for(String a : args) est la boucle foreach en Java

champs, méthodes et classes /interfaces internes sont appelés membres

Un membre **static** est partagé par tous les objets de la classe

Pour un static (methode ou champs), pas besoin d’un objet pour accéder : il suffit de NomClasse.nomMembre ou NomClasse.nomMethode.

Une méthode static peut accéder uniquement à des membres static de sa classe, et n’a pas d’accès à this.

this() pour appeler un constructeur dans un autre constructeur

**contrôle d’accès**

* private (visible uniquement par la classe)
* public (visible par tout le monde) pas de modificateur (package) (visible uniquement par les classes du même package)
* protected (visible uniquement par les classes du même package et par les sous-classes)

Toute modification de la référence p par q.distance n’affecte pas le main (si p = null)

Toute modification de l’objet référencé par p faite par q.distance affecte le main (puisqu’il partage l’objet)

Toute modification des paramètres par p.deplace n’affecte pas les variables du main

**Surcharge** (overloading) : dans une même classe signatures différentes pour un même nom de fonction

class **Ambiguity** { void g ( int i, double d) {… } void g (double d, int i) {… }} deux méthodes sont compatibles avec l’appel même si leurs signatures ont l’air diff. Ça produit une erreur de compilation.

Les méthodes d’accès retournent normalement une copie des objets mutables. Ce qui fait que des données protected / private restent protégées.

**Héritage** et **extends**

Un champ avec le même nom d’un champ de la super-classe cache ce dernier

Les constructeurs dans les sous classes sont plus détaillés que dans les classes mères.

Pour initialiser les champs hérités il est souvent nécessaire d’invoquer le constructeur de la classe parent : super (…)

super(…) doit être la première instruction du constructeur

super() et this() ne peuvent pas être invoqués dans un même constructeur.

Si aucun des deux n’est invoqué, un appel super() (sans arguments) est implicite.

pour qu’il y ait redéfinition d’une méthode, la méthode redéfinie dans la sousclasse doit avoir la même signature (nom de méthode et types des paramètres) que la méthode de la classe parent à redéfinir

**polymorphisme** : une même variable peut référencer des objets de plusieurs types différents pendant sa vie

Donc un objet d’une sous-classe peut être affecté à une variable d’une super-classe

CompteCredit cc = new CompteCredit(“AG675”, 0, 200);

CompteBancaire cb = new CompteBancaire(“234BD”, 0);

cb = cc; //et pas viceversa

Un objet a un type déclaré (type de la déclaration, associé au nom de la variable, ne change jamais) et un type effectif (classe avec laquelle l’objet référencé a été construit, peut changer dynamiquement, peut être testé dynamiquement).

> Un objet de type effectif C peut être référencé par une variable de type déclaré D pour tout type D de plus haut niveau dans la hiérarchie d’héritage.

> Seuls les champs et méthodes (visibles) de la classe C sont accessibles sur un objet de type *déclaré* C.

le type déclaré détermine quelles méthodes on peut invoquer → fait statiquement (en phase de compilation)

e type effectif détermine quelle définition est à utiliser pour ces méthodes → fait dynamiquement (en phase d’exécution)

> Le dynamic binding ne s’applique pas au variables! Le lien entre le nom d’une variable et sa définition a lieu à la compilation (static binding). Toutes les valeurs (même celles renvoyées par des getters) sont fixées à la compilation.

Transtypage et forcer un type, on vérifie que le type est ok avec instanceof() :

if (cb instanceof CompteCredit) { cc = (CompteCredit) cb; }

Upcasting : Conversion vers un type de plus haut niveau dans la hiérarchie, toujours sûr

Downcasting : Sûr uniquement s’il ne dépasse pas le type effectif.

Toutefois, comme toute autre classe, la sous-classe :

■ n’a pas accès aux membres private de la super-classe

Les membres protected sont visibles uniquement par les sous classes et par les classes du même package

■ a accès aux membres avec visibilité package uniquement si elle se trouve dans le même package que la super-classe

■ a accès aux membres protected et public de la super-classe

□ Le contrôle d’accès est vérifié à la compilation

Quand un constructeur est invoqué, l’objet est déjà crée et ses champs initialisés aux valeurs par défaut (0, null, false)

**Interfaces**

Une interface est une collection de signatures de méthodes. Elle pt contenir des champs constants et des classes internes, mais pas des variables d’instance

une sous-classe peut hériter d’une seule autre classe, alors qu’elle peut implémenter plusieurs interfaces.

N’importe quelle classe (même sous-classe) peut être étiquetée NomDInterface à condition qu’elle définisse NomDInterface. Cela ne serait pas possible si NomDInterface était une classe abstraite.

Si A et I possèdent une méthode avec la même signature B hérite la méthode de la classe mère A, une éventuelle définition de la même signature dans I est ignorée.

**Classes Membres**

Classes membres = membres (possiblement statiques) d'une autre classe

Classes locales = classes définies dans un bloc de code

*Classes membre non-static* : Définir des classes qui ont du sens uniquement en relation à la classe englobante. Si elles sont private, seule la classe englobante y a accès.

