Programmation orientée objets

Mise en œuvre avec Java
Z225DM02

Yannick Loiseau

Université Blaise Pascal

Licence Informatique 2e année

Contenu

Cours

- concepts de la programmation orientée objets
- outil de modélisation (UML, un peu)
- langage Java bases

10h30

TD

exercices sur le cours + règles basiques de conception

- conception
- mise en œuvre en Java

15h

TP

- bases de Java
- mise en œuvre
  - structures de données
  - systèmes complets
  - programmes élaborés

25h30

Présentation

Paradigmes

- nombreux langages de programmation
  - https://en.wikipedia.org/wiki/List_of_programming_languages
  - http://99-bottles-of-beer.net/
  - http://rosettacode.org/
- points communs
- modélisation de problèmes
- $\Rightarrow$ *paradigmes*

Définition (Paradigme)

- représentation (données, traitements)
- concepts
- style
- modèle de calcul

Impératif

- état (global)
- suite d'instructions
- séquentielles ordre !
- $\Rightarrow$ changement de l'état : effet de bord
- machine à états (Turing)

Exemple

C, Pascal, Fortran, ...

- ✓ proche de la machine
- ✓ « naturel » (recette, . . .)
- ✗ proche de la machine
- ✗ état dépend de l'historique
- ✗ comportement dépend de l'état
- ⇒ ⛔ debug / tests

Fonctionnel

- pas d'état
- ensemble de fonctions (sens mathématique)
  - expressions
  - pures ⇒ pas d'ordre
- composition de fonctions
- λ-calcul

Exemple

Lisp, Caml, Haskel, ...

- ✓ raisonnement
- ✓ exact
- ✓ abstrait
- ✓ comportement indépendant de l'état / historique
- ⇒ ✌ debug / tests
- ✗ abstrait
- ✗ moins naturel

Orienté objets

- « objets » distincts
- comportement propre
- collaboration par « messages »
- état global ⇒ par objet

Exemple

Smalltalk, C++, Java, ...

Exemple

- Simula : 1967 (premier langage à classe)
- SmallTalk : 1972
- C++ : 1983
- Objective-C : 1983
- Object Pascal : 1986
- Python : 1990
- PHP : 1995
- Java : 1995
- Ruby : 1995
- Javascript : 1995 (langage à prototype)
- C# : 2001
- ...

Pourquoi autant de langages ?

⇒ applications spécifiques

Exemple

- C : embarqué, pilotes (performances)
- Perl : traitement de texte, scripts système
- Web : PHP (coté serveur), Javascript (coté client)

Pourquoi différents paradigmes ?

⇒ différentes problématiques

Exemple

- fonctionnel : parallélisme, concurrent, distribué
- objet : IHM

⇒ langages multi-paradigmes

⇒ plateformes multi-langages

Pourquoi Java ?

- libre et multi-plateformes
- très objet
- très utilisé
- plateforme et environnement riche
- rigoureux
- *simple*
- autres langages vus dans d'autres cours

Java

Java ?

- langage : spécifications
- plateforme : bibliothèques, frameworks (JSR → JDK)
  - Java SE
  - Java EE
  - Java ME

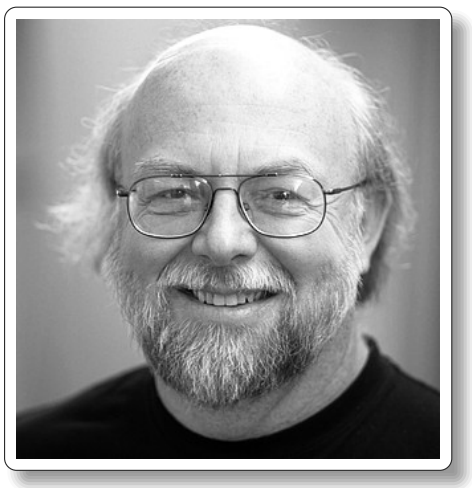
  ⇒ API (spécifications)
- environnement d'exécution (JRE) : machine virtuelle
- programme : implémentation (compilateur, VM, bibliothèques)

Implémentations

- Hotspot (Oracle)
- OpenJDK
- IBM
- Microsoft
- Apple
- Gnu : classpath, gcj
- ...

Historique

- James Gosling 1991
- Sun
- 1995 : JDK 1.0
- 2002 : J2SE 1.4
- 2009-2010 : Oracle
- 2011 : Java SE 7
- 2014 : Java SE 8
- 2017 : Java SE 9
- 2019-09 : Java SE 13



James Gosling

Caractéristiques

- orienté objets
- basé sur les classes
- typage
  - fort
  - statique
  - explicite
- héritage simple

Syntaxe de base

Similaire au C :

- Commentaires :
  // ligne de commentaire
  ou */* commentaire sur plusieurs lignes */*
- Fin d'instruction : `;`
- Déclaration / affectation : `int i = 3;`

Affectation

Gestion de mémoire

- ✌ gestion automatique ⇒ *garbage collector*
- ✌ références sans pointeurs

Conventions

- constantes : majuscules `static final int MA_CONSTANTE = 5;`
- variables : camelCase `int maConstante = 5;`

Types de base

- valeurs booléennes : `boolean` : `true`, `false`
- valeur indéfini : `null`
- type vide : `void`

Types numériques

- entiers :
  - `byte` (octet) 8 bits signé $[\![-128; 127]\!]$
    ```
    byte valeur = 42;
    ```
  - `short` 16 bits signé $[\![-32\,768; 32\,767]\!]$
    ```
    short valeur = 42;
    ```
  - `int` 32 bits signé $[\![-2^{31}; 2^{31} - 1]\!]$
    ```
    int valeur = 42; int valeur = 0x2A; int valeur = 052;
    int valeur = 0b101010;
    ```
  - `long` 64 bits signé $[\![-2^{63}; 2^{63} - 1]\!]$
    ```
    long valeur = 42L; long valeur = 1_337L;
    ```
- réels :
  - `float` IEEE 754, 32 bits, simple précision
    ```
    float valeur = 133.7f;
    ```
  - `double` IEEE 754, 64 bits, double précision
    ```
    double valeur = 13.37d; double valeur = 1.337e2;
    ```

Texte

- caractères : `char` unicode 16 bits `\u0000` → `\uffff`
  `char valeur = 'a'; char valeur = '\u03BB';` (λ)
- chaînes de caractère (plus ou moins)
  `String valeur = "hello"`

Tableau

- taille fixe
- type fixe
- commence à 0

- `int[] tabInt = {1, 2, 3, 4};`
  `char[] tabChar = {'a', 'b', 'c'};`
- `int[] tabInt = new int[10];`
- `tabChar[0] = 'd';`

Opérateurs

- numériques : `+ - * / % ++ --`
- binaires : `& | ^ ~ << >> >>>`
- logiques : `&& || !`

  ⚠ expression ternaire :

  `condition ? valeur si vrai : valeur si faux`
- comparaisons : `== != > >= < <=`
- affectation : `=`

  `+= -= *= /= %=`

  `&= ^= |= <<= >>= >>>=`

Contrôle

Tests

```
if (condition) {
  traitement si vrai
}
```

```
if (condition) {
  traitement si vrai
} else {
  traitement si faux
}
```

```
if (cond1) {
  traitement si cond1 vrai
} else if (cond2) {
  traitement si cond2 vrai
} else {
  traitement si tout faux
}
```

```
switch (variable) {
  case val1 : traitement 1
  case val2 : traitement 2
  case val3 : traitement 3
  default : traitement sinon
}
```

⚠ cascade ⇒ `break`

```
switch (variable) {
  case val1 :
    traitement 1
    break;
  case val2 :
    traitement 2
    break;
  default :
    traitement sinon
}
```

entiers, `char`, `String`, enum

Boucles

```
while (condition) {
  traitement
}
```

```
int i = 1;
while (i <= 10) {
  System.out.println(i);
  i++;
}
```

```
do {
  traitement
} while (condition);
```

Boucles

```
for (init ; condition ; inc) {
  traitement
}
```

```
for (int i = 1 ; i <= 10 ; i++) {
  System.out.println(i) ;
}
```

```
int[] lst = {1, 2, 8, 13, 14, 5, 2, 1} ;
for (int i=0 ; i < lst.length ; i++) {
  System.out.println(lst[i]) ;
}
```

```
for (element  : collection) {
  traitement
}
```

```
int[] lst = {1, 2, 8, 13, 14, 5, 2, 1} ;
for (int elt  : lst) {
  System.out.println(elt) ;
}
```

Boucles

Sortie prématurée

- `break` → sort de la boucle
- `continue` → passe à l'itération suivante
- `return`

Routines

```
int foisDeux (int a) {
  return 2 * a;
}
```

Convention

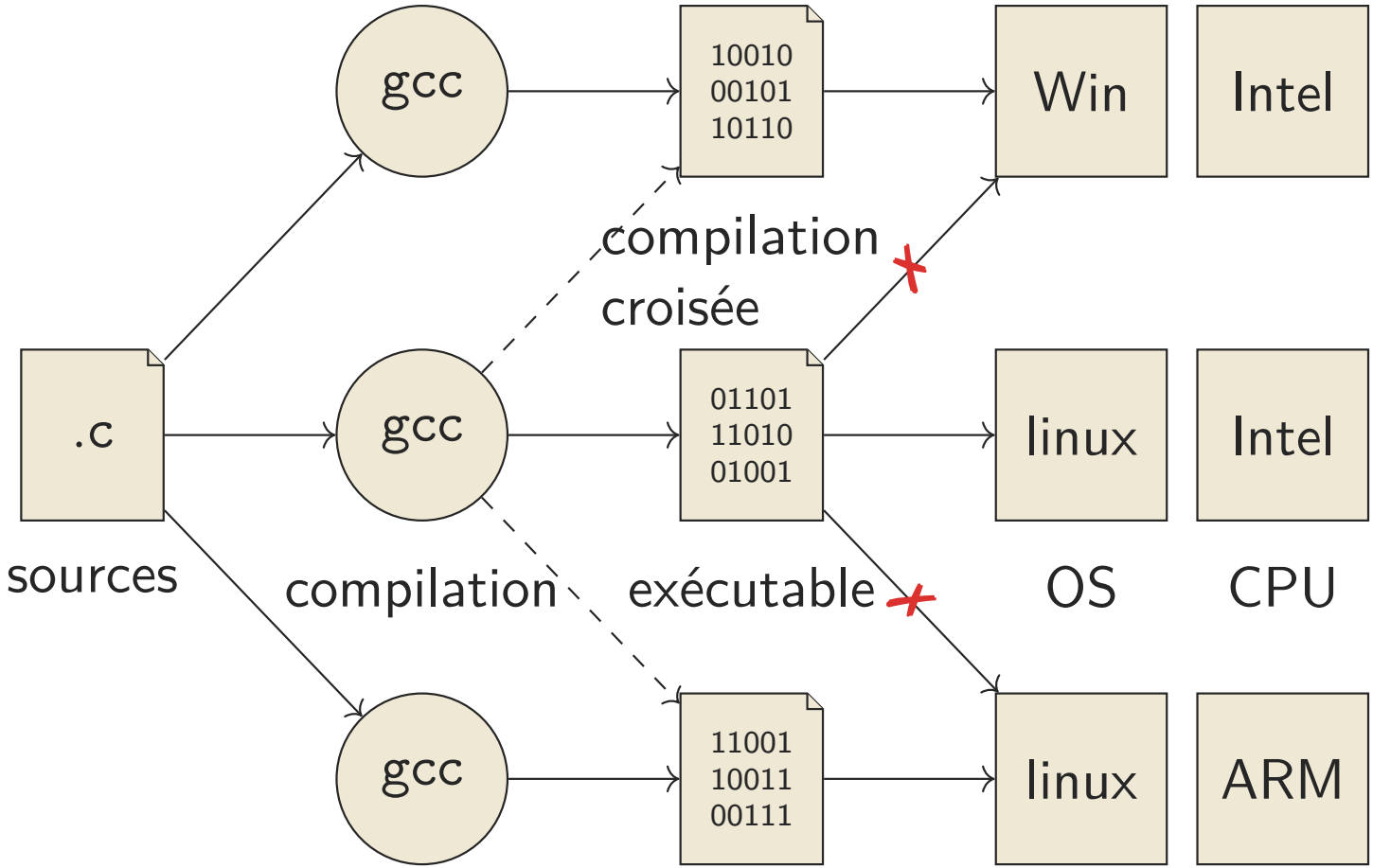
- idem variables : camelCase
- verbe
- test : *is* : `isEven` ou *has* `hasChanged`
- conversion : *as* : `asList` ou *to* : `toArray`
- ...

Compiler un programme Java

Compilation native

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("Hello\n");
    return 0;
}
```

```
gcc -o hello hello.c
./hello
```


gcc
10010
00101
10110
Win
Intel
compilation
croisée
.c
gcc
01101
11010
01001
linux
Intel
sources
compilation
exécutable
OS
CPU
gcc
11001
10011
00111
linux
ARM

⛔ distribution

- sources
- $n \times m$ binaires
- 1 seul système

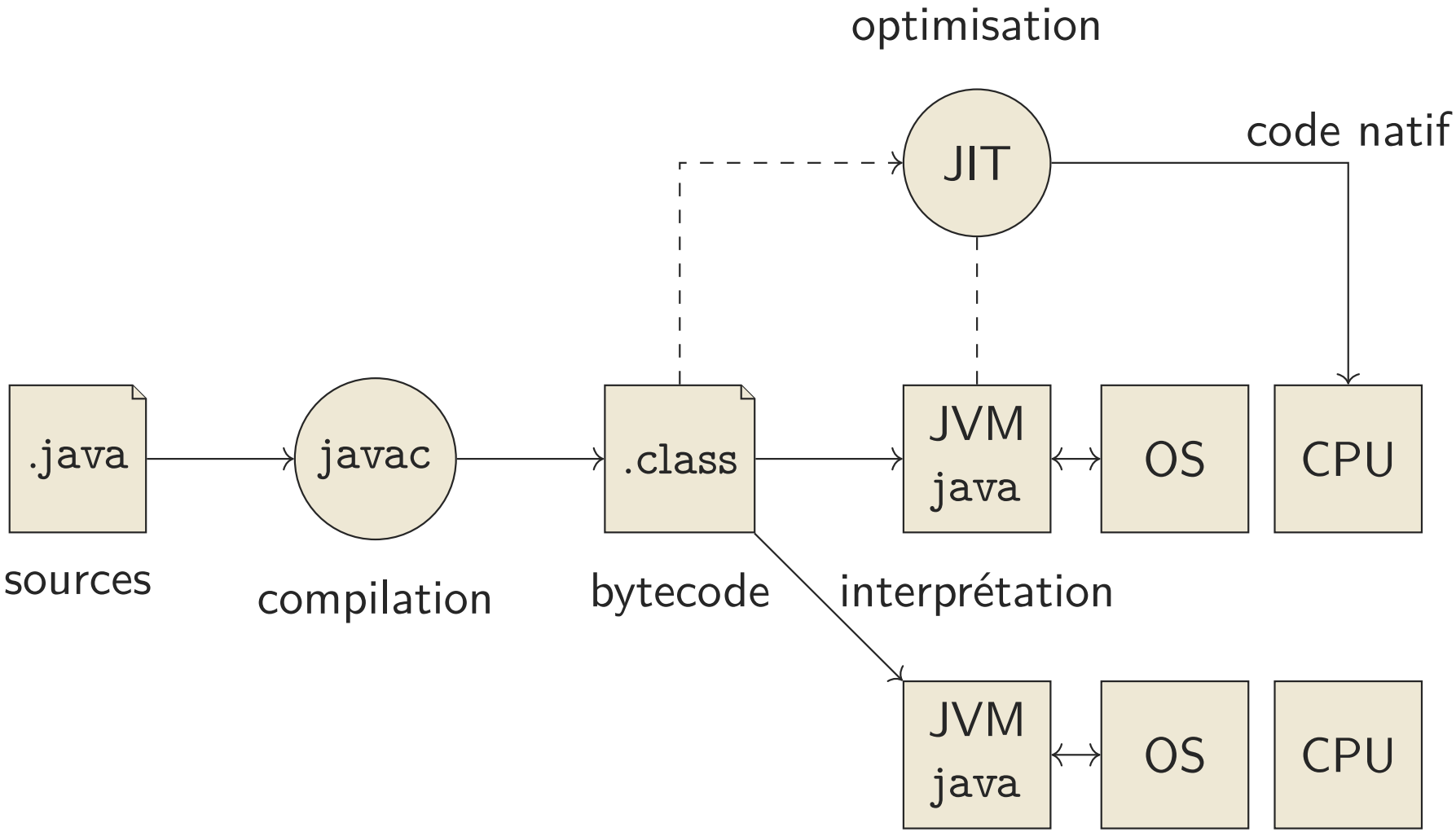
$\Rightarrow$ machine virtuelle

Définition (Machine virtuelle)

- abstraction machine réelle
- interprétation
- *bytecode* $\Rightarrow$ précompilation

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

