}

class Parent extends GrandParent {
    protected static int a; // masque le hérité de GrandParent (tjs accessible via super.a)
    // masque g() hérité de GrandParent (tjs appelable via super.g()).
    protected static void g() {}
    // redéfinit f() hérité de GrandParent (tjs appelable via super.f()).
    @Override protected void f() {}
}

class Enfant extends Parent { @Override protected void f() {} }
```

Liaisons statique et dynamique

Compléments en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Héritage des membres

Relations d'héritage

Héritage des méthodes

Liasons statique et dynamique

abstrait et final

Énumérations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Compléments en POO

Aldric Degoire

À la compilation : dans tous les cas, les usages d'un nom de méthode sont traduits comme référence vers une méthode existant dans le contexte d'appel (classe).

À l'exécution :

- Autres membres que méthodes d'instance : la méthode trouvée à la compilation sera effectivement appelée.
- Mécanisme de liaison statique (ou précoce).

- La classe Enfant hérite a, g et f de Parent et b de GrandParent via Parent.
- a et g de GrandParent masqués mais accessibles via préfixe GrandParent..
- f de Parent héritée mais redéfinie dans Enfant. Appel de la version de Parent avec super.f().
- f de GrandParent masquée par celle de Parent mais peut être appellée sur le récepteur de classe GrandParent. Remarque : super.super n'existe pas.

233. Sauf méthodes privées et sauf appel avec préfixe "super". → liaison statique.

Liaison statique

Liaisons statique et dynamique

Exemple	Principes	Compléments en POO
<pre>class A { public A() { f(); g(); } static void f() { System.out.println("A::f"); } void g() { System.out.println("A::g"); } }</pre>	<p>Attention : la liaison statique concerne tous les membres sauf les méthodes d'instance²³⁴.</p> <p>Lorsqu'on fait référence à un membre, lequel choisir ?</p> <p>Principe de la liaison statique : dès la compilation, on décide quelle définition sera effectivement utilisée pour l'exécution (c.-à-d. toutes les exécutions).</p> <p>Pour l'explication, nous distinguons cependant :</p> <ul style="list-style-type: none">d'abord le cas simple (tout membre sauf méthode)ensuite le cas moins simple des méthodes (possible surcharge)	<p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Inertie et avérifications Héritage des membres Liasons statique et dynamique abstract et final Interfaces graphiques Gestion des erreurs et</p> <p>Discussion Généricité Concurrence Interfaces graphiques Gestion des erreurs et</p>
<pre>class B extends A { public B() { super(); } static void f() { // masque System.out.println("B::f"); } void g() { // redéfinition System.out.println("B::g"); } }</pre>	<p>Si on fait new B(), alors on verra s'afficher</p> <p>A::f B::g</p>	<p>234. Sauf les méthodes d'instance private pour lesquelles la liaison est statique.</p>
<pre>class C { static void m() { System.out.println("C::m"); } }</pre>	<p>Pour trouver la bonne définition d'un membre (non méthode) de nom m, à un point donné du programme.</p> <ul style="list-style-type: none">Si m membre statique, soit C le contexte²³⁵ d'appel de m. Sinon, soit C la classe de l'objet sur lequel on appelle m.On cherche dans le corps de C une définition visible et compatible (même catégorie de membre, même "staticté", même type ou sous-type...), puis dans les types parents de C (superclasse et interfaces implémentées), puis les parents des parents (et ainsi de suite). <p>On utilise la première définition qui convient.</p>	<p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Inertie et avérifications Héritage des membres Liasons statique et dynamique abstract et final Interfaces graphiques Gestion des erreurs et</p> <p>Discussion Généricité Concurrence Interfaces graphiques Gestion des erreurs et</p>
<pre>class D { static void m() { System.out.println("D::m"); } }</pre>	<p>Pour les méthodes : même principe, mais on garde toutes les méthodes de signature compatible avec l'appel, puis on applique la résolution de la surcharge. Ce qui donne...</p>	<p>235. le plus souvent classe ou interface, en toute généralité une définition de type</p>

Liaison statique

Principe : cas des méthodes statiques
Principe : cas des méthodes statiques

Quelle définition de **f** utiliser, quand on appelle **f(x1, x2, ...)**, avec **f** statique ?

- 1 **C** := contexte de l'appel de la méthode (classe ou interface).
- 2 Soit **Mf** := \emptyset .
- 3 **Mf** := { méthodes statiques de nom **f** dans **C**, compatibles avec **x1, x2, ...**}.
- 4 Pour tout supertype direct **P** de **C**, **Mf** += { méthodes statiques de nom **f** dans **P**, compatibles avec **x1, x2, ..., non masquées**²³⁶ par autre méthode dans **Mf**}. On répète l'étape avec les supertypes des supertypes, et ainsi de suite.²³⁷
- 5 On choisit la méthode de **Mf** dont les paramètres ont les types les plus petits.

Le compilateur ajoute au code-octet l'instruction **invokestatic** avec pour paramètre une référence vers la méthode trouvée.

236. à cause de la surcharge il peut exister des méthodes de même nom non masquées
237. Jusqu'aux racines du graphe de sous-type.

Lors de l'appel `x.f(y)`, quelle définition de `f` choisir?

→ Principe de la liaison dynamique :

- à la compilation, une méthode de nom `f` et de signature compatible avec `y` est recherchée depuis le **type statique** de `x`; le compilateur ajoute au code-octet l'instruction **invokevirtual** avec, pour paramètre, une référence vers la méthode trouvée²³⁸.
- à l'exécution, une redéfinition de cette méthode est cherchée depuis le **type dynamique**²³⁹ de `x`²⁴⁰, la redéfinition "la plus proche" sera exécutée.

238. Pour une méthode d'instance privée ou un appel précédé de "**super**", l'instruction utilisée est **invokespecial** et le comportement sera similaire à **invokesstatic**.

239. Rappel : type dynamique = classe de l'objet référencé par l'expression à l'exécution.

240. Mais pour `y`: seulement du type de l'expression `y` (à la compilation). On parle de *single dispatch*. D'autres langages que Java ont au contraire un mécanisme de *multiple dispatch*.

• Classe dérivée : certaines méthodes peuvent être redéfinies

```
class A { void f() { System.out.println("classe A"); } }
class B extends A { void f() { System.out.println("classe B"); } }

public static void main(String args[]) {
    B b = new B();
    b.f(); // <-- affiche "classe B"
}
```

• mais aussi...

```
public class Test {
    public static void main(String args[]) {
        A b = new B(); // <-- maintenant variable b de type A
        b.f(); // <-- affiche "classe B" quand-même
    }
}
```

Imaginons le cas suivant, avec redéfinition et surcharge :

```
class Y1 {}
class Y2 extends Y1 {}

class X1 { void f(Y1) { System.out.print("X1 et Y1:"); } }

class X2 extends X1 {
    void f(Y1 y) { System.out.print("X2 et Y1:"); }
    void f(Y2 y) { System.out.print("X2 et Y2:"); }
}

class X3 extends X2 { void f(Y2 y) { System.out.print("X3 et Y2:"); } }

public class Liaisons {
    public static void main(String args[]) {
        X3 x = new X3(); Y2 y = new Y2();
        // notez tous les upcastings explicités ci-dessous (servent-il s vraiment à rien ?)
        ((X1)x).f((Y1)y);
        ((X2)x).f((Y1)y);
        ((X2)x).f((Y2)y);
        x.f((Y1)y);
        x.f(y);
    }
}
```

2 étapes :

À la compilation, on vérifie si l'appel `x.f(y)` est correct (i.e. bien typé).

Pour cela, algorithme de recherche identique à celui de la liaison statique, mais recherche parmi les méthodes d'instance.

Si on trouve finalement une méthode `m`, alors, la 2ème étape est garantie de fonctionner : À l'exécution, choix effectif de la méthode à appeler. Recherche²⁴¹ de méthode comme à la liaison statique, mais :

- recherche restreinte aux méthodes qui redéfinissent `m`²⁴²
- point de départ de la recherche = classe de l'objet référencé par `x`.

Qu'est-ce qui s'affiche ?

241. En réalité, recherche exécutée une seule fois, car résultat mis en cache.

242. Ainsi les éventuelles surcharges existant dans la classe de l'objet mais pas dans celle de l'expression sont ignorées à cette étape.

```

class Y1 {}  

class Y2 extends Y1 {}  

class X1 extends Y1 {} { void f(Y1 y) { System.out.print("X1_let,Y1_;" ); } }  

class X2 extends X1 {} { void f(Y1 y) { System.out.print("X2_let,Y1_;" ); } }  

void f(Y2 y) { System.out.print("X2_let,Y2_;" ); }  

class X3 extends X2 {} void f(Y2 y) { System.out.print("X3_let,Y2_;" ); } }

public class Liasons {
    public static void main(String args[]) {
        X3 x = new X3(); Y2 y = new Y2();
        // notez tous les upcastings explicites ci-dessous (servent-ils vraiment à rien ?)
        ((X1)x).f(y); ((X1)x).f(y);
        ((X2)x).f(y); ((X2)x).f(y);
        ((X3)x).f(y); ((X3)x).f(y);
    }
}

```

Affiche : X2 et Y1 ; X2 et Y1 ; X2 et Y1 ; X3 et Y1 ; X3 et Y2 ;

- Pour les instructions commençant par ((X1)x) : la phase statique cherche les signatures dans X1 → les surcharges prenant Y2 sont ignorées à l'exécution.
- Les instructions commençant par ((X2)x) . se comportent comme celles commençant par x . : les mêmes signatures sont connues dans X2 et X3.

```

class Y1 {}  

class Y2 extends Y1 {} { void f(Y1 y) { System.out.print("X1_let,Y1_;" ); } }  

class X1 extends Y1 {} { void f(Y1 y) { System.out.print("X2_let,Y1_;" ); } }  

void f(Y2 y) { System.out.print("X2_let,Y2_;" ); }  

class X3 extends X2 {} void f(Y2 y) { System.out.print("X3_let,Y2_;" ); } }

public class Object {
    // la 'vraie', c.-à-d. java.lang.Object
    ...
    public boolean equals(Object obj) { return this == obj; }
    ...
}

class C /* sous-entendu : extends Object */ {
    public boolean equals(C obj) { return ...; } // <- c'est une surcharge, pas une redéfinition !
}

```

- **Recommandé :** placer l'annotation @Override devant une définition de méthode pour demander au compilateur de générer une erreur si ce n'est pas une redéfinition.
 - Exemple : @Override **public boolean** equals(**Object** obj) { return ...; }
243. même pas un masquage

```

public class A {
    int f(int x) { return 0; }
    abstract int g(int x); // <- oh, une méthode abstraite !
}

```

Méthode abstraite : méthode déclarée sans être définie.

- Pour déclarer une méthode comme abstraite, faire précéder sa déclaration du mot-clé **abstract**, et ne pas écrire son corps (reste la signature suivie de « ; »).
- Classe abstraite** : classe déclarée comme **non directement instanciable**. Elle se déclare en faisant précéder sa déclaration du modificateur **abstract**:

```

abstract class A {
    int f(int x) { return 0; }
    abstract int g(int x); // <- oh, une méthode abstraite !
}

```

- Le lien entre les 2** : une méthode abstraite ne peut être pas déclarée dans un type directement instanciable → seulement dans interfaces et classes abstraites.
- Interprétation** : tout objet instantié doit connaître une implémentation pour chacune de ses méthodes.

- Une méthode abstraite a vocation à être redéfinie dans une sous-classe.
- Conséquence** : **abstract static, abstract final** et **abstract final private** sont des non-sens !

```

class Employee {
    private String name;
    public final String getName() { return name; }
    ...
}

```

On peut déclarer une méthode avec modificateur **final**²⁴⁴. Exemple :

```

class Employee {
    private String name;
    public final String getName() { return name; }
    ...
}

==> ici, final empêche une sous-classe de Employee de redéfinir getName()
Aussi possible :
final class Employee { ... }

```

```

class Y1 {}  

class Y2 extends Y1 {} { void f(Y1 y) { System.out.print("X1_let,Y1_;" ); } }  

class X1 extends Y1 {} { void f(Y1 y) { System.out.print("X2_let,Y1_;" ); } }  

void f(Y2 y) { System.out.print("X2_let,Y2_;" ); }  

class X3 extends X2 {} void f(Y2 y) { System.out.print("X3_let,Y2_;" ); } }

public class Object {
    // la 'vraie', c.-à-d. java.lang.Object
    ...
    public boolean equals(Object obj) { return this == obj; }
    ...
}

class C /* sous-entendu : extends Object */ {
    public boolean equals(C obj) { return ...; } // <- c'est une surcharge !
}

```

- si on se trompe dans le type ou le nombre de paramètres, ce n'est pas une redéfinition²⁴³, mais un ajout de méthode surchargée. Erreur typique :

```

public class A {
    int f(int x) { return 0; }
    abstract int g(int x); // <- oh, une méthode abstraite !
}

```

Méthode abstraite : méthode déclarée sans être définie.

- Pour déclarer une méthode comme abstraite, faire précéder sa déclaration du mot-clé **abstract**, et ne pas écrire son corps (reste la signature suivie de « ; »).
- Classe abstraite** : classe déclarée comme **non directement instanciable**. Elle se déclare en faisant précéder sa déclaration du modificateur **abstract**:

```

abstract class A {
    int f(int x) { return 0; }
    abstract int g(int x); // <- oh, une méthode abstraite !
}

```

- Le lien entre les 2** : une méthode abstraite ne peut être pas déclarée dans un type directement instanciable → seulement dans interfaces et classes abstraites.
- Interprétation** : tout objet instantié doit connaître une implémentation pour chacune de ses méthodes.

- Une méthode abstraite a vocation à être redéfinie dans une sous-classe.
- Conséquence** : **abstract static, abstract final** et **abstract final private** sont des non-sens !

244. Attention : une variable peut aussi être déclarée avec le mot-clé **final**. Sa signification est alors différente : il interdit juste toute nouvelle affectation de la variable après son initialisation.

245. Ainsi, pour résumer, on a le droit de redéfinir les méthodes héritées non **static** et non **final**.

Méthodes et classes abstraites

Méthodes et classes finales ou abstraites

Notation UML

Compléments en POO

Alfric Degorre

```
abstract class Figure {
    Point2D centre; String nom; // autres attributs éventuellement
    public abstract Point2D getVertexNumber();
    public final double perimetre() {
        Point2D courant = getVertex(0);
        double peri = 0;
        for (int i=1; i < getVertexNumber(); i++) {
            Point2D suivant = getVertex(i);
            peri += courant.distance(suivant);
            courant = suivant;
        }
        return peri + courant.distance(getVertex(0));
    }
}

final class Triangle extends Figure {
    private Point2D a, b, c;
    @Override public int getVertexNumber() {
        return 3;
    }
    @Override public Point2D getVertex(int i) {
        switch (i) {
            case 0: return a;
            case 1: return b;
            case 2: return c;
            default: throw new NoSuchElementException();
        }
    }
}
```

Exemple

Alfric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Interet et avertissements

Relation d'héritage

Héritage des membres

Liasons statique et dynamique

abstract et tool

Discussions

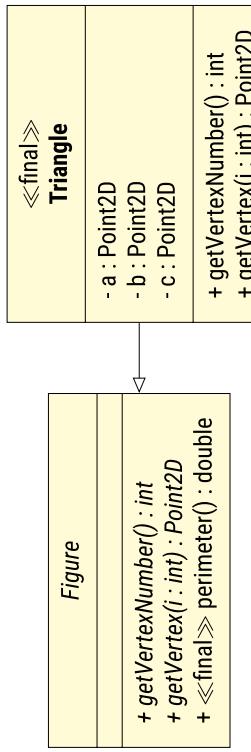
Énumérations

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et



Remarquez l'italique pour les méthodes et classes abstraites. En revanche, **final** n'a pas de typographie particulière²⁴⁶.

246. **final** n'est pas un concept de la spécification d'UML, mais heureusement, UML autorise à ajouter des informations supplémentaires en tant que « stéréotypes », écrits entre doubles chevrons.

abstract et final

abstract et final

le bon usage pour les classes (2)

Compléments en POO

Alfric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Interet et avertissements

Relation d'héritage

Héritage des membres

Liasons statique et

dynamique

abstract et tool

Discussions

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Constat : une classe non finale correspond à une implémentation complétable.
Idéologie : si c'est complétable c'est que c'est donc probablement incomplet.²⁴⁸
Si cela est vrai, alors une classe ni finale ni abstraite est louche!²⁴⁹. Comme **abstract - final** est exclus d'office, toute classe devrait alors être soit (juste) **abstract** soit (juste) **final**.

pas abstract	louche	OK	interdit
final			

248. Ce n'est pas toujours vrai : certaines classes proposent un comportement par défaut tout à fait valable, tout en laissant la porte ouverte à des modifications (cf. composants Swing).
249. On parle de code smell. Cela dit, c'est « louche », mais pas absurde, cf. remarque précédente.

Dans les deux cas, on interdit la possibilité d'instances absurdes (respectivement incohérents ou incomplets) de la classe marquée.
247. **abstract**, appliqué à une méthode, n'est une contrainte que dans la mesure où cela force à marquer aussi **abstract** la classe la contenant.

abstract et final

le bon usage pour les classes : exemple

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Intérêt et avertissements	Héritage des membres	Relations d'héritage	Liasons statique et dynamique	abstract et final	Énumérations	Discussion
le bon usage pour les classes : illustration en UML												

En UML, une bonne structure d'héritage donnerait des diagrammes comme celui-ci :

```

classDiagram
    class A {
        <<final>>
    }
    class C {
        <<final>>
    }
    class D {
        <<final>>
    }
    A --> B
    A --> C
    C --> D
  
```

Mieux :

```

class Personne {
    public String getNom() { return null; } // mauvaise implémentation par défaut
}
class PersonneImpl extends Personne {
    private String nom;
    @Override public String getNom() { return nom; }
}

final class Personne {
    private String nom;
    @Override public String getNom() { return nom; }
}
  
```

abstract et final

le bon usage pour les méthodes

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Intérêt et avertissements	Héritage des membres	Relations d'héritage	Liasons statique et dynamique	abstract et final	Énumérations	Discussion
le bon usage pour les méthodes : contre-exemple												

Quand on programme une classe extensible :

- Si possible, éviter tout appel, depuis une autre méthode de la classe 250, de méthode redéfinissable (= non **final**) = « ouverte »).
- À défaut le signaler dans la documentation.
- Objectif : éviter des erreurs bêtes dans les futures extensions.
- Par exemple : appels mutuellement récursifs non voulus.
- La documentation devra donner une spécification des méthodes redéfinissables assurant de conserver un comportement globalement correct.

abstract et final

le bon usage pour les méthodes : contre-exemple

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Intérêt et avertissements	Héritage des membres	Relations d'héritage	Liasons statique et dynamique	abstract et final	Énumérations	Discussion
le bon usage pour les méthodes : contre-exemple												

Sans **final, Personne est une classe de base fragile. Quelqu'un pourrait écrire :**

```

class Personne {
    private String prenom, nom;
    public String getPrenom() { return prenom; } // il faudrait final
    public String getNom() { return nom; } // à aussi
    public String getNomComplet() {
        return getPrenom() + " " + getNom(); // appeler à méthodes redéfinissables → danger !!!
    }
}
  
```

À ne pas faire non plus :

Compléments en POO

Aldric Degoire

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Intérêt et avertissements	Héritage des membres	Relations d'héritage	Liasons statique et dynamique	abstract et final	Énumérations	Discussion
le bon usage pour les méthodes : contre-exemple												

Récursion non bornée ! → StackOverflowError.

... puis exécuter **new Personne2(...).getNom()**, qui appelle **getNomComplet()**, qui appelle **getPrenom()** et **getNom()**, qui appellent **getNomComplet()** qui appelle...

abstract et final

L'héritage casse-t-il l'encapsulation ?

abstract et final
le bon usage : résumé

On entend souvent dire « L'héritage casse l'encapsulation. » (mais c'est exagéré).

Signification : pour qu'une classe soit étendue correctement, documenter ses membres **public** ne suffit pas ; certains points d'implémentation²⁵¹ doivent aussi l'être.

→ Cela contredit l'idée que l'implémentation d'une classe devrait être une « boîte noire ».

À défaut de pouvoir faire cet effort de documentation pour une classe, il est plus raisonnable d'intégrer d'hériter de celle-ci (→ **final class**).

EJ3, Item 19 : « Design and document for inheritance or else prohibit it »

251. À commencer par l'évidence : les membres **protected**. Mais même cela ne suffit pas.

Compléments en POO
Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Héritage des membres

Méthode des membres

Utilisation statique et dynamique

abstract et final

Enumérations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Une stratégie simple et extrême :

- Déclarer **final** toute classe destinée à être instanciée.
↔ feuilles de l'arbre d'héritage.
- Déclarer **abstract** toute classe destinée à être étendue²⁵².
↔ nœuds internes de l'arbre d'héritage.
- Dans ce dernier cas, déclarer en **private** ou **final** tous les membres qui peuvent l'être, afin d'empêcher que les extensions cassent les contrats déjà implémentés.
- Écrire la spécification de toute méthode redéfinissable (telle que, si elle est respectée, les contrats soient alors aussi respectés).

252. Voir, si la classe n'a pas d'attribut d'instance, déclarer plutôt une interface !

Application : les types scellés

Compléments en POO
Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avantage

Héritage des membres

Méthode des membres

Utilisation statique et dynamique

abstract et final

Enumérations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Autre mécanisme : une classe à constructeurs privés est instanciable et extensible seulement depuis l'intérieur de son corps ou de celui de son type englobant.

Théorème : ²⁵⁴ en Java²⁵⁵, les types scellés sont exactement ceux définis par une classe **final** ou à constructeurs privés telle que ses sous-classes directes sont aussi scellées.

Preuve :

- Par récurrence, toute l'arborescence de sous-types est imbriquée dans un même type englobant, donc définie dans le même fichier .java. Il n'est donc pas possible d'ajouter un sous-type sans modifier le fichier et le recompiler.
- Réciproquement, si une classe n'est pas **final** et a un constructeur non privé, on peut alors l'étendre depuis un autre fichier.

253. Notamment :
- besoin d'écrire une méthode dont les comportements varient en fonction des types de plusieurs paramètres (impossible : la liaison dynamique est *single dispatch*);
 - besoin d'ajouter un comportement à un type fourni par un tiers (impossible d'y ajouter une méthode).

254. Correct si on considère que les classes privées et locales sont à constructeurs privés.
255. Dans d'autres langages (Scala, Kotlin), un mot-clé **sealed** permet de déclarer un type scellé sans bricoler avec les constructeurs. Un tel mot-clé est en discussion pour une future version de Java.

Application : les types scellés

Application : les types scellés

Exemple typique : type algébrique

```

public abstract class BoolExpr {
    public abstract boolean eval(); // classe scellée (et abstraite !)
}

private BoolExpr() {}

public static final class Var extends BoolExpr {
    private final boolean value;
    public Var(boolean value) { this.value = value; } // appel super() implicite (super est accessible car Var est imbriquée)
    @Override public boolean eval() { return value; }
}

public static final class Not extends BoolExpr {
    private final BoolExpr subexpr;
    public Not(BoolExpr subexpr) { this.subexpr = subexpr; } // même remarque
    @Override public boolean eval() { return !subexpr.eval(); }
}

public static final class And extends BoolExpr {
    private final BoolExpr subexpr1, subexpr2;
    public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 = subexpr1; } // même remarque
    @Override public boolean eval() { return subexpr1.eval() && subexpr2.eval(); }
}

public static final class Or extends BoolExpr {
    private final BoolExpr subexpr1, subexpr2;
    public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 = subexpr1; } // même remarque
    @Override public boolean eval() { return subexpr1.eval() || subexpr2.eval(); }
}

```

Introduction

Compléments en POO

Aldric Degoire

```

public abstract class BoolExpr {
    public static boolean eval(BoolExpr expr) { // en vrai, la version précédente était mieux
        if (expr instanceof Var) return ((Var) expr).value;
        else if (expr instanceof Not) return eval(((Not) expr).subexpr);
        else if (expr instanceof And) return eval(((And) expr).subexpr) && eval(((And) expr).subexpr2);
        else if (expr instanceof Or) return eval(((Or) expr).subexpr) || eval(((Or) expr).subexpr2);
        else { assert false; "Cannot happen...the pattern matching is exhaustive!"; return false; }
    }
}

private BoolExpr() {}

public static final class Var extends BoolExpr {
    private final boolean value;
    public Var(boolean value) { this.value = value; } // valeur = value;
    @Override public boolean eval() { return value; }
}

public static final class Not extends BoolExpr {
    private final boolean value;
    public Var(boolean value) { this.value = value; }
    @Override public boolean eval() { return !value; }
}

public static final class And extends BoolExpr {
    private final BoolExpr subexpr;
    public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    @Override public boolean eval() { return subexpr.eval(); }
}

public static final class Or extends BoolExpr {
    private final BoolExpr subexpr;
    public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    @Override public boolean eval() { return subexpr.eval(); }
}

public static final class And extends BoolExpr {
    private final BoolExpr subexpr1, subexpr2;
    public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 = subexpr1; } // même remarque
    @Override public boolean eval() { return subexpr1.eval() && subexpr2.eval(); }
}

public static final class Or extends BoolExpr {
    private final BoolExpr subexpr1, subexpr2;
    public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 = subexpr1; } // même remarque
    @Override public boolean eval() { return subexpr1.eval() || subexpr2.eval(); }
}

```

Introduction

Compléments en POO

Aldric Degoire

Compléments en POO

Aldric Degoire

Types finis

Pour quoi faire ?

Pourquoi définir un type fini plutôt que réutiliser un type existant ?

- Typiquement, types de Java trop grands²⁵⁷. Si utilisés pour représenter un ensemble fini, difficile voire impossible de prouver que les variables ne prennent pas des valeurs absurdes.
- Même si on l'a prouvé sur papier, le programme peut comporter des typos (ex : "lundi" au lieu de "lundi"), que le compilateur ne les verra pas.

Avec un type fini, le compilateur garantit que la variable reste dans le bon ensemble.

→ Situation intéressante car, théoriquement, nombre fini de cas à tester/vérifier.

256. C'est donc un type scellé (très contraint) : clairement, si un type n'est pas scellé, il ne peut pas être fini.

257. Soit très grands (p. ex., il y a 2³² **ints**), soit quasi-infinis (il ne peut pas exister plus de 2³² références en même temps, mais à l'exécution, un objet peut être détruit et un autre reçue à la même adresse...).

258. Il pourrait aussi théoriquement vérifier l'exhaustivité des cas d'un **switch** (sans **default**) ou d'un **if / else if** (sans **else** seul) : ça existe dans d'autres langages, mais **javac** ne le fait pas²⁵⁹. Intérêt : éviter des **default** et des **else** que l'on sait inatteignables.

Types finis

Pour quoi faire ?

Un type fini est un type ayant un ensemble fini d'instances, toutes définies statiquement dès l'écriture du type, sans possibilité d'en créer de nouvelles lors de l'exécution.

- Certaines variables ont, en effet, une valeur qui doit rester dans un ensemble fini, prédefini :
- les 7 jours de la semaine
 - les 4 points cardinaux
 - les 3 (ou 4 ou plus) états de la matière
 - les n états d'un automate fini (dans protocole ou processus industriel, par exemple)
 - les 3 mousquetaires, les 7 nains, les 9 nazgûls...

Avec un type fini, le compilateur garantit que la variable reste dans le bon ensemble.

→ Situation intéressante car, théoriquement, nombre fini de cas à tester/vérifier.

256. C'est donc un type scellé (très contraint) : clairement, si un type n'est pas scellé, il ne peut pas être fini.

Écrire un type fini

Comment faire ?

- Mauvaise idée :** réserver un nombre fini de constantes dans un type existant (ça ne résout pas les problèmes évoqués précédemment).
- Remarque :** c'est ce que fait la construction **enum** du langage C. Les constantes déclarées sont en effet des **int**, et le type créé est un alias de **int**.
- On a déjà vu qu'il fallait créer un nouveau type.
- Il faut qu'il soit impossible d'en créer des instances en dehors de sa déclaration...
- ... qu'elles soient directes (appel de son constructeur) ou indirectes (via extension).

- Bonne idée :** implémenter le type fini comme classe à constructeurs privés et créer les instances du type fini comme constantes statiques de la classe :

```
public class Piece { // peut être final... mais le constructeur privé suffit
    private Piece() {}
    public static final Piece PILE = new Piece(), FACE = new Piece();
}
```

→ les **enum** de Java sont du sucré syntaxique pour écrire cela (+ méthodes utiles).

