Concepts et langages orientés-objet

Jean-Luc Falcone

22 Février 2022

L’approche Orientée-Objet

Définition: Programmation Orientée-Objet

- Sur les 20 langages en tête du classement TIOBE, seuls 3 ne se décrivent pas comme *orientés-objet* (C, SQL, Assembleur)
- Les 17 langages restants sont très variés

Problème 1: Définition tautologique

- La programmation orientée-objet est une approche qui consiste à décomposer un problème en objets
- Les objets sont les éléments de base de la programmation orientée-objet

Problème 2: Définition puriste

- Prendre position et affirmer ce qu'est la vraie programmation orientée-objet
- Evacuer les contre-exemples en utilisant le sophisme du *vrai écossais*

Approche pragmatique

- Partir d'un exemple concret: Java
- Dégager des principes et des concepts
- Comparer aux autres langages

Bonnes pratiques et usage recommandé

- Qu’est-ce que c’est ?
- A quoi ça sert ?
- Quand l’utiliser ?
- Pour quelle raison ?

Organisation du cours

Evaluation

- Note finale: moyenne note TP et note examen oral
- L'examen oral porte sur tous le programme, cours et TPs
- Tous les TPs ne seront pas évalués et n'auront pas le même poids

Travaux pratiques

- Installer Java 17 (consignes et aide durant le premier TP)
- Exercices et petits projets
- Les TPs évalués et les modalités seront annoncées à l'avance

Java Procédural

Jean-Luc Falcone

22 Février 2022

Le langage Java

Création

- Créé par James Gosling (alias Dr. Java), Mike Sheridan, et Patrick Naughton
- Ingénieurs chez Sun Microsystems (achetée par Oracle Corporation en 2009)
- Version 1.0 en 1996

But: remplacer C++

- Langage orienté-objet et procédural
- Syntaxe familière, dérivée du C
- Robuste, performant et *multithreaded*
- Portable (*compile once, run anywhere*)