```
javac Hello.java
java Hello
```


optimisation
JIT
code natif
.java
javac
.class
JVM
java
OS
CPU
sources
compilation
bytecode
interprétation
JVM
java
OS
CPU

```
public static int count(int sup) {
  int total = 0;
  for (int i = 1; i <= sup; i++) {
    total += i;
  }
  return total;
}
```

```
public static int count(int);
  Code:
     0: iconst_0
     1: istore_1
     2: iconst_1
     3: istore_2
     4: iload_2
     5: iload_0
     6: if_icmpgt     19
     9: iload_1
    10: iload_2
    11: iadd
    12: istore_1
    13: iinc          2, 1
    16: goto          4
    19: iload_1
    20: ireturn
```

javap -c

“Write once, run anywhere”

+ lent $\Rightarrow$ JIT

Définition (JIT)

- *Just In Time*
- optimisation dynamique
- à l'exécution

Autres langages pour JVM

- Scala
- Clojure
- Groovy
- Ceylon
- Golo
- ...

Concepts Objets

impératif → gestion de l'état ⇒ difficultés à raisonner

- instruction ⇒ changement d'état
- ordre des instructions
- connaissance de l'état global

fonctionnel : pas d'état

objet : séparation de l'état

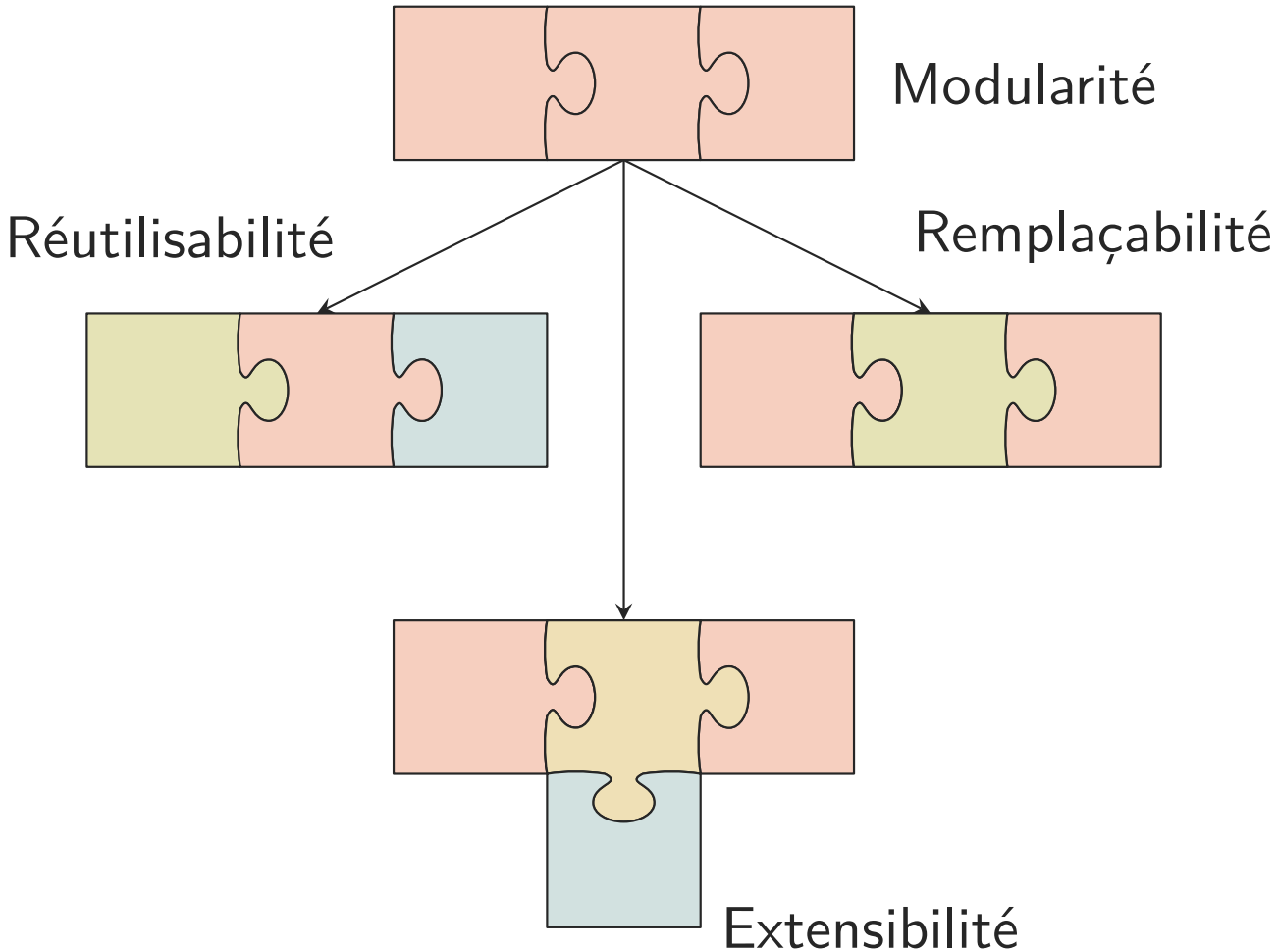
Définition (Objet)

- entité *autonome*
- état (masqué)
- comportement → manipulation de l'état

- objet
  → responsable de *son* état
  → dépendant *que* de *son* état
- état global ⇒ « somme » des états
- *isolation* ⇒ encapsulation

Définition (Modularité)

- décomposition
- $\Rightarrow$ entités autonomes
- indépendants
- collaboration
- découplage


Modularité
Réutilisabilité
Remplaçabilité
Extensibilité

autonomie → indépendant du contexte ⇒ réutilisable

Définition (Encapsulation)

- état (données)
- traitement (fonctions)
- $\Rightarrow$ entité unique : objet
- détails *masqués* $\Rightarrow$ boîte noire

$\Rightarrow$ responsabilité

collaboration $\rightarrow$ envoie de message (interface)

Principe : Tell don't ask

- décision → objet
- responsabilité
- encapsulation

encapsulation : interface ↔ implémentation

Définition (Interface)

- contrat (abstrait)
- messages
- accessibles → mise à disposition

⇒ spécification de l'interaction

Définition (Implémentation)

- réalisation (concrète)
- code effectif
- respect de l'interface

implémentations d'une même interface

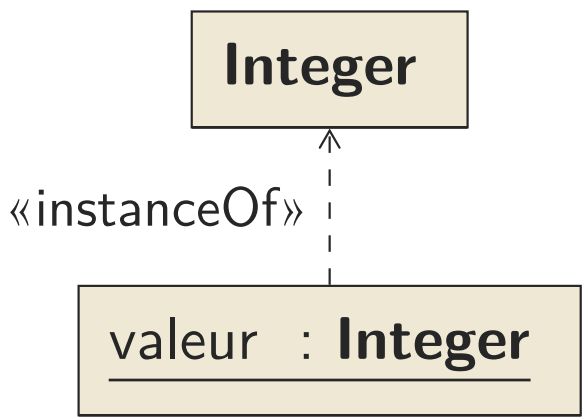
encapsulation $\Rightarrow$ remplaçabilité

Définition (Classe)

- définition de type (concret)
- ensemble d'objets → caractéristiques communes
- modèle d'objet ⇒ instance

Définition (Instance)

- objet défini par la classe
- valeur du type de la classe




```
Integer valeur = new Integer(1);
```

```
assert valeur instanceof Integer;
```

UML

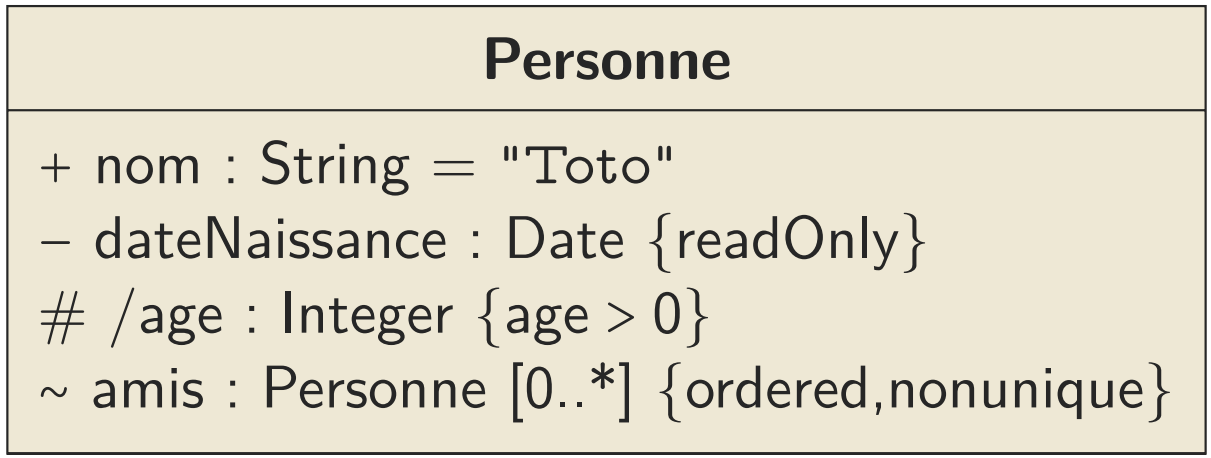
Unified Modeling Language

http://www.uml.org/

- langage modélisation
- graphique
- objet
- indépendant d'un langage
- conceptuel → implémentation
- standard : OMG `http://www.omg.org/spec/UML/`
  - 1.4.2 (ISO/IEC 19501) : 2001
  - 2.4.0 (ISO/IEC 19505-1 19505-2) : 2011
  - 2.5 : 2012 (*in process*)

Définition (Attribut)

- variable d'un objet
- donnée
- locale
- $\Rightarrow$ état




```
class Personne {
    public String nom = "Toto";
    private final Date dateNaissance;
    protected Integer age;
    List<Personne> amis;
}
```

Visibilité

- publique : + → `public`
- privée : − → `private`
- de paquet : ~ defaut
- protégée : # → `protected`

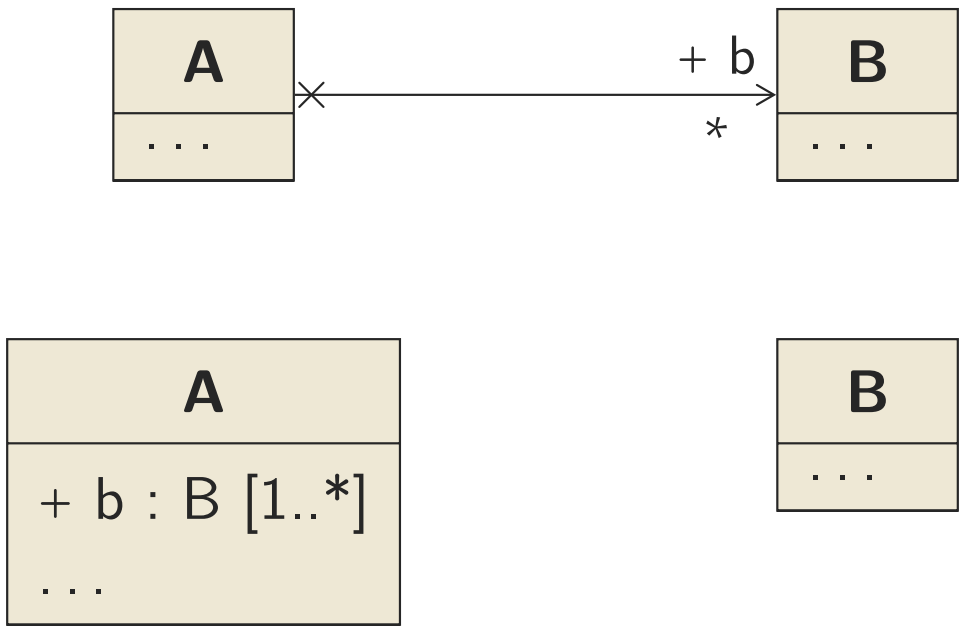
attribut dérivé : /age

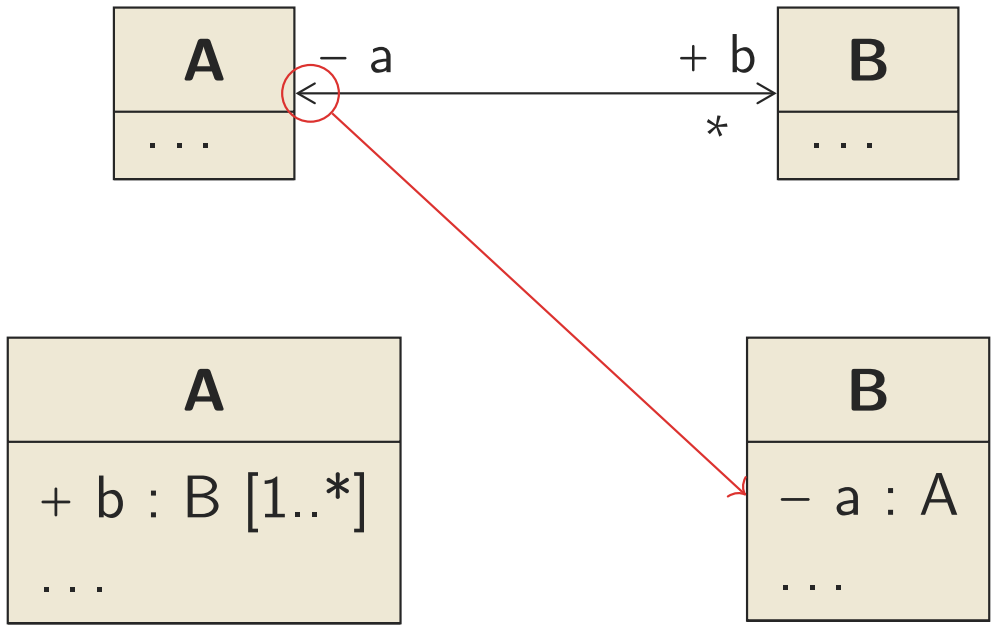
⇒ encapsulation

Association


A
. . .
+ a
0..1
nom ▸
– b
∗{ordered}
B
. . .

Navigabilité

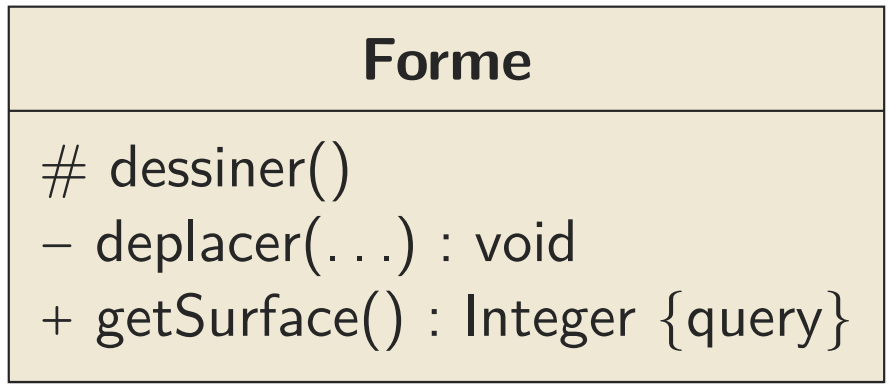




Modifier A.b ⇒ modifier B.a

Définition (Méthode)

- fonction
- traitement
- manipulation de l'état
- envoie de messages



visibilité nomOpération(*paramètres*) : *type retour* [*mult.*] {*prop.*}

```
class Point { }

class Forme {
   protected void dessiner() {
      /* traitement de dessin */
   };

   private void deplacer(Point p) {
      /* deplacement de la forme */
   };

   public Integer getSurface() {
      Integer s = 0;
      /* calcul de la surface */
      return s;
   }
}
```