```
public enum ETAT { SOLIDE, LIQUIDE, GAZ, PLASMA }
```

Une **classe d'énumération** (ou juste énumération) est une classe particulière, déclarée par un bloc syntaxique **enum**, dans lequel est donnée la liste (exhaustive et définitive) des instances (= « constantes » de l'**enum**).

Elle définit un **type énuméré**, qui est un type :

- fini : c'est la raison d'être de cette construction;
- pour lequel l'opérateur « == » teste bien l'égalité sémantique 260 (toutes les instances représentent des valeurs différentes);
- utilisable en argument d'un bloc **switch**;
- et dont l'ensemble des instances s'itere facilement :
- for** (**MonEnum val**: **MonEnum.values()** {...})

260. Pour les enums, identité et égalité sont synonymes.

Compléments
en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

interne et avérifications

Héritage des membres

Relation d'héritage

héritage des membres

Liasons statique et dynamique

abstrait et final

Enumerations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Compléments
en POO

Aldric Degorre

Introduction

Généralités

Objets et classes

Types et polymorphisme

Héritage

interne et avérifications

Héritage des membres

Relation d'héritage

héritage des membres

Liasons statique et dynamique

abstrait et final

Enumerations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Énumération = classe particulière

L'exemple précédent pourrait (presque 262) s'écrire :

```
public final class Day extends Enum<Day> {
    public static final Day SUNDAY = new Day("SUNDAY", 0),
        MONDAY = new Day("MONDAY", 1), TUESDAY = new Day("TUESDAY", 2),
        WEDNESDAY = new Day("WEDNESDAY", 3), THURSDAY = new Day("THURSDAY", 4),
        FRIDAY = new Day("FRIDAY", 5), SATURDAY = new Day("SATURDAY", 6);

    private Day(String name, int ordinal) {
        super(name, ordinal);
    }

    // plus méthodes statiques valueOf() et values()
}
```

Énumérations

La base

Compléments
en POO

Aldric Degorre

Introduction

Généralités

Objets et classes

Types et polymorphisme

Héritage

interne et avérifications

Héritage des membres

Relation d'héritage

héritage des membres

Liasons statique et dynamique

abstrait et final

Enumerations

Discussion

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et

Exemple simple :

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

public class Test {
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            switch (d) {
                case SUNDAY:
                case SATURDAY:
                    System.out.println(d + " : sleep");
                    break;
                default:
                    System.out.println(d + " : work");
            }
        }
    }
}
```

Enum<E> est la superclasse directe de toutes les classes déclarées avec un bloc **enum**. Elle contient les fonctionnalités communes à toutes les énumérations.

261. Puisque c'est du sucre syntaxique !

262. En réalité, ceci ne compile pas : javac n'autorise pas le programmeur à étendre la classe **Enum** à la main. Cela est réservé aux vraies **enum**. Si on voulait vraiment toutes les fonctionnalités des **enum**, il faudrait réécrire les méthodes de la classe **Enum**.

Énumérations

Extensions
Compléments en POO

Énumérations
Extensions
Compléments en POO

On peut donc y ajouter des membres, en particulier des méthodes :

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;  
    public boolean isWorkDay() {  
        switch (this) {  
            case SUNDAY:  
            case SATURDAY:  
                return false;  
            default:  
                return true;  
        }  
    }  
    public static void main(String[] args) {  
        for (Day d : Day.values()) {  
            System.out.println(d + ":" + (d.isWorkDay() ? "work" : "sleep"));  
        }  
    }  
}
```

On peut même ajouter des constructeurs (privés seulement). Auquel cas, il faut passer les paramètres du(d'un des) constructeur(s) à chaque constante de l'enum :

```
public enum Day {  
    SUNDAY(false), MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;  
    final boolean isWorkDay;  
    private Day(boolean work) {  
        isWorkDay = work;  
    }  
    private Day() { // constructeur sans paramètre -> on peut aussi déclarer les  
        // constantes d'enum sans paramètre  
        isWorkDay = true;  
    }  
    public static void main(String[] args) {  
        for (Day d : Day.values()) {  
            System.out.println(d + ":" + (d.isWorkDay ? "work" : "sleep"));  
        }  
    }  
}
```

On peut même ajouter des constructeurs (privés seulement). Auquel cas, il faut passer les paramètres du(d'un des) constructeur(s) à chaque constante de l'enum :

```
public enum Day {  
    SUNDAY(false), MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;  
    final boolean isWorkDay;  
    private Day(boolean work) {  
        isWorkDay = work;  
    }  
    private Day() { // constructeur sans paramètre -> on peut aussi déclarer les  
        // constantes d'enum sans paramètre  
        isWorkDay = true;  
    }  
    public static void main(String[] args) {  
        for (Day d : Day.values()) {  
            System.out.println(d + ":" + (d.isWorkDay ? "work" : "sleep"));  
        }  
    }  
}
```

Énumérations

Types énumérés et sous-type
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérit et
avènement
Héritage des membres
Listes statique et
dynamique
abstrait et final.
Énumérations
Discussion
Généricité
Concurrence
Interfaces graphiques
Gestion des erreurs et

Énumérations

Types énumérés et sous-type
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérit et
avènement
Héritage des membres
Listes statique et
dynamique
abstrait et final.
Énumérations
Discussion
Généricité
Concurrence
Interfaces graphiques
Gestion des erreurs et

Énumérations

Types énumérés et sous-type
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérit et
avènement
Héritage des membres
Listes statique et
dynamique
abstrait et final.
Énumérations
Discussion
Généricité
Concurrence
Interfaces graphiques
Gestion des erreurs et

Énumérations

Types énumérés et sous-type
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérit et
avènement
Héritage des membres
Listes statique et
dynamique
abstrait et final.
Énumérations
Discussion
Généricité
Concurrence
Interfaces graphiques
Gestion des erreurs et

Chaque déclaration de constante énumérée peut être suivie d'un corps de classe, afin d'ajouter des membres ou de redéfinir des méthodes juste pour cette constante.

```
public enum Day {  
    SUNDAY { @Override public boolean isWorkDay() { return false; } },  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY { @Override public boolean isWorkDay() { return false; } };  
  
    public boolean isWorkDay() { return true; }  
    public static void main(String[] args) {  
        for (Day d : Day.values()) {  
            System.out.println(d + ":" + (d.isWorkDay() ? "work" : "sleep"));  
        }  
    }  
}
```

Chaque déclaration de constante énumérée peut être suivie d'un corps de classe, afin d'ajouter des membres ou de redéfinir des méthodes juste pour cette constante.

```
public enum Day {  
    SUNDAY { @Override public boolean isWorkDay() { return false; } },  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY { @Override public boolean isWorkDay() { return false; } };  
  
    public boolean isWorkDay() { return true; }  
    public static void main(String[] args) {  
        for (Day d : Day.values()) {  
            System.out.println(d + ":" + (d.isWorkDay() ? "work" : "sleep"));  
        }  
    }  
}
```

Chaque déclaration de constante énumérée peut être suivie d'un corps de classe, afin d'ajouter des membres ou de redéfinir des méthodes juste pour cette constante.

```
public enum Day {  
    SUNDAY { @Override public boolean isWorkDay() { return false; } },  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY { @Override public boolean isWorkDay() { return false; } };  
  
    public boolean isWorkDay() { return true; }  
    public static void main(String[] args) {  
        for (Day d : Day.values()) {  
            System.out.println(d + ":" + (d.isWorkDay() ? "work" : "sleep"));  
        }  
    }  
}
```

Chaque déclaration de constante énumérée peut être suivie d'un corps de classe, afin d'ajouter des membres ou de redéfinir des méthodes juste pour cette constante.

```
public enum Day {  
    SUNDAY { @Override public boolean isWorkDay() { return false; } },  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY { @Override public boolean isWorkDay() { return false; } };  
  
    public boolean isWorkDay() { return true; }  
    public static void main(String[] args) {  
        for (Day d : Day.values()) {  
            System.out.println(d + ":" + (d.isWorkDay() ? "work" : "sleep"));  
        }  
    }  
}
```

Dans ce cas, la constante est l'instance unique d'une sous-classe 263 de l'**enum**.

Remarque : comme d'habitude, toute construction basée sur des **@Override** et la liaison dynamique est à préférer à un **switch** (quand c'est possible et que ça a du sens).

263. Version exacte : l'énumération E étend **Enum<E>**. Voir la généricité.

265. Elles sont même **final** si aucune des constantes énumérées n'est muni d'un corps de classe.

266. Ainsi, toutes les instances d'une **enum** sont connues dès la compilation.

267. On ne peut pas l'étendre « à la main », mais des sous-classes (singletons) sont compilées pour les constantes de l'enum qui sont munies d'un corps de classe.

Toute énumération `E` a les méthodes d'instance suivantes, héritées de la classe `Enum` :

- `int compareTo(E o)` (de l'interface `Comparable`, implémentée par la classe `Enum`) : compare deux éléments de `E` (en fonction de leur ordre de déclaration).
- `String toString()` : retourne le nom de la constante (une chaîne dont le texte est le nom de l'identificateur de la constante d'`enum`)
- `int ordinal()` : retourne le numéro de la constante dans l'ordre de déclaration dans l'`enum`.

Par ailleurs, tout type énuméré `E` dispose des deux méthodes statiques suivantes :

- `static E valueOf(String name)` : retourne la constante d'`enum` dont l'identificateur est égal au contenu de la chaîne `name`
- `static E[] values()` : retourne un tableau contenant les constantes de l'`enum` dans l'ordre dans lequel elles ont été déclarées.

- **Évidemment** : pour implémenter un type fini (cf. intro de ce cours). Remarquez au passage toutes les erreurs potentielles si on utilisait, à la place d'une `enum` :
 - des `int` : tentation d'utiliser directement des littéraux numériques (1, 0, -42) peu parlants au lieu des constantes (par flemme). Risque très fort d'utiliser ainsi des valeurs sans signification associée.
 - des `String` sous forme littérale : risque fort de faire une typo en tappant la chaîne entre guillemets.
- **Cas particulier** : quand une classe ne doit contenir qu'une seule instance (`singleton`) → le plus sûr pour garantir qu'une classe est un singleton c'est d'écrire une enum à 1 élément.

```
enum MaclassesSingleton /* insérer implements Machin */ {
    INSTANCE; // <--- l'instance unique !
    /* insérer ici tous les membres utiles */
}
```

Tout cela peut être fait sans les `enums` mais c'est fastidieux et risque d'être mal fait.

Il existe des implémentations de collections optimisées pour les énumérations.

- `EnumSet<E extends Enum<E>`, qui implémente `Set<E>` : on représente un ensemble de valeurs de l'énumération `E` par un champ de bits (le bit nⁱ vaut 0 si la constante d'`ordinal` i est dans l'ensemble, 1 sinon). Cette représentation est très concise et très rapide.

Création via méthodes statiques

```
Set<DAY> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY), voire
Set<Day> week = EnumSet.allOf(Day.class).
```

L'usage d'`EnumSet` est à préférer à l'usage direct des champs de bits 268 (EJ3 Item 36). On gagne en clarté et en sécurité.

268. Vous savez, ces entiers qu'on manipule bit à bit via les opérateurs <-, >, |, & et ~ et dont les programmeurs en C sont si friands...

Énumérations et collections (2)

Introduction	Compléments en POO Aldric Degorre	Autour de l'héritage Le bon usage de l'héritage de classe (1)
Généralités		
Style		

• **EnumMap<K extends Enum<K>, V>** qui implémente **Map<K, V>** : une **Map** représentée (en interne) par un tableau dont la case d'indice *i* référence la valeur dont la clé est la constante d'ordinal *i* de l'enum **K**.

Construire un **EnumMap** :

```
Map<Day, Activity> edt = new EnumMap<(Day, class)>;
```

EnumMap est à préférer à tout tableau ou toute liste où l'on utiliserait les **ordinaux des constantes** d'une **enum** en tant qu'indices (EJ3 Item 37).

On a coutume de dire que :

- la classe **B** hérite de la classe **A** seulement si un **B est un A**.
- on compose²⁶⁹ **A** dans **B** quand un **B possède un A**.

Mais attention, « est un » peut être interprété de plusieurs façons :

- une instance de **A** peut être utilisée à la place d'**'une instance de B ↔ sous-type'**.
- une instance de **A** est faite comme une instance de **B** ↔ héritage.

269. Il s'agit juste d'utiliser une instance de **A** comme attribut de **B**. On reparle de composition juste après.

270. Mêmes champs en mémoire, appel du constructeur parent, même code sauf si redéfini.

Introduction	Compléments en POO Aldric Degorre	Autour de l'héritage Le bon usage de l'héritage de classe et avertissements
Généralités		
Style		

... ce n'est pas parce qu'on peut le faire que c'est une bonne idée!

- Si la classe **B** hérite de **A**, elle récupère toutes les fonctionnalités héritées de **A**, y compris celles qui n'auraient pas de rapport avec l'objectif de **B**.²⁷²
- (Très utile pour le sous-typage, mais la vraie question est : est-ce mon intention de créer un sous-type ? Cette classe sera-t-elle utilisée dans un contexte polymorphe?)
- Les instances de **B** contiennent tous les champs de **A** (y compris privés), même devenus inutiles → « surpoids » et risque d'incohérences.
- Étendre une classe qui n'était pas conçue pour cela expose à des comportements inattendus²⁷³ (non documentés par son auteur... qui n'avait pas prévu ça!).

272. Et c'est le cas maintenant, évitez les constructions fragiles, comme par exemple, appeler une méthode héritée **f** depuis une méthode redéfinie **g** de **B**. En effet : sauf indication contraire, **f** est susceptible d'appeler **g** → risque de **StackOverflowError**.

273. Cf. cas de la « classe de base fragile » vu précédemment.

Autour de l'héritage

Autour de l'héritage

Alternative : ce qu'on entend par « composition »

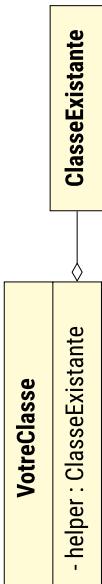
Compléments en POO

Aldric Degoire

Pour des objectifs simples, préférer des techniques alternatives :

- créer du sous-typage → implémentation d'interface 274/275.
- Une interface ne crain pas le syndrôme de la « classe de base fragile ». 276
- réutiliser des fonctionnalités déjà programmées → composition 277 (utiliser un objet auxiliaire possédant les fonctionnalités voulues pour les ajouter à votre classe).

Ici aussi, on ne risque pas de « perturber » des fonctionnalités déjà programmées.



274. Obligatoire et assumé dans des langages comme Rust, où l'héritage ne crée pas de sous-typage.
 275. EJ3 20 : « Prefer interfaces to abstract classes »
 276. Faux en cas de méthodes **default** → même besoin de documentation que pour l'héritage de classe.
 277. EJ3 18 : « Favor composition over inheritance »

Autour de l'héritage

Alternative : ce qu'on entend par « composition »

Compléments en POO

Aldric Degoire

Composition : utilisation d'un objet à l'intérieur d'un autre pour réutiliser les fonctionnalités codées dans le premier. Exemple :

```

class Vendeur {
    private double marge;
    public Vendre(double marge) { this.marge = marge; }
    public vend(Bien b) { return b.prixRevenu() * (1. + marge); }
}

class Boutique {
    private Vendeur vendeur;
    private final List<Bien> stock = new ArrayList<>();
    private double caisse = 0.;

    public Boutique(Vendeur vendeur) { this.vendeur = vendeur; }

    public void vend(Bien b) {
        if (stock.contains(b)) { stock.remove(b); caise += vendeur.vend(b); }
    }
}
  
```

Boutique réutilise des fonctionnalités de **Vendeur** sans en être un sous-type.
 Ces mêmes fonctionnalités pourraient aussi être réutilisées par une autre classe **SuperMarche**.

Objectif : ne pas hériter d'un « bagage » inutile

Exemple problématique

Objectif : ne pas hériter d'un « bagage » inutile

Modélisation à l'aide d'interfaces

Mieux :

```

public interface Rationnel { int getNumer(); int getDenom(); // + operations }

public interface Entier extends Rationnel {
    int getValeur();
    default int getNumer() { return getValeur(); }
    default int getDenom() { return 1; }
}

public final class RationnelImmutable implements Rationnel {
    private final int numerateur, denominateur;
    // + getteurs et opérations
}

public final class EntierImmutable implements Entier {
    public RationnelImmutable(int p, int q) { numerateur = p; denominateur = q; }
    // + getteurs et opérations
}

public final class EntierImmutable @Override public int getValeur() { return intValue; }
  
```

Ainsi nos types existent en version immutable (via les classes) et en version à mutabilité non précisée (via les interfaces), le tout sans trainer de « bagage » inutile.

Autour de l'héritage

Alternative : ce qu'on entend par « composition »

Compléments en POO

Aldric Degoire

Mathématiquement les entiers sont sous-type des rationnels. Mais comment le coder?

Pas terrible :

```

public class Rationnel {
    private final int numerateur, denominateur;
    public Rationnel(int p, int q) { numerateur = p; denominateur = q; }
    // + getteurs et opérations
}

public class Entier extends Rationnel { public Entier (int n) { super(n, 1); } }
  
```

Ici, toute instance d'entiers contient 2 champs (certes non visibles) : numérateur et dénominateur. Or 1 seul **int** aurait dû suffire.

- utilisation trop importante de mémoire (pas très grave)
- risque d'incohérence à cause de la redondance (plus grave)

Autour de l'héritage

Alternative : ce qu'on entend par « composition »

Compléments en POO

Aldric Degoire

Autre problème : la classe **Rationnel** visant à être immutable (attributs **final**) serait typiquement **final** (pour empêcher des sous classes avec attributs modifiables).

Objectif : ne pas laisser l'héritage casser l'encapsulation

Objectif : ne pas laisser l'héritage casser l'encapsulation

Solutions (1)

```
/** RéelPositif : représente un réel positif modifiable.  
 * Contrat : getValeur et racine retournent toujours un réel positif.  
 */  
class ReelPositif {  
    double valeur;  
    public ReelPositif(double valeur) { setValeur(valeur); }  
    public ReelPositif() { setValeur(0); } // on veut retourner 0  
    public void setValeur(double valeur) { if (valeur < 0) throw new IllegalArgumentException(); valeur = valeur; }  
    public double racine() { return Math.sqrt(valeur); }  
  
    class ReelPositifArrondi extends ReelPositif {  
        double valeur2; // et hop, on remplace l'attribut de la superclasse  
        public ReelPositifArrondi(double valeur) { this(valeur); }  
        public void getValeur() { return valeur2; }  
        public void setValeur(double valeur) { this.valeur2 = Math.floor(valeur); }  
    }  
}
```

Ici, **racine** peut toujours retourner NaN. Pourquoi ?

Point fort : on restreint le strict nécessaire pour assurer le contrat.

Point faible : il faut réfléchir, sinon on a vite fait de manquer une faille.

Compléments en POO

Aldric Degorre

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérité et avertissements
Héritage des membres
Héritage des méthodes
Listes statique et dynamique
abstrait et final
Interfaces graphiques
Gestion des erreurs et

Discussion
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et

Objectif : ne pas laisser l'héritage casser l'encapsulation

Compléments en POO

Aldric Degorre

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérité et avertissements
Héritage des membres
Héritage des méthodes
Listes statique et dynamique
abstrait et final
Interfaces graphiques
Gestion des erreurs et

Discussion
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et

Exemple problématique

Objectif : ne pas laisser l'héritage casser l'encapsulation

Solutions (2)

```
/** RéelPositif : représente un réel positif modifiable.  
 * Contrat : getValeur et racine retournent toujours un réel positif.  
 */  
class ReelPositif {  
    public static void main(String[] args) {  
        ReelPositif x = new ReelArrondi(- Math.PI);  
        System.out.println(x.racine()); // affiche "NaN"  
    }  
  
    public class Test {  
        public void setValeur(double valeur) { this.valeur = Math.floor(valeur); }  
    }  
}
```

Point fort : la composition (on perd le sous-type) :

Point faible : on ne peut pas créer de sous-classe ReelPositifArrondi, mais on peut contourner grâce à la composition (on perd le sous-type) :

```
class ReelPositifArrondi {  
    private ReelPositif valeur;  
    public ReelPositifArrondi(double valeur) { this.valeur = new ReelPositif(Math.floor(valeur)); }  
    public void getValeur() { return valeur.getValeur(); }  
    public void setValeur(double valeur) { this.valeur.setValeur(Math.floor(valeur)); }  
}
```

Compléments en POO

Aldric Degorre

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérité et avertissements
Héritage des membres
Héritage des méthodes
Listes statique et dynamique
abstrait et final
Interfaces graphiques
Gestion des erreurs et

Discussion
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et

Objectif : ne pas laisser l'héritage casser l'encapsulation

Le patron décorateur (1)

Solutions (1)

```
class ReelPositifArrondi extends ReelPositif {  
    double valeur2; // et hop, on remplace l'attribut de la superclasse  
    public ReelPositifArrondi(double valeur) { this(valeur); }  
    public void getValeur() { return valeur2; }  
    public void setValeur(double valeur) { this.valeur2 = Math.floor(valeur); }  
}
```

Point fort : rendre valeur privé et et passer getValeur en final. Cette solution garantit que le contrat sera respecté par toute classe dérivée.

Point fort : on restreint le strict nécessaire pour assurer le contrat.

Point faible : il faut réfléchir, sinon on a vite fait de manquer une faille.

Compléments en POO

Aldric Degorre

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérité et avertissements
Héritage des membres
Héritage des méthodes
Listes statique et dynamique
abstrait et final
Interfaces graphiques
Gestion des erreurs et

Discussion
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et

Objectif : ne pas laisser l'héritage casser l'encapsulation

Le patron décorateur (1)

Solutions (1)

```
interface Nombre {  
    double getValeur();  
    void setValeur(double valeur);  
}  
  
final class Real implements Nombre {  
    private double valeur;  
    public Real(double valeur) { this.valeur = valeur; }  
    @Override public double getValeur() { return valeur; }  
    @Override public void setValeur(double valeur) { this.valeur = valeur; }  
}  
  
final class Arrondi implements Nombre {  
    private final Nombre valeur;  
    public Arrondi(Nombre valeur) { this.valeur = valeur; }  
    @Override public double getValeur() { return Math.abs(valeur.getValeur()); }  
    @Override public void setValeur(double valeur) { this.valeur.setValeur(valeur); }  
}  
  
final class Positif implements Nombre {  
    private final Nombre valeur;  
    public Positif(Nombre valeur) { this.valeur = valeur; }  
    @Override public double getValeur() { return Math.abs(valeur.getValeur()); }  
    @Override public void setValeur(double valeur) { this.valeur.setValeur(valeur); }  
}
```

Point fort : la sûre, simple mais rigide : valeur → private, ReelPositif → final.

Point fort : sans faille et très facile

Point faible : on ne peut pas créer de sous-classe ReelPositifArrondi, mais on peut contourner grâce à la composition (on perd le sous-type) :

Compléments en POO

Aldric Degorre

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérité et avertissements
Héritage des membres
Héritage des méthodes
Listes statique et dynamique
abstrait et final
Interfaces graphiques
Gestion des erreurs et

Discussion
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et

Objectif : ne pas laisser l'héritage casser l'encapsulation

Le patron décorateur (1)

Solutions (1)

```
class ReelPositifArrondi {  
    private ReelPositif valeur;  
    public ReelPositifArrondi(double valeur) { this.valeur = new ReelPositif(Math.floor(valeur)); }  
    public void getValeur() { return valeur.getValeur(); }  
    public void setValeur(double valeur) { this.valeur.setValeur(Math.floor(valeur)); }  
}
```

Pour retrouver le polymorphisme : écrire une interface commune à implémenter (argument supplémentaire pour toujours programmer à l'interface).

→ on a alors mis en œuvre le patron de conception « décorateur » (GoF).

Compléments en POO

Aldric Degorre

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Inérité et avertissements
Héritage des membres
Héritage des méthodes
Listes statique et dynamique
abstrait et final
Interfaces graphiques
Gestion des erreurs et

Discussion
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et

Objectif : ne pas laisser l'héritage casser l'encapsulation

Comment « hériter » de plusieurs classes ?

Compléments en P00

Compléments en P00

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avertissements

Rélation d'héritage

Héritage des membres

Héritage des méthodes

Liasons statique et dynamique

abstrait et final

Enumerations

Discussion

Généricité

Concurrency

Interfaces graphiques

Gestion des erreurs et

Discussion

Généricité

Concurrency

Interfaces graphiques

Gestion des erreurs et

Discussion

- Le patron décorateur permet, via la composition, d'ajouter/modifier plusieurs fois du comportement en réutilisant plusieurs classes existantes.

- Mais, ce patron est limité à créer des objets d'interface constante²⁷⁸.

- Pour obtenir à la fois le bénéfice de la réutilisation d'implémentation et d'un type enrichi (plus de méthodes), il faut s'y prendre autrement.

- Le besoin décrit serait pourvu si la clause **extends** admettait plusieurs superclasses. Malheureusement, Java ne permet pas l'héritage multiple.

- À la place, il faut donc « bricoler » avec la composition et l'implémentation d'interfaces → patron déléguéation²⁷⁹.



Dans l'exemple, les décorateurs sont les classes **Positif** et **Arrondi**. Pour obtenir un réel positif arrondi, on écrit juste : **new Arrondi(new Positif(new Reel(42)))**. On n'a pas eu besoin de créer la classe **Reel**!
Pour obtenir un **Nombre** arrondi, on écrit : **new Arrondi(new Nombre)**.

Objectif : simuler un héritage multiple

Patron « délégation » en UML

Compléments en P00

Compléments en P00

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Intérêt et avertissements

Rélation d'héritage

Héritage des membres

Héritage des méthodes

Liasons statique et dynamique

abstrait et final

Enumerations

Discussion

Généricité

Concurrency

Interfaces graphiques

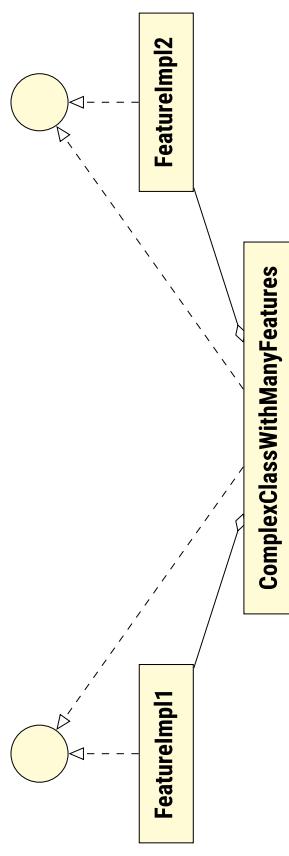
Gestion des erreurs et

Discussion

- Diagramme à 2 interfaces déléguées (mais ça pourrait être 1, 3 ou autant qu'on veut) :

HasFeature1

HasFeature2



Ce qu'on ne voit pas sur le diagramme : les implémentations dans **ComplexClassWithManyFeatures** des méthodes de **HasFeature1** et **HasFeature2** ne comportent qu'un simple appel vers la méthode de **FeatureImpl1** de même nom.

Si les classes auxiliaires sont fournies par un tiers et n'implémentent pas d'interface, on peut créer les interfaces manquantes et les implémenter dans **PossedePropAetB**.²⁸¹

Supposées moins triviales que dans l'exemple (sinon c'est le marteau-pilon pour écraser une mouche!).
On revient au patron adaptateur déjà introduit dans ce cours.

On peut alors écrire une classe ayant les 2 propriétés de la façon suivante :

```

class PossedePropAetB implements AvecPropA, AvecPropB {
    PossedePropA aProxy; PossedePropB bProxy;
    void setA(int newA) { aProxy.setA(newA); }
    int getA() { return aProxy.getA(); }
    void setB(int newB) { bProxy.setB(newB); }
    int getB() { return bProxy.getB(); }
}
  
```

Ce qu'on ne voit pas sur le diagramme : les implémentations dans **ComplexClassWithManyFeatures** des méthodes de **HasFeature1** et **HasFeature2** ne comportent qu'un simple appel vers la méthode de **FeatureImpl1** de même nom.

Si les classes auxiliaires sont fournies par un tiers et n'implémentent pas d'interface, on peut créer les interfaces manquantes et les implémenter dans **PossedePropAetB**.²⁸¹

Supposées moins triviales que dans l'exemple (sinon c'est le marteau-pilon pour écraser une mouche!).
On revient au patron adaptateur déjà introduit dans ce cours.

Généricité

Pourquoi ? (1)

Généricité

Compléments en POO

Aldric Degorre

Exemple (de l'API) :

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
    public boolean equals(Object o);
```

→ Que veut dire ce « <T> » ?