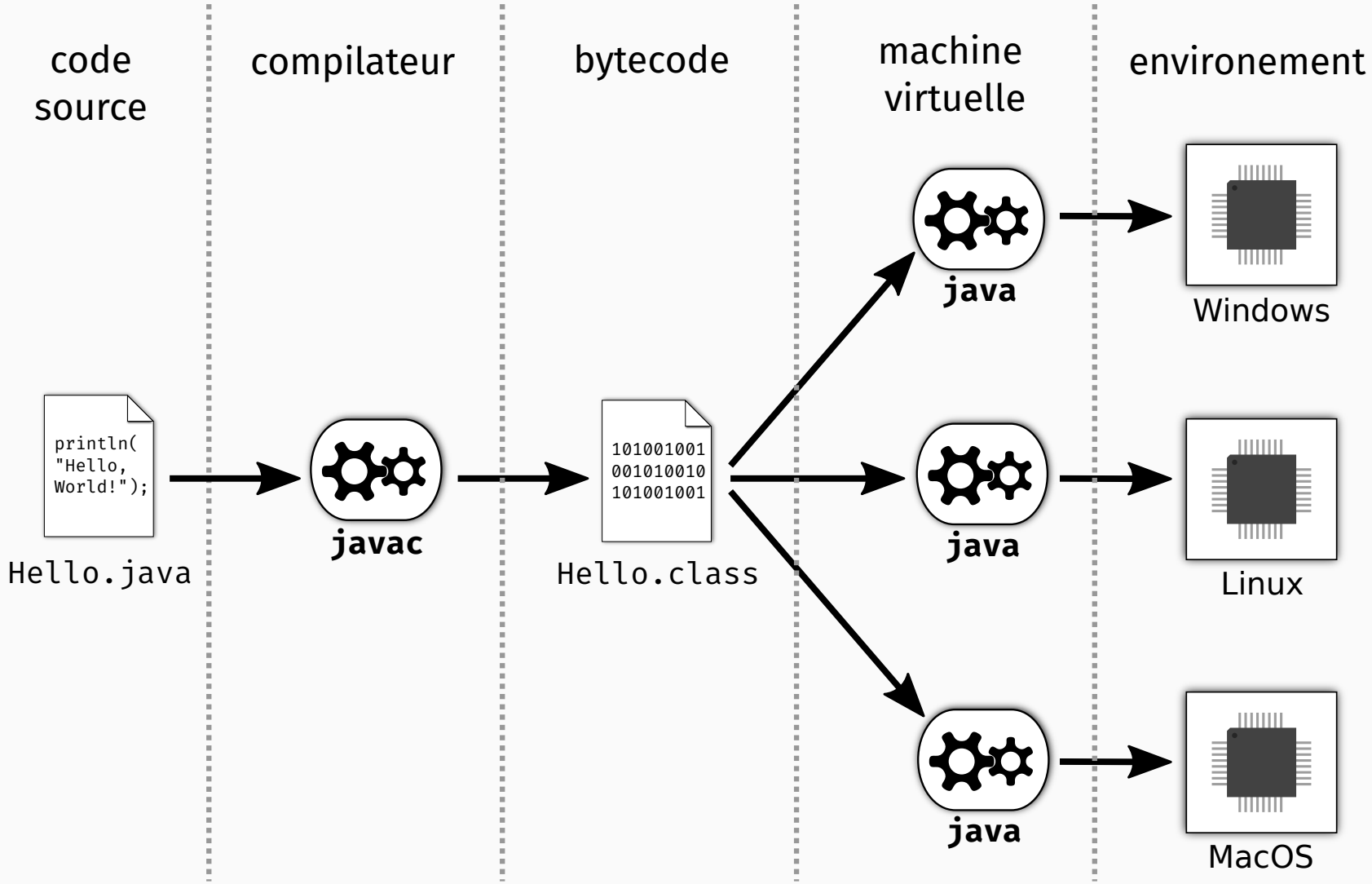
Java vs. Javascript

- Syntaxe dérivée du C
- Fondamentalement différents !
- Confusion volontaire: accord marketing
- Profite de l'essor du Web (premiers navigateurs grand public en 1994)

Machine virtuelle (JVM)

- Code source compilé en *bytecode*, similaire à du langage machine
- Une machine virtuelle exécute le *bytecode* sur un environement hôte
- Très bon compromis portabilité vs. performance

Machine virtuelle: schéma



Plateformes et environnements

- On peut directement produire du code natif (*AOT*, *GraalVM*, etc.)
- Il existe d'autre types de VM (*Dalvik*, *ART*, etc.)
- Il existe d'autres lanages pour la JVM: *clojure*, *scala*, *kotlin*, etc.
- Il existe des compilateurs *bytecode* pour beaucoup de langages pré-existants: *python*, *ruby*, *pascal*, etc.

Popularité

Langage très répandu, très utilisés en entreprise.

Avantages

- Beaucoup de librairies
- Main d'oeuvre disponible
- *Lingua franca*

Désavantages

- Beaucoup de ressources de mauvaise qualité
- Forte concurrence sur le marché du travail
- Applications massives qui commencent à dater

Syntaxe de base

Commentaires

Similaire à C99:

```
//Commentaire sur une ligne

/* Commentaire
   sur
   plusieurs lignes */
```

Types primitifs

Type	Description
`byte`	entier signé 8 bits
`short`	entier signé 16 bits
`int`	entier signé 32 bits ∗
`long`	entier signé 64 bits
`float`	virgule flottante 32 bits
`double`	virgule flottante 64 bits ∗
`boolean`	booléen: `true` ou `false`
`char`	charactère unicode 16 bits

∗ à utiliser par défaut

Types primitifs: remarques

- Utiliser uniquement `short` et `float` pour économiser de la mémoire dans des très grands tableaux. Leurs opérations sont plus lentes.
- Pareil pour `byte` mais usage justifié pour les IO binaires.
- Les types entiers sont signés, mais `int` et `long` peuvent être interprétés comme non signés à partir de Java 8 (cf. classes `Integer` et `Long`)

Expressions littérales

```
12345;     //int
12345L;    //long
0xEE;      //int en notation hexadécimal
0b11001;   //int en notation binaire
1.23e45;   //double
1.23e45f;  //float

/* underscore pour lisibilité (JSE7) */
12_345;          //int
123_456_789L;    //long
0xBD_55_1C_E9; //int en notation hexadécimal
```

Variables

Comme en C:

```
int i = 12_345;
boolean done = true;
short s = 12;
float x = 0.01f;
```

Constantes

Mot-clé `final`:

```
final int x = 12;
final double quasiPi = 3.14159;

x = 12; // error: cannot assign a value
        // to final variable x
```

Remarques

- Le mot clé `final` à d'autres significations
- Tout ce qui n'a pas de raison de changer devrait être déclaré `final`

Inférence du type

Le mot-clé `var` permet d'inférer le type:

```
var i = 12;          //int i = 12;
var x = 0.12;        //double x = 0.12;
var b = false;       //boolean b = false;
final var j = -1;    //final int j = -1;

short s = i; //error: incompatible types...
```

Attention
Privilégier la lisibilité (supprimer le bruit, documenter les types, éviter les surprises).

Arithmétique

- Entiers et virgule flottante: presque comme en C
- Logique booléenne distincte des entiers:

```
true && false == ! true; //AND logique
47 & 19 == ~(-4);        //AND bit-à-bit
```

Structures de contrôle

```
if( condition ) { /*...*/ } else { /*...*/ }

for( int i=0; i<N; i++ ) { /*...*/ }

while( condition ) { /*...*/ }

do { /*...*/ } while( condition );

switch( x ) {
  case y : /*...*/; break;
  /*...*/
}
```

Structures de contrôle: conditions

Vaies expressions logiques dans `if`, `while`, et `do ... while`

```
int i = 0;
if( i + 1 ) {  //error: incompatible types:
               //int cannot be converted
               //to boolean
}
```

Fonctions

Proche de C:

```
static long permutations( int n, int k ) {
        long res = 1;
        for( int i = n; i > (n-k); i-- ) {
                res *= i;
        }
        return res;
}
```

Mot réservé `static`

- Permet la programmation non orientée-objet (procédurale comme en C ou en Pascal).
- Différent du mot-clé `static` en C
- Permet aussi de définir des variables (ou des constantes) globales en dehors des fonctions
- Les fonctions `static` sont parfois nommées *méthodes de classe*.