```java
import java.util.*;

class Personne {
    public String nom = "Toto";
    private final Date dateNaissance;
    List<Personne> amis;

    protected Integer age() {
        Integer age = 0;
        /* calcul de l'age */
        return age;
    }
}
```

accès par le .

- `String sonNom = personne.nom;`
- `Integer sonAge = personne.age();`
- `forme.dessiner();`

Principe : CQS (*Command Query Separation*)

une opération est :

- une requête (query) sans effet de bord, retourne une valeur
- une commande (command) avec effet de bord, retourne *void*

- procédure vs. fonction
- return vs. output
- passage par valeur vs. passage par référence
- transparence référentielle

Définition (Constructeur)

→ création d'instance (initialisation)

- constructeur par défaut → pas de paramètre
- constructeur paramétré

mot-clé new

- `Personne p = new Personne();`
- `Personne p = new Personne(new Date(1990,4,1), "Joe");`

mot-clé `this`

- l'instance elle-même : `this.nom` (optionnel)
- le constructeur : `this()` (dans un autre constructeur)

```java
import java.util.*;

class Personne {
    public String nom = "Toto";
    private final Date dateNaissance;
    List<Personne> amis;

    public Personne() {
        this.dateNaissance = new Date();
    }

    public Personne(String nom) {
        this();
        this.nom = nom;
    }

    public Personne(Date naissance, String nom) {
        this.dateNaissance = naissance;
        this.nom = nom;
    }
}
```

Définition (Destructeur)

méthode spéciale appelée à la suppression de l'objet

```
public void finalize() {/* */}
```

Exemple

- suppression de fichier temporaire
- fermeture de connexion SGBD
- ...

⚠ appel non déterministe

Accesseurs

Définition (Accesseur)

méthodes d'accès aux attributs privé : « getter » / « setter »

Accesseurs

Convention « Beans »

- `TypeAttr getAttr() {return this.attr;}`
- `void setAttr(TypeAttr val) {this.attr = val;}`

```
MyObjet obj = new MyObjet();
obj.setA(1);
obj.setB("b");
obj.setC(42);
Integer a = obj.getA();
```

Accesseurs

Convention « builder »

- `TypeAttr attr() {return this.attr;}`
- `void attr(TypeAttr val) {this.attr = val;}`
- `TypeObjet attr(TypeAttr val) {this.attr = val; return this;}`

```
MyObjet obj = new MyObjet().a(1).b("b").c(42);
Integer a = obj.a();
```

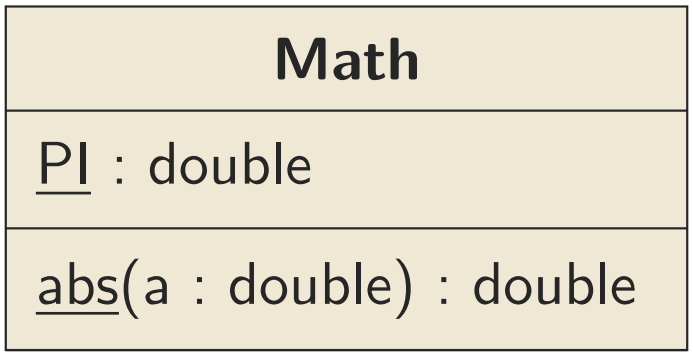
pourquoi ?

- contrôle d'accès
- contraintes

⇒ encapsulation

utiliser prudemment (voir *Tell don't ask*)

Attribut et méthode statique



- attribut de la classe : `Math.PI`
- commun à toutes les instances
- méthode qui s'applique à la classe : `Math.abs(42.0)`
- ✗ pas de `this`

Principe : OCP : open-closed principle (88,96)

- fermée à la modification (encapsulation)
- ouverte par extension (polymorphisme)

SOLID

Polymorphisme

Définition (Polymorphisme)

utilisation indépendante du type réel des éléments manipulés (arguments)

- *ad hoc*
  - transtypage
  - surcharge
- universel
  - générique / paramétré
  - inclusion

Définition (Transtypage)

conversion implicite du type d'une valeur

cast

Exemple

opérateurs : `1 + 2.5`, `"a" + 1`

Définition (Surcharge)

- plusieurs fonctions (méthodes)
- même nom
- paramètres différents (types, nombre) ⇒ signature

Exemple

- `void adopter(Personne p) {/*...*/}`
- `void adopter(Animal a) {/*...*/}`

Définition (Polymorphisme générique)

- traitement indépendant des types
- types génériques / paramétré
- structures complexes

Exemple

fonction calculant la longueur d'une liste chaînée.

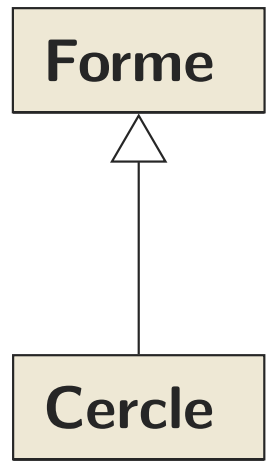
Définition (Polymorphisme d'inclusion)

- sous-type
- héritage / redéfinition

Sous-type

Sous-type

- « est un »
- sous-ensemble



remplaçablilité

$\Rightarrow$ même interface

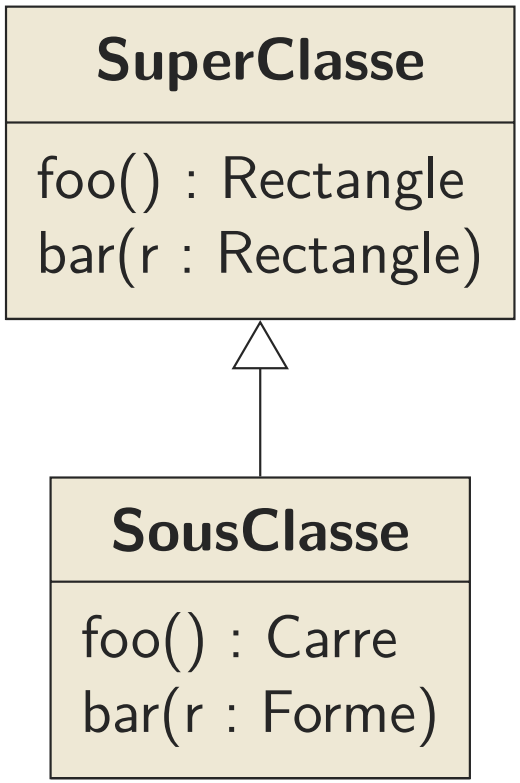
$\rightarrow$ interface compatible

Principe : Substituabilité Liskov (87)

Sous-type comportemental

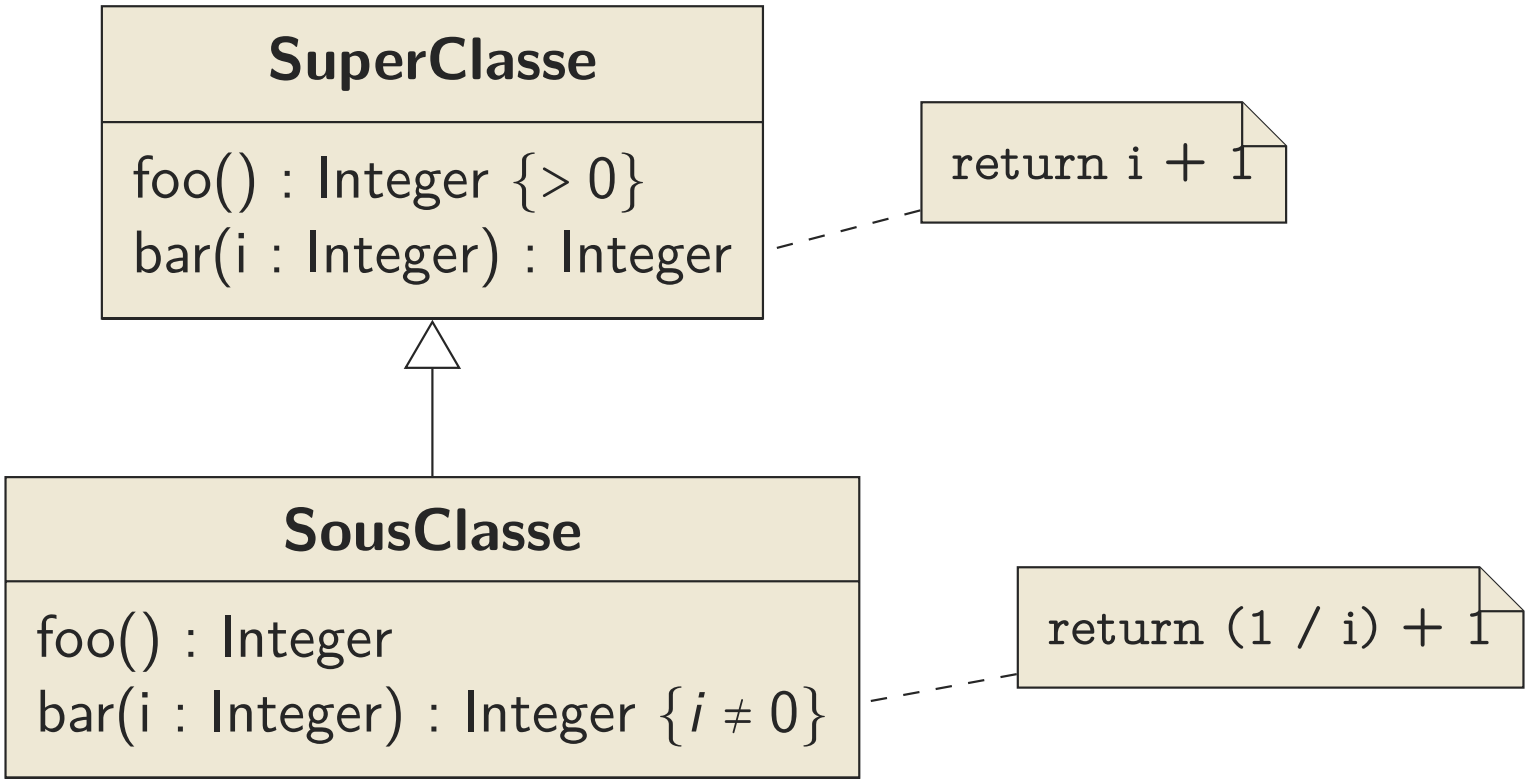
- *covariance* des sorties (retours, exceptions)
- pas de relachement des *postconditions*
- *contravariance* des entrées
- pas de renforcement des *préconditions*
- conservation des *invariants*
- conservation de la *mutabilité* et des *états*

SOLID



Exemple

- donne + restrictif : `Rectangle r = c.foo()`
  - ✓ `Carre`
  - ✗ `Forme` quelconque
- accepte − restrictif : `c.bar(new Rectangle())`
  - ✓ `Forme` quelconque
  - ✗ `Carre`



Exemple

- sortie + restrictif :
  ✗ `42 / c.foo()`
- entrée - restrictive :
  ✗ `c.bar(0)`

2 types de sous-typage :

- *nominal* : nommage explicite du super-type
- *structurel* : bonne interface (Liskov) ⇒ sous-type (duck-typing)

Héritage

Définition (Héritage)

transfert de propriété : super-type → sous-type

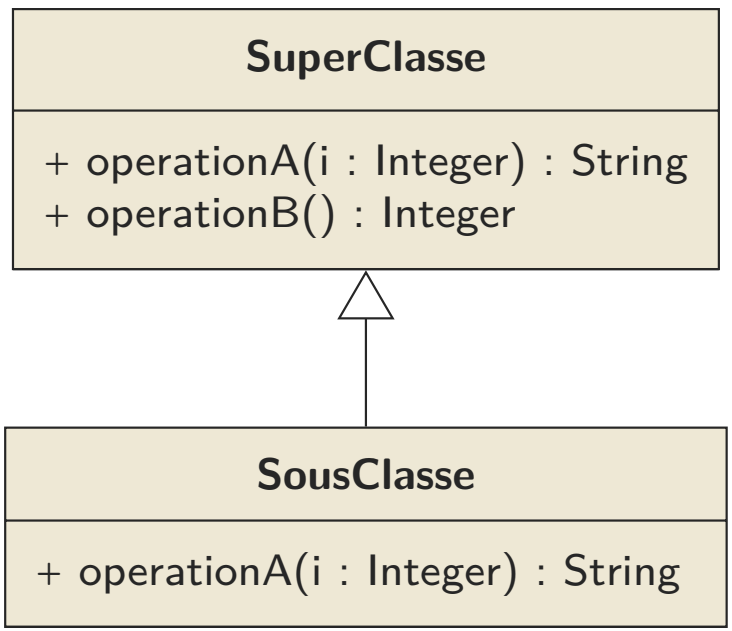
⇒ redéfinition

Définition (Redéfinition)

implémentation alternative pour une méthode héritée

- annotation : `@Override` → vérification
- mot-clé : `final` → redéfinition impossible
- mot-clé : `super` → super implémentation

Java




```
class SuperClasse {
  public String operationA(final Integer i) {
    return "a" + i;
  }
  public Integer operationB() { return 1; }
}
```

```
class SousClasse extends SuperClasse {
  @Override
  public String operationA(final Integer i) {
    return super.operationA(i) + "b";
  }
}
```

```
public class Heritage {
  public static void main(final String[] args) {
    SousClasse s = new SousClasse();
    assert s.operationA(1).equals("a1b");
    assert s.operationB() == 1;
  }
}
```

Smalltalk

```
class SuperClasse {
  public String operationA(final Integer i) {
    return "a" + i;
  }
  public Integer operationB() { return 1; }
}
```

```
Object subclass : SuperClasse [
  operationA : i [ ^('a', i asString). ]

  operationB [ 1 ]
]
```

```
class SousClasse extends SuperClasse {
  @Override
  public String operationA(final Integer i) {
    return super.operationA(i) + "b";
  }
}
```

```
SuperClasse subclass : SousClasse [
  operationA : i [
    ^((super operationA : i), 'b').
  ]
]
```

```
public class Heritage {
  public static void main(final String[] args) {
    SousClasse s = new SousClasse();
    assert s.operationA(1).equals("a1b");
    assert s.operationB() == 1;
  }
}
```

```
| s |
s  := SousClasse new.
(s operationA : 1) = 'a2b' ifFalse : [Error
      signal].
(s operationB) = 1 ifFalse : [Error signal].
```

Python

```
class SuperClasse {
  public String operationA(final Integer i) {
    return "a" + i;
  }
  public Integer operationB() { return 1; }
}
```

```
class SousClasse extends SuperClasse {
  @Override
  public String operationA(final Integer i) {
    return super.operationA(i) + "b";
  }
}
```

```
public class Heritage {
  public static void main(final String[] args) {
    SousClasse s = new SousClasse();
    assert s.operationA(1).equals("a1b");
    assert s.operationB() == 1;
  }
}
```

```
class SuperClasse :
    def operationA(self, i) :
        return "a" + str(i)

    def operationB(self) :
        return 1
```

```
class SousClasse(SuperClasse) :
    def operationA(self, i) :
        return super().operationA(i) + "b"
```

```
s = SousClasse()
assert s.operationA(1) == "a1b"
assert s.operationB() == 1
```

C#

```
class SuperClasse {
  public String operationA(final Integer i) {
    return "a" + i;
  }
  public Integer operationB() { return 1; }
}
```

```
class SousClasse extends SuperClasse {
  @Override
  public String operationA(final Integer i) {
    return super.operationA(i) + "b";
  }
}
```

```
public class Heritage {
  public static void main(final String[] args) {
    SousClasse s = new SousClasse();
    assert s.operationA(1).equals("a1b");
    assert s.operationB() == 1;
  }
}
```

```
class SuperClasse {
  public virtual string OperationA(int i) {
    return "a" + i;
  }

  public int OperationB() { return 1; }
}
```

```
class SousClasse  : SuperClasse {
  public override string OperationA(int i) {
    return base.OperationA(i) + "b";
  }
}
```

```
class Heritage {
  public static void Main(string[] args) {
    SousClasse s = new SousClasse();
    if (s.OperationA(1) != "a1b")
      throw new Exception("error");
    if (s.OperationB() != 1)
      throw new Exception("error");
  }
}
```

PHP

```
class SuperClasse {
  public String operationA(final Integer i) {
    return "a" + i;
  }
  public Integer operationB() { return 1; }
}
```

```
class SousClasse extends SuperClasse {
  @Override
  public String operationA(final Integer i) {
    return super.operationA(i) + "b";
  }
}
```

