POO

Introduc° à Rappel

- this() => appel à un autre constructeur
- encapsula° : champ privé récupéré par des méthodes
- règle de visibilité
  - ↳ déf ds bloc de class => visible / modifiable ds tt le bloc
  - ↳ déf ds un bloc => unig^{t} ce bloc + imbrig'

Heritage

- Object est au dessus de l'arbre
- super() => constructeur du parent
- un champ avec le m nom qu'un champ super, le cache (idem méthode)
  - ↳ on peut les appeler avec super.champ (super.methode)
  - ↳ final interdit la redefini° d'une méthode
- un objet a un type effectif & déclaré
  - getClass()
  - obj instanceof A => ancêtre du t.e de obj
  - q^{t} méthode on peut invoquer
  - q^{t} déf^{o} de la méthode
- pour les champs, on doit preciser manu^{t}
  - ↳ b.c => 4
  - ↳ ab.c => 3
  - ↳ ((B)ab).c => 4
- la sous-class n'a pas accès aux membres privés de la super-class

Interface

- class abstract : class dont cert^{s} méthodes n'ont pas de déf
- collec° de signature de méthode (automatiq^{t} public)
  - ↳ abstract ou default
- pas de variables d'instance
- les sous class peuvent imp. pls interface

Héritage multiple

- cas d'ambigüité : Class.constante
- A (class) & I (interface) ont la m méthode => c^{l} de A
- I_1 & I_2 => redefinir la méthode

Class interne

- class définie ds une autre class
  - ↳ class membre (CM)
  - ↳ class locale (CL) => ds un bloc de code
  - ↳ class anonyme (CA) => sans nom
- obj de la CM B : B b = a.new B()
  - ↳ class engloblante (CE)
- ds la CI, la ref à un obj de la CE avec CE.this
- la CM a accès aux autres membres de la CE
- la CE a accès à ses CM qu'importe leur visibili^{t}
  - ↳ la CF d'une CM respecte les règles habitu^{t} de visibili^{t}
- endehors de la CE, on note la CM : CE.CM
  - ↳ A a = new A() A.B b = a.new B()
- CM statiq a accès aux autres membres statiq
- CMS m règle d'accès
- accès avec CE.CMS (ou l'importer)

	obj, int, ...		méthode	
	statiq	non statiq	statiq	non statiq
créa st.	✓	&	✓ accès CI.m	&
// n-st.	✓	✓	✓ accès avec l'obj CE.CI b = new CE.CI()	✓
accès st.	✓	✓ CE.m	✓ accès CE.m	✓ CE.m
// n-st.	& sauf ds CI	✓ CE.this.m	&	✓ CE.this.m
CI	A a = new A(); a.membre			
st -> ...	✓	✓	/	/
n-st -> MCI	& sauf ds n-s	✓	/	/
CI	A a = new A(); Obj b = a.m			

Façons d'étendre une CM

- ds CE : CIE extends CI
- CEE extends CE => CIE extends CI
  - ↳ CI r = new CIE()
- Autre extends CE.CI
  - ↳ Autre (CE r, int i) { r.super(i); }

Class locale

- CIL => par CM, visible unig^{t} ds code
- a accès : • membres CM
  - • var locale ssi effectively final

Class anonyme

- CIL sans nom
- on la déclare : A a = new A() { ... } ;

POO

Expressions lambda

- bloc de code, précédé de ses param utilisées par le code
  - ↳ `(int k) -> {`
    `for (int i=k; i >= 0; i--)`
    `System.out.print(i);`
    `}`
  - ↳ `(String s1, String s2) -> s1.length() - s2.length()`

Interface fonct°

- interface ayant une seule méthode abstraite
- une E.L peut être affectée à une var de type I.F
  - ↳ fournit une déf° pour la méthode abs.

ex `Comparator<String> c = (first, second) -> first.length() - second.length();`
=> `c.compare("abc", "a")`

ex `jButton1.addActionListener(event -> System.out.println(event));`

- réf à une méthode existante : `Class::méthode` / `obj::méthode`
  - ↳ `String::concat` <=> `(x,y) -> x.concat(y)`
  - ↳ idem avec les constructeurs : `I i = Point::new`
- ≠ visibilité qu'une class locale

Généricité

```
public class Cellule<E>
    private E elem;        ← type variable
```

↳ `Cellule<Integer> c = new Cellule<>();`

- pls type de var possible : `Nom Class <V,K>`
- convent° :
  - collec° (liste, file ...) : E
  - clé a valeur : K, V
  - type arbitraire : T, U, S

constructeur générique
`class <String> m; ...`
`m.f(String::new)`

`void f(Supplier<E> cons)`
`E v = cons.get();`
...

constructeur tab
`E[] tab = (E[]) new Obj[1]`

`static <T> T pick(T[] tab)`
↳ déclara° du type de var avant le type retour
↳ on peut l'expliciter : `String s = class.<String>pick(tab)`

Types bornés

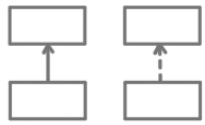
`<T extends obj>`
↳ sous-class de obj
↳ A s.c B ≠ C<A> s.c C<B>
↳ covariance ✗

`<?>` => wildcard
↳ `<? extends obj>` => lecture ✓ ecriture ✗
↳ `<? super obj>` => lecture ✓ ecriture ✓
↳ restreint
↳ covariance ✓

Design Pattern

- descrip° informt° d'un pb → implementa°
  - ↳ analyse
    - décrit ce q doit faire le logicl
  - ↳ design
    - identifica° des class, leur responsabilt & rela°
  - ↳ implementa°

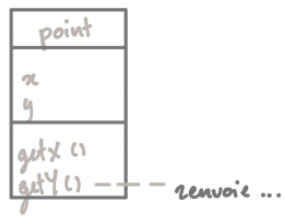
- 3 types de rela° de class
  - ↳ Heritage ("être") => implemt°



  - ↳ Agrégra° ("avoir")

  - ↳ Dépendance ("utilisen")

- notat° d'une classe :



TD · correc° importante

n°4.1

- Class A : `f(A x) => A.f(A)` & `g() => f(A)`
- Class B : `f(A x) => B.f(A)` & `f(B x) => B.f(B)`

- `A a = new A()`
  - `a.g()` => `A.f(A)`
  - `a.f(a)` => `A.f(A)`
  - `a.f(b)` => `A.f(A)`
  - `a.f(c)` => `A.f(A)`
- `B b = new B()`
  - `b.g()` => `B.f(A)`
  - `b.f(a)` => `B.f(A)`
  - `b.f(b)` => `B.f(B)`
  - `b.f(c)` => `B.f(A)`
- `A c = new B()`
  - `c.g()` => `B.f(A)`
  - `c.f(a)` => `B.f(A)`
  - `c.f(b)` => `B.f(A)`
  - `c.f(c)` => `B.f(A)`

=> affichage : `TE.f(TD)`

n°6.1

- On ne peut pas instancier une interface ni une class abs
- une interf. ne peut avoir de constructeur, une CA, oui.
- `A a = new B()` marche si A est une interf / CA (si B pas CA)
- les interf n'ont q des attributs public static final
- Depuis J8 : méth. abs, méth. default, static possible unigt avec default
  - ↳ méth. static d'interf. => acc avec `I.f()` unigt
- Inter. ← imp — CA
  - Inter. : ext
  - CA : ext