→ **Comparator** est une interface **générique** : un type paramétrable par un autre type.²⁸²

Types génériques de l'API :

- Les collections de Java ≥ 5 (interfaces et classes génériques).

Ce fait seul suffit à justifier l'intérêt des génériques et leur introduction dans Java 5.

- Les interfaces fonctionnelles de Java ≥ 8 (pour les lambda expressions).

- **Optional**, **Stream**, **Future**, **CompletableFuture**, **ForkJoinTask**, ...

282. Ou « constructeur de type ». Mais cette terminologie est rarement utilisée en Java.

La généricité est un procédé permettant d'augmenter la réutilisabilité du code de façon maîtrisée²⁸³ grâce à des relations fines entre les types utilisés.

Sur un exemple :

```
class Boite { // non polymorphe
    public int x;
    void echange(Boite autre) {
        int ech = x; x = autre.x; autre.x = ech;
    }
}
```

Inconvénient : définition qui ne marche que pour les boîtes à entiers.

Réutilisabilité : proche de zéro !

283. Par opposition au polymorphisme par sous-typage, où, par exemple, pour les arguments d'appel de méthode, tout sous-type fait l'affaire indépendamment des autres types utilisés en argument.

Généricité

Pourquoi ? (3)

Généricité

Compléments en POO

Aldric Degorre

Cas d'utilisation problématique :

```
Boite b1 = new Boite();
Boite b2 = new Boite();
System.out.println(7 * (Integer) b1.x); // <- là c'est ok
b1.echange(b2);
System.out.println(7 * (Integer) b1.x); // <- ClassCastException !
```

En fait on aurait dû tester le type à l'exécution :

```
if (b.x instanceof Integer) System.out.println(7 * (Integer) b.x);
```

... mais on préfèrerait vraiment que le code soit garanti par le compilateur²⁸⁵.

285. Remarque : dans cet exemple, probablement l'IDE (à défaut de javac) signalera que la conversion est hasardeuse.

Normalement, on aura donc pensé à mettre **instanceof**. Il n'en reste pas moins que c'est un test à l'exécution qu'on aimeraient éviter (en plus d'être une lourdeur à l'écriture du programme).

Généricité

Pourquoi ? (1)

Généricité

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Collections

Opérations

Lambda-expressions

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Généricité

Pourquoi ? (1)

Généricité

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Collections

Opérations

Lambda-expressions

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

La bonne solution : hôte générique (\rightarrow polymorphisme générique)

```
class Boîte<C> {
    public C x;
    void échange(Boîte<C> autre) { C ech = x; x = autre.x; autre.x = ech; }

    ... // plus loin :
        Boîte<Integer> b1 = new Boîte<String>(); Boîte<String> b2 = new Boîte<x>();

        b1.x = 6; b2.x = "toto";
        System.out.println(7 * b1.x); // <- là c'est toujours ok (et sans cast, SVP !)
        // b1échange(b2); // <- ici erreur à la compilation ! (ouf !)
        System.out.println(7 * b1.x);
}
```

La généricité consiste à introduire des types dépendants d'un paramètre de type.

La concrétisation du paramètre est vérifiée dès de la compilation²⁸⁶ et uniquement à la compilation. Celle-ci est oubliée aussitôt²⁸⁷ (**effacement de type / type erasure**).

286. Or le plus tôt on détecte une erreur, le mieux c'est!

287. Conséquence : les objets de classe générique ne savent pas avec quel paramètre ils ont été instanciés.

- **A l'usage, le type générique sert de constructeur de type : on remplace le paramètre par un type concret et on obtient un type paramétré.**

Exemple : `List<String>` un des types paramétrés que `List` permet de construire.

- Le type concret substituant le paramètre doit être un type référence : `Triplet<String, String, Integer>` est interdit²⁸⁹ !

289. Pour l'instant. Il semble qu'il soit prévu de permettre cela dans une prochaine version de Java.

```
class Boîte<C> {
    public C x;
    void échange(Boîte<C> autre) { C ech = x; x = autre.x; autre.x = ech; }

    ... // plus loin :
        Boîte<Integer> b1 = new Boîte<String>(); Boîte<String> b2 = new Boîte<x>();

        b1.x = 6; b2.x = "toto";
        System.out.println(7 * b1.x); // <- là c'est toujours ok (et sans cast, SVP !)
        // b1échange(b2); // <- ici erreur à la compilation ! (ouf !)
        System.out.println(7 * b1.x);
}
```

La généricité consiste à introduire des types dépendants d'un paramètre de type.

La concrétisation du paramètre est vérifiée dès de la compilation²⁸⁶ et uniquement à la compilation. Celle-ci est oubliée aussitôt²⁸⁷ (**effacement de type / type erasure**).

286. Or le plus tôt on détecte une erreur, le mieux c'est!

287. Conséquence : les objets de classe générique ne savent pas avec quel paramètre ils ont été instanciés.

- **Utilisation de classe générique par instantiation directe :**

```
// à partir de Java 5 :
new Triplet<String, String, Integer> t1 =
    new Triplet<String, String, Integer>("Marcel", "Durand", 23);

// à partir de Java 7 :
Triplet<String, String, Integer> t2 = new Triplet<>("Marcel", "Durand", 23);

// à partir de Java 10 (si t3 est une variable locale) :
var t3 = new Triplet<String, String, Integer>("Marcel", "Durand", 23);
```

Le type de `t1`, `t2` et `t3` est le type paramétré `Triplet<String, String, Integer>`.

289. Pour l'instant. Il semble qu'il soit prévu de permettre cela dans une prochaine version de Java.

Types génériques et types paramétrés

Parallèle entre type générique et méthode

Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques	Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques
Usage de base (4)					
<p>Utilisation de classe générique par <u>extension</u> non générique (spécialisation) :</p> <pre>class TroisChars extends Triplet<Char, Char, Char> { public TroisChars(Char x, Char y, Char z) { super(x,y,z); }</pre> <p>TroisChars étend la classe paramétrée Triplet<Char, Char, Char>.</p> <p>Autre cas (spécialisation partielle) :</p> <pre>class DeuxCharsEtAutre<T> extends Triplet<Char, Char, T> { public DeuxCharsEtAutre(Char x, Char y, T z) { super(x,y,z); }</pre> <p>La classe générique DeuxCharsEtAutre<T> étend la classe générique partiellement paramétrée Triplet<Char, Char, T>.</p>		<p>La déclaration et l'utilisation des types génériques rappellent celles des méthodes.</p> <p>Similitudes :</p> <ul style="list-style-type: none"> introduction des paramètres (de type ou de valeur) dans l'en-tête de la déclaration; utilisation des noms des paramètres dans le corps de la déclaration seulement; pour utiliser le type générique ou appeler la méthode, on passe des <u>concrétisations des paramètres</u>. <p>Principales différences :</p> <ul style="list-style-type: none"> Les paramètres des génériques représentent des <u>types</u> alors que ceux des méthodes représentent des <u>valeurs</u>. Pour les paramètres de type, le « remplacement »²⁹⁰ a lieu à la compilation. Pour les paramètres des méthodes, remplacement par une valeur à l'exécution. <p>290. Rappel : remplacement oublié, effacé, aussitôt que la vérification du bon typage a été faite.</p>			

Types génériques et types paramétrés

Parallèle entre type générique et méthode

Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques	Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques
Usage de base (4)					
<p>Utilisation de classe générique par <u>extension</u> non générique (spécialisation) :</p> <pre>class TroisChars extends Triplet<Char, Char, Char> { public TroisChars(Char x, Char y, Char z) { super(x,y,z); }</pre> <p>TroisChars étend la classe paramétrée Triplet<Char, Char, Char>.</p> <p>Autre cas (spécialisation partielle) :</p> <pre>class DeuxCharsEtAutre<T> extends Triplet<Char, Char, T> { public DeuxCharsEtAutre(Char x, Char y, T z) { super(x,y,z); }</pre> <p>La classe générique DeuxCharsEtAutre<T> étend la classe générique partiellement paramétrée Triplet<Char, Char, T>.</p>		<p>La déclaration et l'utilisation des types génériques rappellent celles des méthodes.</p> <p>Similitudes :</p> <ul style="list-style-type: none"> introduction des paramètres (de type ou de valeur) dans l'en-tête de la déclaration; utilisation des noms des paramètres dans le corps de la déclaration seulement; pour utiliser le type générique ou appeler la méthode, on passe des <u>concrétisations des paramètres</u>. <p>Principales différences :</p> <ul style="list-style-type: none"> Les paramètres des génériques représentent des <u>types</u> alors que ceux des méthodes représentent des <u>valeurs</u>. Pour les paramètres de type, le « remplacement »²⁹⁰ a lieu à la compilation. Pour les paramètres des méthodes, remplacement par une valeur à l'exécution. <p>290. Rappel : remplacement oublié, effacé, aussitôt que la vérification du bon typage a été faite.</p>			

Types bruts

Parallèle entre type générique et méthode

Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques	Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques
Usage de base (4)					
<p>Un nom de type générique seul, sans paramètre (comme « Triplet »), est aussi un type légal, appelé un type brut (raw type).</p> <p>Son utilisation est fortement déconseillée, mais elle est permise pour assurer la compatibilité ascendante²⁹¹.</p> <p>Un type brut est <u>super-type direct</u>²⁹² de tout type paramétré correspondant (ex : Triplet est supertype direct de Triplet<Number, Object, String>).</p> <p>Pour faciliter l'écriture, le downcast implicite²⁹³ est malgré tout possible :</p> <pre>List l1 = new ArrayList(); // déclaration de l1 avec raw type List<Integer> l2 = l1; // downcast implicite de l1 vers type paramétré</pre> <p>compile avec l'avertissement unchecked conversion sur la douzième ligne.</p>		<p>Il est aussi possible d'introduire un paramètre de type <u>dans la signature d'une méthode</u> (dans une classe pas forcément générique) :</p> <pre>static <E> List<E> inverseListe(List<E> l) { ... ; E x = get(0); ... ; }</pre> <p>Dans l'exemple ci-dessus, on garantit que la liste renournée par inverseListe() a le même type d'éléments que celle donnée en paramètre.</p> <p>Usages possibles :</p> <ul style="list-style-type: none"> contraindre plusieurs types apparaissant dans la signature de la méthode à être les mêmes... sans pour autant dire quel type concret ce sera! introduire localement un nom de type utilisable dans le corps de la méthode (type non défini, mais dont les contraintes sont connues, ex : type intersection, voir plus loin). <p>Remarque : il est donc, malgré tout²⁹⁴, possible d'écrire une méthode statique générique.</p>			

Compléments en POO

Parallèle entre type générique et méthode

Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques	Compléments en POO	Aldric Degorre	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Collections Optimale Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards Concurrence Interfaces graphiques
Usage de base (4)					
<p>Un source Java < 5 compile avec javac ≥ 5. Or certains types sont devenus génériques entre temps.</p> <p>C'est une des règles de sous-type relatives aux génériques, omises dans le début de ce cours.</p> <p>Je crois que c'est l'unique occurrence de downcast implicite en Java.</p>		<p>291. Un source Java < 5 compile avec javac ≥ 5. Or certains types sont devenus génériques entre temps.</p> <p>292. C'est une des règles de sous-type relatives aux génériques, omises dans le début de ce cours.</p> <p>293. Je crois que c'est l'unique occurrence de downcast implicite en Java.</p>			

Bornes de paramètres de type (1)

Bornes de paramètres de type (2)

Bornes de paramètres de type (3)

Bornes de paramètres de type (2)

Bornes de paramètres de type (3)

Compléments en POO	Aldric Degorre
Introduction	
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité :	
généricité : introduction	
Collections	
Optimisation	
Lambda-expressions	
Les "Taramas"	
Wildcards	
Concurrence	
Interfaces graphiques	

- Pour limiter les concrétisations autorisées, un paramètre de type admet des **bornes supérieures**²⁹⁵ (se comportant comme supertypes du paramètre) :

```
class Calculator<Data extends Number>
```

Ici, **Data** devra être concrétisé par un sous-type de **Number** : une instance de **Calculator** travaillera sur nécessairement avec un certain sous-type de **Number**, celui choisi à son instantiation.

295. On verra dans la suite que les bornes inférieures existent aussi, mais elles ne s'appliquent qu'aux wildcards (et non aux paramètres de type).

Compléments en POO	Aldric Degorre
Introduction	
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité :	
généricité : introduction	
Collections	
Optimisation	
Lambda-expressions	
Les "Taramas"	
Wildcards	
Concurrence	
Interfaces graphiques	

- Pour définir des bornes supérieures multiples (p. ex. pour implémenter de multiples interfaces), les supertypes sont séparés par le symbole « & » :

```
class RunWithPriorityList<T extends Comparable<T> & Runnable> implements List<T>
```

« **Comparable**<**T**> & **Runnable** » est un **type intersection**²⁹⁶, il est sous-type direct de **Comparable**<**T**> et de **Runnable**.

Ainsi, **T** est sous-type de l'intersection (et donc de de **Comparable**<**T**> et de **Runnable**).

296. Remarque : c'est le seul contexte où on peut écrire un type intersection (type non dénotable). Ainsi, il n'est pas possible de déclarer explicitement une variable de type intersection. Implicitement, à l'aide d'une méthode générique et du mot-clé **var**, cela est cependant possible : **public static** <**T** extends **A** & **B** > **T** intersectionFactory(...) { ... }

plus loin :

var **x** = **intersectionFactory**(...); // **x** est de type **A** & **B**

La technique assez « tirée par les cheveux » et d'utilité toute relative....

Collections : pourquoi ?

Compléments en POO	Aldric Degorre
Introduction	
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité :	
généricité : introduction	
Collections	
Optimisation	
Lambda-expressions	
Les "Taramas"	
Effacement des génériques vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

Bornes de paramètres de type (3)

Compléments en POO	Aldric Degorre
Introduction	
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité :	
généricité : introduction	
Collections	
Optimisation	
Lambda-expressions	
Les "Taramas"	
Effacement des génériques vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

On peut prolonger l'analogie avec les méthodes et leurs paramètres : en effet, les paramètres des méthodes sont eux-mêmes « bornés » par les types déclarés dans la signature.

- Besoin** : représenter des « paquets », des « collections » d'objets similaires.
- Plusieurs genres de paquets/collections** : avec ou sans doublon, accès séquentiel ou aléatoire (= par indice), avec ou sans ordre, etc.
- Mais nombreux points communs** : peuvent contenir plusieurs éléments, possibilité d'itérer, de tester l'appartenance, l'inclusion etc.
- Pour chaque « genre » plusieurs représentations/implémentations de la structure (optimisant telle ou telle opération...).

Collections : pourquoi ?

La hiérarchie des sous-interfaces d'Iterable

Interfaces de java.util et java.util.concurrent²⁹⁹

Compléments en POO

Aldric Degorre

Les collections génériques, introduites dans Java SE 5, remplacent avantagusement :

- Les tableaux ²⁹⁷.
- Les collections non génériques ²⁹⁸ de Java < 5, avec leurs éléments de type statique **Object**, qu'il fallait caster avant usage.

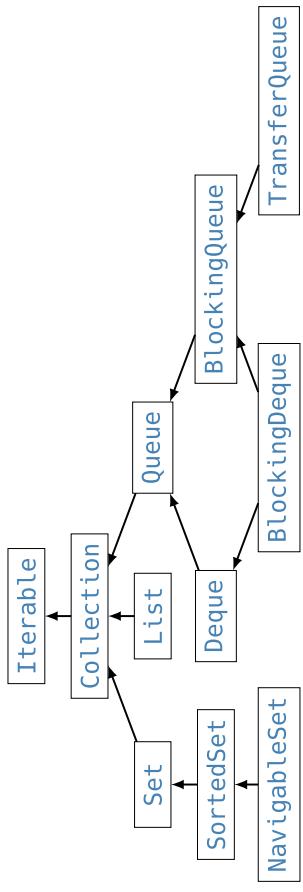
Ex. : classe **Vector** (listes implémentées par tableaux dynamiques synchronisés).

Les collections justifient à elles seules l'introduction de la généricité dans Java.

297. Qui gardent quelques avantages : syntaxe pratique, efficacité et disposent déjà d'un « genre de générique » (un **String[]** contiendra des éléments **String** et rien d'autre), dont nous reparlerons.

298. **NB** : les anciennes collections ont été transformées en types génériques (**Vector**<E> au lieu de **Vector**) implémentant l'interface **Collection<E>**. Les types sans paramètre sont désormais considérés comme des types bruts et sont à éviter.

Si vous migrez du code Java < 5 vers Java ≥ 5, remplacez **ArrayList<TypeElements>** par **ArrayList** par [ArrayList](#).



299. Autres sous-interfaces dans [java.nio.file](#) et [java.beans.beancontext](#).

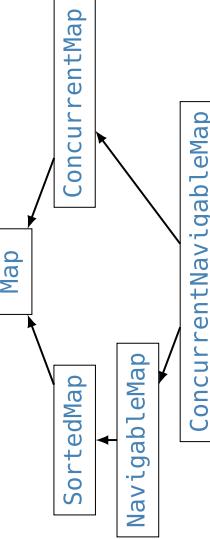
La hiérarchie des sous-interfaces de Map

L'interface Iterable (1)

Interfaces de java.util et java.util.concurrent³⁰⁰

Compléments en POO

Aldric Degorre



Chacune de ces interfaces possède une ou plusieurs implementations.

300. Autres dans [javax.xml.script](#), [javax.xml.ws.handler](#) et [javax.xml.ws.handler.soap](#).

La hiérarchie des sous-interfaces d'Iterable

Interfaces de java.util et java.util.concurrent²⁹⁹

Compléments en POO

Aldric Degorre

Les collections génériques, introduites dans Java SE 5, remplacent avantagusement :

- Les tableaux ²⁹⁷.
- Les collections non génériques ²⁹⁸ de Java < 5, avec leurs éléments de type statique **Object**, qu'il fallait caster avant usage.

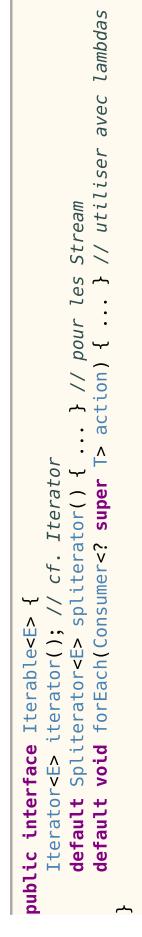
Ex. : classe **Vector** (listes implémentées par tableaux dynamiques synchronisés).

Les collections justifient à elles seules l'introduction de la généricité dans Java.

297. Qui gardent quelques avantages : syntaxe pratique, efficacité et disposent déjà d'un « genre de générique » (un **String[]** contiendra des éléments **String** et rien d'autre), dont nous reparlerons.

298. **NB** : les anciennes collections ont été transformées en types génériques (**Vector**<E> au lieu de **Vector**) implémentant l'interface **Collection<E>**. Les types sans paramètre sont désormais considérés comme des types bruts et sont à éviter.

Si vous migrez du code Java < 5 vers Java ≥ 5, remplacez **ArrayList<TypeElements>** par **ArrayList** par [ArrayList](#).



Chacune de ces interfaces possède une ou plusieurs implementations.

- soit avec la construction **for-each** (conseillé) :

for (Object o : monIterable) System.out.println(o);

- soit avec la méthode **forEach** et une lambda-expression (cf. chapitre dédié) :

monIterable.forEach(System.out::println);

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Optimisation	Lambda-expressions	Effacement de type	Invariance des génériques vs covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques	Compléments en POO

- soit en utilisant explicitement l'**itérateur** (rare, mais utile pour accès en écriture) :

```
Iterator<String> it = monIterable.iterator();
while (it.hasNext()) {
    String s = it.next();
    if (s.equals("À enlever")) it.remove();
    else System.out.println("On garde:" + s);
}
```

Remarque : la construction `for-each` et la méthode `forEach` ne permettent qu'un parcours en lecture seule.

- soit en réduisant un **Stream** (cf. chapitre dédié) basé sur cet **Iterable** 301 :

```
macollection.stream()
    .filter(x -> !x.equals("À enlever"))
    .forEach(System.out::println);
```

Les paramètres des méthodes de **Stream** sont typiquement des **lambda-expressions**.

301. En réalité, pour des raisons assez obscures, la méthode `stream` n'existe que dans la sous-interface **Collection**. Mais il est facile de programmer une méthode équivalente pour **Iterable**.

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Optimisation	Lambda-expressions	Effacement de type	Invariance des génériques vs covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques	Compléments en POO

- soit en utilisant explicitement l'**itérateur** (rare, mais utile pour accès en écriture) :

Un itérateur :

- sert à parcourir un itérable et est habituellement utilisé implicitement;
- s'instancie en appelant la méthode `iterator` sur l'objet à parcourir;
- est un objet respectant l'interface suivante :

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // opération optionnelle
}
```

remove, si implémentée, permet de supprimer un élément en cours de parcours sans provoquer **ConcurrentModificationException** (au contraire des méthodes de l'itérable). Cette possibilité justifie de créer une variable pour manipuler explicitement l'itérateur (sinon, on préfère `for-each`).

- Pas de méthodes autres que celles héritées de **Collection**.

- Déférence : le contrat de **Set** garantit l'**unicité** de ses éléments (pas de doublon).

Exemple :

```
Set<Integer> s = new HashSet<Integer>();
s.add(1); s.add(2); s.add(3); s.add(1);
for (int i : s) System.out.print(i + " ");
```

Ceci affichera : 1, 2, 3,
La classe **HashSet** est une des implementations de **Set** fournies par Java. C'est celle que vous utiliserez le plus souvent.

Unicité ? un élément **x** est unique si pour tout autre élément **y**, **x.equals(y)** retourne **false**.
⇒ importance d'avoir une redéfinition correcte de `equals()`.

L'API ne fournit pas d'implémentation directe de **Collection**, mais plutôt des collections spécialisées, décrites dans la suite.

Comme Set, mais les éléments sont triés.

... ce qui permet d'avoir quelques méthodes en plus.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Implémentation typique : classe TreeSet.

List : c'est une Collection ordonnée avec possibilité de doublons. C'est ce qu'on utilise le plus souvent. Permet d'abstraire les notions de tableau et de liste chainée.

Fonctionnalités principales :

- acès positionnel (on peut accéder au *i*ème élément)
 - recherche (si on connaît un élément, on peut demander sa position)
- ```
public interface List<E> extends Collection<E> {
 // Positional access
 E get(int index);
 E set(int index, E element); //optional
 boolean add(E element); //optional
 void add(int index, E element); //optional
 E remove(int index); //optional
 boolean addAll(List<int> index,
 Collection<? extends E> c); //optional

 // Search
 int indexOf(Object o);
 int lastIndexOf(Object o);
 ...
}
```

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
généricité :  
introduction  
Collections  
Optimale  
Lambda-expressions  
Les "Streams"  
Effacement de type  
Invariance des génériques vs.  
covariance des tableaux  
Wildcards

Concurrence  
Interfaces graphiques

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
généricité :  
introduction  
Collections  
Optimale  
Lambda-expressions  
Les "Streams"  
Effacement de type  
Invariance des génériques vs.  
covariance des tableaux  
Wildcards

Concurrence  
Interfaces graphiques

**Mais aussi :**

- itérateurs plus riches (peuvent itérer en arrière)
- « vues » de sous-listes<sup>302</sup>

**Un itérateur de liste sert à parcourir une liste. Il fait la même chose qu'un itérateur, mais aussi quelques autres opérations, comme :**

- parcourir à l'envers
  - ajouter/modifier des éléments en passant
  - un itérateur de liste est un objet respectant l'interface suivante :
- ```
public interface ListIterator<E> extends Iterator<E> {
    void add(E e);
    boolean hasPrevious();
    int nextIndex();
    E previous();
    int previousIndex();
    void set(E e);
}
```

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
généricité :
introduction
Collections
Optimale
Lambda-expressions
Les "Streams"
Effacement de type
Invariance des génériques vs.
covariance des tableaux
Wildcards

Concurrence
Interfaces graphiques

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
généricité :
introduction
Collections
Optimale
Lambda-expressions
Les "Streams"
Effacement de type
Invariance des génériques vs.
covariance des tableaux
Wildcards

Concurrence
Interfaces graphiques

302. Vue d'un objet **o** : **objet v dominant accès à une partie des données de o** sans en être une copie (partielle), les modifications des 2 objets restent liées.

Implémentations principales : `ArrayList` (basée sur un tableau, avec redimensionnement dynamique), `LinkedList` (basée sur liste chaînée).

Exemple :

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(1); l.add(2); l.add(3); l.add(1);
for (int i : l) System.out.print(i + " ");
System.out.println("n3e élément:" + l.get(2));
l.set(2, 9);
System.out.println("Nouveau 3e élément:" + l.get(2));
```

Ceci affichera :

```
1, 2, 3, 1
3e élément: 3
Nouveau 3e élément: 9
```

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

généricité :

introduction

Collections

Optimisation

Langage expressions

Effacement de type

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Aldric Degorre

Une `Queue` représente typiquement une collection d'éléments en attente de traitement (typiquement FIFO : `first in, first out`).