```
public class Heritage {
  public static void main(final String[] args) {
    SousClasse s = new SousClasse();
    assert s.operationA(1).equals("a1b");
    assert s.operationB() == 1;
  }
}
```

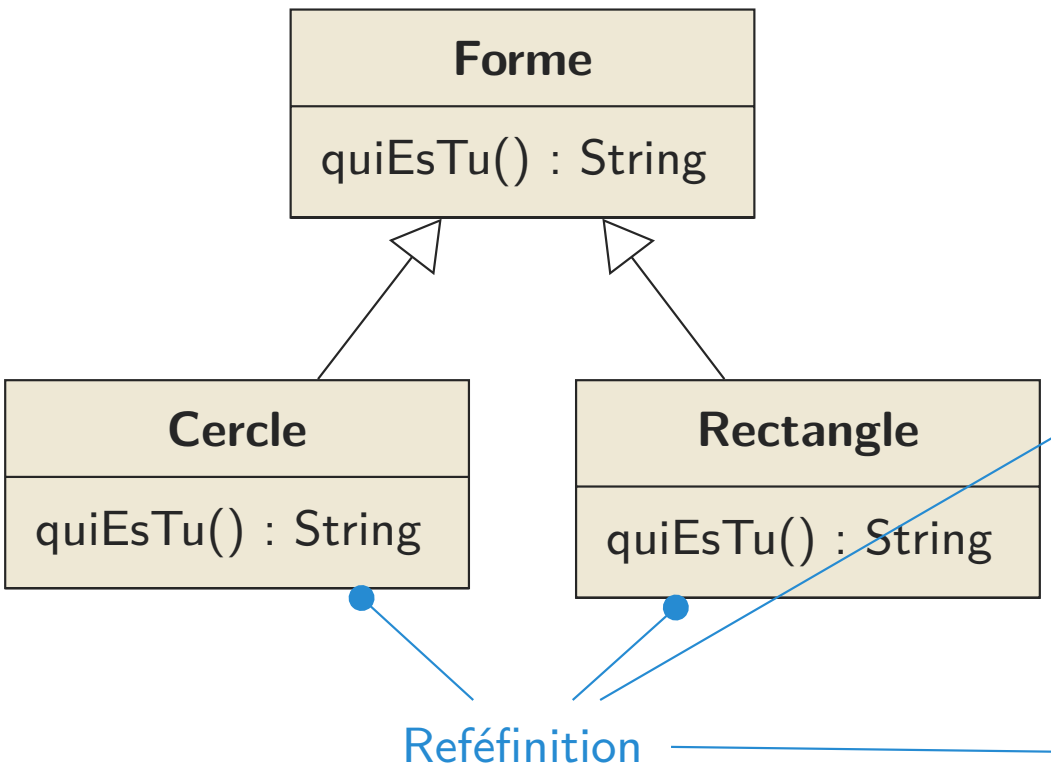
```
class SuperClasse {
  function operationA($i) {
    return "a" . $i;
  }

  function operationB() {return 1;}
}
```

```
class SousClasse extends SuperClasse {
  function operationA($i) {
    return parent : :operationA($i) . "b";
  }
}
```

```
$s = new SousClasse();
if ($s->operationA(1) !== "a1b") {
  throw new Exception();
}
if ($s->operationB() !== 1) {
  throw new Exception();
}
```

Redéfinition et surcharge

Redéfinition

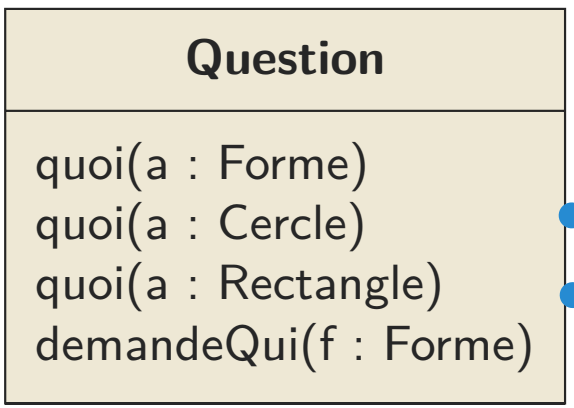



```
class Forme {
  public String quiEsTu() {
    return "une forme" ;
  }
}
```

```
class Cercle extends Forme {
  @Override
  public String quiEsTu() {
    return "un cercle" ;
  }
}
```

```
class Rectangle extends Forme {
  @Override
  public String quiEsTu() {
    return "un rectangle" ;
  }
}
```

Surcharge



Surcharge

```
class Question {
  public void quoi(Forme a) {
    System.out.println("j'ai une forme");
  }

  public void quoi(Cercle a) {
    System.out.println("j'ai un cercle");
  }

  public void quoi(Rectangle a) {
    System.out.println("j'ai un rectangle");
  }

  public void demandeQui(Forme f) {
    System.out.println("? " + f.quiEsTu());
  }
}
```

Comportement

Redéfinition (appel direct)

```
Question q = new Question();

Forme f = new Forme();
Cercle c = new Cercle();
Rectangle r = new Rectangle();
Forme fc = (Forme) (new Cercle());

Forme[] t = {f, c, r, fc};
```

```
System.out.println("# Redefinition directe");
System.out.println(f.quiEsTu());
System.out.println(c.quiEsTu());
System.out.println(r.quiEsTu());
System.out.println(fc.quiEsTu());
```

```
# Redefinition directe
une forme
un cercle
un rectangle
un cercle
```

Comportement

Redéfinition (dans un tableau)

```
Question q = new Question();

Forme f = new Forme();
Cercle c = new Cercle();
Rectangle r = new Rectangle();
Forme fc = (Forme) (new Cercle());

Forme[] t = {f, c, r, fc};
```

```
System.out.println("# Redefinition boucle");
for (int i = 0; i < t.length; i++) {
  System.out.println(t[i].quiEsTu());
}
```

```
# Redefinition boucle
une forme
un cercle
un rectangle
un cercle
```

Comportement

Sous-typage

```
Question q = new Question();

Forme f = new Forme();
Cercle c = new Cercle();
Rectangle r = new Rectangle();
Forme fc = (Forme) (new Cercle());

Forme[] t = {f, c, r, fc};
```

```
System.out.println("# Sous-typage");
q.demandeQui(f);
q.demandeQui(c);
q.demandeQui(r);
q.demandeQui(fc);
```

```
# Sous-typage
? une forme
? un cercle
? un rectangle
? un cercle
```

Comportement

Surcharge (appel direct)

```
Question q = new Question();

Forme f = new Forme();
Cercle c = new Cercle();
Rectangle r = new Rectangle();
Forme fc = (Forme) (new Cercle());

Forme[] t = {f, c, r, fc};
```

```
System.out.println("# Surcharge directe");
q.quoi(f);
q.quoi(c);
q.quoi(r);
q.quoi(fc);
```

```
# Surcharge directe
j'ai une forme
j'ai un cercle
j'ai un rectangle
j'ai une forme
```

Comportement

Surcharge (dans un tableau)

```
Question q = new Question();

Forme f = new Forme();
Cercle c = new Cercle();
Rectangle r = new Rectangle();
Forme fc = (Forme) (new Cercle());

Forme[] t = {f, c, r, fc};
```

```
System.out.println("# Surcharge boucle");
for (int i = 0; i < t.length; i++) {
  q.quoi(t[i]);
}
```

```
# Surcharge boucle
j'ai une forme
j'ai une forme
j'ai une forme
j'ai une forme
```

sélection en fonction du type résolution (*dispatch*)

- dynamique : exécution
- statique : compilation

Java (et beaucoup d'autres)

- redéfinition : dynamique
- surcharge : statique

liaison statique ↔ liaison tardive/dynamique
early/static binding ↔ *late/dynamic binding*

Classe abstraite et interface

Définition (Classe abstraite)

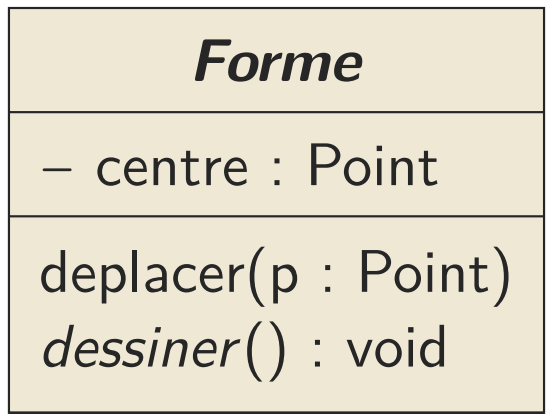
classe non instanciable

mot-clé `abstract`

Définition (Méthode abstraite)

Méthode sans implémentation

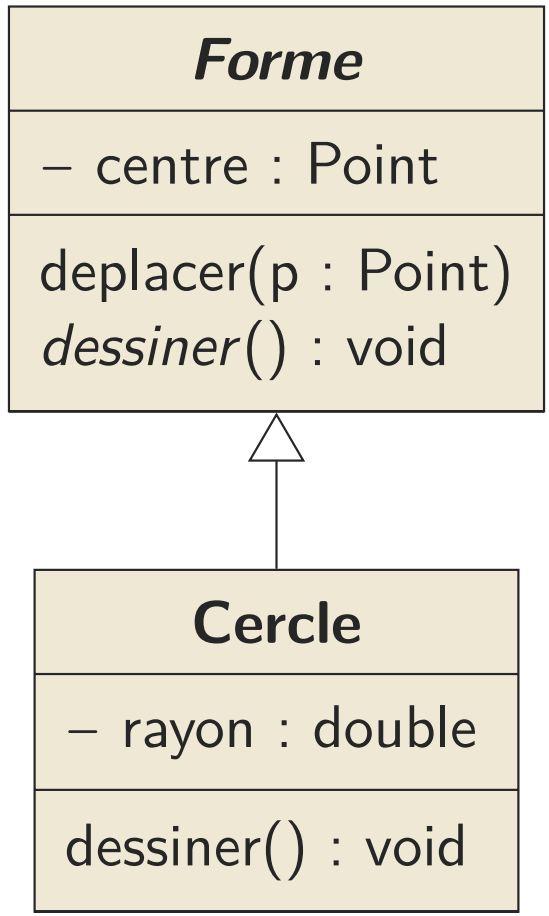
mot-clé `abstract`


Forme
– centre : Point
deplacer(p : Point)
dessiner() : void

- méthode abstraite $\Rightarrow$ classe abstraite
- classe concrète ✗ méthode abstraite
- classe abstraite
  - ✓ constructeur
  - ✓ methode concrète
  - ✓ attributs

```
abstract class Forme {

  private Point centre;

  Forme(Point centre) {
    deplacer(centre);
  }

  public void deplacer(Point p) {
    this.centre = requireNonNull(p);
  }

  public abstract void dessiner();
}
```

⇒ extension par classe concrète



```
class Cercle extends Forme {
  private double rayon;

  Cercle(Point centre, double rayon) {
    super(centre);
    this.rayon = rayon;
  }

  public void dessiner() {
    System.out.println("ceci est un cercle");
  }
}
```

Définition (Interface)

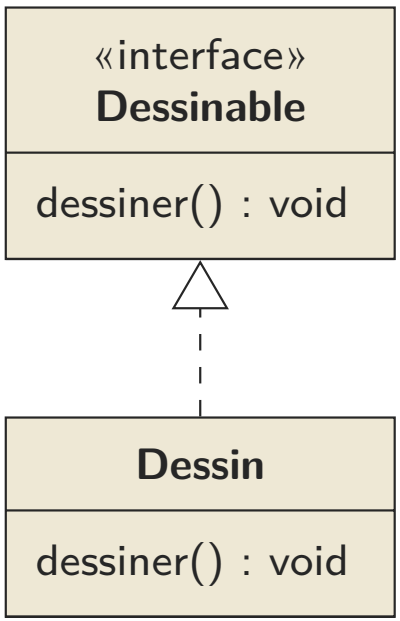
- ensemble de signatures de méthodes
- pas d'implémentation
- définie un type (abstrait)
- classe :
  - → *implémente* 1..*n* interface
  - ⇒ fournir une implémentation aux méthodes de l'interface

≅ classe « purement » abstraite

mots-clés : `interface`, `implements`

convention de nommage :

- ≈ `I*`
- ✓ `*able`



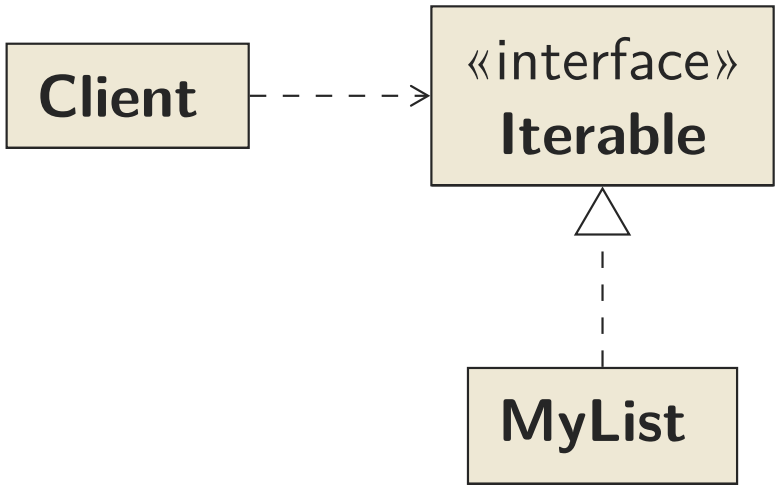

```
interface Dessinable {
  void dessiner();
}
```

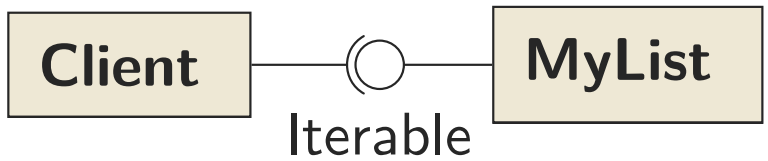
```
class Dessin implements Dessinable {
  public void dessiner() {
    System.out.println("Voici un joli dessin...");
  }
}
```

```
class Peinture implements Dessinable, Vendable {
  public void dessiner() {
    System.out.println("Une belle peinture !");
  }