Sémantique par valeur

Lorsque l'on appelle une fonction, les primitives passées en argument sont copiées.

Une modfication de ces arguments n'a pas d'effet hors de la fonction.

Sémantique par valeur: Exemple 1

```
static void increment( int i ) {
  i += 1; //l'argument i est modifié
}

int x = 1;
increment(x); //x n'est pas modifier
```

Sémantique par valeur: Exemple 2

```
static void swap( int a, int b ) {
  int tmp = a;
  a = b;
  b = tmp;  //les valeurs des args a et b
            //sont modifiées
}

int x = 10;
int y = 100;
swap(x,y);  //les valeurs de x et y ne changent pas
```

Tableaux

Déclaration du type

On déclare le type d'un tableau en ajoutant `[]` après le type des éléments:

```
int[] is      //Tableau d'entier
double[] xs   //Tableau de doubles
boolean[] bs //Tableau de booleans
int[][] js    //Tableau de tableaux d'entiers
// etc.
```

Déclaration du tableau

On déclare le tableau grâce au mot réservé `new` et en passant la taille entre crochet:

```
int[]  is = new int[3]; //3 entiers
double[] xs = new double[N] ;// 'N' doubles
var bs = new boolean[15]; // 15 booléens
```

Remarques

- Par défaut les nombres sont initialisés à `0` ou `0.0`, les booléens à `false`, et les caractères à `\0`.
- La taille maximale et la taille max d'un entier ($2^{31} - 1 \approx 2$ millions d'éléments)

Déclaration du tableau (2)

On peut aussi initialiser en le déclarant avec des accolades:

```
//Tableau de 4 entier
int[]  is = {10, 20, 30, 40};

//Tableau de 3 booléens
boolean[] bs = {false, true, false};

//Tableau de 0 doubles (vide)
double[] xs = {};
```

Tableaux: Remarques

- Les tableaux sont toujours mutables
- La taille d'un tableau est fixe
- Les tableaux sont initialisés dans le tas (*heap*)
- Les valeurs des tableaux des primitives sont contigües en mémoire
- Il n'y a pas de vrais tableaux en 2D (ou plus)

Opérations

Accès par indices commençant à 0 (comme en C).

La taille est accessible par la propriété `length`

```
static double max( double[] x ) {
  double m = x[0];
  for( int i=1; i < x.length; i++ ) {
    if( x[i] > m )  {
              m = x[i];
    }
  }
  return m;
}
```

Références

Dans la déclaration:

```
int[] i = new int[10];
```

- `i` est une référence: similaire à un pointeur, mais sans arithmétique et sans déréférencement explicite
- `new` est l'équivalent de `malloc`
- les données sont automatiquement désallouées: *Garbage-collector* (ramasse-miettes).

Sémantique par référence

Lorsque l'on passe une référence à une fonction, celle-ci est copiée dans les arguments, mais pas les données.

- Modification de l'argument sans effet hors de la fonction
- Modification des données pointées, visible hors de la fonction

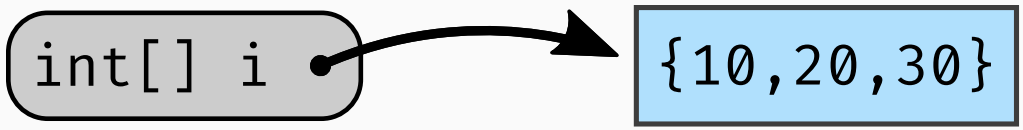
```
static void abs( double[] x ) {
  for( int i=0; i < x.length; i++ ) {
    if( x[i] < 0 )  {
      //Modifie les données pointées par x
      x[i] = -x[i];
    }
  }
}
```

Sémantique par référence: exemples

```
//Mauvais exemple
static void reset( double[] x ) {
  //Modifie la référence
  x = new double[x.length];
}

//Bon exemple
static void reset( double[] x ) {
  for( int i=0; i<x.length; i++ ) {
    x[i] = 0.0;  //Modifie les données
  }
}
```