Opérations de base : insertion, suppression et inspection.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Exemple : la classe `PriorityQueue` présente ses éléments selon l'ordre naturel de ses éléments (ou un autre ordre si spécifié).

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

généricité :

introduction

Collections

Optimisation

Langage expressions

Effacement de type

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Aldric Degorre

Deque = « *double ended queue* ».

C'est comme une `Queue`, mais enrichie afin d'accéder à la collection aussi bien par le début que par la fin.

Le même Deque peut ainsi aussi bien servir de structure FIFO que LIFO (last in, first out).

```
public interface Queue<E> extends Collection<E> {
    boolean addFirst(E e);
    boolean addLast(E e);
    Iterator<E> descendingIterator();
    E getFirst();
    E getLast();
    boolean offerFirst(E e);
    boolean offerLast(E e);
    E peekFirst();
    E peekLast();
    E removeFirst();
    E removeLast();
}
```

Implémentations typiques : `ArrayDeque`, `LinkedList`

Une Map est un ensemble d'associations (clé → valeur), où chaque clé ne peut être associée qu'à une seule valeur.

Nombreuses méthodes communes avec l'interface Collection, mais particularités.

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    ...
}
```

Introduction	Généralités
Aldric Degorre	Style
Objets et classes	Types et polymorphisme
Héritage	Généricité
généricité : introduction	introduction
Collections	Optimisation
Optimisation	Lambda-expressions
Lambda-expressions	Effacement de type
les "Streams"	Invariance des génériques vs. covariance des tableaux
Effacement de type	Wildcards
Invariance des génériques vs. covariance des tableaux	Concurrence
Wildcards	Interfaces graphiques

```
... // Bulk operations
void putAll(Map<? extends K, ? extends V> m);
void clear();
// Collection View
public Set<K> keySet();
public Set<Map.Entry<K,V> entrySet();
// Interface for entrySet elements
public interface Entry {
    K getKey();
    V getValue();
    V setValue(V value);
}
```

Implémentation la plus courante : la classe `HashMap`

Introduction	Généralités
Aldric Degorre	Style
Objets et classes	Types et polymorphisme
Héritage	Généricité
généricité : introduction	introduction
Collections	Optimisation
Optimisation	Lambda-expressions
Lambda-expressions	Effacement de type
les "Streams"	Invariance des génériques vs. covariance des tableaux
Effacement de type	Wildcards
Invariance des génériques vs. covariance des tableaux	Concurrence
Wildcards	Interfaces graphiques

SortedMap est à `Map` ce que `SortedSet` est à `Set` : ainsi les associations sont ordonnées par rapport à l'ordre de leurs clés.

```
public interface SortedMap<K,V> extends Map<K,V>{
    Comparator<? super K> comparator();
    SortedMap<K,V> subMap(K fromKey, K toKey);
    SortedMap<K,V> headMap(K fromKey);
    SortedMap<K,V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

SortedMap est à `Map` ce que `SortedSet` est à `Set` : ainsi les associations sont ordonnées par rapport à l'ordre de leurs clés.

```
public interface SortedMap<K,V> extends Map<K,V>{
    Comparator<? super K> comparator();
    SortedMap<K,V> subMap(K fromKey, K toKey);
    SortedMap<K,V> headMap(K fromKey);
    SortedMap<K,V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Introduction	Généralités
Aldric Degorre	Style
Objets et classes	Types et polymorphisme
Héritage	Généricité
généricité : introduction	introduction
Collections	Optimisation
Optimisation	Lambda-expressions
Lambda-expressions	Effacement de type
les "Streams"	Invariance des génériques vs. covariance des tableaux
Effacement de type	Wildcards
Invariance des génériques vs. covariance des tableaux	Concurrence
Wildcards	Interfaces graphiques

Introduction	Généralités
Aldric Degorre	Style
Objets et classes	Types et polymorphisme
Héritage	Généricité
généricité : introduction	introduction
Collections	Optimisation
Optimisation	Lambda-expressions
Lambda-expressions	Effacement de type
les "Streams"	Invariance des génériques vs. covariance des tableaux
Effacement de type	Wildcards
Invariance des génériques vs. covariance des tableaux	Concurrence
Wildcards	Interfaces graphiques

Fabriques statiques du JDK → alternative intéressante aux constructeurs de collections :

- Nombreuses dans la classe `Collections`³⁰³ : collections vides, conversion d'un type de collection vers un autre, création de vues avec telle ou telle propriété, ...
- Pour obtenir une liste depuis un tableau : `Arrays.asList(tableau)`.
- Fabriques statiques de collections immuables, nommées « `of` », dans les interfaces `List`, `Set` et `Map` (Java ≥ 9) :

```
List<String> semaine = List.of("lundi", "mardi", "mercredi", "jeudi",
    "vendredi", "samedi", "dimanche");
Map<String, String> instruments = Map.of("guitare", "cordes", "piano", "cordes",
    "clarinette", "vent");
Set<Integer> premiers = Set.of(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
```

Appeler une fabrique plutôt qu'un constructeur évite de choisir une implémentation : on fait confiance à la fabrique pour choisir la meilleure pour les paramètres donnés.

303. Noter le 's'

Avantages des tableaux : syntaxe légère et efficacité.

Avantages des collections génériques :

- polyvalence (plein de collections adaptées à des cas différents)
- polymorphisme via les interfaces de collections
- sûreté du typage (« vraie » générativité)

Conclusion, utilisez les collections, sauf :

- si vous prototyppez un programme très rapidement et vous appréciez la simplicité
 - si vous souhaitez optimiser la performance au maximum 304 305
 - si vous pourrez faire mieux ? (peut-être, si besoin très spécifique)
304. Cela dit, les méthodes du *collection framework*, sont écrites et optimisées par des experts et déjà testées par des milliers de programmeurs. Pensez-vous faire mieux ? (peut-être, si besoin très spécifique) 305. Mais pourquoi programmez-vous en Java alors ?

Une solution pas trop mauvaise : 307 la classe `java.util.Optional` 308

- une instance de `Optional<T>` est une valeur représentant soit une instance présente de `T` soit l'absence d'une instance (par définition, de façon non ambiguë).
- ainsi, instance de `Optional<T>` contient juste un champ de type `T`
- La présence d'un élément se teste en appelant `isPresent`.
- On accède à la valeur de l'élément via la méthode `get` qui ne retourne jamais **null** mais lance `NoSuchElementException` si l'élément est absent.

Bien qu'`Optional<T>` n'implémente pas `Collection<T>`, il est pertinent d'imaginer `Optional<T>` comme un type représentant des collections de 0 ou 1 élément.

307. Les valeurs nullables gardent quelques avantages sur `Optional` : pas besoin d'allouer un conteneur supplémentaire, moins de lourds mécanismes syntaxiques (comme la nécessité d'appeler `isPresent` et `get`). De plus, même une expression de type `Optional` est elle-même nullable... Des alternatives existent (hors Java : notamment systèmes de types non nullables, en Java : annotations `@NotNull` et `@Nullable` + outil d'analyse statique).

308. Inspirée des langages fonctionnels : la classe `Option` en Scala, la monade `Maybe` en Haskell

Solutions (pas très bonnes) :

- retourner une valeur qui peut être `null` (« `nullable` »). **Inconvénients :** 306
 - si `getVal()` retourne une valeur nullable, l'appel `getVal().doSomething()` peut causer une `NullPointerException`. En toute généralité, cette exception peut se déclencher bien plus loin dans le programme (débogage difficile).
 - `null` peut aussi représenter une variable pas encore initialisée (ambiguité)
 - lancer une exception pour l'absence de valeur.

- Inconvénient :** obligation d'utiliser des `try catch` (lourdeur syntaxique, s'intègre mal au flot du programme) ; mécanisme coûteux à l'exécution.
- Utiliser une liste à 0 ou 1 élément.
 - **Inconvénient :** le type liste autorise les listes à 2 éléments ou plus.
306. L'invention de `null` a été qualifiée *a posteriori* par son auteur, Tony Hoare, d'*« erreur à un milliard de dollars »*, ce n'est pas peu dire !

Les optionnels

Compléments en POO

Les optionnels

Compléments en POO

Pourquoi c'est plus sûr qu'un type nullable :

- On ne peut pas appeler directement les méthodes de `T` sur une expression de type `Optional<T>` : il faut d'abord extraire son contenu (méthode `get()`).
- Ainsi pas de risque de `NullPointerException` (ni sur l'instance d'`Optional<T>` ni sur le résultat de `get()`).
- `get` peut bien lancer `NoSuchElementException`, mais ça se produit là où `get` est appelée. On voit donc tout de suite si et où on a oublié d'appeler `isPresent`.

Exemple :

```
Optional<Client> maybeRes = seat.getReservation();
if (maybeRes.isPresent()) {
    Client res = maybeRes.get(); // on est sûr qu'il n'y a pas d'exception
    res.sendReminder(); // aucun risque de NPE car res est résultat de get()
}
```

Remarque : cela peut aussi s'écrire

```
seat.getReservation().ifPresent(res -> res.sendReminder());
```

Les optionnels

Compléments en POO

Les optionnels

Compléments en POO

Ah... et comment les instancier au fait ?

- La classe `Optional` est munie de 2 fabriques statiques principales :
 - `<T> Optional<T> of(T elem) : si elem est non null, retourne un optional contenant elem (sinon NullPointerException)`
 - `<T> Optional<T> empty() : retourne un optional vide du type désiré (en fonction du contexte)`

Exemple :

```
public static Optional<Integer> findIndex(int[] elems, int elem) {
    for (int i = 0; i < elems.length; i++) {
        if (elems[i] == elem) return Optional.of(i);
    }
    return Optional.empty();
}
```

Les optionnels

Compléments en POO

Fonctions de première classe et fonctions d'ordre supérieur

Le besoin (1)

Appeler 5 fois une méthode `f` déjà connue :

```
f(); f(); f(); f(); f();
```

Appeler `f` un nombre de fois inconnu à l'avance :

```
// Facile ! On ajoute un paramètre int :
public static void repeat(int n) { for (int i = 0; i < n; i++) f(); }
```

Appeler 5 fois une méthode inconnue à l'avance ?

```
// Hm... il faudrait passer une méthode en paramètre ? Tentative :
public static void repeat(??? f) { // quel type pour f ?
    for (int i = 0; i < 5; i++) f(); // si f une variable, "f()" -> erreur de syntaxe
}
```

- `repeat5` = fonction avec paramètre fonction = **fonction d'ordre supérieur** (FOS).
- Pour que cela existe, il faut des fonctions considérées comme des valeurs (passables en paramètre) par le langage : des **fonctions de première classe** (FPC).

Fonctions de première classe et fonctions d'ordre supérieur		Fonctions de première classe et fonctions d'ordre supérieur	
Compléments en POO Aldric Degoire	Le besoin (2)	Compléments en POO Aldric Degoire	Plus d'exemples (1)
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : généricité ; Collections Optimisation Lambdas-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards	Concurrence Interfaces graphiques	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : généricité ; Collections Optimisation Lambdas-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards	Concurrence Interfaces graphiques
<ul style="list-style-type: none"> Une FPC peut être affectée à une variable, être le paramètre d'une FOS, ou bien sa valeur de retour : c'est une valeur comme une autre. Avec des valeurs fonction, il devient possible de manipuler des instructions sans les exécuter/évaluer immédiatement (évaluation paresseuse). Elles peuvent ainsi : <ul style="list-style-type: none"> être transformées, composées avant d'être évaluées; être évaluées plus tard, une, plusieurs fois ou pas du tout, en fonction de critères programmables ; (condition, répétition, déclenchement par événement ultérieur, ...); exécutées dans un autre contexte (p. ex. autre <i>thread</i>³⁰⁹). <p>De telles modalités d'exécution sont programmables en tant que FOS qui se comportent, en gros, comme de nouvelles structures de contrôle 310 .</p> <p><u>309.</u> Voir chapitre programmation concurrente. La programmation concurrente a probablement été un argument primordial pour l'introduction des lambdas-expressions en Java. <u>310.</u> À comparer avec while(...) ..., for(...) ..., switch(...) ..., if (...) ... else ... , ...</p>	<p>Bloc if/else :</p> <pre>// types et syntaxe d'appel des FPC toujours fantaisistes dans cet exemple public static void ifElse??? condition, ??? ifBlock, ??? elseBlock) { if (condition()) ifBlock(); else elseBlock(); }</pre> <p>Impossible d'écrire la signature d'une méthode mimant le bloc if/else sans paramètres FPC. Une telle méthode est nécessairement une FOS.</p>	<p>Encore un exemple :</p> <pre>// toujours en syntaxe fantaisiste --- SURTOUT NE PAS RECOPIER OU MÊME RETENIR ! public static <U, V> List<V> map(List<U> l, ??? f) { List<V> ret = new ArrayList<V>(); for (U x : l) ret.add(f(x)); return ret; }</pre> <p>Ou encore : 312</p> <pre>public static readPacket(Socket s, ??? callback) { ... }</pre>	<p>callback = FPC pour traiter le prochain paquet reçu = fonction de rappel/callback.</p> <p><u>311.</u> L'API <code>java.util.stream</code> contient plein de méthodes de traitement par lot dans ce genre.</p> <p><u>312.</u> Lecture asynchrone, similaire à ce qu'on trouve dans l'API <code>java.nio</code>.</p>
Compléments en POO Aldric Degoire	Le besoin (2)	Compléments en POO Aldric Degoire	Plus d'exemples (2)
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : généricité ; Collections Optimisation Lambdas-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards	Concurrence Interfaces graphiques	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : généricité ; Collections Optimisation Lambdas-expressions Les "Streams" Effacement de type Invariance des génériques vs. covariance des tableaux Wildcards	Concurrence Interfaces graphiques
<p>Évidemment, plus intéressant d'écrire de nouveaux blocs de contrôle, par exemple :</p> <p>Bloc retry :</p> <pre>// pareil, ne faites pas ça à la maison ! public static void retry(?? instructions, int tries) { while (tries > 0) { try { instructions(); return; } catch (Throwable t) { tries--; } } throw new RuntimeException("Failure persisted, after all tries."); }</pre>	<p>Fonctions de première classe et fonctions d'ordre supérieur</p> <p>Plus d'exemples (3)</p>	<p>Encore un exemple :</p> <pre>// toujours en syntaxe fantaisiste --- SURTOUT NE PAS RECOPIER OU MÊME RETENIR ! public static <U, V> map(List<U> l, ??? f) { List<V> ret = new ArrayList<V>(); for (U x : l) ret.add(f(x)); return ret; }</pre>	<p>Ou encore : 312</p> <pre>public static readPacket(Socket s, ??? callback) { ... }</pre>

Programmation fonctionnelle

Que faut-il pour qu'un langage supporte les FPC ?

Compléments en POO

Aldric Degoire

Concepts de FPC et de FOS essentiels pour la **programmation fonctionnelle** (PF) :

- PF = paradigme de programmation, au même titre que la POO.
- **Idée de base** : on conçoit un programme comme une fonction mathématique, elle-même obtenue par composition d'un certain nombre d'autres fonctions.
- Or pour pouvoir composer les fonctions, il faut supporter les FOS et donc les FPC.
- Langages fonctionnels connus : Lisp (et variantes : Scheme, Emacs Lisp, Clojure...), ML (et variantes : OCaml, F#...), Haskell, Erlang...
- Rien n'empêche d'être à la fois objet et fonctionnel (Javascript, Scala, OCaml, Common Lisp ...). Les langages sont souvent multi-paradigme (avec préférence).

Java (≥ 8) possède quelques concepts fonctionnels³¹³.

313. Mais n'est pas un vrai langage de PF pour autant (on verra plusieurs raisons). Remarque : quasiment tous les langages modernes supportent les FPC, bien qu'ils ne soient pas tous des LPPF.

Compléments en POO

Aldric Degoire

Principalement 3 choses :

- si langage à typage statique, un système de types permettant d'écrire les types des FPC
- une yntaxe adaptée pour les expressions décrivant les FPC.
Notamment, il faut des littéraux fonctionnels (appelés aussi **lambda-expressions**³¹⁴ ou encore fonctions anonymes).
- une représentation en mémoire adaptée pour les FPC
(en Java, forcément des objets particuliers)

314. En particulier en Java. C'est donc le nom « lambda-expression » que nous allons utiliser.
Pourquoi « lambda » ? Référence au lambda-calcu d'Alonzo Church : la fonction $x \mapsto f(x)$ s'y écrit $\lambda x.f(x)$.

Compléments en POO

Aldric Degoire

FPC avant Java 8

Bilan

Bilan pour les FPC avant Java 8 :

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
- **syntaxe** : lourde et peu pratique.
- **représentation en mémoire** : instance d'une classe contenant juste une méthode.

Compléments en POO

Aldric Degoire

FPC avant Java 8

Analyse de l'existant

Dans tout LOO, une FPC est représentable par un objet ayant la fonction comme méthode.

En Java (toute version) on implémente et instance une interface à méthode unique.

Typiquement, pour créer un *thread* à l'aide d'une classe anonyme :

```
new Thread( /* début de l'expression-fonction */ new Runnable {
    @Override public void run() { /* choses à faire dans l'autre thread */ }
} /* fin de l'expression-fonction */ ).start()
```

Ici, on passe une fonction (décrise par la méthode *run*) au constructeur de *Thread*.

Inconvénients :

- syntaxe lourde et peu lisible, même avec classes anonymes,
- obligation de se rappeler et d'écrire des informations sans rapport avec la fonction qu'on décrit (nom de l'interface : **Runnable** ; et de la méthode implémentée : **run**).

Compléments en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généralité : généricité

Introduction

Collections

Optimisation

Lambda-expressions

Effacement des types

Invocatrice des génératrices vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Conclusion

FAQ

Autres

Introduction	Compléments en POO
Généralités	Aldric Degoire
Style	Objets et classes Types et polymorphisme Héritage Généricité généricité : introduction Collections Optimisation Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs covariance des tableaux Wildcards
Concurrence	
Interfaces graphiques	Introduction à la programmation graphique

Bilan pour les FPC avant Java 8 :

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
→ à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.
Simon, rien n'est changé au système de type de Java (même sa syntaxe).
- **syntaxe** : lourde et peu pratique.
→ à partir de Java 8, on peut écrire des **lambda-expressions**.
- **représentation en mémoire** : instance d'une classe contenant juste une méthode.

Introduction	Compléments en POO
Généralités	Aldric Degoire
Style	Objets et classes Types et polymorphisme Héritage Généricité généricité : introduction Collections Optimisation Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs covariance des tableaux Wildcards
Concurrence	
Interfaces graphiques	Introduction à la programmation graphique

Bilan pour les FPC avant Java 8 :

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
→ à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.
Simon, rien n'est changé au système de type de Java (même sa syntaxe).
- **syntaxe** : lourde et peu pratique.
→ à partir de Java 8, on peut écrire des **lambda-expressions**.
- **représentation en mémoire** : instance d'une classe contenant juste une méthode.

Introduction	Compléments en POO
Généralités	Aldric Degoire
Style	Objets et classes Types et polymorphisme Héritage Généricité généricité : introduction Collections Optimisation Lambda-expressions Les "Streams" Effacement de type Invariance des génériques vs covariance des tableaux Wildcards
Concurrence	
Interfaces graphiques	Introduction à la programmation graphique

Bilan pour les FPC avant Java 8 :

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
→ à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.
Simon, rien n'est changé au système de type de Java (même sa syntaxe).
- **syntaxe** : lourde et peu pratique.
→ à partir de Java 8, on peut écrire des **lambda-expressions**.
- **représentation en mémoire** : instance d'une classe contenant juste une méthode.
→ comme c'est une idée raisonnable, ça ne change pas.

`java.util.function` contient toute une série d'interfaces fonctionnelles standard.

Interfaces génériques :

Interface	Type représenté	Méthode unique
<code>BiConsumer<T, U></code>	$T \times U \rightarrow \{\}$	<code>void accept(T, U)</code>
<code>BiFunction<T, U, R></code>	$T \times U \rightarrow R$	<code>R apply(T, U)</code>
<code>BinaryOperator<T></code>	$T \times T \rightarrow T$	<code>T apply(T, T)</code>
<code>BiPredicate<T, U></code>	$T \times U \rightarrow \{\bot, \top\}$	<code>boolean test(T, U)</code>
<code>Consumer<T></code>	$T \rightarrow \{\}$	<code>void accept(T)</code>
<code>Function<T, R></code>	$T \rightarrow R$	<code>R apply(T)</code>
<code>Predicate<T></code>	$T \rightarrow \{\bot, \top\}$	<code>boolean test(T)</code>
<code>Supplier<T></code>	$\{\} \rightarrow T$	<code>T get()</code>
<code>UnaryOperator<T></code>	$T \rightarrow T$	<code>T apply(T)</code>

De plus, ce package contient aussi des interfaces pour les fonctions prenant ou retournant des types primitifs `int`, `long`, `double` ou `boolean` (page suivante).

316. Moins d'allocations et d'indirections.

Introduction
Généralités

Style
Objets et classes

Types et polymorphisme
Héritage

Généricité :
généricité ;
introduction

Optimisation
Collections

Lambda-expressions
Les "streams"

Effacement de type
Invariance des génériques vs.
compatibilité des tableaux

Wildcards

Concurrency
Interfaces graphiques

Interfaces spécialisées :

Interface

BooleanSupplier

DoubleBinaryOperator

DoubleConsumer

DoubleFunction<R>

DoublePredicate

DoubleSupplier

DoubleToIntFunction

...

...

...

...

Catalogue des interfaces fonctionnelles de Java 8

dans le package java.util.function... mais pas seulement!

Attention, catalogue incomplet :

- L'interface `java.lang.Runnable` reste le standard pour les « fonctions » 317 de $\{\() \rightarrow \{\}\}$ (`void` vers `void`).
- Pas d'interfaces standard pour les fonctions à plus de 2 paramètres → il faut définir les interfaces soi-même :

```
@FunctionalInterface public interface TriConsumer<T, U, V> {
    void apply(T t, U u, V v);
}
```

L'annotation facultative `@FunctionalInterface` demande au compilateur de signaler une erreur si ce qui suit n'est pas une définition d'interface fonctionnelle.

317. Ces fonctions sont intéressantes pour leurs effets de bord et non pour la transformation qu'elles représentent. En effet, en mathématiques, $\text{card}(\{\}) \rightarrow \{\()\} = 1$.

Retour sur les exemples...

... avec les bons types et la bonne syntaxe

```
public static void repeat5(Runnable f) { for (int i = 0; i < 5; i++) f.run(); }
```

```
public static void ifElse(BooleanSupplier cond, Runnable ifBlock, Runnable elseBlock) {
    if (cond.get()) ifBlock.run();
    else elseBlock.run();
}
```

```
public static void retry(Runnable instructions, int tries) {
    while (tries > 0) {
        try {
            instructions.run();
        } catch (Throwable t) { tries--; }
    }
    throw new RuntimeException("Failure persisted after all tries.");
}
```

```
public static <U, V> List<V> map(List<U> l, Function<U, V> f) {
    List<V> ret = new ArrayList<V>();
    for (U x : l) ret.add(f.apply(x));
    return ret;
}
```

```
}
```

Compléments
en POO

Aléric Degoire

Introduction
Généralités

Style
Objets et classes

Types et polymorphisme
Héritage

Généricité :
généricité ;
introduction

Optimisation
Collections

Lambda-expressions
Les "streams"

Effacement de type
Invariance des génériques vs.
compatibilité des tableaux

Wildcards

Concurrency
Interfaces graphiques

Interfaces fonctionnelles

Écrire les types des FPC, c'est bien. Mais comment exécuter une fonction ?

Comme on a déjà pu voir sur les exemples :

- Il n'y a pas de yntaxe réservée pour exécuter une expression fonctionnelle.
- Il faut donc à chaque fois appeler explicitement la méthode de l'interface fonctionnelle concernée (et donc connaître son nom...).

Exemple :

```
Function<Integer, Integer> carre = n -> n * n;
System.out.println(carre.apply(5)); // <--- ici c'est apply
```

mais...

```
Predicate<Integer> estPair = n -> (n % 2) == 0;
System.out.println(estPair.test(5)); // <--- là c'est test
```

Syntaxe des lambda-expressions

lambda-abstraction : par l'exemple

Compléments
en POO

Aldric Degoire

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Généricité :
 Introduction
 Collections
 Optimisation
Lambda-expressions
 Les "Streams"
 Effacement de type
 Invariance des génériques vs covariance des tableaux
 Wildcards
Concurrence
Interfaces graphiques

Écrire une fonction anonyme par lambda-abstraction :

<paramètres> -> <corps de la fonction>

Exemples :

$x \rightarrow x + 2$

(raccourci pour `(int x) -> { return x + 2; }`)

$(x, y) \rightarrow x + y$

(raccourci pour `(int x, int y) -> { return x + y; }`)

$(a, b) \rightarrow \{$

```
int q = 0;
while (a >= b) { q++; a -= b; }
return q;
```

mais...