  public void vendre(Personne p) {
    System.out.println("vendue a " + p);
  }
}
```

- séparation interface / implémentation
- dépendance sur interface
- $\Rightarrow$ découplage

interfaces minimales → sous-typage structurel

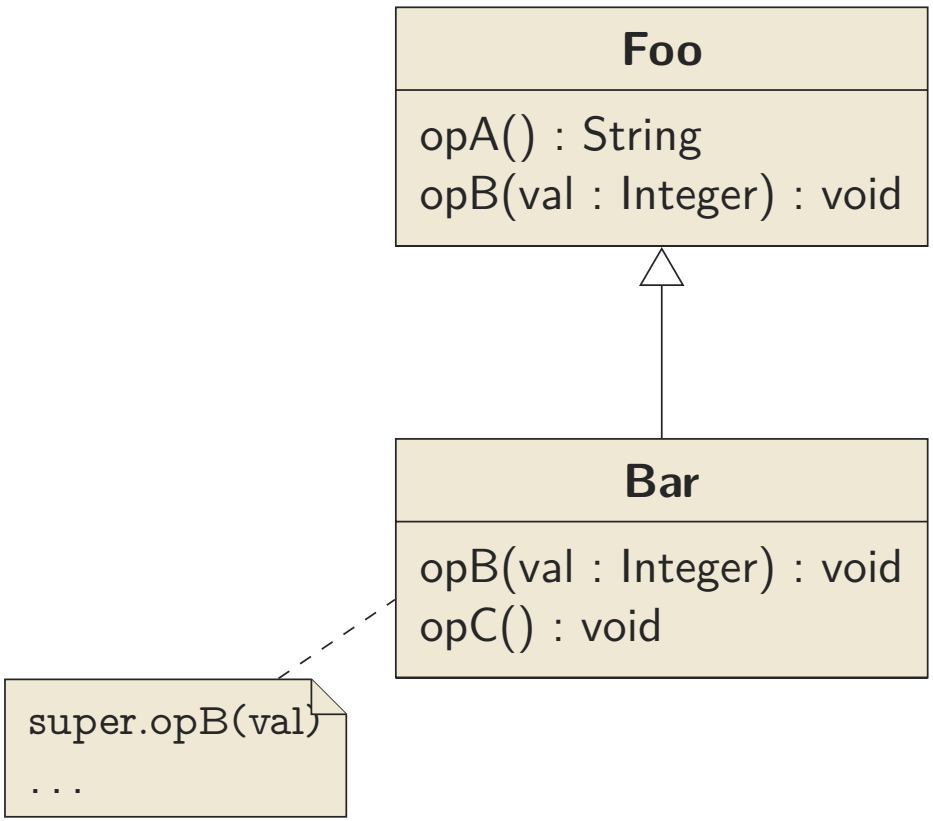




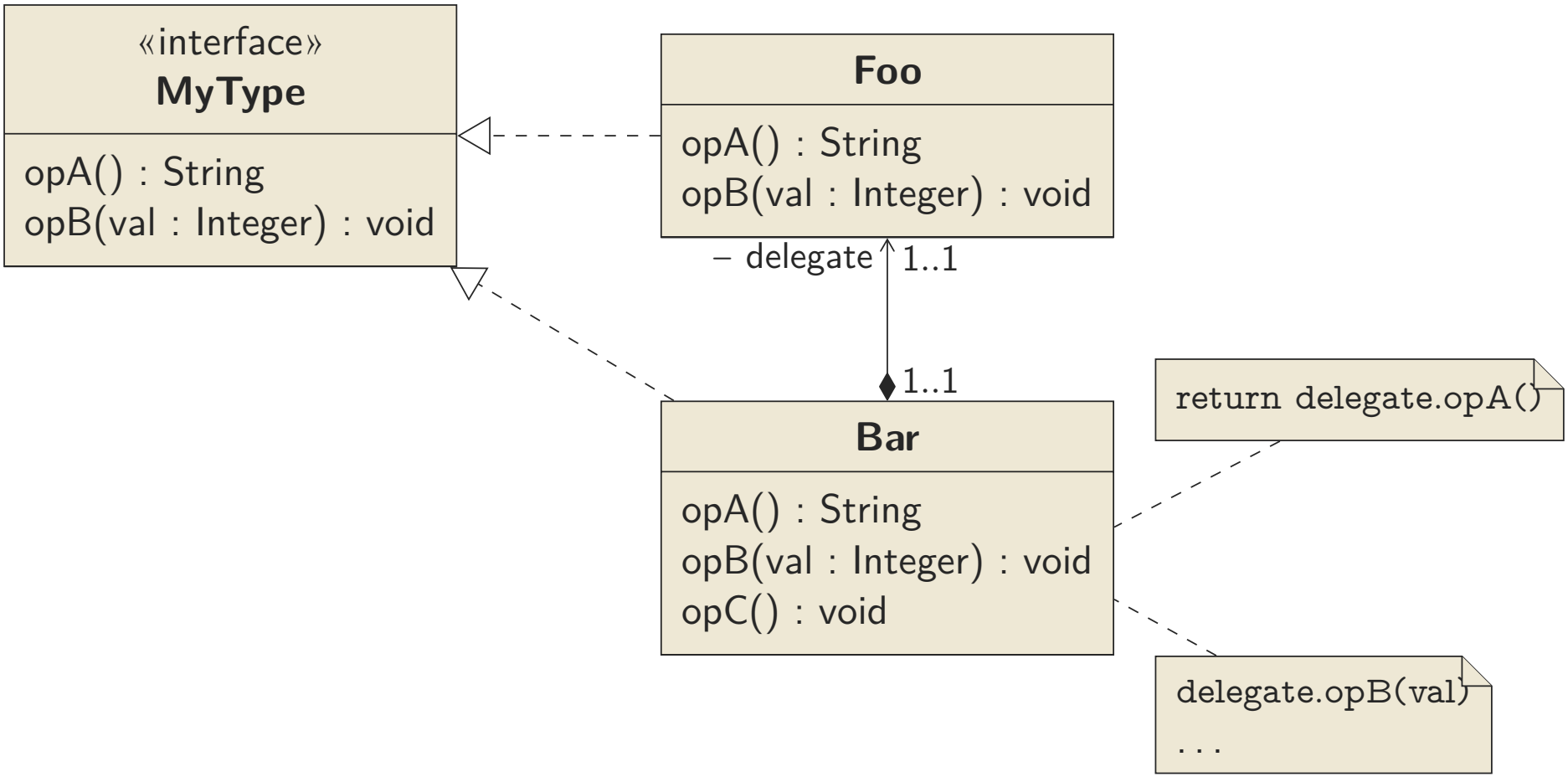

```
interface Iterable {/* */}

class MyList implements Iterable {/* */}

class Client {
  private Iterable elements;
  /* */
}
```

Délégation

- réutilisation
- sans sous-typage
- ≈ alternative à l'héritage




```java
class Foo {
  public String opA() {
    return "Hello World";
  }

  public void opB(int val) {
    // ...
  }
}
class Bar extends Foo {
  @Override
  public void opB(int val) {
    super.opB(val);
    // ...
  }

  public void opC() {
    //...
  }
}
```


«interface»
MyType
opA() : String
opB(val : Integer) : void
Foo
opA() : String
opB(val : Integer) : void
– delegate
1..1
1..1
Bar
opA() : String
opB(val : Integer) : void
opC() : void
return delegate.opA()
delegate.opB(val)
...

```
interface MyType {
  String opA();
  void opB(int val);
}

class Foo implements MyType {
  public String opA() {
    return "Hello World";
  }

  public void opB(int val) {
    //...
  }
}
```

```
class Bar implements MyType {
  private final Foo delegate = new Foo();

  public String opA() {
    return delegate.opA();
  }

  public void opB(int val) {
    delegate.opB(val);
    //...
  }

  public void opC() {
    //...
  }
}
```

```
class Bar extends Foo {
  @Override
  public void opB(int val) {
    super.opB(val);
    // ...
  }

  public void opC() {
    //...
  }
}
```

```
class Bar implements MyType {
  private final Foo delegate = new Foo();

  public String opA() {
    return delegate.opA();
  }

  public void opB(int val) {
    delegate.opB(val);
    //...
  }

  public void opC() {
    //...
  }
}
```

✗ si

- `Bar` même type que `Foo`
- pas d'interface du type commun
- `Foo` pas modifiable

Principe :

Préférer la composition/délégation à l'héritage

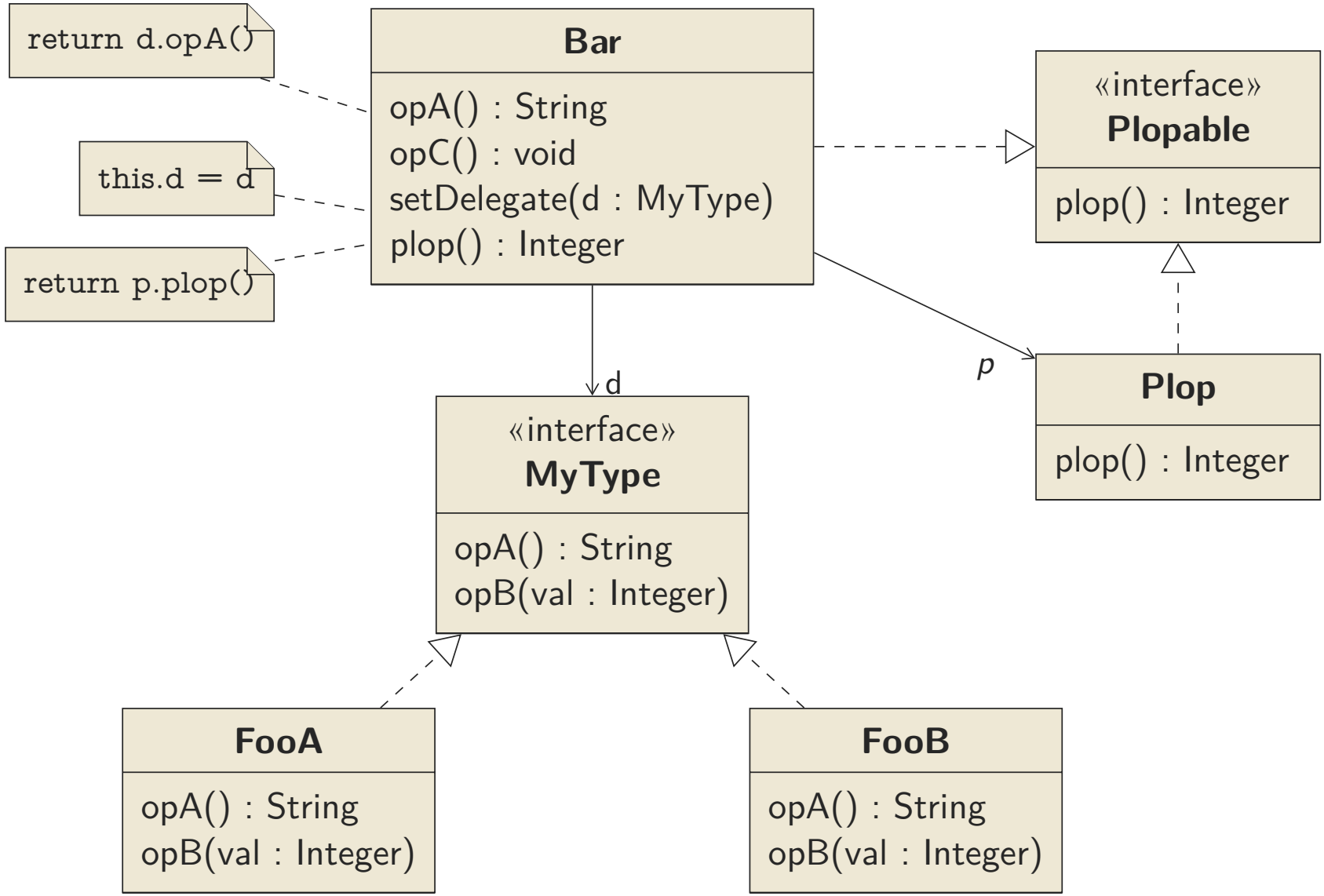
- hiérarchie de types : interfaces
- implémentations : finales
- réutilisation : composition/délégation

⇒ séparation « héritage »/sous-typage

Exemple

Go

- ✗ redéfinition complexe : e.g. `equals` ou `compareTo`
  ⇒ commutatif, transitif, . . .
- ✗ couplage fort
- ✓ composition plus flexible
  - sélective : pas toutes les méthodes ⇒ pas forcément sous-type
  - dynamique : changement à l'exécution (*stratégie*)
  - délégation multiple
- ≈ gestion « manuelle » (selon le langage)
  ⇒ simule l'héritage
  méta-programmation → automatique (et plus propre)


return d.opA()
this.d = d
return p.plop()
Bar
opA() : String
opC() : void
setDelegate(d : MyType)
plop() : Integer
«interface»
Plopable
plop() : Integer
d
p
«interface»
MyType
opA() : String
opB(val : Integer)
Plop
plop() : Integer
FooA
opA() : String
opB(val : Integer)
FooB
opA() : String
opB(val : Integer)

Types génériques

- type générique
- type paramétré
- *generics*
- *template*

⇒ polymorphisme paramétré

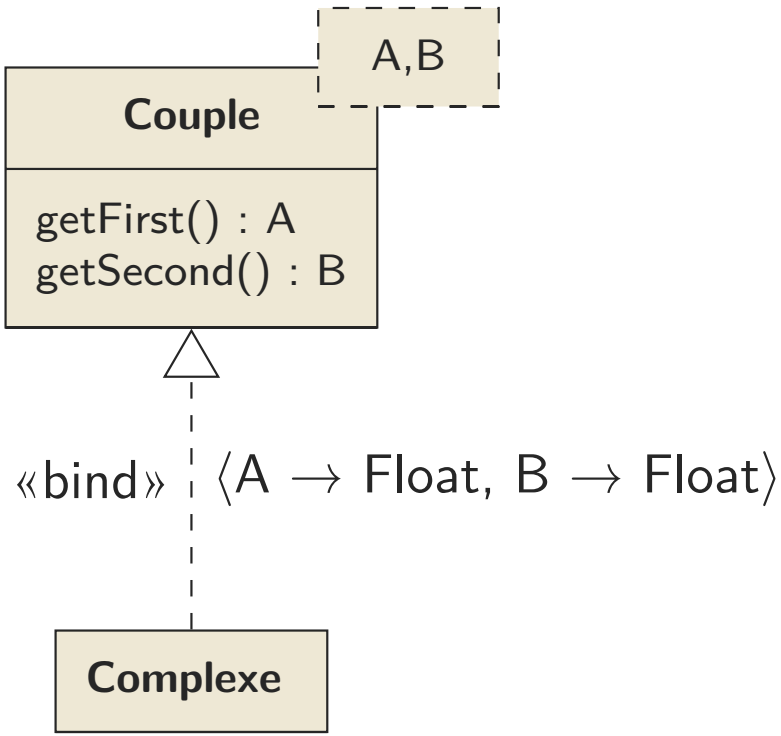
Java 1.5

Définition (Type générique)

- structure (classe)
- méthode
- type des éléments inconnu à la définition
- → paramétré
- défini :
  - instanciation
  - sous-typage (*binding*)

Exemple

- collections
- classe *comparable*
- ...




```java
class Couple<A, B> {
  private A a;
  private B b;

  public Couple(final A first, final B second) {
    this.a = first;
    this.b = second;
  }

  public A getFirst() { return a; }
  public B getSecond() { return b; }

  @Override
  public String toString() {
    return String.format("(%s; %s)", a, b);
  }
}
```

```java
class Complexe extends Couple<Double, Double> {
  public Complexe(final Double first, final Double
      second) {
    super(first, second);
  }
}
```

```
Complexe c = new Complexe(new Double(1.3), new Double(5.5));
```

```
Complexe c2 = new Complexe(1.3, 5.5);
```

autoboxing, depuis Java 1.5

```
Couple<String, Integer> a = new Couple<String, Integer>("Coucou", 2);
```

```
Couple<String, Integer> a2 = new Couple<>("Coucou", 2);
```

inférence de type, depuis Java 1.7

Conventions

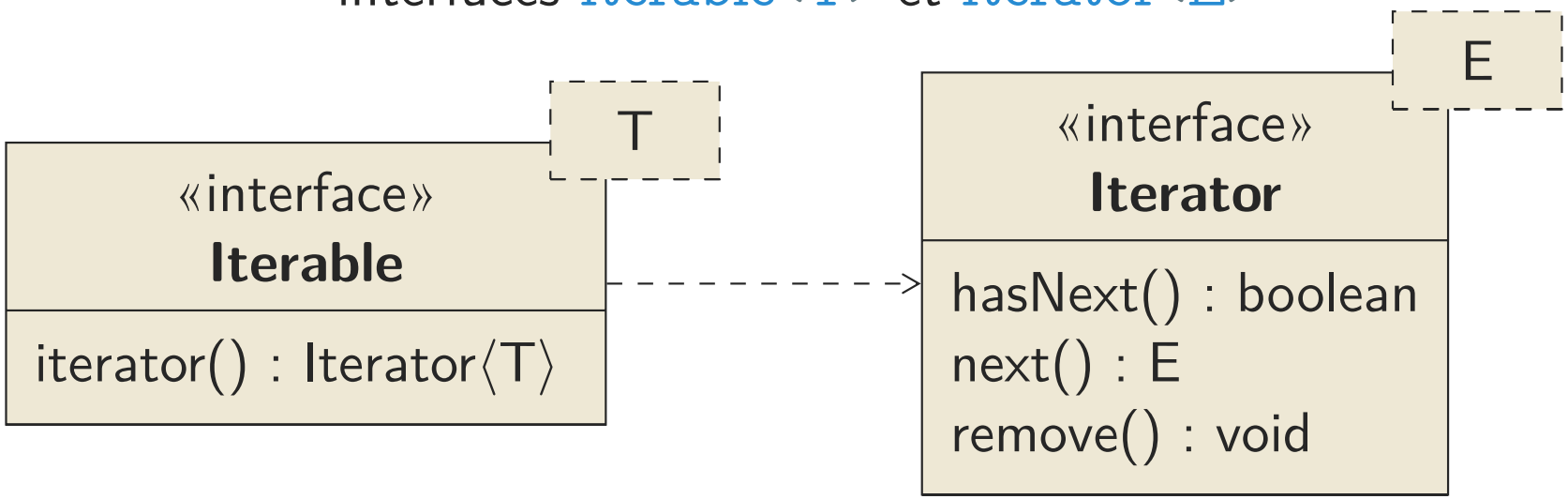
- `T` : Type
- `E` : Element (collections)
- `N` : Number
- `K` : Key
- `V` : Value

Collections

```
import java.util.*;
```

Iterateur

interfaces `Iterable<T>` et `Iterator<E>`



→ boucles simplifiées