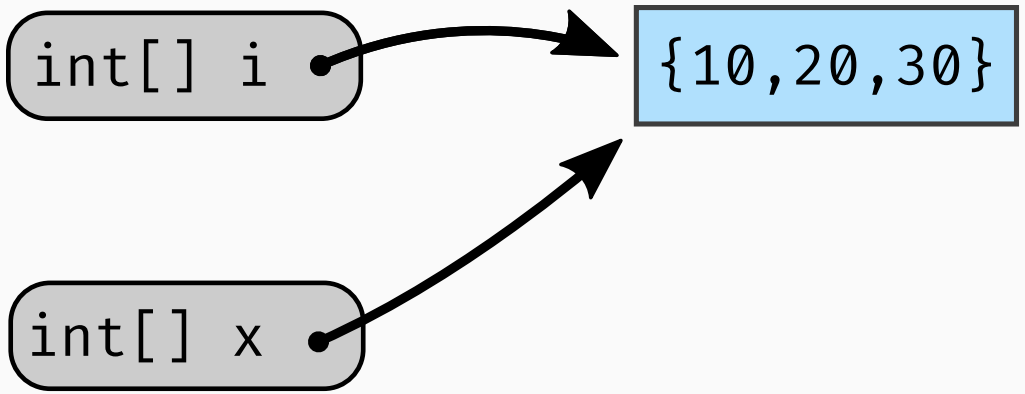
Sémantique par référence: pas à pas (1)

```
static void foo( int[] x ) {
  x[1] = 99;
  x = new int[3];
}

int[] i = {10, 20, 30}; //<-- ici
foo(i);
```



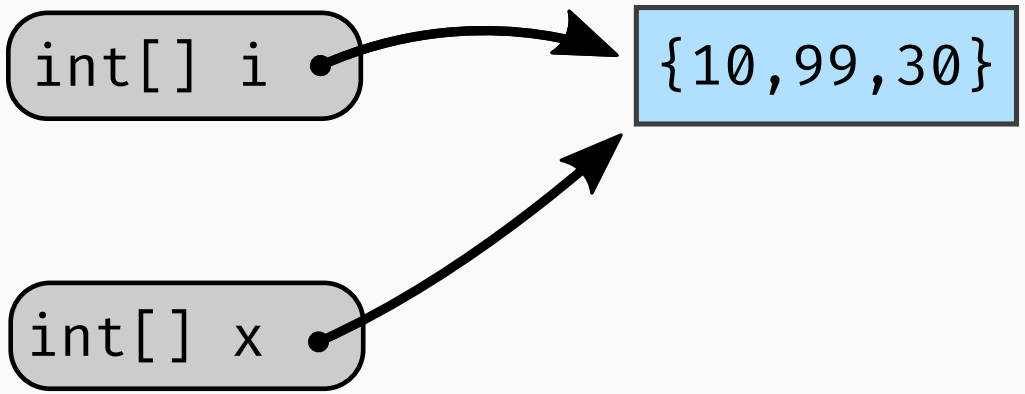
Sémantique par référence: pas à pas (2)

```
static void foo( int[] x ) { //<-- ici
  x[1] = 99;
  x = new int[3];
}

int[] i = {10, 20, 30};
foo(i);
```



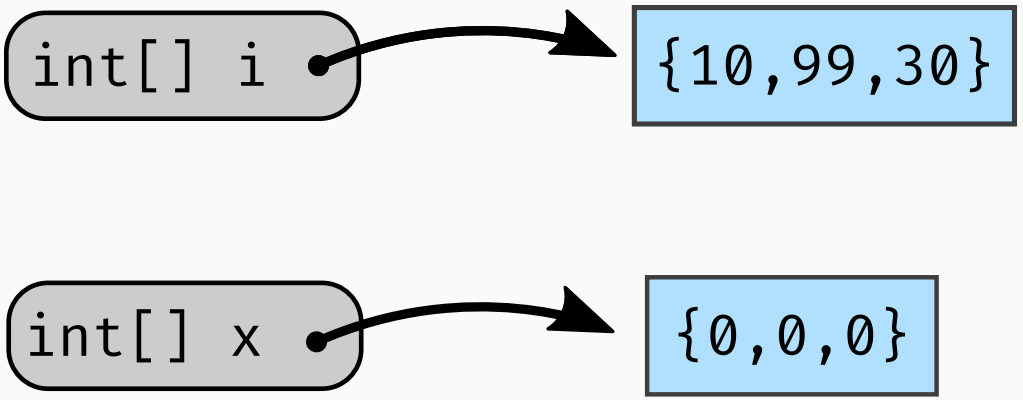
Sémantique par référence: pas à pas (3)

```
static void foo( int[] x ) {
  x[1] = 99;  //<-- ici
  x = new int[3];
}

int[] i = {10, 20, 30};
foo(i);
```



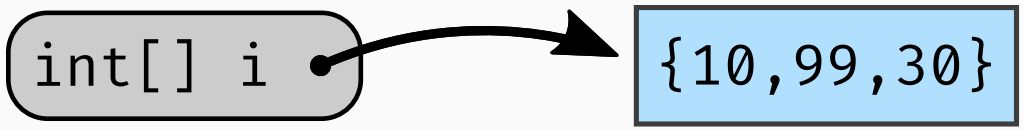
Sémantique par référence: pas à pas (4)

```
static void foo( int[] x ) {
  x[1] = 99;
  x = new int[3]; //<-- ici
}

int[] i = {10, 20, 30};
foo(i);
```



Sémantique par référence: pas à pas (5)

```
static void foo( int[] x ) {
  x[1] = 99;
  x = new int[3];
}

int[] i = {10, 20, 30};
foo(i);  //<-- ici
```



Bonjour, Monde!