```
Predicate<Integer> estPair = n -> (n % 2) == 0;
System.out.println(estPair.test(5)); // <--- là c'est test
```

Compléments
en POO

Aldric Degoire

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Généricité :
 Introduction
 Collections
 Optimisation
Lambda-expressions
 Les "Streams"
 Effacement de type
 Invariance des génériques vs covariance des tableaux
 Wildcards
Concurrence
Interfaces graphiques

Syntaxe des lambda-expressions

Référence de méthode

Pour créer une lambda-expression contenant juste l'appel d'une méthode existante :

- on peut utiliser la lambda-abstraction :
 $x \rightarrow \text{Math.sqrt}(x)$
- mais il existe une notation encore plus compacte :
Math::sqrt

Ceci s'appelle une **référence de méthode**

Remarque :

- `Math::sqrt` est bien équivalent à $x \rightarrow \text{Math.sqrt}(x)$, et non à
(double x) -> Math.sqrt(x). Cela a une incidence pour l'inférence de type (cf. la suite).

retour non **void**

Syntaxe des lambda-expressions

Référence de méthode

Pour créer une lambda-expression contenant juste l'appel d'une méthode existante :

- on peut utiliser la lambda-abstraction :
 $x \rightarrow \text{Math.sqrt}(x)$
- mais il existe une notation encore plus compacte :
Math::sqrt

Ceci s'appelle une **référence de méthode**

Remarque :

- `Math::sqrt` est bien équivalent à $x \rightarrow \text{Math.sqrt}(x)$, et non à
(double x) -> Math.sqrt(x). Cela a une incidence pour l'inférence de type (cf. la suite).

retour non **void**

Syntaxe des lambda-expressions

Référence de méthode : les cas de figure

Supposons la classe suivante définie :

```
class C {  
    int val;  
    C(int val) { this.val = val; }  
    static int f(int n) { return n; }  
    int g(int n) { return val + n; }  
}
```

La notation « référence de méthode » se décline pour différents cas de figure :

- Méthode statique → `C::f` pour `n -> C.f(n)`
- Méthode d'instance avec récepteur donné → avec `x = new C()`, on écrit `x::g` pour `n -> x.g(n)`
- Méthode d'instance sans récepteur donné → `C::g` pour `(x, n)-> x.g(n)`
- Constructeur → `C::new` pour `n -> new C(n)`

En cas de surcharge, Java déduit la méthode référencée du type attendu.

Utiliser les méthodes (FoS) de nos exemples...

... en leur passant des lambda-expressions

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

```
// Dire 5 fois "Bonjour !" :  
repeat5(() -> { System.out.println("Bonjour\u2192!"); }) ;  
  
// Tirer pile ou face  
ifElse( () -> Math.random() > 0.5,  
() -> { System.out.println("pile"); },  
() -> { System.out.println("face"); }) ;  
  
// Essayer d'ouvrir un fichier jusqu'à 3 fois  
retry(() -> { ouvre("monFichier.txt"); }, 3);  
  
// Calculer les racines carrées des nombres d'une liste  
List<Double> racines = map(maListe, Math::sqrt);
```

Ce qui se cache derrière les lambda-expressions

Compléments en POO

Aldric Degorre

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Ce qui se cache derrière les lambda-expressions

Concernant le type (1)

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Deux questions se posent alors :

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Exemple, on peut écrire `Function<Integer, Double> f = x -> Math.sqrt(x);`

car l'interface `Function` est comme suit :

```
public interface Function<T,R> { R apply(T t); } // apply a une signature compatible
```

318. Ou bien, dans le cas où les types des arguments ne sont pas précisés, s'il existe une façon de les ajouter qui rend la signature compatible.

Ce qui se cache derrière les lambda-expressions

Concernant le typage (2) – petits pièges

Le fait de préciser le type des paramètres d'une lambda-expression restreint les possibilités d'utilisation.

Ces exemples compilent :

```
Function<Integer, Double> f = x -> Math.sqrt(x);  
Function<Integer, Double> f = (Integer x) -> Math.sqrt(x);
```

Mais ceux-ci ne compilent pas :

```
Function<Integer, Double> f = (Double x) -> Math.sqrt(x);  
IntFunction<Double> f = (double x) -> Math.sqrt(x); // pourtant la lambda-expression  
accepte double (plus large que int).
```

Remarquablement, ceci compile (malgré le fait que `sqr` ait un paramètre `double`) :

```
Function<Integer, Double> f = Math::sqrt;
```

Ce qui se cache derrière les lambda-expressions

Concernant le typage (3)

Attention : n'importe quel type SAM peut être le type d'une lambda-expression. Pas seulement ceux définis dans `java.util.function`.

Partout où une expression de type SAM attendue, on peut utiliser une lambda-expression, même si ce type (ou la méthode qui l'attend) date d'avant Java 8.

Ainsi, en Swing, à la place de la classique « invocation magique » :

```
SwingUtilities.invokeLater(  
    new Runnable() {  
        public void run() { MonIG.build(); }  
    });
});
```

on peut écrire : `SwingUtilities.invokeLater(() -> MonIG.build());`

ou encore mieux : `SwingUtilities.invokeLater(MonIG::build);`

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Instanceance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Avertissement

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Instanceance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Ce qui se cache derrière les lambda-expressions

Concernant l'évaluation

Avertissement

Attention aux variables locales !

À l'évaluation, Java construit un objet singleton (instance d'une classe anonyme) qui implémente l'interface fonctionnelle en utilisant la fonction décrite dans la lambda-expression.

Toujours dans le même exemple, `javac` sait alors qu'il doit compiler l'instruction

```
Function<Integer, Double> f = x -> Math.sqrt(x);
```

de la même façon 319 que :

```
Function<Integer, Double> f = new Function<Integer, Double>() {  
    @Override public Double apply(Integer x) {  
        return Math.sqrt(x);  
    }  
};
```

Avertissement

Attention aux variables locales !

- Valeur d'une lambda-expression = instance de classe locale (anonyme).
- Ainsi, une lambda-expression de Java ne peut utiliser que les variables locales effectivement `final`.³²⁰
- Comparaison avec OCaml : en OCaml, cette « limitation » n'est pas perçue car, en effet, les « variables » ne sont pas réaffectables³²¹ (tout est « `final` »).
- Comme Java, OCaml se contente de recopier les valeurs des variables locales dans la clôture de la fonction (λ l'instance de la classe locale).

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Instanceance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

319.

En fait, pour les lambdas, la JVM construit la classe anonyme à l'exécution seulement. À cet effet, java a en fait compilé l'expression en écrivant l'instruction `invokedynamic` (introduite dans Java 8 dans ce but). Autrement, les classes, même anonymes, sont créées à la compilation et existent déjà dans le code octet.

Compléments en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Instanceance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

320. Rappel : dans une classe locale, on a accès aux variables locales seulement si elles sont **effectivement final** (c.-à-d. jamais modifiées après leur initialisation). Cette restriction permet d'éviter les incohérences (modifications locales non partagées).

321. On simule des données locales modifiables en manipulant des « références » mutables :
`let ref x = 42 in x := !x + 1; x;`

Dans l'exemple, `x` n'est pas réaffectable, mais la valeur stockée à l'adresse contenue dans `x` l'est. L'équivalent en Java serait un objet-boîte contenant un unique attribut non `final`. D'ailleurs, rien n'empêche d'utiliser cette technique en Java ; il faut juste l'écrire « à la main ».

Fonctions récursives

Avertissement

Compléments en POO

Aldric Degorre

Incorrect :

```
int a = 1;
a++; // a réaffecte
Function<Integer, Integer> f = x -> { return x + a; };
```

Correct :

```
final int a = 1; // a final (non réaffectable)
Function<Integer, Integer> f = x -> { return x + a; };
```

Aussi correct :

```
int a = 1; // a effectivement final (non réaffectée)
Function<Integer, Integer> f = x -> { return x + a; };
```

Et correct aussi :

```
class IntRef { int val; IntRef(int val) { this.val = val; } }
final IntRef a = new IntRef(12);
Function<Integer, Integer> f = x -> { return a.val += x; /* modification de a */ };
```

Exemple :

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

définition : introduction

Collections

Optimisations

Lambda-expressions

Les "Streams"

Effacement de type

Instance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Graphiques

- Supposons qu'on veuille définir une fonction récursive, comme en OCaml :

```
let rec fact = function
| 0 -> 1
| n -> n * fact (n - 1);
```

- Sachant qu'il n'existe pas l'équivalent de **let rec** en Java, peut-on définir ?

```
IntUnaryOperator fact = n -> (n==0)?1:(n*fact.applyAsInt(n-1));
```

- Problème : la variable **fact** n'est pas encore déclarée quand on compile la lambda-expression \Rightarrow la compilation échoue.

Il existe des dizaines de façons de contourner cette limite, mais la seule qui soit élégante consiste à définir d'abord une méthode récursive.

Fonctions récursives

Compléments en POO

Aldric Degorre

Fonctions récursives

Pour définir cette FPC, il faudrait donc faire en 2 temps :

- initialiser **fact** (à **null** par exemple)
- écrire la lambda-expression (utilisant **fact**) et l'affecter à **fact**.

Il faudrait donc que la variable **fact** soit réaffectable... donc **fact** ne pourrait pas être locale.

→ Il faudrait que **fact** soit un attribut, mais alors attention à l'encapsulation.

Une possibilité, respectant l'encapsulation, à l'aide d'une classe locale auxiliaire :

```
class FunRef { IntUnaryOperator val; }
final FunRef factAux = new FunRef();
factAux.val = n -> (n==0)?1:(n*factAux.val.applyAsInt(n-1));
IntUnaryOperator fact = factAux.val;
```

Inconvénient : niveau d'indirection supplémentaire.

Compléments en POO

Aldric Degorre

Fonctions récursives

Alternative : déclarer la factorielle comme méthode privée récursive, puis manipuler une référence vers celle-ci.

```
class Autre2 {
    ...
    private int fact(int n) { return (n==0)?1:(n*fact(n-1)); }
    ...
}
```

Et si on veut tout encapsuler correctement, on revient à une classe anonyme classique :

```
IntUnaryOperator fact = new IntUnaryOperator() {
    @Override public int applyAsInt(int n) { return (n==0)?1:(n*applyAsInt(n-1)); }
}
```

Cette dernière technique n'a pas d'inconvénient³²². C'est donc celle qu'il faut privilégier.
322. si, un : on troque la syntaxe des lambda-expressions contre celle, plus verbuse, des classes anonymes

Bilan des FPC dans Java 8

Bilan des FPC dans Java 8

Le bon

Compléments en POO
Aldric Degoire

- Pas de notation (flèche) dédiée aux types fonctionnels, juste interfaces classiques.

- Plusieurs interfaces possibles pour une même fonction.

- Pas de syntaxe réservée, unique, pour exécuter une FPC.

À la place : appel de la méthode de l'interface, dont le nom peut varier.

- Clôture contenant variables effectivement finales seulement.

Implication : nécessité de « contourner » pour capturer un état mutable.³²³

(⇒ Impossible de définir simplement une lambda-expression récursive³²⁴)

- Malgré les apports de Java 8 à 13, le JDK contient peu d'APIs dans le style fonctionnel (On peut néanmoins citer Stream et CompletableFuture).

Java n'est toujours pas un langage de PF, mais juste un LOO avec support limité des FPC.

323. Cela dit, on évite d'utiliser un état mutable en PF.

324. De plus Java n'optimise pas la récursivité terminale. Ainsi, l'appel d'une méthode récursive sur des données de taille modérément grande risque facilement de provoquer un StackOverflowError.
Ainsi rien n'est fait pour encourager la programmation récursive.

Compléments en POO
Aldric Degoire

- Java supporte les FPC et les FOS.

Or les FPC sont amenées à jouer un rôle de plus en plus important, notamment pour la programmation concurrente³²⁵, qui devient de plus en plus incontournable³²⁶.

- Les API Stream et CompletableFuture sont d'excellents exemples d'API concurrentes, introduites dans Java 8 et utilisant les FOS.

À la place : appel de la méthode de l'interface, dont le nom peut varier.

- Clôture contenant variables effectivement finales seulement.

Implication : nécessité de « contourner » pour capturer un état mutable.³²⁵

(⇒ Impossible de définir simplement une lambda-expression récursive³²⁶)

- D'anciennes API se retrouvent immédiatement utilisables avec les lambda-expressions car utilisant déjà des interfaces fonctionnelles (e.g. JavaFX).

→ nouvelle concision, « gratuite ».

Malgré ses défauts, le support des FPC et FOS dans Java est un apport indéniable.

325. Que est le rapport entre FPC/FOS et programmation concurrente ? Plusieurs réponses :

la programmation concurrente incite à utiliser des structures immuables pour garantir la correction du programme.

Or le style fonctionnel est naturel pour travailler avec les structures immuables.

en programmation concurrente, on demande souvent l'exécution asynchrone d'un morceau de code. Pour ce faire, ce dernier doit être passé en argument d'une fonction (FOS) sous la forme d'une FPC.

326. En particulier à cause de la multiplication du nombre de cœur dans les microprocesseurs.

Compléments en POO
Aldric Degoire

- Java supporte les FPC et les FOS.

Or les FPC sont amenées à jouer un rôle de plus en plus important, notamment pour la programmation concurrente³²⁵, qui devient de plus en plus incontournable³²⁶.

- Les API Stream et CompletableFuture sont d'excellents exemples d'API concurrentes, introduites dans Java 8 et utilisant les FOS.

À la place : appel de la méthode de l'interface, dont le nom peut varier.

- Clôture contenant variables effectivement finales seulement.

Implication : nécessité de « contourner » pour capturer un état mutable.³²⁵

(⇒ Impossible de définir simplement une lambda-expression récursive³²⁶)

- D'anciennes API se retrouvent immédiatement utilisables avec les lambda-expressions car utilisant déjà des interfaces fonctionnelles (e.g. JavaFX).

→ nouvelle concision, « gratuite ».

Malgré ses défauts, le support des FPC et FOS dans Java est un apport indéniable.

325. Que est le rapport entre FPC/FOS et programmation concurrente ? Plusieurs réponses :

la programmation concurrente incite à utiliser des structures immuables pour garantir la correction du programme.

Or le style fonctionnel est naturel pour travailler avec les structures immuables.

en programmation concurrente, on demande souvent l'exécution asynchrone d'un morceau de code. Pour ce faire, ce dernier doit être passé en argument d'une fonction (FOS) sous la forme d'une FPC.

326. En particulier à cause de la multiplication du nombre de cœur dans les microprocesseurs.

Compléments en POO
Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité / introduction

Collections

Opérations

Lambda-expressions

Les "streams"

Effacement de type

initialisation des génériques vs. covariance des tableaux

Wildcards

Concurrente

Interfaces graphiques

Concurrente

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité / introduction

Collections

Opérations

Lambda-expressions

Les "streams"

Effacement de type

initialisation des génériques vs. covariance des tableaux

Wildcards

Concurrente

Interfaces graphiques

Concurrente

Compléments en POO
Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité / introduction

Collections

Opérations

Lambda-expressions

Les "streams"

Effacement de type

initialisation des génériques vs. covariance des tableaux

Wildcards

Concurrente

Interfaces graphiques

Concurrente

Opérations d'agrégation

Opérations d'agrégation

Quelques exemples (1)

Opérations d'agrégation

Opérations d'agrégation

Définition et quelques exemples

Compléments en POO
Aldric Degoire

Opération d'agrégation : traitement d'une séquence de données de même type qui produit un résultat synthétique³²⁷ dépendant de toutes ces données.

Exemples :

- calcul de la taille d'une collection

- concaténation des chaînes d'une liste de chaînes

- transformation d'une liste de chaînes en la liste de ses longueurs (ex : "bonjour", "le", monde → 7, 2, 5)

- recherche d'un élément satisfaisant un certain critère

Tous ces calculs pourraient s'écrire à l'aide de boucles for très similaires...

Calcul de la taille d'une collection :

```
public static int size(Collection<?> dataSource) {  
    int acc = 0;  
    for (Object e: dataSource) acc++;  
    return acc;  
}
```

Concaténation des chaînes d'une liste de chaînes :

```
public static String concat(List<String> dataSource) {  
    String acc = "";  
    for (String e: dataSource) acc += e.toString();  
    return acc;  
}
```

Opérations d'agrégation

Quelques exemples (2)

Opérations d'agrégation

Généralisation

Transformation d'une liste de chaînes en la liste de ses longueurs :

```
public static List<Integer> Lengths(List<String> dataSource) {  
    List<Integer> acc = new LinkedList<()>;  
    for (String e : dataSource) acc.add(e.length());  
    return acc;  
}
```

Recherche d'un élément satisfaisant un certain critère 328 :

```
public static <E> E find(List<E> dataSource, Predicate<E> criterion) {  
    E acc = null;  
    for (E e : dataSource) acc = (criterion.test(e)) ? e : null;  
    return acc;  
}
```

Voyez-vous le motif commun ?

328. Remarque : on peut optimiser cette boucle, mais cette présentation illustre mieux le propos.

Opérations d'agrégation

Généralisation

Compléments en POO

Aldric Degorre

On garde ce qui est commun dans une méthode prenant en argument ce qui est différent :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, ??? op) {  
    R acc = zero;  
    for (E e : dataSource) acc = op(acc, e); // comment on écrit ça déjà ?  
    return acc;  
}
```

Opérations d'agrégation

Généralisation

Opérations d'agrégation

Généralisation

Opérations d'agrégation

Généralisation

Compléments en POO

Aldric Degorre

On garde ce qui est commun dans une méthode prenant en argument ce qui est différent :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, ??? op) {  
    R acc = zero;  
    for (E e : dataSource) acc = op(acc, e); // comment on écrit ça déjà ?  
    return acc;  
}
```

Opérations d'agrégation

Généralisation

Compléments en POO

Aldric Degorre

On peut alors écrire :

```
public static int size(Collection<?> dataSource) {  
    return fold(dataSource, 0, (acc, e) -> acc + 1);  
}  
  
public static String concat(List<String> dataSource, "") {  
    return fold(dataSource, "", (acc, e) -> acc + e.toString());  
}
```

Opérations d'agrégation

Généralisation

Opérations d'agrégation

Généralisation

Opérations d'agrégation

Généralisation

Compléments en POO

Aldric Degorre

... et on se rappelle le cours sur les fonctions de première classe (FPC) et les fonctions d'ordre supérieur (FOS) :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, BiFunction<R, E, R> op) {  
    R acc = zero;  
    for (E e : dataSource) acc = op.apply(acc, e);  
    return acc;  
}
```

Opérations d'agrégation

Généralisation

Opérations d'agrégation

Généralisation

Opérations d'agrégation

Généralisation

Compléments en POO

Aldric Degorre

// "Bof" : on modifie l'argument de op dans op (incorrect si fold est concurrent).
// On doit pouvoir faire mieux (à méditer en TP...) !

```
public static <E> E find(List<E> dataSource, Predicate<E> criterion) {  
    return fold(dataSource, null, (acc, e) -> (criterion.test(e) ? e : null));  
}
```

Opérations d'agrégation

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Éffacement de type	Intégration avec génériques vs covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques

Pour écrire des traitements similaires à ces exemples, on aimera une API fournissant des FOS analogues à **fold** pour les principaux schémas d'itération.³²⁹

C'est justement ce que fait l'API **stream**.³³⁰

- 329. Similaires aux fonctions de manipulation de liste en OCaml.
- 330. Mais pas seulement...

Compléments
en POO

Streams : API introduite dans Java 8 pour effectuer des opérations d'agrégation.

- API dans le style fonctionnel, avec fonctions d'ordre supérieur;
- distincte de l'API des collections (nouvelle interface **Stream**, au lieu de méthodes ajoutées à **Collection**)³³¹;
- optimisée pour les grands volumes de données : **évaluation paresseuse** (calculs effectués seulement au dernier moment, seulement lorsqu'ils sont nécessaires);
- qui sait utiliser les CPUs multi-cœur pour accélérer ses calculs (implémentation parallèle *multi-threadée*).

- Avertissement :** ce chapitre traite du package `java.util.stream` introduit dans Java 8. Ces streams n'ont aucun rapport avec les classes `InputStream` et `OutputStream` de `java.io`.
331. Heureusement on obtient facilement une instance de `Stream` depuis une instance de `Collection`, grâce à la méthode `Stream.of(Collection)`.

- Quelques streams :**
- Pour toute collection `coll`, le stream associé est `coll.stream()`.
 - `Stream.of(4, 39, 2, 12, 32)` représente la séquence 4, 39, 2, 12, 32.
 - `Stream.of(4, 39, 2, 12, 32).map(x -> 2 * x)` représente la séquence 8, 78, 4, 24, 64.

Inversement, on peut ensuite obtenir une collection depuis un stream :

```
Stream.of(4, 39, 2, 12, 32).map(x -> 2 * x).collect(Collectors.toList)
```

→ on obtient un `List<Integer>(collect, opération terminale, force le calcul de la séquence).`

- 332. Souvent une collection, mais peut aussi être un tableau, une fonction productrice d'éléments, un canal d'entrées/sorties

Qu'est-ce qu'un stream ? → **2 points de vue :**

- ① la représentation implicite d'une séquence d'éléments finie ou infinie
- ② la description d'une suite d'opérations permettant d'obtenir cette séquence.

Remarques importantes :

- Un objet **stream n'est pas une collection** : il ne contient qu'une référence vers une source d'éléments (parfois une collection, souvent un autre **stream**) et la description d'une opération à effectuer.
- Un objet **stream n'est pas le résultat d'un calcul**, mais la description d'un calcul à effectuer ³³³.
- 333. Pour les fans de programmation fonctionnelle : le type `Stream<T>` muni des opérations `of` et `flatMap` est une monade.

Les streams et les iterators ont beaucoup en commun :

- intermédiaires techniques pour parcourir les collections
- contiennent juste l'information pour faire cela ; pas les éléments eux-mêmes ;
- usage unique (après le premier parcours de la source, l'objet ne peut plus servir)

Et une grosse différence :

- **streams** : opérations agissant sur l'ensemble des éléments (itération implicite). Ce sont les méthodes fournies dans le JDK qui gèrent l'itération (et proposent notamment une implémentation en parallèle sur plusieurs threads ³³⁴).
 - **iterators** : 1 opération (`next`) = lire l'élément suivant (→ itération explicite avec **for** ou **while**)
334. Le **stream** construit sur une collection utilise en fait le **splitterator** de celle-ci : sorte d'itérateur évolué capable, en plus d'itérer séquentiellement, de couper une collection en morceaux ("split") pour partager le travail entre plusieurs **threads**.

- la plupart des collections ne sont pas thread safe (comportement incorrect quand utilisées dans plusieurs **threads** en même temps, notamment à cause des accès en compétition)
- on peut y ajouter de la synchronisation (voir collections synchronisées), mais toujours risque de dead-lock.

Pourtant, accélérer le traitement les grandes collections, il est utile de profiter du parallélisme.

15 nombres entiers aléatoires positifs inférieurs à 100 en ordre croissant :

```
Stream.generate(Math::random) // on obtient un Stream<Double>
  .limit(15) //
  .map(x -> (int)(100 * x)) //
  .sorted() //
  .collect(Collectors.toList()); //
```

Nombre d'usagers d'une bibliothèque ayant emprunté un livre d'Alexandre Dumas :

```
bibli.getLivres() // List<Livre>
  .stream() //
  .filter(livre -> livre.getAuteur().equals("Dumas", "Alexandre")) //
  .flatMap(livre -> livre.getEmprunteurs().stream()) //
  .distinct() //
  .count(); //
```

Streams et parallélisme

Problème à résoudre... une solution ?

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

généricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Compléments en POO
Aldric Degorre

Les streams, semblent une réponse naturelle à ce problème. En effet :

- leur opérations ne modifient pas le contenu de leur source ;

- l'objet de type Stream est lui-même à usage unique.

→ protection naturelle maximale contre les accès en compétition.

→ **Ce serait bien que les opérations d'agrégation d'un stream puissent être réparties sur plusieurs threads** (lors de l'appel à l'opération terminale)...

Compléments en POO

Aldric Degorre

Streams et parallélisme

Streams séquentiels et streams parallèles

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

généricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Compléments en POO
Aldric Degorre

... et bien justement :

Java permet de lancer les opérations d'agrégation en parallèle³³⁵, sans presque rien changer à l'invocation du même traitement en séquentiel :

- Il suffit de créer le stream avec `maCollection.parallelStream()` à la place de `maCollection.stream()`.
 - Alternative : à partir d'un stream séquentiel, on peut obtenir un stream parallèle avec la méthode `parallel()`, et vice-versa avec la méthode `sequential()`.
- Un stream est soit (entièrement) parallèle, soit (entièlement) séquentiel. L'opération terminale prend seulement en compte le dernier appel à `sequential()`³³⁶.
-
335. En utilisant (de façon cachée) `ForkJoinPool.ForkJoinTask`. Sauf mention contraire, le `thread pool` par défaut `ForkJoinPool.commonPool()` est utilisé.
336. Rappel : l'effectuation des calculs étant seulement déclenchée par l'opération terminale, il est logique que ses modalités concrètes d'exécution ne soient prises en compte qu'à ce moment.

Compléments en POO

Aldric Degorre

Streams et parallélisme

Quelques explications

Compléments en POO
Aldric Degorre

Streams et parallélisme

Piège à éviter : les effets de bord

Streams et parallélisme

Piège à éviter : les effets de bord

En général, évitez les effets de bord³³⁷ dans le pipeline, préférez les fonctions pures .

- Pour les entrées/sorties, au mieux, pas de contrôle sur leur ordre.
 - Pour les modifications d'objets partagés :
 - sans synchronisation, risque d'accès en compétition. Or, dans ce cas, le modèle de concurrence de Java ne garantit rien (→ résultats incorrects).
 - avec Synchronisation : comme on ne contrôle pas l'ordre d'exécution des tâches du pipeline, risque de dead lock.
Sinon, de toute façon, la synchronisation ralentit l'exécution.
- Heureusement, habituellement³³⁹, il est inutile de modifier des objets extérieurs dans les opérations d'un stream.
-

337. Effet de bord : tout effet externe d'une fonction, c'est à dire toute sortie physique ou modification de mémoire en dehors de son espace propre (= variables locales + champs des objets non partagés).
338. Fonction pure : fonction (méthode ou FPC) sans effet de bord.
339. À part à des fins de débogage ou de monitoring.

Appendice : les méthodes de l'API stream

L'interface Stream

L'interface principale du package

```
public interface Stream<T> { // pour des éléments de type T
...
}
```

Les transparents qui suivent :

- sont un résumé des méthodes proposées dans l'API stream.
- ne sont pas détaillés en cours magistral
- doivent servir de référence pour les TPs et pour la relecture approfondie du cours.
- doivent servir de référence pour les TPs et pour la relecture approfondie du cours.

Compléments en POO

Aldric Degorre

Introduction	Introduction
Généralités	Généralités
Style	Style
Objets et classes	Objets et classes
Types et polymorphisme	Types et polymorphisme
Héritage	Héritage
Généricité :	Généricité :
Généricité :	Généricité :
Introduction	Introduction
Collections	Collections
Optimisation	Optimisation
Lambda expressions	Lambda expressions
Les "streams"	Les "streams"
Effacement de type	Effacement de type
Instanciation des génériques vs. covariance des tableaux	Instanciation des génériques vs. covariance des tableaux
Wildcards	Wildcards
Concurrence	Concurrence
Interfaces graphiques	Interfaces graphiques

- (Il existe aussi `DoubleStream`, `IntStream` et `LongStream`.)
- Cette interface contient un grand nombre de méthodes. 3 catégories :
- des méthodes statiques 340 servant à créer des streams depuis des sources diverses.
 - des méthodes d'instance transformant des streams en streams (pour les opérations intermédiaires)
 - des méthodes d'instance transformant des streams en autre chose (pour les opérations terminales).