```
import java.util.*;

class TestIterator {
   public static void main(final String[] args) {
      List<String> l = new ArrayList<String>();
      l.add("a"); l.add("b"); l.add("c");

      for (Iterator<String> i = l.iterator(); i.hasNext();) {
         System.out.println(i.next());
      }
   }
}
```

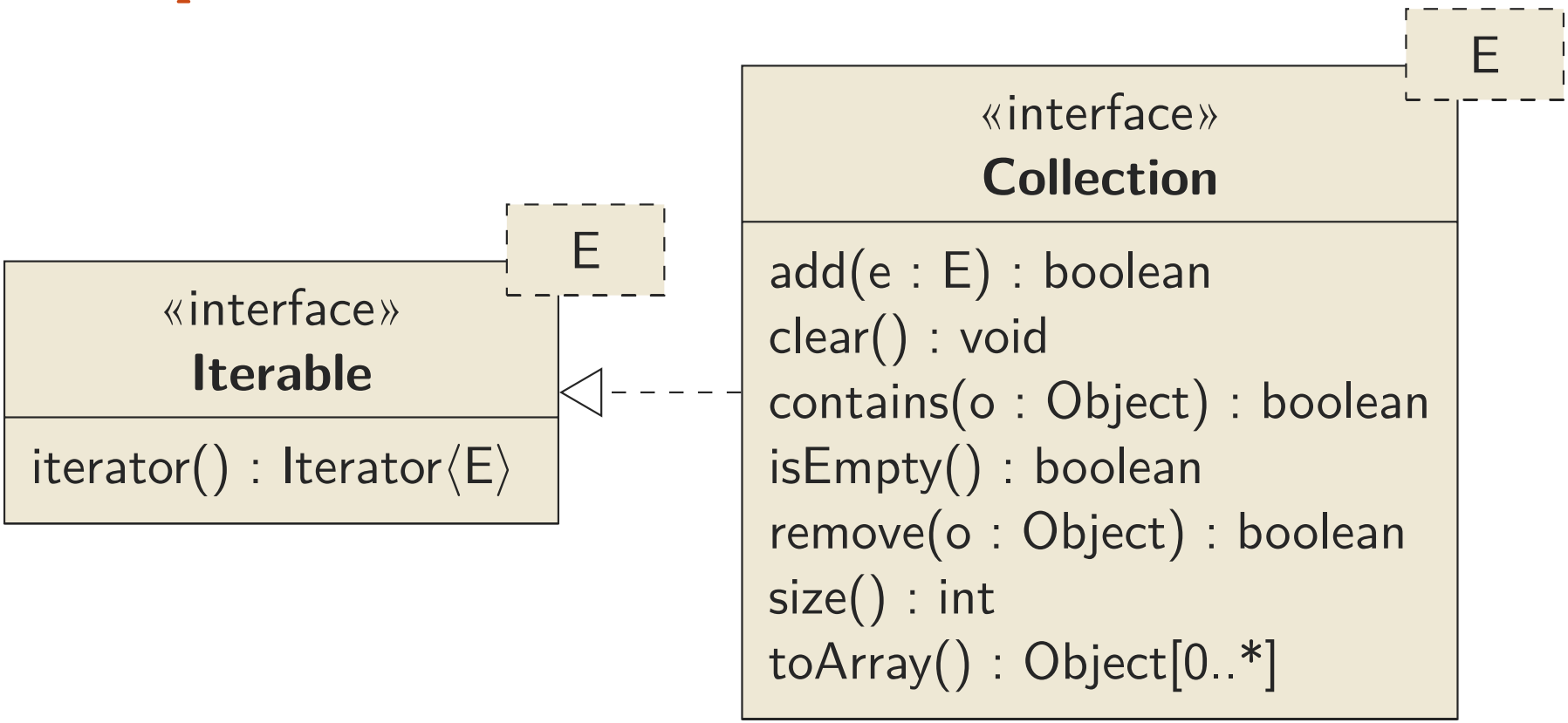
→ boucles encore plus simplifiées

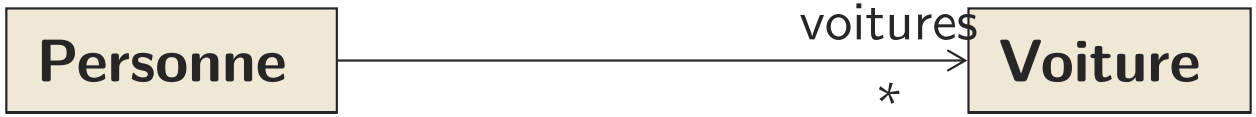
```
import java.util.*;

class TestIteratorE {
   public static void main(final String[] args) {
      List<String> l = new ArrayList<String>();
      l.add("a"); l.add("b"); l.add("c");

      for (String s : l) {
         System.out.println(s);
      }
   }
}
```

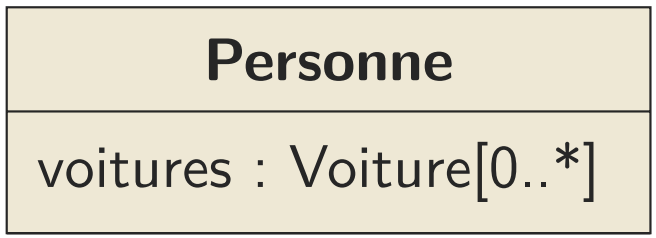
Collections

Ensemble d'interfaces pour les conteneurs

```
public interface Collection<E> extends Iterable<E>
```

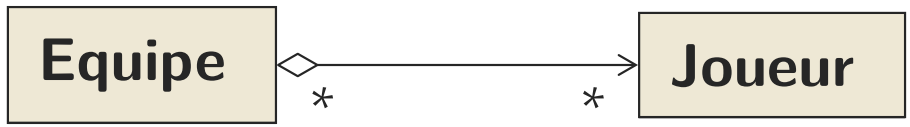







```
class Voiture {/* */}

class Personne {
  Collection<Voiture> voitures = new ArrayList<Voiture>();
}
```

Agrégation



- collection d'instance
- interface de type collection
- indépendance des éléments

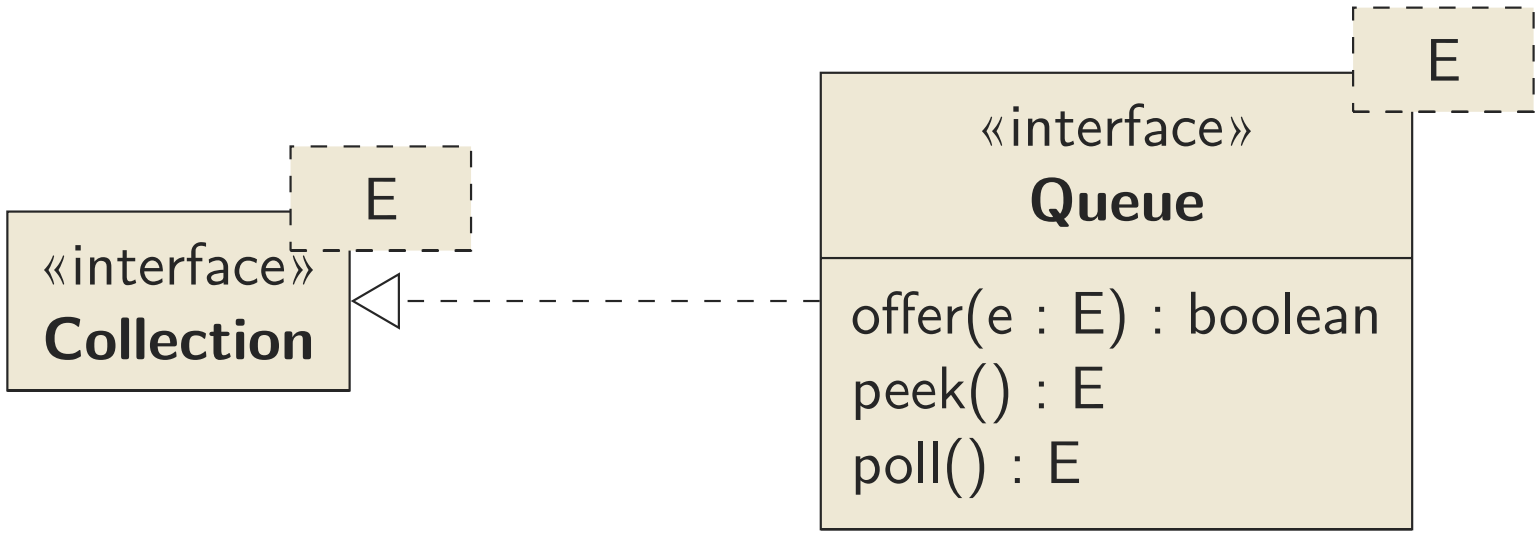
```
class Joueur { /* */ }

class Equipe implements Collection<Joueur> {
  private Collection<Joueur> joueurs = new ArrayList<Joueur>();
}
```

- encapsulation
- délégation

Queue

Fille d'attente (FIFO)

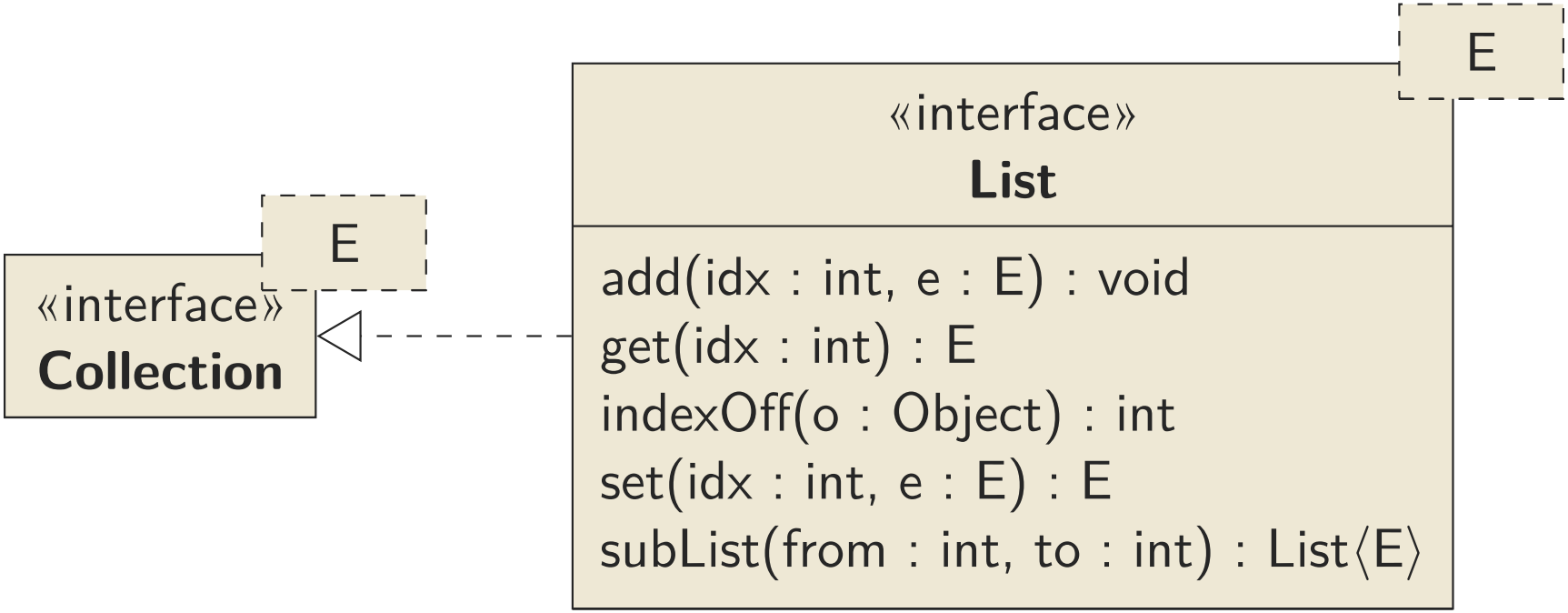
```
public interface Queue<E> extends Collection<E>
```



voir aussi Deque

List

```
public interface List<E> extends Collection<E>
```

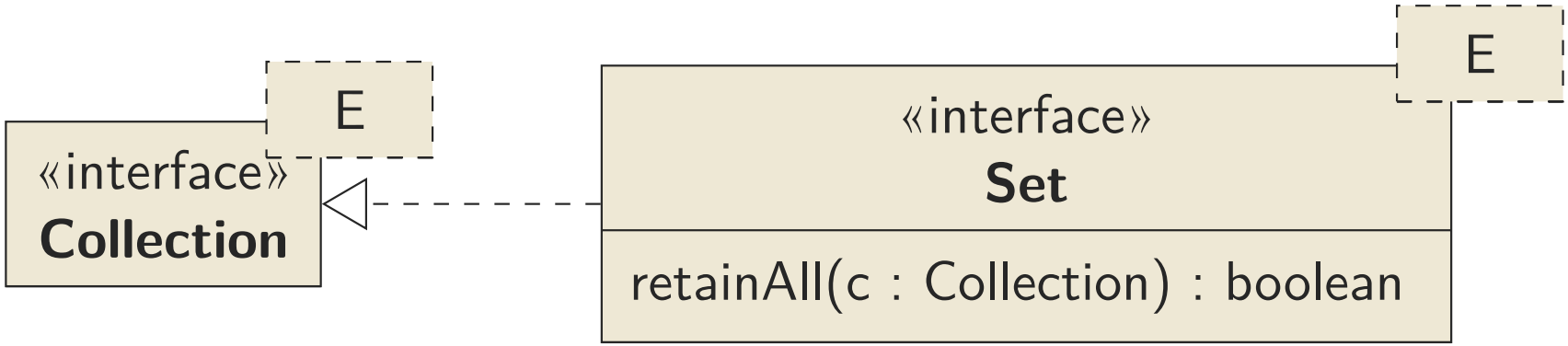


- ArrayList
- LinkedList

voir aussi Stack et Vector

Set

public interface Set<E> extends Collection<E>

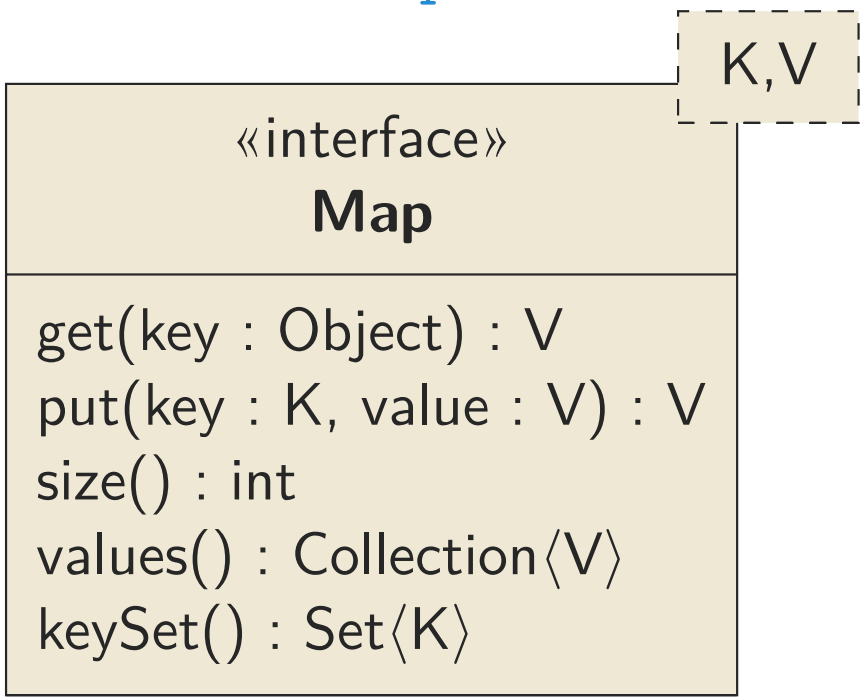


- EnumSet
- HashSet
- LinkedHashSet
- TreeSet

voir aussi SortedSet

Map