Chaînes de caractères

Les chaînes de caractères sont représentées par le type `String`:

- Toujours immutables
- Acceptent tout l'Unicode
- Littéral défini entre guillemets
- Opérateur '+' pour concaténer (conversion automatique).

```
String s = "Hello" + ", " + "World!";
double pi = 3.14;
String z = "Résultat=" + pi;
```

Programme minimal

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

Programme minimal: Anatomie

```java
//Tout code doit être déclaré dans une classe
class HelloWorld {
  //Visible depuis l'extérieur
  public
  //Méthode non OO
  static
  //fonction principale (point d'entrée)
  void main(
   //arguments passés au programme
   String[] args
   ) {
    //Affiche sortie standard
    System.out.println("Hello, World!");
  }
}
```

Classes et Instances

Jean-Luc Falcone

1 mars 2022

Classes, Instances et Objets

Classe en Java: Définition

Dans le langage `Java` une classe est un type composé, regroupant des données et des fonctions associées.

Classe en Java: Définition

Dans le langage `Java` une classe est un type composé, regroupant des données et des fonctions associées.

Quelques termes de jargon

- Classe: type composé
- Instance: valeur d'une classe
- Objet: instance (ou classe)
- Membre de la classe: variables ou fonction déclarées dans la classe
- Champs: variable membre
- Méthode: fonction membre

Exemple médiocre: définition

Spécifications: points en 2D, distance en 2 points.

```
class Point {//Classe
  double x;  //Champ
  double y;  //Champ

  double distance( Point that ) {  //Méthode
    return Math.sqrt(
          Math.pow(x - that.x, 2),
          Math.pow(y - that.y, 2)
        );
  }
}
```

Exemple médiocre: utilisation

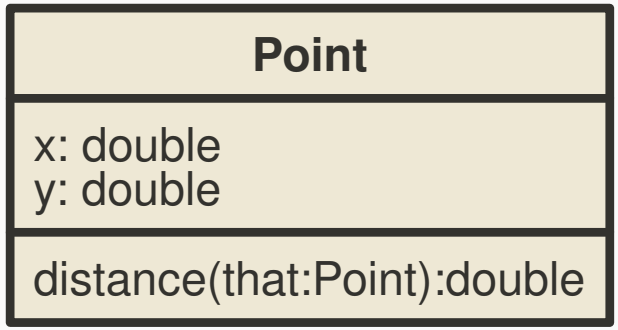
```
Point p = new Point();     //Instanciation
Point q = new Point();     //Instanciation

p.x = 1.0;  p.y = 0.5;     //Assignation
q.x = 2.0;  q.y = 0;       //Assignation

double d = p.distance(q); //Appel de méthode
```

Exemple médiocre: Remarques

Dans l'exemple ci-dessus:

- l'opérateur `.` (point) permet d'accéder aux champs et méthodes d'une instance.
- `new` alloue la mémoire (comme pour un tableau)
- `p` et `q` sont des références pointant sur les instances
- la mémoire sera désallouée si plus aucune référence ne pointe sur les instances.
- A la création les champs prennent leur valeur par défaut (`0.0`)

Exemple médiocre: représentation UML



Constructeur

On peut définir une fonction particulière qui se charge de construire l'instance: le constructeur

- porte le même nom que la classe
- pas de type de retour (pas de `return` non plus)
- appelé après l'allocation de l'instance

Constructeur: Déclaration

```
class Point {
  double x;
  double y;

  Point( double xx, double yy ) { //Constructeur
    x = xx; y = yy;
  }

  double distance( Point that ) {
  //...
```

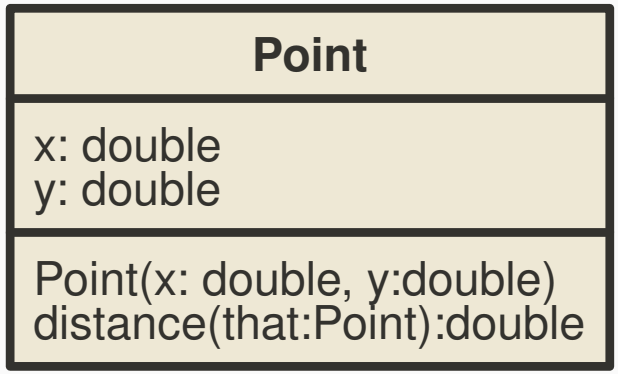
Constructeur: Utilisation

```
Point p = new Point( 1.0, 0.5 ); //Construction
Point q = new Point( 2.0, 0 );   //Construction

double d = p.distance(q); //Appel de méthode
```

Constructeur: Remarques

- On peut surcharger les constructeurs: plusieurs constructeurs si arguments différents.
- S'il n'y pas de constructeur, un constructeur par défaut est généré, vide et sans arguments.
- Restriction dans l'appel de méthodes depuis le constructeur.
- Pour la logique complexe, préférer une fonction `static`

Constructeur: Représentation UML



Référence `this`

Chaque instance a toujours accès à une référence vers elle-même: `this`

Les utilisations sont:

- Utiliser des membres qui serait localement cachés par des variables locales ou des arguments.
- Accroître la lisibilité
- Retourner une instance de soi-même

Référence this: exemple (1)

```
class Point {
  //..
  Point( double x, double y ) {
    this.x = x; this.y = y;
  }
  double distance( Point that ) {
    return Math.sqrt(
          Math.pow( this.x - that.x, 2 ),
          Math.pow( this.y - that.y, 2 )
        );
  }
}
```

Référence `this`: exemple (2)

Nouvelle spec: obtenir le point le plus haut (axe `y`).

```
class Point {
  //...

  Point topMost( Point that ) {
     if( this.y < that.y )
           return that;
     else
           return this;
  }
}
```

Immutabilité

Considérer toujours l'immutabilité par défaut:

- plus simple à analyser
- moins de bugs
- *thread safe*

Plus facile de changer d'avis.

Exemple un peu moins médiocre

Aucune spec ci-dessus, ne nous impose la mutabilité

```
class Point {
  final double x;  //Immutable
  final double y;  //Immutable
  Point( double x, double y ) {
    this.x = x; this.y = y;
  }
  //...
}
```

Aggrégation

Les membres d'une classe peuvent aussi être des références vers des instances.

```
class Segment {
  final Point from;
  final Point to;
  Segment( Point p, Point q ) {
    from = p;
        to = q;
  }
  double length() {
    return from.distance(to);
  }
}
```

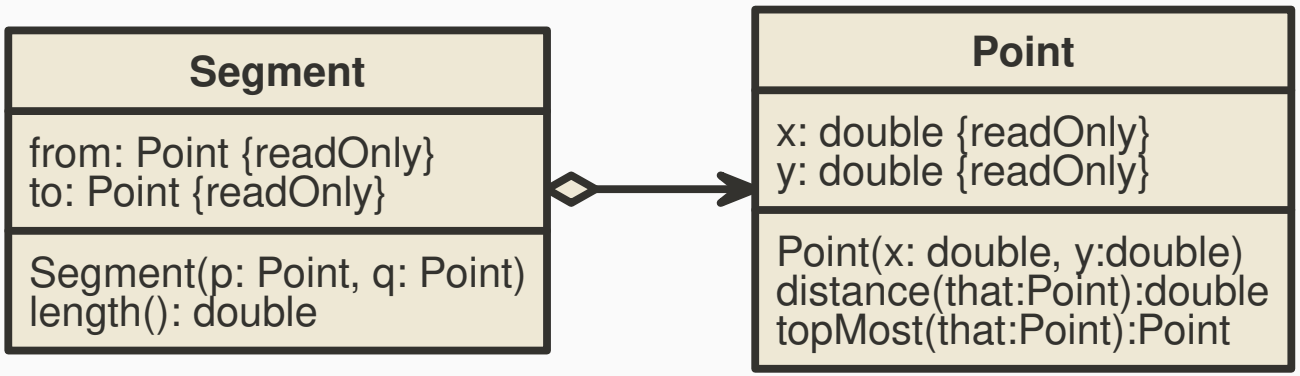
Aggrégation: remarques (1)

- La valeur par défaut d'une référence vers une instance est `null`
- Tout appel de méthode sur une valeur `null` se traduit par une erreur d'exécution.
- Eviter d'utiliser la valeur `null` autant que possible
- Toujours initialiser les référence avec des valeurs acceptables.

Aggrégation: remarques (2)

```
class Segment {
  Point from; //Pas de valeur par défaut
  Point to;   //Pas de valeur par défaut
  double length() {
    return from.distance(to);
  }
}
//CODE CLIENT
Segnment seg = new Segment();
seg.from = new Point(2.1, 0.8);

sig.length(); //Erreur d'exécution car
              //'to' sera 'null'
```

Aggrégation et immutabilité Représentation UML



Sémantique par référence (1)

La sémantique de passage par référence s'applique toujours aux instances (cf. tableaux).

```
class Foo {
  int i;
  void click() { i += 1 };
}
```

Sémantique par référence (2)

```
void bar( Foo foo ) {
  foo.click();
  foo = new Foo();
  foo.i = 12;
  foo.click();
}

Foo foo = new Foo();
foo.i = 3;
bar(foo);
System.out.println(foo.i);
//Quelle valeur sera affichée ?
// 3, 4, 12 ou 13?
```

Surcharge de fonction (*overload*, 1)

On peut définir plusieurs méthodes ou constructeurs avec le même nom dans la même classe si les arguments sont différents.