340. Rappel : oui, c'est possible depuis Java 8.

Fabriquer un stream depuis une source

Transformer un stream : les opérations intermédiaires (1)

Compléments en POO

Aldric Degorre

- Depuis une collection : méthode `Stream<T> stream()de Collection<T>`.
- À l'aide d'une des méthodes statiques de `Stream` :
- `<T> Stream<T> empty()` : retourne un stream vide
 - `<T> Stream<T> generate(Supplier<T> s)` : retourne la séquence des éléments générés par `s.get()` (Rappel : `Supplier<T>` = fonction de `{ } → T`).
 - `<T> Stream<T> iterate(T seed, UnaryOperator<T> f)` : retourne la séquence des éléments `seed, f.apply(seed), f.apply(f.apply(seed))` ...
 - `<T> Stream<T> of(T... values)` : retourne le stream constitué de la liste des éléments passés en argument (méthode d'arité variable).

En utilisant un builder³⁴¹ (`Stream.Builder`) :

- Un `Stream.Builder` est un objet mutable servant à construire un stream.
- On instancie un builder vide avec l'appel statique `b = Stream.builder()`
- On ajoute des éléments avec les appels `b.add(T e)` ou `b.accept(T e)`.
- On finalise en créant le stream contenant ces éléments : appel `s = b.build()`

341. On parle du patron de conception `builder` (ou "monteur"), ici appliqué aux streams. Ainsi, par exemple, il existe une classe `StringBuilder` jouant le même rôle pour les `String`. Voir le TP sur le patron `builder`.

Compléments en POO

Aldric Degorre

- Pour `this` instance de `Stream<T>` :
- `Stream<T> distinct()` retourne un stream qui parcourt les éléments de `this` sans les doublons.
 - `Stream<T> filter(Predicate<? super T> p)` retourne le stream parcourant les éléments `x` de `this` qui satisfont `p.test(x)`
 - ` Stream flatMap(Function<? super T, ? extends Stream> mapper)` retourne la concaténation des streams `mapper . apply(x)` pour tout `x` dans `this`.
 - `Stream<T> limit(long n)` tronque le stream après `n` éléments.
 - ` Stream map(Function<T, B > f)` retourne le stream des éléments `f . apply(x)` pour tout `x` élément de `this`.

Compléments en POO

Aldric Degorre

- Introduction
- Généralités
- Style
- Objets et classes
- Types et polymorphisme
- Héritage
- Généricité :
- Généricité :
- Introduction
- Collections
- Optimisation
- Lambda expressions
- Les "streams"
- Effacement de type
- Instanciation des génériques vs. covariance des tableaux
- Wildcards
- Concurrence
- Interfaces graphiques

Transformer un stream : les opérations intermédiaires (2)

`Stream<T> peek(Consumer<? super T> c)`

retourne un stream avec les mêmes éléments que **this**. À l'étape terminale, pour chaque élément **x** parcouru, **c.consume(x)** sera exécuté³⁴².

`Stream<T> skip(long n)`

retourne le suffixe de la séquence en sautant les **n** premiers éléments.

`Stream<T> sorted()`

retourne la séquence, triée dans l'ordre naturel.

`Stream<T> sorted(Comparator<? super T> comparator)`

idem mais en suivant l'ordre fourni.

³⁴² Remarque : **c** ne sert que pour ses effets de bord. **peek** peut notamment être utile pour le débogage.

Calculer et extraire un résultat : les opérations terminales

`Complements<T> allMatch(Predicate<? super T> p)` retourne vrai si et seulement si **p** est vrai pour tous les éléments du stream.

`boolean allMatch(Predicate<? super T> p)` retourne vrai si et seulement si **p** est vrai pour tous les éléments du stream.

`boolean anyMatch(Predicate<? super T> p)` retourne vrai si et seulement si **p** est vrai pour au moins un élément du stream.

`long count()` retourne le nombre d'éléments dans le stream.

`Optional<T> findAny()` : retourne un élément (quelconque) du stream ou rien si stream vide (voir interface `Optional<T>`).

`Optional<T> findFirst()` : pareil, mais premier élément.

Calculer et extraire un résultat : les opérations terminales

Calculer et extraire un résultat : les opérations terminales

`Optional<T> reduce(BinaryOperator<T> op)` : effectue la réduction du stream par l'opération d'accumulation associative op.

`T reduce(T zero, BinaryOperator<T> op)` : idem, avec le zéro fourni.

`<U> U reduce(U z, BiFunction<U, ? super T, U> acc, BinaryOperator<U> comb)` : idem avec accumulation vers autre type.³⁴³

`<R> R collect(Supplier<R> z, BiConsumer<R, ? super T> acc, BiConsumer<R, R> comb)` : comme reduce avec accumulation dans objet mutable.

`<R,A> R collect(Collector<? super T,A,R> collector)` : on parle juste après.

³⁴³ Opération appelée `fold` dans d'autres langages. Les définitions varient...

L'interface Collector

interface Collector<T,A,R>

Compléments en POO
Aldric Degorre

- Un **Collector** est un objet servant à “réduire” un stream en un résultat concret (en effectuant le calcul). Pour ce faire,
 - il initialise un accumulateur du type désiré (p. ex : liste vide),
 - puis transforme et aggège les éléments issus du calcul du stream dans l’accumulateur (ex : ajout à la liste)
 - enfin il “finalise” l’accumulateur avant retour (p. ex : suppression des doublons).

Trois techniques pour fabriquer un tel objet :

- (cas courants) utiliser une des fabricues statiques de la classe Collectors**
- utiliser la fabrique statique **Collector.of()** (“constructeur” généraliste)
- programmer à la main une classe qui implémente l’interface **Collector**

Introduction	Compléments en POO
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : introduction	
Collections	
Optimisations	
Lambda expressions	
Les “streams”	
Effacement de type	
Instance des génériques vs. covariance des tableaux	
Wildcards	
Concurrent	
Interfaces graphiques	

345. mais ici : implémentation “mutable”, utilisant un attribut accumulateur, alors que dans Stream, les réductions utilisent de fonctions “pures”

345. ... mais le plus simple reste `monStream.count()` !

Effacement de type (type erasure)

De quoi il s’agit.

Compléments en POO

Aldric Degorre

- Effacement d'un type** : sur-approximation permettant d'obtenir un **type réifiable** (i.e. : « classique », façon Java 4) à partir de n'importe quel type. Plus précisément (JLS 4.6) :
 - L'effacement d'un type générique ou paramétré de forme `G<...>`, est le type brut `G`.
 - L'effacement d'une variable de type est l'effacement de sa borne supérieure.
 - L'effacement de tout autre type `T` est `T`.

L'idée principale du phénomène appelé **effacement de type** (ou **type erasure**) c'est que le système de types de la JVM ne connaît que les types réifiables.

Autrement dit : la paramétrisation générique n'a pas d'impact à l'exécution.

Plus de détails juste après.

Construire un collector via l'interface Collector

Compléments en POO

Aldric Degorre

- Au cas où la bibliothèque Collectors ne contient pas ce qu'on cherche, on peut créer un **Collector** autrement :
 - créer et instancier une classe implémentant Collector.
 - Méthodes à implémenter : `accumulator()`, `characteristics()`, `combiner()`, `finished()` et `supplier()`.
 - sinon, créer directement l'objet grâce à la méthode statique `Collector.of()` :

```
c2 = Collector<Integer, List<Integer>, Integer> c2 = Collector.of(
    ArraysList<Integer>::new, List::add,
    (l1, l2) -> {l1.addAll(l2); return l1;}, List::size
);
```

(façon... un peu alambiquée de calculer la taille d'un stream...)

Utiliser la méthode `of()` est plus “légère” syntaxiquement, mais ne permet pas d'ajouter des champs ou des méthodes à l'objet fabriqué.

Introduction	Compléments en POO
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : introduction	
Collections	
Optimisations	
Lambda expressions	
Les “streams”	
Effacement de type	
Instance des génériques vs. covariance des tableaux	
Wildcards	
Concurrent	
Interfaces graphiques	

Effacement de type (type erasure)

Explication
Effacement de type (type erasure)

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

électricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "streams"

Effacement de type

Invariance des généraux vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Aldric Degorre

L'effacement de type commence en réalité dès la compilation.

Descripteur de méthode : information, dans la **table des constantes** d'une classe compilée, permettant d'identifier une méthode (peut-être surchargée) de façon unique. Tout appel de méthode³⁴⁶ dans le code-octet fait référence à un tel descripteur. Or un descripteur consiste en un couple : (nom de méthode, types réifiables des paramètres).

Conséquences :

- aucun paramètre de type n'est réellement passé aux constructeurs (et méthodes)
- les objets ne stockent donc pas les valeurs de leurs paramètres de types. Ils ne peuvent donc connaître que leur classe³⁴⁷. Les « objets paramétrés » n'existent pas.

346. `invokespecial`, `invokevirtual` et `invokeinterface` prennent un index de la table des constantes comme paramètre.
 347. Qui n'existe qu'en un seul exemplaire dans la mémoire, quelle que soit la paramétrisation.

Compléments
en POO

Aldric Degorre

Effacement de type (type erasure)

Conséquences concrètes
Effacement de type (type erasure)

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

électricité :

Introduction

Collections

Optimisation

Lambda-expressions

Les "streams"

Effacement de type

Invariance des généraux vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Aldric Degorre

À l'exécution, il est impossible de savoir comment un paramètre de type a été concrétisé.

En particulier :

- Faute d'être raisonnablement exécutables, ces expressions ne compilent pas :
 - `x instanceof P` (avec `P` paramètre de type);
 - `x instanceof Type<Y>`³⁵⁰ (avec `Y` type quelconque).

• On ne peut pas déclarer d'exception générique `Ex<T>` car `catch(Ex<X> ex)` ne serait pas non plus évaluable (même problème qu'`instanceof`).

```
// ne compile pas
public class A extends ArrayList<Integer> implements List<String> { ... }
```

Conséquences concrètes
Effacement de type (type erasure)

- Dans une classe, on ne peut pas définir plusieurs méthodes dont les signatures seraient identiques après effacement (leurs descripteurs seraient identiques).

```
// ne compile pas
public class A {
    List<Integer> f() { return null; }
    List<String> f() { return null; }
}
```

- Une classe ne peut pas implémenter plusieurs fois une interface générique avec des paramètres différents (on se retrouverait dans le cas précédent).

```
// ne compile pas
public class A extends GenericException<T> implements Exception { ... }
```

350. Mais `x instanceof Type` compile, c'est un des rares cas où on tolère le raw type.

Effacement de type (type erasure)

Mais quand les paramètres de type servent-ils alors ?

Le problème : examinons l'exemple suivant (qui ne compile pas).

```
public static void main(String[] args) {
    List<Voiture> listVoituresP = new ArrayList<>();
    // l1: ceci est en fait interdit... mais supposons que ça passe...
    List<Voiture> listVoiture = listVoituresP;
    // l2: instruction bien typée (pour le compilateur), mais...
    listVoiture.add(new Voiture());
    // l3: ... logiquement ça afficherait "Voiture" (contradictoire)
    System.out.println(listVoitureSP.get(0).getClass());
}
```

S'il compilait, en l'exécutant, à la fin, `listVoitureSP = listVoiture` contiendrait des `Voiture` → **contredit la déclaration de `List<VoituresP!`**

Ainsi, Java interdit l1 : deux spécialisations différentes du même type générique sont incompatibles. On dit que les génériques de Java sont **invariants**.

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	généricité : introduction	Collections	Optimisation	Lambda-expressions	Effacement de type	Invariance des génériques vs. covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques
Compléments en POO	Alidric Degoire														

Types génériques et sous-typage

Invariance des génériques

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	généricité : introduction	Collections	Optimisation	Lambda-expressions	Effacement de type	Invariance des génériques vs. covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques
Compléments en POO	Alidric Degoire														

Types génériques et sous-typage

Invariance des génériques

Tableaux – génériques ou pas ? (1)

Remarque : l'analogue à l'exemple précédent utilisant `Voiture[]` au lieu de `List<Voiture>` compile sans avertissement :

```
public static void main(String[] args) {
    VoitureSansPermis[] listVoituresP = new VoitureSansPermis[100];
    // l1: ceci est autorisé !
    Voiture[] listVoiture = listVoituresP;
    // l2: instruction bien typée (pour le compilateur), mais... ArrayStoreException !
    listVoiture[0] = new Voiture();
    // l3: on ne va pas jusque là
    System.out.println(listVoitureSP[0].getClass());
}
```

→ Les tableaux sont **covariants** (l1 autorisé) : `[A <: B] ⇒ [A[] <: B[]]`.

Mais on crashe plus loin, lors de l'exécution de l2.

Types génériques et sous-typage

Invariance des génériques

Tableaux – génériques ou pas ? (1)

Note : cependant le compilateur détecte la conversion « `louche` » et signale un avertissement (`warning`) « `unchecked conversion` » pour la ligne l1'.

Moralité : si avertissement, alors garanties usuelles supprimées.

```
// l1': version avec "triche" (pas d'exception car type erasure)
List<Voiture> listVoiture = (List<Voiture>)(Object) listVoituresSP;
/* l2: */ listVoiture.add(new Voiture());
// l3: ça affiche effectivement "Voiture" (oooooh !)
System.out.println(listVoituresSP.get(0).getClass());
// l4: et pour la forme, une petite ClassCastException :
VoitureSansPermis vsp = listVoitureSP.get(0);
```

→ Les tableaux sont **covariants** (l1 autorisé) : `[A <: B] ⇒ [A[] <: B[]]`.

Tableaux – génériques ou pas ? (2)

Génériques et tableaux

Différence de philosophie (1)

Compléments en POO
Aldric Degorre

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Collections	Opérations	Lambda-expressions	Les "Streams"	Effacement de type	Invariance des génériques vs. covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques	Autres
<ul style="list-style-type: none">• covariance à la place d'invariance : \Rightarrow vérifications moins strictes à la compilation,• rendant possibles des problèmes à l'exécution³⁵¹• pour détecter les problèmes au plus tôt : à l'instanciation, un tableau enregistre le nom du type déclaré pour ses éléments (pas d'effacement de type)• cela permet à la JVM de déclencher ArrayStoreException lors de toute tentative d'y stocker un élément du mauvais type, au lieu de ClassCastException lors de son utilisation (donc bien plus tard). <p>→ « Genre de » généricité, mais conception obsolète : avec la généricité moderne, la compilation garantit une exécution sans erreur.</p> <p>351. Raison : un tableau est à la fois producteur et consommateur. D'un point de vue théorique, une telle structure de données ne peut être qu'invariante, si on veut des garanties dès la compilation.</p>	<ul style="list-style-type: none">• Tableaux : (vérification à l'exécution, mais le + tôt possible)• usage normal : conversion sans warning de SousType[] à SuperType[] par upcasting (implicite).	<ul style="list-style-type: none">• Possibilité d'ArrayList<String>															

Génériques et tableaux

Différence de philosophie (2)

Compléments en POO
Aldric Degorre

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Collections	Opérations	Lambda-expressions	Les "Streams"	Effacement de type	Invariance des génériques vs. covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques	Autres
<h2>Génériques et tableaux</h2> <p>Différence de philosophie (2)</p>	<h2>Génériques et tableaux</h2> <p>Différence de philosophie (2)</p>																

Génériques et tableaux

Différence de philosophie (1)

Compléments en POO
Aldric Degorre

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Collections	Opérations	Lambda-expressions	Les "Streams"	Effacement de type	Invariance des génériques vs. covariance des tableaux	Wildcards	Concurrence	Interfaces graphiques	Autres
<h2>Génériques et tableaux</h2> <p>Différence de philosophie (1)</p>	<h2>Génériques et tableaux</h2> <p>Différence de philosophie (1)</p>	<ul style="list-style-type: none">• Génériques : (vérification à la compilation... puis plus rien)• usage normal, le compilateur rejette toute tentative de conversion implicite de Gen<A> à Gen, garantissant qu'à l'exécution toute instance de Gen<T> sera bien utilisée avec le type T → exécution cohérente et sans exception garantie.• usage abnormal, conversion forcée : (Gen) (Object) (new Gen<A>()) compile avec un warning et... provoque des erreurs à retardement à l'exécution (très mal, mais on a été prévenu) ! Exemple :	<pre>List<String> ls = new ArrayList<String>(); List<Integer> li = (List<Integer>)(Object) ls; //exécution ok ! (oh !) li.add(5); // toujours pas de crash... (double oh !) ls.get(0); // BOOM à retardement ! (ClassCastException)</pre>	<ul style="list-style-type: none">• Avec T, paramètre de type, new T[taille] est impossible.Raison : pour instancier un tableau, Java doit connaître dès la compilation le type concret des éléments du tableau.	<p>Or à la compilation, T n'est pas associé à un type concret.</p> <ul style="list-style-type: none">• Les types tableau de types paramétrés, comme List<Integer>[], sont illégaux.Raison : à l'exécution, Java ne sait pas distinguer List<Integer> et List<String> et donc ne peut pas accepter de mettre des List<Integer> dans un tableau sans aussi accepter List<String>.	<p>⇒ Tout ce qui est dans le tableau pouvant ensuite être affecté à une variable de type List<Integer>, la garantie promise par la généricité serait cassée.</p>											

Génériques et tableaux

Différence de philosophie (2)

Compléments en POO
Aldric Degorre

Génériques et tableaux

Deuxième interdit : `new T[10]` (1)

Génériques et tableaux

Premier interdit : `new T[10]`

Supposons `T extends Up` paramètre de type.

`new T[10]` est aussi interdit.

Raison : après compilation, `T` est oublié et remplacé par `Up`. Au mieux `new T[10]` pourrait être compilé comme `new Up[10]`. Mais si c'était ce qui se passait, on pourrait trop facilement « polluer » la mémoire sans s'en rendre compte :

```
static <T> T[] makeArray(int size) { return new T[10]; /* interdit ! */ }

String[] tString = makeArray(10); // à l'exécution on affacterait un Object[]

Object[] tobject = tString; // toujours autorisé (covariance)

tObject[0] = 42; // et BOOM ! Maintenant tString contient un Integer !
```

En pratique, pour faire compiler cela, il faut « tricher » avec cast explicite :

```
T[] tab = (T[]) new Up[10];
```

Ça n'empêche pas le problème ci-dessus, mais au moins le compilateur affiche un warning (« unchecked conversion »).

Compléments en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

définition :

introduction

Collections

Optimisation

Lambda-expressions

les "Streams"

Effacement de type

Invariance des

génériques vs.

covariance des

tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Mauvais scenario, pouvant se produire si on autorisait les tableaux de génériques :

```
class Box<T> {
    final T x;
    Box(T x) { this.x = x; }
}

class Loophole {
    public static void main(String[] args) {
        Box<String>[] bsa = new Box<String>[3];
        // supposons que cette ligne compile
        // autorisé car tableaux covariants

        /* autorisé à la compilation (Box < Object) le test à l'exécution est aussi ok
         * parce que le tableau référencé par oa est celui instancié à la première ligne
         * et que le type enregistré dans la JVM est juste Box */ /
        oa[0] = new Box<Integer>(3);

        /* ... et là , c'est le drame !
         * Willcards (ClassCastException), alors que l'instruction est bien typée ! */
        String s = bsa[0].x;
    }
}
```

Génériques et tableaux

Deuxième interdit : `new T[10]` (2)

Les génériques invariant c'est bien mais...

Invariance des génériques → garanties fortes : très bien, mais... très rigide à l'usage!

Le besoin : quand `B <: A` 353, on aimerait pouvoir écrire `Gen<A> g = new Gen();`.

- Cela favoriserait le polymorphisme (par sous-typage).
- On le fait bien avec les tableaux (`Object[] t = new String[10];`).
- C'est souvent conforme à l'intuition (cf. tableaux).

Mais on sait que ça risque d'être difficile :

- On a vu un contre-exemple pathologique (on provoque facilement `ClassCastException` si on force le compilateur à outrepasser l'invariance).
- On a vu les problèmes que posent les tableaux covariants (`ArrayListException` possible même dans programme sans `warning`).

353. À cause de l'effacement de type, `Box<Integer>` n'existe pas à l'exécution. Toutes les spécialisations ont le même type : `Box` !

Compléments en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

définition :

introduction

Collections

Optimisation

Lambda-expressions

les "Streams"

Effacement de type

Invariance des

génériques vs.

covariance des

tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Compléments en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

définition :

introduction

Collections

Optimisation

Lambda-expressions

les "Streams"

Effacement de type

Invariance des

génériques vs.

covariance des

tableaux

Wildcards

Concurrence

Interfaces graphiques

Annotations

Pour les quelques pages qui suivent, **oublions que javac impose l'invariance.**

Question : parmi les variables `x`, `y`, `z` et `t`, ci-dessous, lesquelles devrait-on, idéalement³⁵⁴, pouvoir affecter à quelles autres ?

```
class A {}  
class B extends A {}  
// Interface pour fonctions F -> U (extraite de java.util.function) :  
interface Function<T, U> { U apply(T t); }  
class Test {  
    Function<A, x> x;  
    Function<A, B> y;  
    Function<B, A> z;  
    Function<B, B> t;  
}
```

Le critère : on cherche les cas où une instance `Function<X, Y>` fournit au moins le service d'une instance de `Function<Z, T>`.

354. par exemple dans un langage où les générïques pourraient ne pas être invariants

Réponse : affecter `u` à `v` a un sens si la méthode `apply` de `u` peut remplacer celle de `v` (en toute situation). C.-à-d. :

- si elle accepte tous les paramètres effectifs acceptés par celle-ci
- et si les valeurs rentrées appartiennent à un type au moins aussi restreint.

(En résumé : une instance de `Function<X, Y>` peut remplacer une instance de `Function<Z, T>` si `X >: Z` et `Y <: T`.)
→ en appliquant ce principe, on voudrait donc que le compilateur accepte :

```
z = t; z = x; t = y; x = y; z = y;
```

Considérations autour de la variance (4)

Ainsi, inclusion des formes si et seulement s'il y a sous-typepage :

type de `expr2`



type de `expr1`



`varx = expr1; ?`
type de `varx` (type attendu, en négatif)



(L'encoche à gauche représente `T` et l'excroissance à droite, `U`.)

→ seul `varx = expr2;` doit fonctionner³⁵⁵ (pas de chevauchement) :

355. Attention, on ne parle pas de Java, mais seulement d'un système de type « idéal ».

Considérations autour de la variance (5)

Considérations autour de la variance (6)

Compléments en POO

Aldric Degoire

La variance souhaitée n'est donc pas la même pour tous les types génériques :

- intuitif et logique de vouloir `Function<Object, Integer> <: Function<Double, Number>`.

Justification : le premier paramètre de type est utilisé uniquement pour l'argument de `apply` alors que l'autre est uniquement son type de retour.

→ Emboîtement de `Function<T, U>` dans `Function<V, W>` possible dès que `T >: V` et `U <: W`.

Remarque : tailles de l'encoché et de l'excroissance de `Function<T, U>` indépendantes l'une de l'autre car elles représentent 2 paramètres différents. Si le même paramètre de type est utilisé en entrée et en sortie, ça ne marche plus (cf. page d'après).

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité : introduction

Collections

Opérations

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrency

Interfaces graphiques

Concurrence

Interfaces graphiques

- mais `List<Integer><:> List<Number>` serait illogique.
- Justification** : Le même paramètre apparaît à la fois en sortie (méthode `get`) et en entrée (méthodes `set` et `add`)³⁵⁶. Donc tailles de l'encoché et de l'excroissance de `List<T>` liées car représentant le même `T`
→ impossible d'encastrer la pièce de `List<x>` dans le trou `List<y>` si `y ≠ T`.

356. Même topo pour `Integer[] <: Number[] <:> Number[i]` avec les opérations `x = t[i]` et `t[i] = x` (... mais ça c'est autorisé : en contrepartie, il est nécessaire de faire des vérifications à l'exécution, avec risque de `ArrayStoreException`).

Encore un peu de vocabulaire autour de la variance (1)

Encore un peu de vocabulaire autour de la variance (2)

Compléments en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité : introduction

Collections

Opérations

Lambda-expressions

Les "Streams"

Effacement de type

Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrency

Interfaces graphiques

- 3 catégories de paramètres de type :
- **paramètre covariant** (comme `U`) : utilisé seulement en position covariante
→ plus le paramètre effectif est petit, plus le type paramétré devrait être petit;
 - **paramètre contravariant** (comme `T`) : utilisé seulement en position contravariante
→ plus le paramètre effectif est petit, plus le type paramétré devrait être grand;
 - **paramètre invariant** : utilisé à la fois en position covariante et contravariante.

Attention, ces concepts ne sont que théoriques.

Il se trouve que ceux-ci n'ont pas de sens pour le compilateur de Java : rappelez-vous qu'on avait dit que, pour l'instant, on oubliait l'invariance imposée par Java.

2 approches principales pour prendre en compte le phénomène de la variance :

- **annotations de variance sur site de déclaration** (n'existent pas en Java)

Variance définie (définitivement) dans la déclaration du type générique.

Exemple en langage Kotlin, on utilise **in** (contravariance) et **out** (covariance) :

```
interface Function<in T, out U> { fun apply(t: T) : U }
```

Annotations de variance sur site d'usage

Variance choisie lors de l'usage d'un type générique (dans déclarations de variables et signatures de méthodes).

C'est l'approche utilisée par Java, via le mécanisme des **wildcards**.

357. Le compilateur de Kotlin vérifie que les paramètres covariants (resp. contravariants) sont effectivement uniquement utilisés en position covariante (resp. contravariante).

(On revient enfin à Java !)

- Quand on écrit un type paramétré, les paramètres peuvent en fait être soit des types, soit le symbole « ? » (symbolisant un joker, un **wildcard**), parfois muni d'une **borne**.

- Les types paramétrés dont le paramètre est compatible avec la borne du **wildcard** se comportent alors comme des sous-types du type contenant le **wildcard**.
Ainsi `List<Integer>` est sous-type de `List<?>`.

Remarque : « ? » tout seul n'est pas un type. Ce caractère ne peut être utilisé que pour écrire un type paramétré (entre « < » et « > »).

Toute occurrence de « ? » peut se voir associer une borne.

Le principe est similaire aux bornes de paramètres de type, avec quelques différences :

- « ? » bornable à chaque usage (or, paramètres bornables juste à leur introduction).
- Les « ? » admettent des bornes supérieures (`T<?` **extends** `A`), **mais aussi** des bornes inférieures (`T<?` **super** `A`), imposant que toute concrétisation doit être un supertype de la borne.

- Pour un « ? », Java autorise une seule borne à la fois.