```
public interface Map<K,V>
```



- HashMap
- LinkedHashMap
- IdentityHashMap
- TreeMap
- WeakHashMap

voir aussi SortedMap

Méthodes

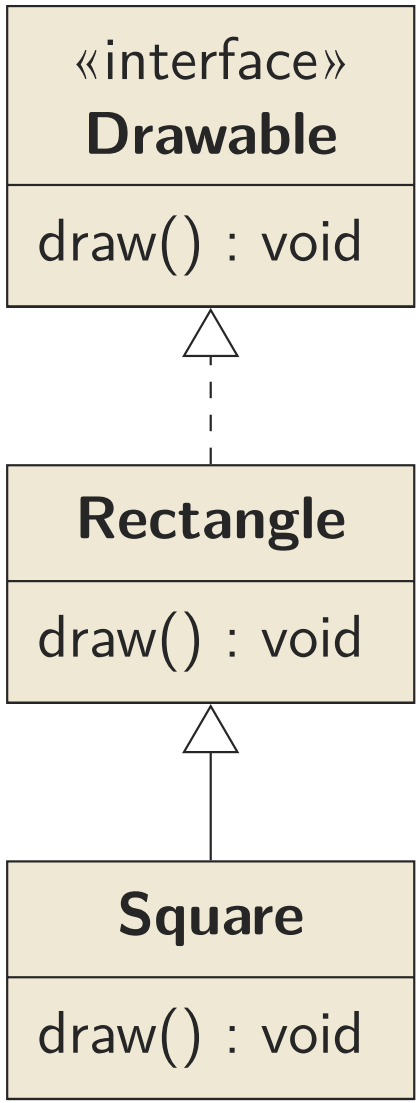
méthodes génériques

```
static public <E> List<E> list(E... elements) {
  List<E> result = new ArrayList<>();
  for (E elt  : elements) {
    result.add(elt);
  }
  return result;
}
```

```
List<String> l = list("a", "b", "c");
```

Sous-typage de types génériques

```java
interface Drawable {
  void draw();
}

class Rectangle implements Drawable {
  public void draw() {
    System.out.println("draw a rectangle...");
  }
}

class Square extends Rectangle {
  @Override
  public void draw() {
    System.out.println("draw a square...");
  }
}
```



✗ `Collection<Drawable> c = new ArrayList<Square>();`

? `c.add(new Rectangle());`

→ type déclaré `Drawable` et résolution statique

```
static void drawAll0(Collection<Drawable> v) {
  for (Drawable e  : v) {
    e.draw();
  }
}
```

```
Collection<Drawable> cd = new ArrayList<Drawable>();
cd.add(new Rectangle());
cd.add(new Square());
cd.add(new Rectangle());
```

✓ `drawAll0(cd);`

```
draw a rectangle...
draw a square...
draw a rectangle...
```

```
static void drawAll0(Collection<Drawable> v) {
  for (Drawable e  : v) {
    e.draw();
  }
}
```

```
Collection<Rectangle> cr = new ArrayList<Rectangle>();
cr.add(new Rectangle());
cr.add(new Square());
cr.add(new Rectangle());
```

✗ `drawAll0(cr);`

```
error : incompatible types : Collection<Rectangle> cannot be converted to
      Collection<Drawable>
```

- ✓ `List<Rectangle>` est un sous-type de `Collection<Rectangle>`
- ⚠ `List<Rectangle>` n'est pas un sous-type de `List<Drawable>`

✓
```
Collection<Square> squares = new ArrayList<Square>();
```

⚠
```
Collection<Rectangle> rectangles = squares;
```

✓
```
rectangles.add(new Rectangle());
```

✗
```
Square sq = squares.get(0);
```

caractères jokers (*wildcards*)

```
static void printAll(Collection<?> v) {
  for (Object e  : v) {
    System.out.println(e);
  }
}
```

✓ `printAll(cd);`

✓ `printAll(cr);`

⚠ `printAll(new ArrayList<Toto>());`

covariance, contravariance, . . .

`<? extends T>`

⇒ n'importe quel sous-type de T

```
static void drawAll(Collection<? extends Drawable> v) {
  for (Drawable e : v) {
    e.draw();
  }
}
```

✓ `drawAll(cd);`

✓ `drawAll(cr);`

✗ `drawAll(new ArrayList<Toto>());`

`<? super T>`

⇒ n'importe quel super-type de T

```
<T extends Comparable<? super T>> T e
```

`e` est de n'importe quel type comparable à un de ses super-type

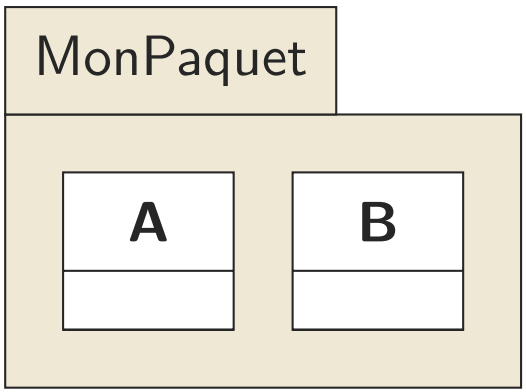
Bibliothèques standard de Java

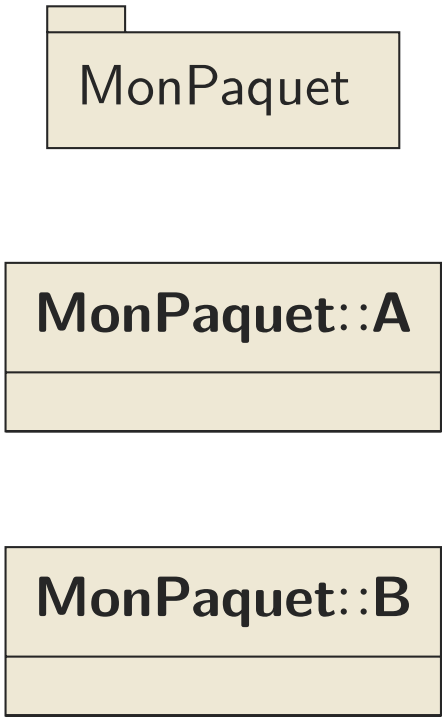
Définition (Paquet)

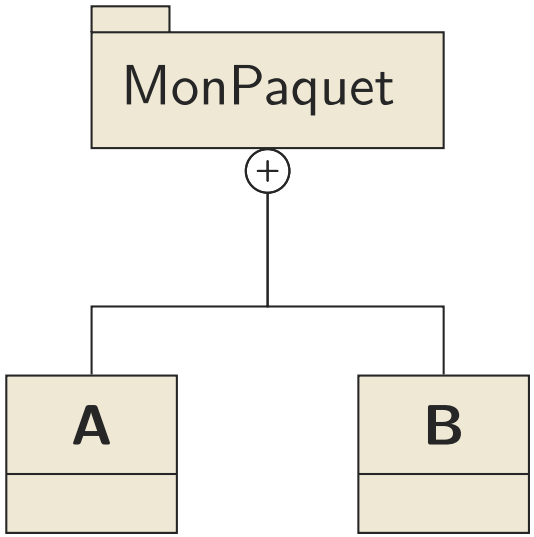
- groupe de classes
- espace de nom

- minimiser le couplage
- maximiser la cohésion

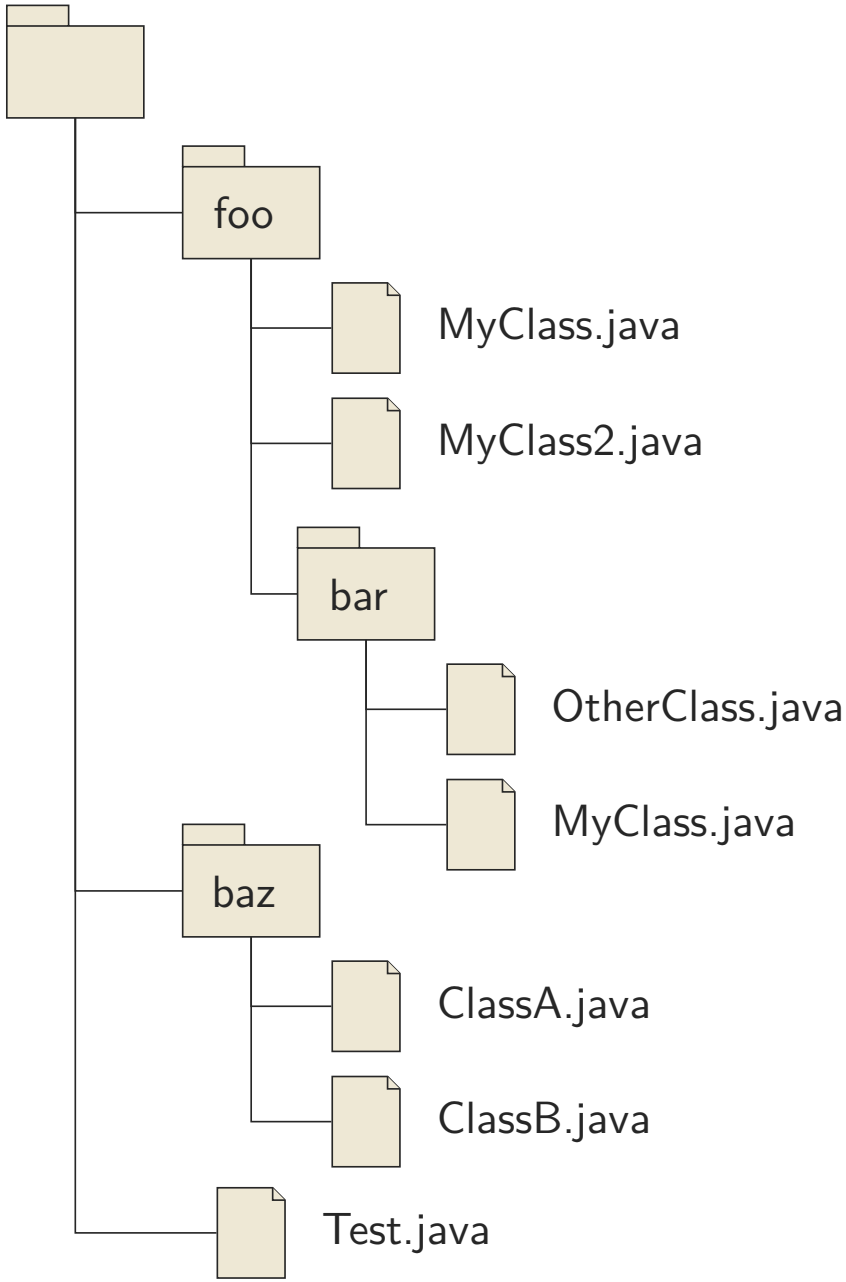
$\Rightarrow$ peu de dépendances entre paquets


MonPaquet
A
B


MonPaquet
MonPaquet::A
MonPaquet::B


MonPaquet
A
B

- mot-clé `package`
- hiérarchie de répertoires



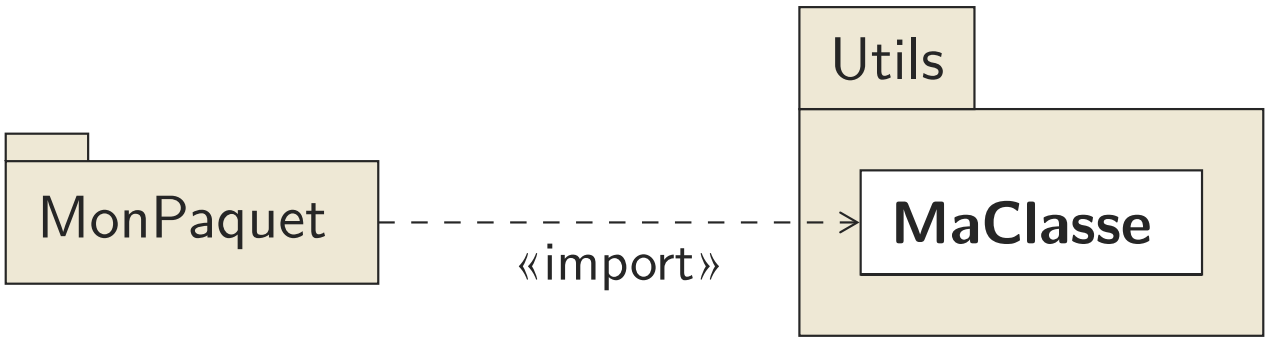

```java
package foo;

public class MyClass { }

class MyClass1 { }
```

```java
package foo;

class MyClass2 { }
```

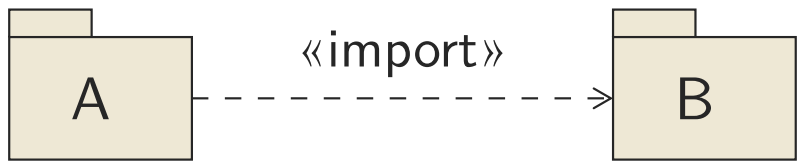
```java
package foo.bar;

public class OtherClass { }
```

```java
package baz;

public class ClassA { }
```

```java
package baz;

public class ClassB { }
```




```
import utils.MaClass ;
```




```
import B.*;
```

```java
package foo;

public class MyClass { }

class MyClass1 { }
```

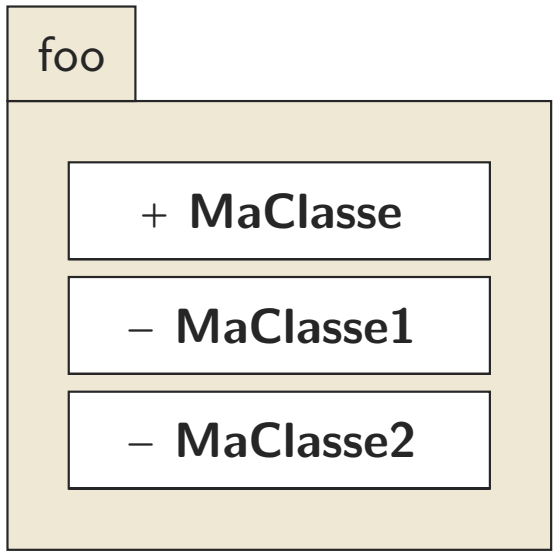
```java
package foo;

class MyClass2 { }
```

```java
package foo.bar;

public class OtherClass { }
```

```java
package baz;

public class ClassA { }
```

```java
package baz;

public class ClassB { }
```

```java
import foo.MyClass;
import foo.bar.OtherClass;
import baz.*;

class Test {
    public static void main(String[] args) {
        MyClass m = new MyClass();
        OtherClass o = new OtherClass();
        foo.bar.MyClass mb = new foo.bar.MyClass();
        ClassA a = new ClassA();
        ClassB b = new ClassB();
    }
}
```



Import statique

```java
package utils;

public class Say {
    public static String hello(String s) {
        return "Hello " + s;
    }
}
```

```java
import static utils.Say.hello;

public class Test {
    public static void main(final String[] args) {
        System.out.println(hello("World !"));
    }
}
```

- ensemble de classes/interfaces
- dans le JDK
- paquets

http://docs.oracle.com/javase/8/docs/api

La bibliothèque java.lang

- outils du langage
- import automatique

Interfaces

- `Comparable<T>` : `int compareTo(T o)`
- `Iterable<T>` : `Iterator<T> iterator()`

- System : purement statique
  - err, out, in
  - exit(int status)
  - static Map<String,String> getenv()
- Object : classe mère
  - protected Object clone()
  - boolean equals(Object obj)
  - protected void finalize()
  - int hashCode()
  - String toString()
  - Class<?> getClass()
- Class<T> : méta-classe
  - String getName()
  - T newInstance()
  - ⇒ java.lang.reflect

Math

- constantes
- méthodes statiques

⇒ opérations numériques

Math

- static double E = e
- static double PI = π
- static double abs(double a) → $|a|$
- static double ceil(double a), static int round(float a), static double floor(double a)
- static double cos(double a) → cos a
- static double sin(double a) → sin a
- static double tan(double a) → tan a
- static double exp(double a) → e^a
- static double log(double a) → ln a
- static double log10(double a) → log a
- static double pow(double a, double b) → a^b
- static double sqrt(double a) → $\sqrt{a}$
- static double random() → $0.0 \leq r < 1.0$
- ...

Types

- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Short

String

- char charAt(int index)
- int compareTo(String anotherString)
- String concat(String anotherString)
- boolean endsWith(String suffix)
- boolean startsWith(String prefix)
- int indexOf(int ch)
- int length()
- String substring(int beginIndex, int endIndex)
- String trim()
- String toLowerCase()
- String toUpperCase()
- String[] split(String regex)
- boolean matches(String regex)
- String replaceAll(String regex, String replacement)
- static String format(String format, Object... args)

StringBuilder

- StringBuilder append(String str)
- StringBuilder insert(int index, String str)
- String substring(int beginIndex, int endIndex)
- String toString()

La bibliothèque java.util

- Date, Calendar, TimeZone
  plutôt java.time depuis 1.8
- Random
- Collections (List, Set,. . .)
- Collections purement statique → outils (trier, min, max)

interface Comparator<T>

- int compare(T o1, T o2)
- paramètre dans les méth. de collections

Programmation Java avancée

Arguments variables

nombre variable d'arguments (*varargs*)