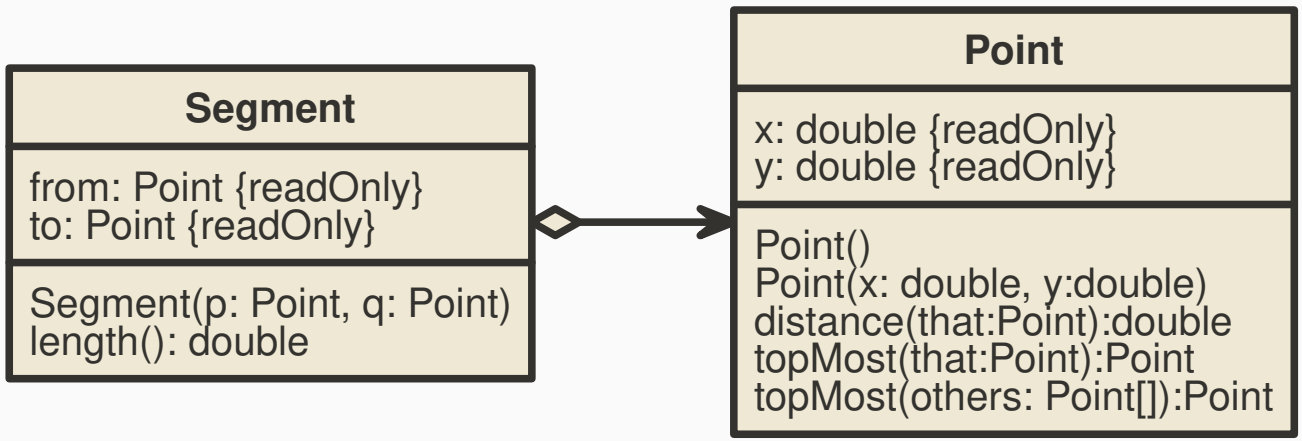
```
class Point {
  final double x;
  final double y;
  Point( double x, double y ) {
    this.x = x; this.y = y;
  }
  Point() { //Surcharge de constructeur
   this.x = 0.0; this.y = 0.0;
  }
}
```

Surcharge de fonction (2)

On peut définir plusieurs méthodes ou constructeurs avec le même nom dans la même classe si les arguments sont différents.

```
class Point {
    Point topMost( Point that ) { /*...*/ }
        //Surcharge de méthode
        Point topMost( Point[] others ) {
          Point top = this;
          for( int i=0; i< others.length; i++ ) {
            top = top.topMost( others[i] );
          }
      return top;
        }
  }
}
```

Surcharge de fonction: représentation UML



Encapsulation

Modificateurs d'accès

On peut contrôler l'accès des membres d'un objet au moyen de modificateurs:

- `public` accessible/utilisable partout
- `private` accessible/utilisable uniquement par les instances de la même classe

Remarques

- Les restrictions s'appliquent en lecture/écriture
- Il existe deux autres niveaux d'accès: celui par défaut et `protected`
- Il ne s'agit pas d'un mécanisme de sécurité

Modificateurs d'accès: Exemple 1

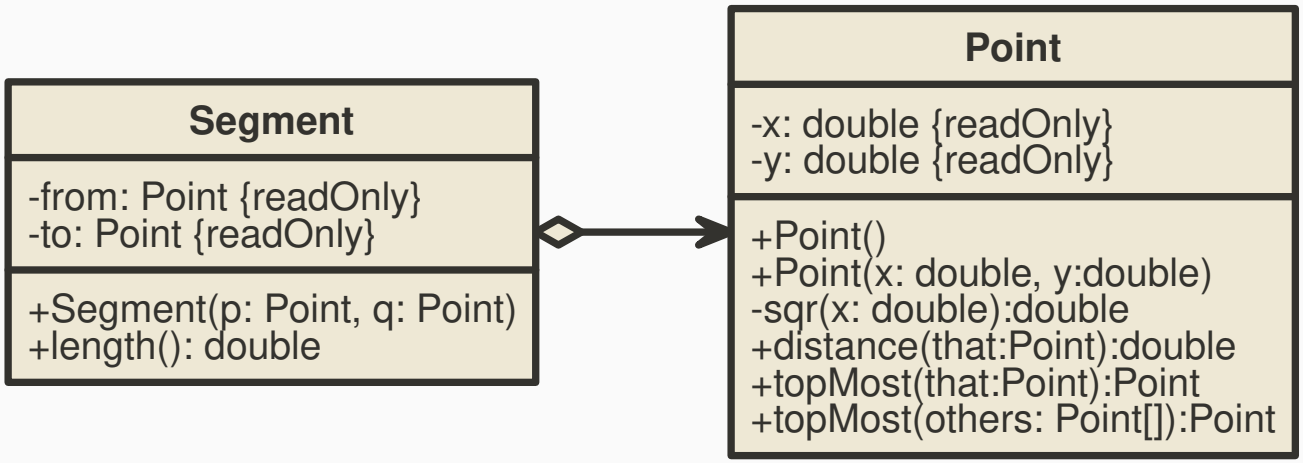
Rien dans nos specs ne dit que les coordonnées doivent être accessibles

```
class Point {
  private final double x;
  private final double y;
  public Point( double x, double y ) {
    this.x = x; this.y = y;
  }
  //...
}
```

Modificateurs d'accès: Exemple 2

```
class Point {
  //...
  private double sqr( double x ) {
    return x*x;
  }
  public double distance( Point that ) {
    return Math.sqrt(
          sqr( this.x - that.x ), //peut voir that.
          sqr( this.y - that.y )  //car même classe
        );
  }
}
```

Modificateurs d'accès: Représentation UML



Recommandations

Pour décider du niveau d'accès:

1. Tous les champs doivent être `private`
2. Aussi peu de méthodes `public` que possible

Plus facile de changer `private` en `public` que l'inverse !