Dans la classe interne la référence implicite à l’objet de classe englobante est NomClasseEnglobante.this, mais elle peut être omise en général (sauf si occultée)

Une classe fille d’une classe membre (bien que interne à la même classe englobante) maintient les règles habituelles de visibilité des champs hérités. CaD qu’elle ne verra pas un champ de la classe englobante de sa classe mère.

*Classe membre statique* d'une autre classe

pas d’accès à NomClasseEnglobante.this

**Expressions lambda**

Bloc de code, précédé par la liste de tous les paramètres utilisés dans ce code :

Ex 1 : (int k) **->** { for (int i = k; i >= 0; i--) System.out.println(i); }

Ex 2 : (String first, String second) **->** first.length() - second.length()

Les interfaces possédant une seule méthode abstraite sont appelées fonctionnelles :

■ Comparator avec méthode int compare(T o1, T o2); censée renvoyer >0,0,<0 si o1o2, resp. ■ ActionListener avec méthode void actionPerformed(ActionEvent e) invoquée quand un événement a lieu (e est l’événement à traiter) ■ Function avec méthode R apply (T t) censé implémenter une fonction arbitraire à un seul paramètre ■ Predicate avec méthode boolean test(T t) censée implémenter une fonction booléenne arbitraire à un seul paramètre ■ IntConsumer avec méthode void accept(int i) ■ IntSupplier avec méthode int getAsInt()

Une expression Lambda peut être affectée à une variable de type interface fonctionnelle; *l’expression lambda fournit alors une définition pour la méthode abstraite de l’interface*

Comparator c = (first, second) -> first.length() - second.length();

Ensuite c.compare (“abc”, “a”);

Predicate positif = (Point p) -> p.getX() > 0 && p.getY() > 0;

Ensuite positif.test (new Point(2,3)) renvoie true et positif.test (new Point(-1,2)) renvoie faux

**Exceptions**

try { } bloc qui pt soulever l’exception

catch (Type exception e) { //exécuté uniquement si le bloc try soulève // une exception de Type exception e }

Remarque : si un exception d’un type incompatible avec est soulevée pendant l’exécution du bloc try, la méthode termine en état d’erreur avant de compléter le bloc try et soulève l’exception non-capturée

Checked : Classes qui dérivent de Exception, mais pas de RuntimeException

Unchecked : Classes qui dérivent de RuntimeException

Toute méthode qui peut soulever une exception checked doit le déclarer : public void fullRead(String filename) throws FileNotFoundException {}

Remarque : throws pas nécessaire pour les exceptions unchecked (toute instruction peut en soulever une…)

On peut capturer plusieurs exceptions dans le même bloc : il y a plusieurs blocs catch à la suite. bloc catch exécuté : le premier dont le type est compatible avec celui de l’exception soulevée dans le bloc try.

**Généricité**

À la declaration de T on peut ajouter des “bornes” pour limiter les types acceptés comme valeur de T. public class A <T extends Figure> T borné par la classe Figure

Tout type variable est implicitement borné : <T> equivalent à <T extends Object>

À l’intérieur de la classe / méthode qui déclare un type borné <T extends C> , les méthodes de C peuvent être utilisés sur les objets de type T

Wildcard : Mais usages différents : □ ? à utiliser quand aucun autre type dépend du type anonyme □ Exemple : dans printList on n’a pas besoin de faire référence à T (pour afficher les éléments on peut les traiter comme des Object)

## **Problèmes liés à la généricité**

■ Types dynamiques identiques

C<A> == C<B> pour getClass et instanceof

■ Instanciation impossible

new E() → (E) new Object() → ERREUR

=> Workaround : @FunctionalInterface public interface Supplier<T> {T get ();}

dans C<E> : methode (Supplier<E> supplier) {E nouveauE = supplier.get();}

Ensuite : C<Type> objet; objet.methode(Type::new); …

■ Tableaux impossibles

<E> … E[] → ERREUR

=> utiliser ArrayList<E> ou Object[]

■ Tableau d’elements de type générique impossible

new ArrayList<String>[]; → ERREUR

=> sauf exception : <?>

■ Référence qu type variable impossible dans un contexte static

Car après effacement il y a une classe raw, il n’y a qu’une version des champs statiques → pas de ref à ce type dans une méthode ou champ statique de la classe

=> excéption : les méthodes génériques, qui peuvent en revanche être statiques

■ Overriding

C’est la signature après effacement qui compte

■ Héritage

On peut hériter d’une instance d’une classe (ou interface) générique (class D<T> extends C<String>)

=> Le compilateur implémente ce genre d’héritage de façon à garantir l’overriding des méthodes instanciées de la classe mère (introduction de méthodes “bridge”)

■ Unchecked warnings

Si utilisation de raw types bc interaction avec vieux code, warnings unchecked

=> @SuppressWarnings("unchecked")