

Interfaces.

- Une interface exprime un contrat (i.e. services que doivent fournir les classes qui implementent l'interface).
- Une interface peut contenir :
 - des méthodes "public abstract", sans implementation
 - des méthodes "public default", ayant une implementation par defaut.
 - des méthodes "public static", ayant une implementation non redéfinissable.
 - des attributs "public static final".
- Syntaxe :

modificateur interface NomInterface {
 // caractéristiques.
}

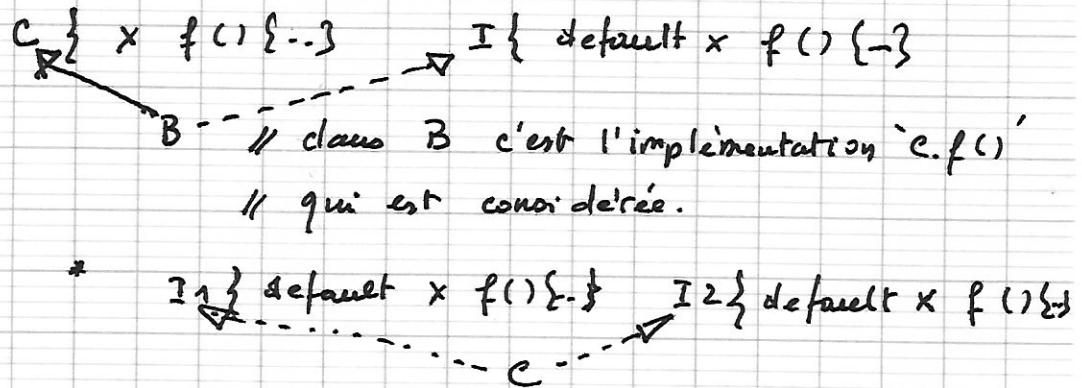
- Une interface définit un type, mais ne servir à la création d'objets (ne possède pas de constructeur)
- Une classe qui implemente une interface :

class A implements I { ... }

doit fournir une implementation pour tous les méthodes abstraites de l'interface, et peut redéfinir les méthodes "default".
- Une classe qui implemente une interface est un sous-type de cette interface.
- Une classe n'implementant qu'une partie des méthodes abstraites d'une interface doit être déclarée "abstract" (et donc ne peut être instantiée même si elle possède des constructeurs)

- Une interface peut hériter d'autres interfaces;
interface I extends I₁, I₂, ... { ... }
 - Une classe peut implementer plusieurs interfaces:
class X implements I₁, I₂, ... { ... }
 - Cette situation d'implementation multiple peut conduire à un conflit de noms:
 - cas de surcharge: méthodes avec des signatures différents (même nom); la classe doit fournir une implementation pour chacune des versions de la méthode.
 - cas de redéfinition:

- méthodes de même signature et type de retour covariant :
 - cas de méthodes abstraites: la classe implemente une version du sous-type.
 - cas de méthodes "default":
 - * Une implementation de classe primaire sur l'implementation par defaut:



can ambiguous: 2 possibilities.

- C implemente les méthodes.
 - on choisit l'une des implementations pour la syntaxe: "I1.super.f()"

→ des interfaces: (visiter "Java API 8")

- interface List <E> { (java.util)

- interface Comparable<E> {

- int compareTo (E o);

- }

- interface Comparator <E>

- int compare (E o1, E o2);

- }

- interface Cloneable { }

→ On appelle interface fonctionnelle, une interface ayant une seule méthode abstraite.

→ Une lambda expression est un bloc de code qui définit une fonction de façon anonyme (i.e. sans la nommer), et qui est compatible avec le type exprimé par une interface fonctionnelle.

Syntaxe:

(T1 x, ..., TA y) → {, bloc de code normal }

Eg:

(String s, String ss) → { return s.length() - ss.length(); }

- le type des paramètres peut être omis, s'il est didactique du contexte.

- le "return" peut être omis si le bloc ne contient qu'une seule instruction.

- dans une lambda expression tous les parenthèses sont obligatoires: () → { System.out.println("xx"); }

→ Exemple:

- interface Predicate<E> {

 boolean test (~~E~~ o);

 }

- dans "List" on a la méthode :

default boolean removeIf (Predicate<filter>)
 qui supprime d'une liste les éléments vérifiant "filter".

soit : List<Integer>=Arrays.asList(1, 2, 3, 4);
 List<Integer>=new LinkedList<Integer>(2);

on cherche à supprimer de "ll" les nombres pairs.

2 méthodes :

- classe anonyme :

```
ll. removeIf ( new Predicate<Integer> {
    public boolean test ( Integer o ) {
        return (o % 2 == 0); }
    } );
```

- Lambda expression :

```
ll. removeIf ( n → n % 2 == 0 );
```

→ Des interfaces :

- public interface Function<T, R> {

 R apply (T t); }
- public interface Iterator<E> {

 boolean hasNext();

 E next(); }
- public interface Iterable<T> {

 Iterator<T> iterator(); }

Classes imbriquées.

- Une classe peut être déclarée à l'intérieur d'une autre classe. Ce qui permet de limiter sa visibilité et d'éviter des conflits de noms.
- Une classe imbriquée a accès à toutes les caractéristiques de la classe englobante, même celles déclarées "private".
- Une classe imbriquée peut être "static", et dans ce cas doit être déclarée dans le bloc le plus externe de la classe :

```

. class A {
    private static class B { ... }
}
. A.B b = new A.B();

```

- Une classe imbriquée peut être interne, déclarée à l'intérieur de la classe (en dehors de toute méthode).

```

class A { int i;
    X f() { --; i++; -- }
    class B { int i;
        Y g() { -- } i = A.this.i + 1; -- }
    }
    --.

```

- Une classe interne possède toujours une référence sur sa classe englobante, et une instance de B "vit" toujours à l'intérieur d'une instance de A.
- A.B b = (new A()).new B();
- "A.this" sert de cible, dans une classe interne, à une caractéristique de la classe englobante A.

- Une classe interne ne peut avoir de membre "static" (à l'exception des constantes)
- peut hériter d'autres classes (y compris la classe englobante)
- peut être "final" ou "abstract".

→ Une classe imbriquée peut être locale, et dans ce cas déclarée à l'intérieur d'une méthode, et donc locale à la méthode.

```
class A {
    void f () { class B { ... } }
```

Mais toute variable locale à la méthode ou tout paramètre, utilisé dans la classe locale doit être déclaré "final".

→ Une classe locale ne servant qu'à la création d'une seule instance peut être anonyme:

```
class X {
    A getAnA () {
        return new A() { ... }; }
```

Dans l'instruction "new A()" si :

- A est une interface, la classe créée implicitement est une implémentation de A.
- A une classe, la classe créée implicitement est une extension de A.