```
public static void printGreeting(String... names) {
  for (String n  : names) {
     System.out.println("Hello " + n + ".");
  }
}
```

names → tableau de String

```
printGreeting("Zaphod");
printGreeting("Arthur", "Trillian");
```

```
String[] names = {"Ford", "Marvin"};
printGreeting(names);
```

similaire à

```
public static void printGreeting(String[] names)
```

```
    printGreeting(new String[]{"Ford", "Marvin"});
```

Exceptions

Définition

- gestion d'erreurs
- imprévue
- $\Rightarrow$ événement
- $\rightarrow$ changement du déroulement du programme

exception → objet

- type
- attributs
- méthodes

⇒ plus riche que code d'erreur

- sous-classe de `Exception`
  - explicite
  - erreur de contexte
  - gérée
  - `FileNotFoundException`
- sous-classe de `RuntimeException`
  - implicite
  - erreur de programmation
  - pas forcément gérée
  - `IllegalArgumentException`, `NullPointerException`, `IndexOutOfBoundsException`

- « levée » au moment de l'erreur
- → « propagation » dans la pile d'appel
- « capturée » pour être gérée
- → fin du programme

```
Exception in thread "main" java.lang.RuntimeException
        at TestExceptions.runtime(Exceptions.java :9)
        at TestExceptions.passeRuntime(Exceptions.java :13)
        at TestExceptions.pafRuntime(Exceptions.java :17)
        at Exceptions.main(Exceptions.java :26)
```

objet → créer des exceptions

```
class MonException extends Exception {}
```

Lancer une exception

mot clé `throw`

```
class TestExceptions {

  public void runtime() {
    throw new RuntimeException();
  }

  public void passeRuntime() {
    runtime();
  }

  public void pafRuntime() {
    passeRuntime();
  }
}
```

si pas runtime :

- déclarée : `throws`
- capturée

```
public void paf(String filename) throws FileNotFoundException {
  throw new FileNotFoundException();
}

public void appel() throws FileNotFoundException {
  paf("foo");
}
```

Capturer une exception

- `try`
- `catch`

```
public void capture() {
  try {
    appel();
  } catch (FileNotFoundException e) {
    System.err.println("le fichier n'existe pas");
  } catch (IOException e) {
    System.err.println("erreur d'ecriture");
  }
}
```

finally toujours exécuté

```
public String sansCaptureFinal(String path) throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    return br.readLine();
  } finally {
    if (br != null) {
      br.close();
    }
  }
}
```

cas fréquent : « fermer » l'objet quoi qu'il arrive

Java 7 → construction spécifique si implémente `AutoCloseable`

```
public String sansCaptureFinal(String path) throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    return br.readLine();
  } finally {
    if (br != null) {
      br.close();
    }
  }
}
```

```
public String sansCaptureResource(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

Exceptions chaînées

- changer une exception
  niveau d'abstraction
- garder les informations d'appel

```
try {
 // ...
} catch (IOException e) {
  throw new MonException("Huston...", e);
}
```

- getCause()
- initCause(Throwable)
- getStackTrace()

Avantages

- séparation claire code normal / gestion d'erreur
- propagation automatique
- hiérarchie de types (gestion par groupes)
- explicite (si pas runtime)
- capture d'informations

Types énumérés

Principe : No Magic Numbers

pas de valeurs « spéciales »

✗

```
public static int getWeekDay(Date d)
```

```
switch (getWeekDay(d)) {
    case 0 :
        System.out.println("lundi");
        break;
    case 1 :
        System.out.println("mardi");
        break;
    /*...*/
    case 6 :
        System.out.println("dimanche");
}
```

difficile

- lire
- maintenir

≈ ⇒ constantes

```
class Days {
   final static int MONDAY = 0;
   final static int TUESDAY = 1;
   /*...*/
   final static int SUNDAY = 6;
}
```

```
      switch (getWeekDay(d)) {
         case Days.MONDAY :
            System.out.println("lundi");
            break;
         case Days.TUESDAY :
            System.out.println("mardi");
            break;
         /*...*/
         case Days.SUNDAY :
            System.out.println("dimanche");
      }
```

✗ pas de type spécifique ⇒ pas de vérification

✗ validation manuelle

```
setWeekDay(42);    ≈
setWeekDay(captain.getAge());    ✓
captain.setAge(Days.SUNDAY)    ✓
Colors.RED == Days.MONDAY → true
Colors.RED * Days.MONDAY + captain.getAge()    ✓
```

✗ Pas affichable : `println(Days.MONDAY)` → 1

⇒ type énuméré : enum

```
enum WeekDay {
   MONDAY,
   TUESDAY,
   /*...*/
   SUNDAY;
}
```

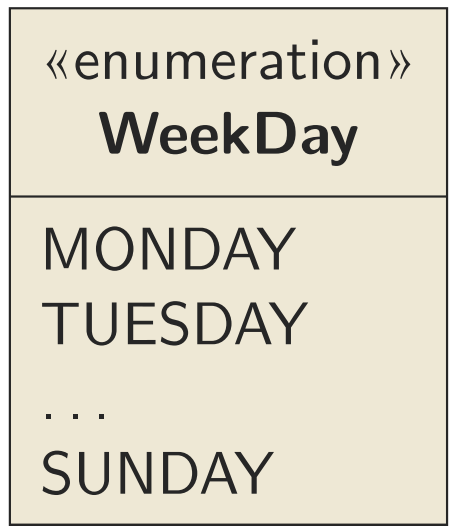
```
public static WeekDay getWeekDay(Date d) {
```

```
    switch (getWeekDay(d)) {
        case MONDAY :
            System.out.println("lundi");
            break;
        case TUESDAY :
            System.out.println("mardi");
            break;
        /*...*/
        case SUNDAY :
            System.out.println("dimanche");
    }
```

- vrai classe
- constructeur privé
- instances finale statiques
- ⇒ comparaison par ==
- ⇒ vérification de types

```
Color.RED == WeekDay.MONDAY → false
captain.setAge(WeekDay.MONDAY) ✗
setWeekDay(5) ✗

WeekDay.MONDAY.toString() → "MONDAY"
```

Effective Java (2001)
Bloch


«enumeration»
WeekDay
MONDAY
TUESDAY
...
SUNDAY

surcharge constructeur

valeur explicites

ajout de méthodes

```
enum WeekDay {
   MONDAY(0),
   TUESDAY(1),
   /*...*/
   SUNDAY(6);

  private final int number;

  private WeekDay(int num) {
    this.number = num;
  }

  public int getNumber() {
    return this.number;
  }
}
```

Classes internes et anonymes

Définition (Classe membre)

classe définie *dans* une autre classe

⇒ outils de la classe externe

- statique
- non statique
- anonyme
- locale

Classe statique

- `static` (`final`)
- classe ordinaire
- « attribut statique »
- visibilité (`public`, `private`)
- accès aux éléments *statiques* de la classe externe (même privés)

- énumérations (`Locale.Category`)
- données internes (`Map.Entry`)
- builder, visiteurs, itérateurs

```
class MaClasse {

  static class MaClasseInterne {
    private void priv() {
      System.out.println("MaClasseInterne.priv");
      plop();
      //foo();
    }
  }

  private static void plop() {
    System.out.println("MaClasse.plop");
  }

  private void foo() {
    System.out.println("MaClasse.foo");
  }

  public void bar() {
    MaClasseInterne ci = new MaClasseInterne();
    ci.priv();
  }
}
```

→ MaClasse$MaClasseInterne.class

```
MaClasse c = new MaClasse();
c.bar();

MaClasse.MaClasseInterne ci = new MaClasse.MaClasseInterne();
```

Classe non statique

- instance interne liée instance externe
- accès aux éléments de la classe externe

- itérateurs
- adaptateurs (`Map.keySet`)
- composition

```
class MaClasse {
  private static void foo() {
    System.out.println("MaClasse.foo");
  }

  private void priv() {
    System.out.println("MaClasse.priv");
  }

  class MaClasseInterne {
    private void priv() {
      System.out.println("MaClasseInterne.priv");
      foo();
      MaClasse.this.priv();
    }
  }

  public void bar() {
    MaClasseInterne ci = new MaClasseInterne();
    ci.priv();
  }
}
```

```
MaClasse c = new MaClasse();
c.bar();
//MaClasse.MaClasseInterne ci = new MaClasse.MaClasseInterne();
```

Classe anonyme

- classe interne
- pas un attribut de la classe externe
- pas de nom
- une seule instance

✗ `instanceof`

✗ une seule interface

⇒ objets « fonctions » (stratégies)

accès aux variables locale ⇒ closure

Collections.sort(List<T> list, Comparator<? super T> c)
int compare(T o1, T o2)

```
List<Integer> l = asList(1, 2, 3, 4);

Collections.sort(l, new Comparator<Integer>(){
  @Override
  public int compare(Integer a, Integer b) {
    return -1 * Integer.compare(a, b);
  }
});
System.out.println(l);
```

→ MaClasse$1.class

Classe locale

- classe interne
- déclarée dans une méthode
- locale à la méthode

```
private static void foo() {

  class Hello {
    public String hello() { return "Hello";}
  }

  Hello h = new Hello();
  System.out.println(h.hello());
}
```

Threads

Définition (Thread)

- processus léger (1 seule VM)
- tâches effectuées en parallèle
- mémoire partagée
- piles d'appel indépendantes

programmes concurents / parallèles

- multi-cœur (1 par cœur)
- application serveur (écoute + 1 par client)
- interface graphique (principal, IHM, traitements)
- gestion du temps / tâches de fond (jeux)
- systèmes asynchrones
- ...

classe `java.lang.Thread`

méthode `void start()` → `void run()`

méthode `void join()` : attend la fin

classe interne

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("debut thread");
        try {
            sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("thread interrompu");
        }
        System.out.println("fin thread");
    }
}
```

```
class TestThread {
    public static void main(final String[] args) {
        System.out.println("debut main");
        MyThread t = new MyThread();
        t.start();
        System.out.println("thread lance");
        try {
            t.join();
        } catch (InterruptedException e) { }
        finally {
            System.out.println("fin main");
        }
    }
}
```

```
debut main
thread lance
debut thread
fin thread
fin main
```

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("runnable");
    }
}
```

```
class TestRunnable {
    public static void main(final String[] args) {
        System.out.println("debut main");
        Thread t = new Thread(new MyRunnable());
        t.start();
        System.out.println("thread lance");
        try {
            t.join();
        } catch (InterruptedException e) { }
        finally {
            System.out.println("fin main");
        }
    }
}
```

⇒ pool de thread (p. ex.)

```java
import java.util.concurrent.*;
class TestPool {
   public static void main(final String[] args) {
      System.out.println("=====");
      System.out.println("debut main");
      ExecutorService pool = Executors.newFixedThreadPool(8);
      Runnable r1 = new Runnable() {
         public void run() {
            System.out.println("in r1");
         }};
      Runnable r2 = new Runnable() {
         public void run() {
            System.out.println("in r2");
         }};
      Runnable r3 = new Runnable() {
         public void run() {
            System.out.println("in r3");
         }};
      pool.execute(r1);
      pool.execute(r2);
      pool.execute(r3);
      pool.execute(r1);
      System.out.println("fin main");
      pool.shutdown();
   }
}
```

```
=====
debut main
in r1
fin main
in r2
in r1
in r3
=====
debut main
in r1
in r2
fin main
in r1
in r3
=====
debut main
in r2
in r1
in r3
fin main
in r1
=====
debut main
in r2
in r1
fin main
in r1
in r3
```

Interruption

- `void interrupt()`
- `boolean isInterrupted()`
- `static boolean interrupted()`

accès concurrents

$\Rightarrow$ synchronisation : `synchronized`

Méthodes

```
public synchronized void setAttr(int val)
```

```
public synchronized int getAttr()
```

appels concurrent impossibles

Objet

```
synchronized (instance) {
  /* modif. de instance */
}
```

interbloquage (*dead lock*)

Exemple

- thread t_1 verrouille o_1 et attend o_2
- thread t_2 verrouille o_2 et attend o_1

volatile

- variable : atomique
- type primitifs
- non bloquant

Attente

- `Object.wait()`
- `Object.notifyAll()`

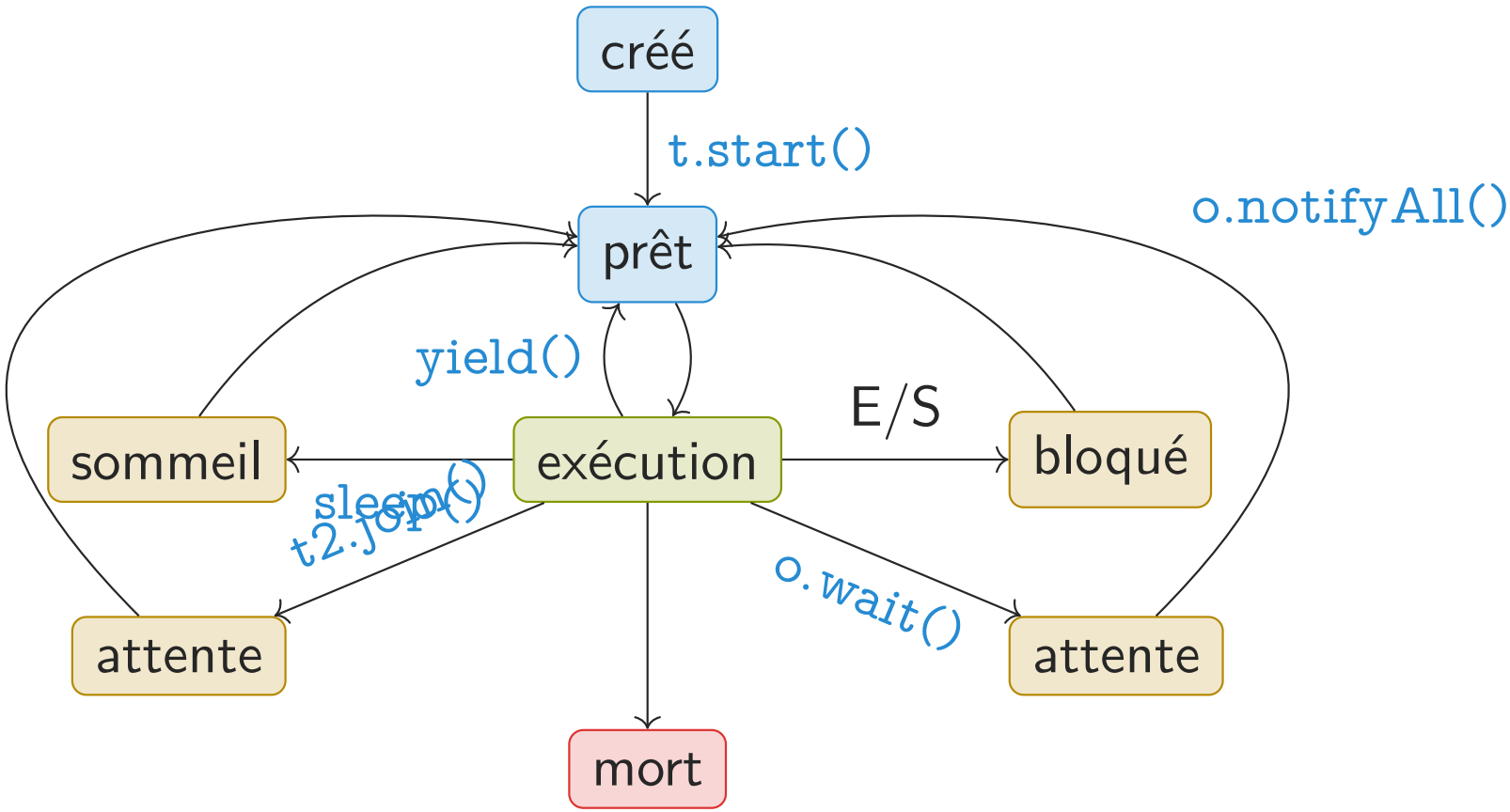
Exemple

- producteur → `notifyAll()`
- consommateur → `wait()`

types et structures

- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks

États


créé
t.start()
prêt
o.notifyAll()
yield()
E/S
sommeil
exécution
bloqué
sleep()
t2.join()
o.wait()
attente
attente
mort