358. Si on veut plusieurs types concrèts comme bornes supérieures, il est possible de contourner cette limite en introduisant un type intermédiaire : **interface** `Borne` **extends** `Borne1, Borne2, ...`. Combiner plusieurs bornes inférieures concrètes (disons `A` et `B`) ne sert à rien : en effet `A` et `B` ont nécessairement un plus petit supertype commun, `C`, qui a un nom déjà connu quand on écrit le programme. `C` est le plus petit type contenant `A` et `B` (qui n'est pas un type de Java). Ainsi `T<?` **super** `C` serait équivalent à `? extends A & B` (syntaxe fictive).

Sinon, pour mixer des bornes qui sont elles-mêmes des paramètres, d'autres techniques basées sur l'introduction d'une variable de type supplémentaire sont envisageables.

Wildcards

Vérification du type

L'affectation suivante est-elle bien typée ?

```
List<? extends Serializable> l = new ArrayList<String>();
```

Pour savoir, on vérifie si le terme droite de l'affectation a un type compatible avec son emplacement.

Son type est **`ArrayList<String>`**, or le type attendu à cet emplacement est **`List<? extends Serializable>`** (= type de la variable à affecter).

- D'une part, **`String`** satisfait la borne de ? (**`String`** implémente **`Serializable`**) et, d'autre part, **`ArrayList<String>`** <: **`List<String>`**.

Donc cette affectation est bien typée.

Wildcards

Compléments
en POO

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité :

 Introduction

 Collections

 Options

 Lambda-expressions

 Les "Streams"

 Effacement de type

 Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Compléments
en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité :

 Introduction

 Collections

 Options

 Lambda-expressions

 Les "Streams"

 Effacement de type

 Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Généralisons :

- Soit une expression **`expr`** utilisées dans un certain contexte (appel de méthode, affectation, ...).
- Soit **`TE`** le type (déjà « converti par capture »³⁵⁹, si applicable) de **`expr`**.
- Supposons que le type attendu dans le contexte soit de la forme **`TA<?`** **`borne`**.
- Alors il est légal d'utiliser **`expr`** à cet endroit si et seulement si il existe un type **T** satisfaisant **`borne`**, tel que **`TE <: TA<T>`**.

359. Explication un peu plus loin. Ceci concerne le cas où le type de l'expression contient un ?.

Compléments
en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité :

 Introduction

 Collections

 Options

 Lambda-expressions

 Les "Streams"

 Effacement de type

 Invariance des génériques vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

En réalité, pour une expression, avoir le type **`Gen<?>`** veut dire qu'il **existe**³⁶⁰ un type **`Gen<?>`** (inconnu mais fixé) tel que cette expression est de type **`Gen<?>`**.

Il faut interpréter le type d'une expression à **wildcards** comme un type inconnu appartenant à l'ensemble des types respectant les contraintes trouvées.

Wildcards
utilisation pour signaler la variance

Pour revenir au problème initial, reprenons notre exemple :

```
interface Function<?, U> { U apply(T t); }
class A {}
class B extends A {}
class Test { Function<A, A> x; Function<A, B> y; Function<B, A> z; Function<B, B> t; }

U → position covariante; T → position contravariante. On « assouplit » donc Test :
```

```
class Test2 {
    Function<?, super A, ? extends A> x;
    Function<?, super B, ? extends A> y;
    Function<?, super A, ? extends B> z;
    Function<?, super B, ? extends B> t;
}
```

Maintenant, les affectations qu'on voulait écrire sont acceptées par le compilateur :

```
z = t; z = x; t = y; x = y; // vérifiez !
/* et aussi... */ a = B; a = x. apply(a); b = y. apply(b); a = z. apply(b); b = t. apply(b);
```

Recette : position covariante → **extends**, position contravariante → **super**.

Se rappeler **PECS** : « producer extends, consumer super ».

Inversez **super** et **extends** et vérifiez que les appels à **apply** ne fonctionnent plus.

360. Et comme on ne sait rien de ce type, vérifier que l'expression est à sa place c'est vérifier qu'elle est à sa place **pour toute** valeur de **QuelqueChose**.

Wildcards

Conversion par capture (2)

Compléments
en POO

Wildcards

Conversion par capture (3)

Concrètement, lors de la vérification de type d'une expression, le compilateur effectue une opération appelée **conversion par capture** 361 :

- À chaque fois qu'un « ? » apparaît au premier niveau 362 du type d'une expression 363, le compilateur le remplace par un nouveau type créé à la volée, recevant un nom de la forme **capture#1-of?** (**capture** de **wildcard**).
- Quand une telle capture est créée, le compilateur se souvient des bornes du « ? » qu'elle remplace (il peut s'en servir dans la suite de l'analyse de types).

361. Pour les logiciens, cette transformation s'apparente à la skolemisation : on remplace une variable quantifiée existentiellement par un nouveau symbole.

362. On ne regarde pas en profondeur : **List<?>** **extends** **Set<capture#1-of?>**, le compilateur se rappelant que **capture#1-of?** est sous-type de **Set<?>**.
363. Cela se produit quand l'expression est une variable typée avec des ?, un appel de méthode dont le type de retour contient des ?, ou une expression castée vers un tel type.

Compléments
en POO

Wildcards

Style

Aléric Degoire

Wildcards

Exemple pathologique

Wildcards

Wildcards

Quid des exemples plus complexes ? (1)

Wildcards

Exemple pathologique

```
List<? super String> l = new ArrayList<?>(); l.add("toto"); //ok
l.add(l.get(0)); // Mal typé ! Mais pourquoi ?
```

Explication : à la 2e ligne,

- la 1e occurrence de **l** est de type **List<capture#1-of?>** \Rightarrow **l.add(...)** attend un paramètre de type **capture#1-of?** ;
- la 2e occurrence de **l** est de type **List<capture#2-of?>** (**capture indépendante**) \Rightarrow **l.get(0)** est de type **capture#2-of?** ;
- or **capture#1-of?** et **capture#2-of?** sont, du point de vue du compilateur, deux types quelconques sans lien de parenté, d'où l'erreur de type.

On peut contourner en forçant une capture anticipée (via méthode auxiliaire) :

```
// méthode auxiliaire. Ici, tout est ok, car l.get(0) de type T, or l.add() prend du T.
<T> static void aux(List<?> l) { l.add(l.get(0)); }
// plus loin
List<? super String> l = new ArrayList<?>(); l.add("toto"); // encore ok
```

Les « ? » peuvent apparaître à différentes profondeurs, y compris dans les bornes :

```
List<A? super String> las = new ArrayList<?>();
List<?> extends A? super Integer> lar = las;
```

Pour chaque niveau de **>>** on vérifie que le type (resp. l'ensemble de types) donné correspond à un élément (resp. un sous-ensemble) de l'ensemble attendu.

```
List<A<? super String> las = new ArrayList<();  
List<? extends A<? super Integer>> lar = las;
```

Dans l'exemple (2e ligne, à droite de =) :

- On veut comparer `List<A<?` **super String>** `las` = `new ArrayList<();`
- **extends** `A<?` **super Integer>** (type attendu = celui de `lar`)
- Au premier niveau, on a `List` et `List` → OK, vérifions les paramètres.
- Il faut que `A<?` **super String>** <: `A<?` **super Integer>**.
- On a `A` des 2 côtés... jusqu'à tout va bien. Vérifions l'inclusion des paramètres.
- À l'intérieur on attend « `? super Integer` », mais on reçoit « `? super String` ».
- Les deux bornes sont dans le même sens, c'est bon signe.
- Malheureusement, on n'a pas `String` >`:> Integer`. Donc « `? super String` » n'est pas inclus dans « `? super Integer` » (p. ex. : le 1er ensemble contient `String` mais pas le `2e`). Donc erreur de type !

Definition (Concurrency)

Deux actions, instructions, travaux, tâches, processus, etc. sont **concurrents** si leurs exécutions sont **indépendantes** l'une de l'autre (l'un n'attend pas de résultat de l'autre).

- **Conséquence** : deux actions concurrentes peuvent s'exécuter simultanément, si la plateforme d'exécution le permet.
- Un **programme concurrent** est un programme dont certaines portions de code sont indépendantes les unes des autres et tel que la plateforme d'exécution sait 364 exploiter ce fait pour optimiser l'exécution. 365 .

364. Le plus souvent, cette connaissance nécessite que les portions concurrentes soient signalées dans le code source.
365. Notamment en exécutant simultanément, en parallèle, ces portions de code si c'est possible.

Introduction
Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

introduction

Concurrence

Concurrence et parallélisme

Les astuces

Théorie en Java

Dominique le JMM

API de haut niveau

Interfaces

graphiques

Gestion des

erreurs et

exceptions

Parallélisme

Deux travaux 368 s'exécutent en parallèle, si ils exécutent en même temps.

- **Simultanéité** au niveau le plus bas : si 2 travaux s'exécutent en parallèle, à un instant t, s'exécutent en même temps une instruction de l'un et de l'autre.
- → exécution sur 2 lieux physiques différents (e.g. 2 coeurs, 2 circuits, ...).
- **Degré de parallélisme** 369 = nombre de travaux simultanément exécutables.

Pour des raisons économiques et technologiques, les microprocesseurs modernes (multi-cœur 370) ont typiquement un degré de parallélisme ≥ 2.

C'est une opportunité qu'il faut savoir saisir !

Util de programmer ces algorithmes de façon concurrente car on peut profiter du parallélisme pour les accélérer (cf. page suivante).

366. Et pas seulement en programmation, mais c'est le sujet qui nous intéresse !

367. Et discutablement naturelle aussi.

368. Pour ne pas dire « processus », qui a un sens un peu trop précis en informatique.

369. D'une plateforme d'exécution.

370. Sur les CPU de moyenne et haut de gamme, le degré de parallélisme est généralement de 2 par cœur, grâce au SMT (simultaneous multithreading), appelé hyperthreading chez Intel

Concurrence			
Enjeu	Compléments en POO	Enjeu	Compléments en POO
Ainsi, l'enjeu de la programmation concurrente est double : <ul style="list-style-type: none"> • Nécessité : pouvoir programmer des fonctionnalités intrinsèquement concurrentes (serveur web, IG, etc.). • Opportunisme : tirer partie de toute la puissance de calcul du matériel contemporain. En effet : des travaux indépendants (concurrents) peuvent naturellement être confiés à des unités d'exécution distinctes (parallèles).	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence et parallelisme Les abstractions Threads en Java Domptez le JMM APIs de haut niveau Interfaces graphiques Gestion des erreurs et exceptions	Exécuter deux travaux réellement concurrents en parallèle est facile ³⁷¹ , mais la réalité est souvent plus compliquée : <ul style="list-style-type: none"> • Si (degré de) concurrence > (degré de) parallélisme, alors partage du temps (→ ordonnanceur nécessaire). • Concurrence de 2 sous-programmes jamais parfaite³⁷² car nécessité de se transmettre/partager des résultats et de se synchroniser.³⁷³ → Différentes abstractions pour aider à programmer de façon correcte et, si possible, intuitive, tout en prenant en compte ces réalités de diverses façons.	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence et parallelisme Les abstractions Threads en Java Domptez le JMM APIs de haut niveau Interfaces graphiques Gestion des erreurs et exceptions
<p>Un ordonnanceur est un programme chargé de répartir les tâches concurrentes sur les unités d'exécutions disponibles. Il s'agit souvent d'un sous-système du noyau de l'OS³⁷⁴.</p> <p>L'ordonnanceur peut mettre en œuvre :</p> <ul style="list-style-type: none"> • un fonctionnement multi-tâches préemptif : l'ordonnanceur choisit quand mettre en pause une tâche pour reprendre l'exécution d'une autre. Cela peut arriver (presque) à tout moment. <p>C'est le cas pour la gestion des processus dans les OS modernes pour ordinateur personnel.</p> <ul style="list-style-type: none"> • ou bien un fonctionnement multi-tâches coopératif : chaque tâche signale à l'ordonnanceur quand elle peut être mise en attente (par exemple en faisant un appel bloquant). 	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence et parallelisme Les abstractions Threads en Java Domptez le JMM APIs de haut niveau Interfaces graphiques Gestion des erreurs et exceptions	<h2>Questions d'ordonnancement</h2> <p>Multi-tâche préemptif vs. coopératif</p> <p>Un ordonnanceur est un programme chargé de répartir les tâches concurrentes sur les unités d'exécutions disponibles. Il s'agit souvent d'un sous-système du noyau de l'OS³⁷⁴.</p> <p>L'ordonnanceur peut mettre en œuvre :</p> <ul style="list-style-type: none"> • un fonctionnement multi-tâches préemptif : l'ordonnanceur choisit quand mettre en pause une tâche pour reprendre l'exécution d'une autre. Cela peut arriver (presque) à tout moment. <p>C'est le cas pour la gestion des processus dans les OS modernes pour ordinateur personnel.</p> <ul style="list-style-type: none"> • ou bien un fonctionnement multi-tâches coopératif : chaque tâche signale à l'ordonnanceur quand elle peut être mise en attente (par exemple en faisant un appel bloquant). 	
<h2>Questions de synchronisation</h2> <p>Transmettre des résultats d'une tâche à l'autre</p> <p>Plusieurs techniques de transmission de résultats :</p> <ul style="list-style-type: none"> • variables partagées : variables accessibles par plusieurs tâches concurrentes. Données partagées de façon transparente, sans synchronisation a priori, mais le langage permet d'insérer des primitives de synchronisation explicite³⁷⁵. • passage de message : données « envoyées »³⁷⁶ d'une tâche à l'autre. Synchronisation implicite de l'émission et de la réception du message : par exemple, une tâche en attente de réception est <u>bloquée</u> tant qu'elle n'a rien reçu.³⁷⁷ <p>La réalité physique est plus proche du modèle des variables partagées³⁷⁸, mais le passage de message est un paradigme plus sûr³⁷⁹.</p> <p>375. En Java : <code>start()</code>, <code>join()</code>, <code>volatile</code>, synchronized et <code>wait()</code>/<code>notify()</code>.</p> <p>376. Sous-entendu : l'envoyeur ne peut plus accéder à ce qui a été envoyé.</p> <p>377. C'est une possibilité. On peut aussi bloquer la tâche émettrice (canal borné, « rendez-vous »).</p> <p>378. Mémoire centrale lisible par plusieurs CPU.</p> <p>379. Pour lequel la sûreté d'un programme est plus facile à prouver.</p>			
<h2>374. Operating System/système d'exploitation</h2>			

Mais on peut simuler le passage de message :

```
public final class MailBox<T> { // classe réutilisable, simulant un passage de message avec "rendez-vous"
    private T content; // mémoire partagée, encapsulée
    public synchronized void sendMessage(T message) throws InterruptedException {
        while (content != null) wait(); // attend la condition content = null
        content = message;
        notifyAll();
    }
    public synchronized T receiveMessage() throws InterruptedException {
        while (content == null) wait(); // attend la condition content = null
        T ret = content;
        content = null;
        notifyAll();
        return ret;
    }
}

public final class PingPong {
    public static void main(String[] args) {
        var box = new MailBox<String>();
        new Thread(() -> {
            try {
                box.sendMessage("ping!");
            } catch (Exception e) { throw new RuntimeException(e); }
        }).start(); // producteur/écrivain
        new Thread(() -> {
            try {
                while (true) System.out.println((box.receiveMessage() == "pong") ? "pong!" : "error!");
            } catch (Exception e) { throw new RuntimeException(e); }
        }).start(); // consommateur/lecteur
    }
}
```

Definition (Thread ou fil d'exécution)

Abstraction concurrente consistant en une séquence d'instructions dont l'exécution **simule une exécution séquentielle** (en interne)³⁸¹ et **parallèle** à celle des autres threads.

- passer **x** en paramètre d'un appel de méthode. Par exemple, on peut soumettre la valeur à une file d'attente synchronisée :

```
... // calcule x
queue.offer(x);
... // fait autre chose (avec interdiction de toucher à x !)
```

Dans ce dernier cas, la tâche consommatrice reçoit le message en appelant **queue.take()** (fonction bloquante).

- Remarque : cela est similaire à l'exemple de la classe **MailBox** donné plus tôt³⁸⁰.

³⁸⁰. Différence : **MailBox** ne stocke qu'un seul message (= « Rendez-vous »), alors qu'une file d'attente peut en stocker plusieurs, permettant au consommateur et au producteur de ne pas suivre le même rythme.

Un nombre quelconque de threads s'exécute sur une plateforme de degré de parallélisme quelconque³⁸². Un ordonnanceur partage les ressources de la plateforme pour que cela soit possible.

- Ainsi, **n threads** en exécution simultanée **simulent un parallélisme de degré n**
- Un **processus**³⁸³ (= 1 application en exécution) peut utiliser plusieurs threads qui ont accès aux mêmes données (mémoire partagée).

³⁸¹. Ce qui permet de le programmer avec les principes habituels de programmation impérative : séquences d'instructions, boucles, branchements, pile d'appels de fonctions, ...

³⁸². Même inférieur au nombre de threads

³⁸³. Cette fois-ci au sens où on l'intend en informatique.

Exemple de deux threads, l'un qui compte jusqu'à 10 alors que l'autre récite l'alphabet :

```
class ReciteNombres extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
    }
}

class ReciteAlphabet extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 26; i++)
            System.out.print((char)('a'+i) + " ");
    }
}
```

Compléments en POO

Aldric Degoire

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Introduction

Concurrence et parallelisation

Les abstractions

Threads en Java

Dominique le JMM

API de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

Exemple

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Introduction

Concurrence et parallelisation

Les abstractions

Threads en Java

Dominique le JMM

API de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

Exemple

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Introduction

Concurrence et parallelisation

Les abstractions

Threads en Java

Dominique le JMM

API de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

- **Dans le runtime des langages de programmation** : des langages de programmation (*Erlang*, *Go*, *Haskell*, *Lua*, ...), mais pas actuellement Java, contiennent une notion de *thread* « léger » (différents noms : *green thread*, *fibre*, *coroutine*, *goroutine*, ...), s'exécutant par dessus un ou des *threads* système.

Langage/runtime

OS (noyau)

Abstractions (fibres, coroutines, acteurs, futurs, événements, ...)

thread | ...

Ordonnanceur

Proc. logique | Proc. logique | Proc. logique | Proc. logique

SMT | SMT | Cœur | Cœur

Cœur | CPU 388

384. On parle typiquement de milliers, pas de millions. La limite pratique est la mémoire disponible.

385. réels ou simulés, cf. *hyperthreading*

386. Contexte = pointeur de pile, pointeur ordinal, différents registres...

387. Sous-entendu : « *thread système* » « *thread* » sans précision = « *thread système* ».

388. Possible aussi : plusieurs CPUs (plusieurs cœurs par CPU, plusieurs processeurs logiques par cœur...)

À propos des <i>threads</i> système		Des APIs de haut niveau pour ne pas manipuler les <i>threads</i>	
Compléments en POO	Aldric Degoire	Compléments en POO	Aldric Degoire
Introduction		Introduction	
Généralités		Généralités	
Style		Style	
Objets et classes		Objets et classes	
Types et polymorphisme		Types et polymorphisme	
Héritage		Héritage	
Généricité		Généricité	
Concurrency		Concurrency	
Introduction		Introduction	
Concurrente et parallèle		Concurrente et parallèle	
Les abstractions		Les abstractions	
Threads en Java		Threads en Java	
Départ du JMM		APIs de haut niveau	
Interfaces graphiques		Interfaces graphiques	
Gestion des erreurs et exceptions		Gestion des erreurs et exceptions	
Avantages et inconvénients		Avantages et inconvénients	
<ul style="list-style-type: none"> Ce sont des <i>threads</i>. Avantage : se programme séquentiellement (respectent les habitudes). Inconvénient : la synchronisation doit être explicitée par le programmeur. 389 Multi-tâche préemptif : l'ordonnanceur peut suspendre un <i>thread</i> (au profit d'un autre), à tout moment Avantage : pas besoin de signaler quand le programme doit « laisser la main ». Inconvénient : changements de contexte fréquents et coûteux. Implémentation dans le noyau : Avantage : compatible avec tous les exécutables de l'OS (pas seulement JVM) Inconvénient : fonctionnalités rudimentaires. P. ex., chaque <i>thread</i> a une pile de taille fixe (1024 ko pour les <i>threads</i> de la JVM 64bits) → peu économique! 		<p>→ les langages de programmation proposent des mécanismes, utilisant les <i>threads</i> système, pour pallier leurs inconvénients tout en essayant 390 de garder leur avantages.</p> <p>Au moins deux objectifs :</p> <ul style="list-style-type: none"> limiter le nombre de <i>threads</i> système utilisés, afin de diminuer l'empreinte mémoire et la fréquence des changements de contexte forcer des procédés sûrs pour le partage de données; ou à défaut, faciliter les bonnes pratiques de synchronisation. <p>390. avec plus ou moins de succès</p>	
Compléments en POO	Aldric Degoire	Compléments en POO	Aldric Degoire
Introduction		Introduction	
Généralités		Généralités	
Style		Style	
Objets et classes		Objets et classes	
Types et polymorphisme		Types et polymorphisme	
Héritage		Héritage	
Généricité		Généricité	
Concurrency		Concurrency	
Introduction		Introduction	
Concurrente et parallèle		Concurrente et parallèle	
Les abstractions		Les abstractions	
Threads en Java		Threads en Java	
Départ du JMM		APIs de haut niveau	
Interfaces graphiques		Interfaces graphiques	
Gestion des erreurs et exceptions		Gestion des erreurs et exceptions	
<p>389. On va voir dans la suite comment Pour l'instant, retenez qu'il n'y a aucune synchronisation, donc aucun partage de données sûr entre <i>threads</i> si on n'ajoute pas « quelque chose ».</p>		<p>390. avec plus ou moins de succès</p>	
La situation de Java		Threads en Java	
Java	Aldric Degoire	Java	Aldric Degoire
Introduction		Introduction	
Généralités		Généralités	
Style		Style	
Objets et classes		Objets et classes	
Types et polymorphisme		Types et polymorphisme	
Héritage		Héritage	
Généricité		Généricité	
Concurrency		Concurrency	
Introduction		Introduction	
Threads en Java		Threads en Java	
La classe Thread		La classe Thread	
Synchronisation		Synchronisation	
Départ du JMM		APIs de haut niveau	
Interfaces graphiques		Interfaces graphiques	
Gestion des erreurs et exceptions		Gestion des erreurs et exceptions	
<ul style="list-style-type: none"> utilise directement les <i>threads</i> système, via la classe Thread. a historiquement (Java 1.1) utilisé des <i>green threads</i> 391, abandonnés pour des raisons de performance 392. pourrait néanmoins, dans le futur, supporter les fibres 393 via le projet Loom. dispose actuellement d'un grand nombre d'APIs facilitant où rendant plus sûre l'utilisation des <i>threads</i> : les boucles d'événements Swing et JavaFX, ThreadPoolExecutor, ForkJoinPool/ForkJoinTask, CompletableFuture, Stream... 		<ul style="list-style-type: none"> 1 <i>thread</i> est associé à 1 pile d'appel de méthodes <i>Thread</i> principal en Java = pile des méthodes appelées depuis l'appel initial à main() → vous utilisez déjà des threads ! Interfaces graphiques (Swing, JavaFX, ...) : un <i>thread</i> 394 ($\neq \text{main}$) est dédié aux événements de l'IG : <ul style="list-style-type: none"> Programmation événementielle → méthodes gestionnaires d'événement Événements → mis en file d'attente quand ils surviennent. Quand le <i>thread</i> des événements est libre, le gestionnaire correspondant au premier événement de la file est appellé et exécuté sur ce <i>thread</i>. Intérêt : 395 pas besoin de prévoir des interruptions régulières dans le <i>IG</i> resterait figée entre deux pour vérifier et traiter les événements en attente (le <i>thread main</i> pour Swing : Event Dispatching Thread (EDT). Pour JavaFX : JavaFX Application Thread.) 	
Compléments en POO	Aldric Degoire	Compléments en POO	Aldric Degoire
Introduction		Introduction	
Généralités		Généralités	
Style		Style	
Objets et classes		Objets et classes	
Types et polymorphisme		Types et polymorphisme	
Héritage		Héritage	
Généricité		Généricité	
Concurrency		Concurrency	
Introduction		Introduction	
Concurrente et parallèle		Concurrente et parallèle	
Les abstractions		Les abstractions	
Threads en Java		Threads en Java	
Départ du JMM		APIs de haut niveau	
Interfaces graphiques		Interfaces graphiques	
Gestion des erreurs et exceptions		Gestion des erreurs et exceptions	
<p>391. Une sorte de <i>threads</i> légers.</p> <p>392. Ils étaient ordonnancés sur un seul <i>thread</i> système, empêchant d'utiliser plusieurs processeurs.</p> <p>393. Autre type de <i>threads</i> légers. Cette fois-ci, le travail peut être distribué sur plusieurs <i>threads</i> système. Des implémentations de fibres pour Java existent déjà : bibliothèques Quasar et Kilim. Mais pour fonctionner, celles-ci doivent modifier le code-octet généré par javac.</p>		<p>394. Pour Swing : Event Dispatching Thread (EDT). Pour JavaFX : JavaFX Application Thread.</p> <p>395. Et l'intérêt de n'avoir qu'un seul <i>thread</i> pour cela : la sûreté du fonctionnement de l'IG. Pas d'entreclacements entre 2 événements, pas d'accès compétition.</p>	

- Tous les *threads* ont accès au même *tas* (mêmes objets) et à la même *zone statique* (mêmes classes)... mais pas à la même *pile*!
- Les *threads* communiquent grâce aux **variables partagées**, stockées dans le *tas*.
- Une même méthode peut être appelée depuis n'importe quel *thread* (pas de séparation syntaxique du code associé aux différents *threads*).
- Pour démarrer un *thread* : un *ObjetThread*.**start()** ; (où un *ObjetThread* instance de la classe *Thread*).
- aussitôt, appel de un *ObjetThread*.**run()** dans le *thread* associé à cet objet.
- A chaque *thread* correspond une pile d'appels de méthode. En bas de la pile :
 - pour le *thread main*, le *frame* de la méthode **main**;
 - pour les autres, celui de l'appel initial à **run** sur l'objet représentant le *thread*.

- Définir et instancier une classe héritant de la classe *Thread* :

```
public class HelloThread extends Thread {
    @Override public void run() { System.out.println("Hello from a thread!"); }
    // plus loin
    new HelloThread().start();
```

- Implémenter *Runnable* et appeler le constructeur *Thread(Runnable target)* :

```
public class HelloRunnable implements Runnable {
    @Override public void run() { System.out.println("Hello from a thread!"); }
    // plus loin
    new Thread(new HelloRunnable()).start();
```

- Mais pour un *thread* simple, on préférera écrire une lambda-expression :

```
new Thread(() -> { System.out.println("Hello from a thread!"); }).start();
```

La classe Thread

- String **getNome()** : récupérer le nom d'un *thread*.
- void join()** : attendre la fin de ce *thread* (voir synchronisation).
- void run()** : la méthode qui lance tout le travail de ce *Thread*. C'est la méthode qui il faudra redéfinir à chaque fois que *Thread* sera étendue !.

```
static void sleep(long millis) : met le thread courant (i.e. en cours d'exécution) en pause pendant tant de ms. (NB : c'est une méthode static). Le thread mis en pause est celui qui appelle la méthode. Il n'y a pas de this!).
```

- void start()** : démarre le *thread* (conséquence : **run()** est exécutée dans le nouveau *thread* : celui décrit par l'objet, pas celui de l'appelant!). ..
- void interrupt()** : interrompt le *thread* (déclenche **InterruptedException**) si le *thread* était en attente sur **wait()**, **join()**, **sleep()**...).

```
static boolean interrupted() : teste si un autre thread a demandé l'interruption du thread courant.
```

- Thread.State getState()** : retourne l'état du *thread*.

La classe Thread

États d'un thread

Compléments
en POO

Aldric Degorre

Une instance de *thread* est toujours dans un des états suivants :

- **NEW** : juste créé, pas encore démarré.
- **RUNNABLE** : en cours d'exécution.
- **BLOCKED** : en attente de moniteur (voir la suite).
- **WAITING** : en attente d'une condition d'un autre *thread* (voir `notify()`/`wait()`).
- **TIME_WAITING** : idem pour attente avec temps limite.
- **TERMINATED** : exécution terminée.

Mais attendons la suite pour en dire plus sur ces états...

La classe Thread

Interruptions (1)

Compléments
en POO

Aldric Degorre

- Si *t* est un *thread*, l'appel `t.interrupt()` demande l'interruption de celui-ci.
- Si *t* est en train d'exécuter une méthode interruptible³⁹⁷, celle-ci quitte tout de suite.

- L'interruption est propagée le long des méthodes de la pile d'appel qui quittent une à une... jusqu'à la méthode principale de la tâche³⁹⁸ qui quitte aussi.
 - Le résultat (non garanti³⁹⁹) est la terminaison de la tâche exécutée sur *t*⁴⁰⁰.
 - La propagation de l'interruption est implémentée par la propagation de l'exception `InterruptedException` et par le contrôle du booléen `Thread.interrupted()` (détails juste après).
-
397. C'est le cas de toutes les méthodes bloquantes de l'API `Thread.wait()`, `sleep()`, `join()` ...
398. Habituellement : `run()`.
399. Si les méthodes exécutées sur *t* n'ont pas prévu d'être interrompues, rien ne se passe.
400. Si exécution directe dans le *thread*, terminaison du *thread*, sinon, si exécution dans un *thread pool*, le *thread* est juste rendu de nouveau disponible.

La classe Thread

Interruptions (3)

Compléments
en POO

Aldric Degorre

Pour écrire une méthode interruptible *f* :

- Quand une interruption est détectée la bonne pratique est de quitter (`return` ou `throw`) au plus tôt, tout en libérant les ressources utilisées.
- L'interruption peut être détectée de deux façons :
 - soit une méthode auxiliaire *g* appelée depuis *f* quitte sur `InterruptedException`
 - soit on a obtenu `true` en appelant `Thread.interrupted()`.
- Le premier cas (exception) doit être traité en mettant tout appels à *g* dans un block `try-finally` (libération explicite des ressources de *f* dans le `finally`) ou bien `try-with-resource` (libération implicite).
- Remarque : il faut absolument vérifier `Thread.interrupted()` dans toute boucle de *f* ne faisant pas d'appel à une méthode interruptible comme *g*.
- Dans tous les cas, il faut veiller à propager le statut « interrompu » au contexte d'exécution, pour qu'il puisse, lui aussi, prendre en compte le fait qu'une interruption a eu lieu. 2 cas de figure (voir la suite).

La classe Thread

Interruptions (2)

Compléments
en POO

Aldric Degorre

2 cas de figure, selon que la signature de *f* est imposée ou non :

- Si ce n'est pas le cas, on propage le statut « interrompu » en quittant sur `InterruptedException`. 2 cas de figure :
 - si une méthode appelée depuis *f* a elle-même lancé `InterruptedException` dans ce cas on ne met pas de `catch` et la propagation est automatique.
 - sinon, on peut ajouter `throw new InterruptedException();`
- `InterruptedException` étant une exception sous contrôle, il faut aussi ajouter `throws InterruptedException` à la signature de *f*.
- Si la signature de *f* est imposée par l'interface implémentée (ex : `Runnable`) et ne contient pas `throws InterruptedException`, on ne peut alors pas quitter sur `InterruptedException`.
- Si non, si la signature de *f* satisfont à la signature de `Runnable`

Compléments
en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Introduction

Threads en Java

Introduction

La classe Thread

Synchronisation

Demande de MM

API de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

Interruptions (4)

Aldric Degoire

Compléments
en POO**Exemples de méthodes interruptibles :**

```
// avec while et acquisition/libération de resource (bloc "try-with-resource")
Data f(Data x) throws InterruptedException {
    try (Scanner s = new Scanner(System.in)) {
        while(test(x)) {
            x = transform(x, s.next());
            if (Thread.interrupted()) throw new InterruptedException(); // <-- ici !
            return x;
        } // s.close() appelée implicitement à la sortie du bloc (par throw ou par return)
    } // exemple sans boucle, mais avec appel bloquant
    void sleep5s() throws InterruptedException {
        System.out.println("Acquisition potentiellement de ressource");
        try {
            Thread.sleep(5000); // on attend 5s
        } finally { System.out.println("Libération de la même ressource"); }
    } // Pas de "catch". Si sleep() envoie InterruptedException, elle est propagée.
}
```

Exemple

Deux principaux problèmes :

- 1 Les entrelacements non maîtrisés :** les instructions de 2 threads **s'entrelacent** 401 et accèdent (lecture et écriture) aux mêmes données dans un ordre imprévisible. Ce phénomène est « naturel » (l'ordonnanceur est libre de faire avancer un *thread*, puis l'autre au moment où il veut) ; il est parfois gênant, parfois non.
- 2 Les incohérences dues aux optimisations matérielles** 402 : la JVM 403 laisse une marge d'interprétation assez large au matériel pour qu'il puisse exécuter le programme efficacement. Principales conséquences :
- ordre des instructions donné dans le code source pas forcément respecté
 - modifications de variables partagées pas forcément vues par les autres *threads*.

Pour l'instant, concentrons nous sur le problème 1.

401. *interleave*
402. en particulier dans le microprocesseur
403. La JVM s'appuie sur le **JMM** : Java Memory Model, un modèle d'exécution relativement laxe.

Compléments
en POO

Aldric Degoire

Introduction

Entrelacements de threads

Exemple (1)

Exemple (2)

Qu'est-ce qui est affiché quand on exécute le programme suivant ?

```
public class ThreadInterferences extends Thread {
    static int x = 0;
    public ThreadInterferences(String name){ super(name); }
    @Override
    public void run(){
        while(x++ < 10) System.out.println("x[incrémenté par " + x + "]. " +
            nouvelle valeur est " + x + ".");
    }
    public static void main(String[] args){
        new ThreadInterferences("t1").start();
        new ThreadInterferences("t2").start();
    }
}
```

On s'attend à voir tous les entier de 1 à 10 s'afficher dans l'ordre.

Exécution possible :

```
x incrémenté par t2, sa nouvelle valeur est 2.
x incrémenté par t1, sa nouvelle valeur est 2.
x incrémenté par t2, sa nouvelle valeur est 3.
x incrémenté par t1, sa nouvelle valeur est 4.
x incrémenté par t2, sa nouvelle valeur est 5.
x incrémenté par t1, sa nouvelle valeur est 6.
x incrémenté par t2, sa nouvelle valeur est 7.
x incrémenté par t2, sa nouvelle valeur est 9.
x incrémenté par t2, sa nouvelle valeur est 10.
x incrémenté par t1, sa nouvelle valeur est 8.
```

Contrairement à ce qu'on pourrait attendre : les nombres ne sont pas dans l'ordre, certains se répètent, d'autres n'apparaissent pas.

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Introduction

Threads en Java

La classe Thread

Synchronisation

Demande de MM

API de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

Exemple

Entrelacements de threads

Exemple (2)

Avec quelle granularité les entrelacements se font-ils ? Peut-on s'arrêter au milieu d'une affectation, faire autre chose sur la même variable, puis finir ? → notion clé : **atomicité**

- **Atomique** = non séparable (étym.), non entrelaçable (ici).

Aucune autre instruction, accédant aux mêmes données, ne peut être exécutée pendant celle des instructions d'une opération atomique.

- Quelques exemples d'opérations atomiques :

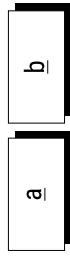
- lecture ou affectation de valeur 32 bits (**boolean, char, byte, short, int, float**) ;
- lecture ou affectation de référence (juste la référence, pas le contenu de l'objet) ;
- lecture ou affectation d'un attribut **volatile**⁴⁰⁴ ;
- exécution d'un bloc **synchronized**⁴⁰⁵

- Exemple d'opération non atomique : **x++** (peut se décomposer ainsi : copie **x** en pile, empile 1, additionne, copie le sommet de pile dans **x**).

404. Notion abordée plus loin.
405. Idem. Dans ce cas, remplacer « accédant aux mêmes données » par « utilisant le même verrou ».

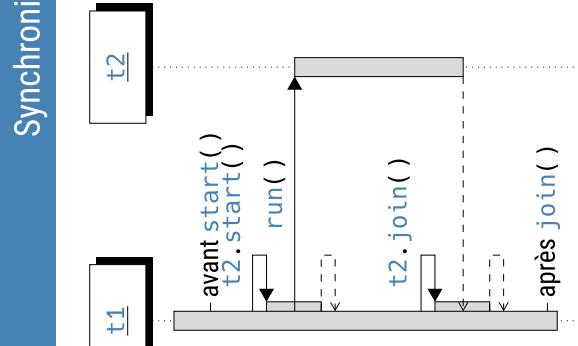
Synchronisation :

- consiste, pour un **thread**, à attendre le « feu vert » d'un autre **thread** avant de continuer son exécution;
- interdit certains entrelacements ;
- contribue à établir la relation "arrivé-avant", limitant les optimisations autorisées⁴⁰⁶.



Ici, **a** arrive avant **b**, **b** arrive avant **a**, mais pas **b1** avant **a2** !

406. À suivre...



affiche 0 alors que le même code sans l'appel à **join()** affichera probablement 1.

Tout ce qui est exécuté dans le **thread t** arrive-avant ce qui suit le **join()** dans le **thread main** (ici, l'incrémentation de **x**).

407. Existe aussi en version temporisée : on bloque jusqu'au délai donné en paramètre maximum.

- En Java tout objet contient un **verrou intrinsèque** (ou **moniteur**).
- À tout moment, le moniteur est soit libre, soit détenu par un (seul) *thread* donné.
- Ainsi un moniteur met en œuvre le principe d'exclusion mutuelle.
- Lors de son exécution, un *thread* **t** peut demander à prendre un moniteur.
 - S'il le moniteur est libre, alors il est pris par **t**, qui continue son exécution.
 - S'il le moniteur est déjà pris, **t** est alors mis en attente jusqu'à ce que le moniteur se libère pour lui (il peut y avoir une liste d'attente).
- Un *thread* peut à tout moment libérer un moniteur qu'il possède.

Conséquence : tout ce qui se produit dans un *thread* avant qu'il libère un moniteur arrive avant ce qui se produit dans le prochain *thread* qui obtiendra le moniteur, après l'obtention de celui-ci.

Bloc synchronisé :

```
class AutreCompteur{
    private int valeur;
    private Object verrou = new Object();
    public void incr(){
        synchronized(verrou){ // --- ici !
            valeur++;
        }
    }
}
```

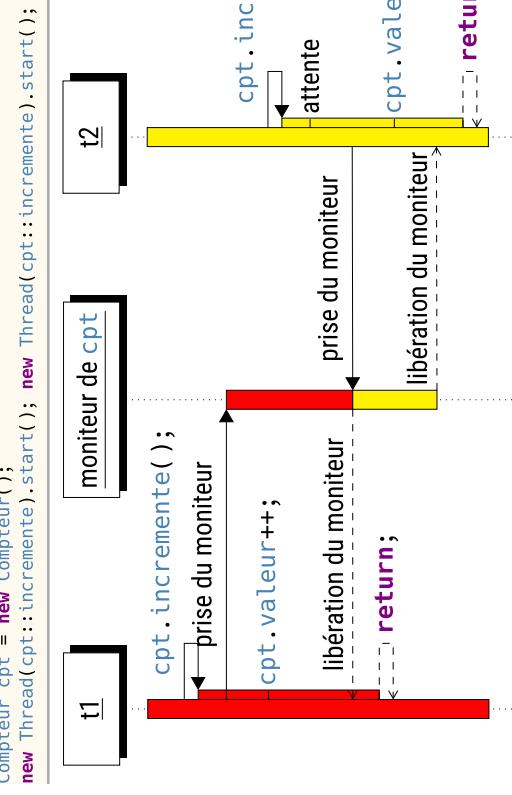
Sémantique : le *thread* qui exécute ce bloc demande le moniteur de **verrou** en y entrant et le libère en en sortant.

Conséquence : pour une instance donnée de **AutreCompteur**, le bloc n'est exécuté que par un seul *thread* en même temps (**exclusion mutuelle**). Les autres *threads* qui essayent d'y entrer sont suspendus (**BLOCKED**).

Le verrou intrinsèque

Qu'est-ce, à quoi cela sert-il ?

Le verrou intrinsèque



Le verrou intrinsèque

Utilisation en pratique : le mot-clé synchronized

Méthode synchronisée : cas particulier avec synchronisation de tout le corps de la méthode sur moniteur de **this**. → syntaxe plus légère, plus souvent utilisée en pratique.

équivalent à...

```
class Compteur {
    private int valeur;
    // méthode contenant bloc synchronisé
    // sur this
    public void incr(){
        synchronized(this){
            valeur++;
        }
    }
}
```

Note : l'exclusion mutuelle porte sur le moniteur (1 par objet) et non sur le bloc synchronisé (souvent plusieurs par moniteur).

Conséquence : 1 bloc synchronisé n'a qu'une seule exécution simultanée. De plus, aucun autre bloc synchronisé sur le même moniteur ne sera exécuté en même temps.

- 3 méthodes concernées (classe **Object**) : **notify()**, **notifyAll()** et **wait()**.
- Ces méthodes sont appelables seulement dans un bloc synchronisé sur l'objet récepteur de l'appel : **synchronized(x){ x.wait(); }**.
- **wait()** : met le *thread* en sommeil et libère le moniteur (**getState()** passe de **RUNNABLE** à **WAITING**).

Le *thread* restera dans cet état tant qu'il n'est pas réveillé (par **notifyAll()** ou **notify()**). Il sera alors en attente pour récupérer le moniteur (**WAITING** → **BLOCKED**).

- **notifyAll()** : réveille tous les threads en attente sur l'objet. Ceux-ci deviennent candidats à reprendre le moniteur quand il sera libéré.
- **notify()** : réveille un *thread* en attente sur l'objet.

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Introduction

Threads en Java

Introduction

La classe Thread

Synchronisation

Domotique le JMM

API de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

- On utilise **wait()** pour attendre une condition **cond**.

- Mais plusieurs *threads* peuvent être en attente. Un autre pourrait être libéré et récupérer le moniteur avant, rendant la condition à nouveau fausse.
- → aucune garantie que **cond** soit vraie au retour de **wait()**.

Ainsi, il faut tester à nouveau jusqu'à satisfaire la condition :

```
synchronized(obj) { // conseil : mettre wait dans un while
    while(!condition(obj)) obj.wait();
    ... // insérer ici instructions qui avaient besoin de condition()
}
```

Il faut absolument retenir la formule ci-dessus!!!
(utilisée dans 99,9% des cas d'usage corrects de **wait...**)

Le mécanisme **notifyAll() / notify() / wait()**

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Introduction

Threads en Java

Introduction

La classe Thread

Synchronisation

Domotique le JMM

API de haut niveau

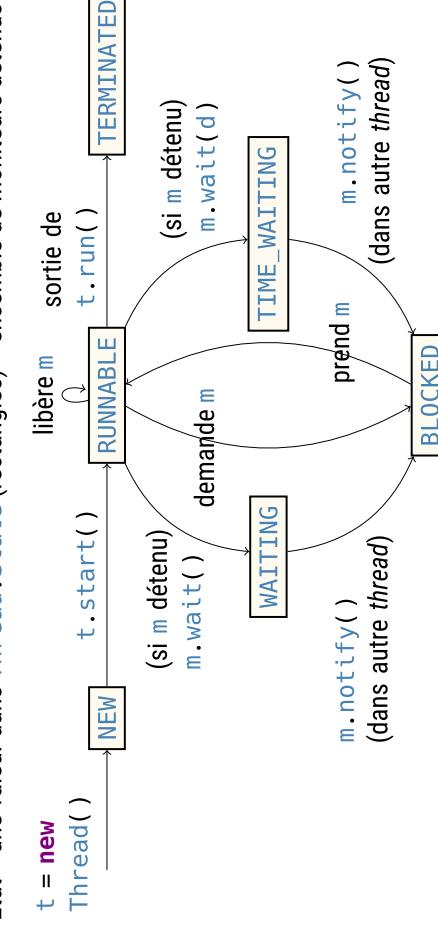
Interfaces graphiques

Gestion des erreurs et exceptions

Résumé

Retour sur les états d'un *thread*

État = une valeur dans **Thread.State** (rectangles) + ensemble de moniteurs détenus



En réalité, raccourcis directement vers **RUNNABLE** plutôt que **BLOCKED** quand le moniteur est déjà disponible.

Variantes acceptables :

- **while(! condition()) Thread.sleep(temps);**
→ utilise quand on sait qu'aucun *thread* ne notifiera quand la condition sera vraie.
- **while(! condition()) Thread.onSpinWait(); (Java ≥ 9) : attente active**
(cest-à-dire : ni blocage ni attente, le *thread* reste **RUNNABLE**).
→ on évite le coût de la mise en attente et du réveil, cette approche est donc conseillée quand on s'attend à ce que la condition soit vraie très vite.

Déconseillé 408 : while(!condition(obj))/*rien*/; ; attente active « bête »

- c'est l'ancienne façon de faire, remplacée avantagereusement par la variante avec **onSpinWait**. En effet, **onSpinWait** signale à l'ordonnanceur que le *thread* peut être mis en pause (laisser sa place sur le processeur) prioritaiement en cas de besoin.

408. Sauf pour faire cuire une omelette sur son microprocesseur...

- moniteurs = principe d'exclusion mutuelle + mécanisme d'attente/notification;
- mais il existe d'autres façons de synchroniser des threads par rapport à l'usage d'une resource (exemple : lecteurs/rédacteur);
- fonctionnalités possibles : savoir à qui appartient le verrou, qui est en attente, etc.;
- → bibliothèque de verrous divers dans `java.util.locks`, implémentant l'interface `java.util.concurrent.locks.Lock`.

L'interface `java.util.concurrent.locks.Lock`:

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    Condition newCondition();
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
}
```

- moniteurs = principe d'exclusion mutuelle + mécanisme d'attente/notification;
- mais il existe d'autres façons de synchroniser des threads par rapport à l'usage d'une resource (exemple : lecteurs/rédacteur);
- fonctionnalités possibles : savoir à qui appartient le verrou, qui est en attente, etc.;
- → bibliothèque de verrous divers dans `java.util.locks`, implémentant l'interface `java.util.concurrent.locks.Lock`.

L'interface `java.util.concurrent.locks.Lock`:

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    Condition newCondition();
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
}
```

Comme le verrouillage et le déverrouillage se font par appels explicites aux méthodes `lock` et `unlock`, ces verrous sont appelés **verrous explicites**.

Inconvénient : l'occupation du verrou n'est pas délimitée par un bloc lexical tel que `synchronized { ... }` 409.

La logique du programme doit assurer que toute exécution de `lock` soit suivie d'une exécution de `unlock`.

Avantages :

- Nombreuses options de configuration.
- Flexibilité dans l'ordre d'acquisition et de libération. 410

409. Mais on peut programmer un tel bloc à la main à l'aide d'une fonction d'ordre supérieur, et encapsuler un tel verrou dans une classe dont l'interface ne permettrait d'acquérir le verrou que via cette FOS.
 410. Concurrent Programming in Java (2.5.1.4) montre un exemple de liste chaînée concurrençante où, lors d'un parcours, il est nécessaire d'exécuter une chaîne d'acquisitions/libérations croisées de la forme :

`m1.lock(); ... ; m2.lock(); m1.unlock(); ... ; m3.lock(); m2.unlock(); ... ; m4.lock(); m3.unlock(); ... ; m5.lock(); m4.unlock(); ... ;`

Dangers de la synchronisation

Exemple de `dead lock`

```
class SynchronizedObject {
    public synchronized void use() {
        public synchronized void useWith(SynchronizedObject other) {
            for (int i = 0; i < 1000; i++) {
                System.out.println(Thread.currentThread() + " claims " + other);
                other.use();
            }
        }
    }
}

public class DeadLock extends Thread {
    private final SynchronizedObject obj1, obj2;
    private DeadLock(SynchronizedObject obj1, SynchronizedObject obj2) {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
    @Override public void run() {
        obj1.useWith(obj2);
        System.out.println(Thread.currentThread() + " is done.");
    }
}

public static void main(String args[]) {
    synchronizedObject obj1 = new SynchronizedObject();
    synchronizedObject obj2 = new SynchronizedObject();
    // dead lock, sauf si le 1er thread arrive à terminer avant que le 2e ne commence
    new DeadLock(obj1, obj2).start();
}
```

Dangers de la synchronisation

quand elle est utilisée à mauvais escient ou à l'excès

Un dernier avertissement : la synchronisation doit rester raisonnable!

En général, plus il y a de synchronisation, moins il y a de parallélisme... et plus le programme est ralenti. Pire, il peut bloquer.

Pathologies typiques :

- **dead-lock** : 2 threads attendent chacun une ressources que seul l'autre serait à même de libérer (en fait 2 ou plus : dès lors que la dépendance est cyclique).
- **famine (starvation)** : une ressource est réservée trop souvent/trop longtemps toujours par la même tâche, empêchant les autres de progresser.
- **live-lock** : boucle infinie causée par plusieurs threads se faisant réagir mutuellement, sans pour autant faire avancer le programme. 411

411. S'imaginer deux individus essayant de se croiser dans un couloir, entamant simultanément une manœuvre d'évitement du même côté, mettant les deux personnes à nouveau l'une face à l'autre, provoquant une nouvelle manœuvre d'évitement, et ainsi de suite...

- Principe pour éviter les **dead-locks** : **toujours acquérir les verrous dans le même ordre**⁴¹² et les libérer dans l'ordre inverse⁴¹³ (ordre LIFO, donc).
- En effet : dans l'exemple précédent, une exécution de `run` veut acquérir `o1` puis `o2`, alors que l'autre exécution veut faire dans l'autre sens.
- quand on écrit un programme concurrent à l'aide de verrous explicites, il faut documenter un ordre unique pour prendre les verrous.

L'autre voie est de se reposer sur des abstractions de plus haut niveau, sur lesquelles il est plus aisément de raisonner (cf. la suite).

412. Pas évident en pratique : verrous créés dynamiquement, difficile de savoir quels verrous existentont à l'exécution. On peut aussi ne pas savoir quels verrous une méthode donnée d'une classe tierce utilise.

413. Pour les verrous intrinsèques, ordre inverse imposé par l'imbrication des blocs **synchronized**. Mais rien de tel pour les verrous explicites. La preuve de l'absence de dead-lock doit alors se faire au cas par cas.

- Rappel : *thread = abstraction*
 - simulant la séquentialité dans le *thread* ;
 - permettant une communication instantanée inter-thread via une mémoire partagée ;
 - (et simulant le parallélisme parfait⁴¹⁴ entre threads).

- Est-ce vraiment la réalité ?
 - Divulgâchage : NON !

En réalité, paradigme idéal trop contraignant, empêchant les optimisations matérielles.

Modèle d'exécution réellement implémenté par la JVM : le **JMM**⁴¹⁵.

Seule garantie : sous condition, ce qu'on observe est indistinguable du paradigme idéal.

- 414. Hors synchronisation, évidemment.
- 415. Java Memory Model

Par exemple, dans le programme suivant :

```
public class ThreadConsistency extends Thread{
    static boolean x = false, y = false;

    public void run(){
        if (x || y) { x = true; y = true; } else System.out.println("Bug!");
        // Affiche "Bug !" si on trouve y vrai alors que x est faux
    }
}
```

→ le **JMM** : ne garantit donc pas une cohérence parfaite (mais un minimum quand-même...)

L'appel `test(100)` peut afficher « **Bug !** ».

Par exemple : si un des threads finit d'exécuter « `x = true`; `y = true` » et un autre reçoit, dans son cache, la modification sur `y` mais pas sur `x`.
417. Le JMM autorise cette possibilité théorique, mais probablement vous ne verrez jamais ce message!

416. Mémoire locale propre au cœur, plus proche physiquement et plus rapide que la mémoire centrale.

Modèle de mémoire Java et optimisations

Modèle de mémoire Java et optimisations

Compléments en POO
Aldric Degorre

Compléments en POO
Aldric Degorre

Réalité physique : Les CPU sont dotés de mécanismes permettant de réordonner des instructions⁴¹⁸ qu'il sait devoir exécuter (afin de mieux occuper tous ses composants).

Interprétation : l'ordre du programme n'est pas toujours respecté (même sur 1 thread). En plus, optimisations différentes d'une architecture matérielle à une autre (p. ex : comportement différent entre x86 et ARM) → ordre peu prévisible.

Solution naïve : ajouter des barrières⁴¹⁹ partout dans le code compilé.

Problème : vitesse d'exécution sous-optimale (le CPU n'arrive plus à donner autant de travail à tous ses composants).

→ **le JMM** : ne garantit pas le respect exact de l'ordre du programme... mais promet que certaines choses importantes restent bien ordonnées.

418. out-of-order execution

419. Instruction spécifique prévue dans les CPU, justement pour empêcher le ré-ordonnancement.

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Concurrence
Interactions
Threads en Java
Domptez le JMM
APIs de haut niveau
Interfaces graphiques
Gestion des erreurs et exceptions

Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Concurrence
Interactions
Threads en Java
Domptez le JMM
APIs de haut niveau
Interfaces graphiques
Gestion des erreurs et exceptions

Exemple :

- Supposons qu'initialement `x == 0 && y == 0`. On veut exécuter :
 - sur le `thread 1` : (1) `a = x;` (2) `y = 1;`
 - et, sur le `thread 2` : (3) `b = y;` (4) `x = 2;`
- (1) arrive-avant (2) et (3) avant (4)
 - à la fin, il semble impossible d'avoir à la fois `a == 2` et `b == 1`.
- Or c'est pourtant possible !
 - En effet, sur chaque `thread` isolé, inverser les 2 instructions ne change pas le résultat. Comme il n'y a pas de synchronisation, rien n'interdit donc ces inversions.
Il est donc possible d'exécuter les 4 instructions dans l'ordre suivant :
`y = 1; x = 2; a = x; b = y;`

Exemple

Compléments en POO
Aldric Degorre

Compléments en POO
Aldric Degorre