Exemple catastrophique: Compte bancaire (1)

Spécifications initiales: compte bancaire, identifiant unique, retraits et versements.

```
class Account{
  public final String ID;
  public long amount;
}

/* Code client */
Account acc = getAccount(2234); //Retourne compte
//Retirons 100 CHF au compte
acc.amount -= 100;
//Position du compte
long remaining = acc.amount;
```

Exemple catastrophique: Compte bancaire (2)

Nouvelles specifications: Afficher débits et crédits totaux

```
class Account{
  public final String ID;
  public long debits;
  public long credits;
}

/* Code client */
Account acc = getAccount(2234);
//Retirons 100 CHF au compte
acc.debits += 100;
//Position du compte
long remaining = acc.credits-acc.debits;
```

Exemple catastrophique: Compte bancaire (3a)

Nouvelles specifications: Pas de découvert (crédit >= débit)

```
class Account{
  public final String ID;
  public long debits;
  public long credits;
  public long isValid() {
     return credits >= debits;
  }
}
```

Exemple catastrophique: Compte bancaire (3b)

Nouvelles specifications: Pas de découvert (crédit >= débit)

```
Account acc = getAccount(2234);

//Retirons 100 CHF au compte
long before = acc.debits;
acc.debits += 100;
if( !acc.isValid() ) {
  acc.debits = before;
}

//Position du compte
long remaining = acc.credits-acc.debits;
```

Meilleur exemple: Compte bancaire (1a)

Spécifications initiales: compte bancaire, identifiant unique, retraits et versements.

```
class Account {
  private final String ID;
  private long amount;
  Account( String ID, long amount ) {
    this.ID = ID;
        this.amount = amount;
  }
  public String getID() { return ID; }
  public long position() { return amount; }
  public void withdraw( long amount ) {
    this.amount -= amount:
  }
}
```

Meilleur exemple: Compte bancaire (1b)

```
/* Code client */
Account acc = getAccount(2234);

//Retirons 100 CHF au compte
acc.widthraw( 100 );

//Position du compte
long remaining = acc.position();
```

Meilleur exemple: Compte bancaire (2a)

Nouvelles specifications: Afficher débits et crédits totaux

```
class Account {
  private final String ID;
  private long debit;
  private long credit;

  Account( String ID, long amount ) {
    this.ID = ID;
        if( amount > 0 ) this.credit = amount;
        else this.debit = -amount;
  }
  //...
}
```

Meilleur exemple: Compte bancaire (2c)

```
class Account {
  //...
  public String getID() { return ID; }
  public long getCredit() { return credit; }
  public long getDebit() { return debit; }

  public long position() { return credit-debit; }

  public void withdraw( long amount ) {
    this.debit += debit:
  }
}
```

Meilleur exemple: Compte bancaire (2c)

Aucun changement côté client!

```
/* Code client */
Account acc = getAccount(2234);

//Retirons 100 CHF au compte
acc.widthraw( 100 );

//Position du compte
long remaining = acc.position();
```

Meilleur exemple: Compte bancaire (3a)

Nouvelles specifications: Pas de découvert (crédit >= débit)

```
class Account {
  //...
  //Change type de retour
  public boolean withdraw( long amount ) {
        if( debit + amount <=  credit ) {
          debit += amount;
          return true;
    } else {
          return false;
    }
  }
```

Meilleur exemple: Compte bancaire (3b)

Toujours aucun changement de ce côté !

```
/* Code client */
Account acc = getAccount(2234);

//Retirons 100 CHF au compte
acc.widthraw( 100 );

//Position du compte
long remaining = acc.position();
```

Accesseurs: *getters & setters*

On appelle accesseurs les méthodes permettant d'accéder à des membres privés, en lecture (*getter*) ou en écriture (*setter*).

Leur pratique systématique est déconseillées car:

- fuite des détails d'implémentation
- contraintes et coordination plus difficile
- synchronisation plus difficile (env. concurrent)

Variante de catastophe: Compte bancaire (1a)

```
class Account {
  private long debit;
  private long credit;
  //....
  public long getCredit() { return credit; }
  public void setCredit(long amount) {
    credit = amount;
  }
  public long getDebit() { return debit; }
  public void setDebit(long amount) {
    debit = amount;
  }
}
```

Variante de catastophe: Compte bancaire (1b)

```
/* Code client */
Account acc = getAccount(2234);

//Retirons 100 CHF au compte
long deb = acc.getDebit();
long cred = acc.getCredit();
if( deb + 100 <= cred ) {
  acc.setDebit( deb + 100 );
}

//Position du compte
long remaining = acc.getDebit() - acc.getCredit();
```

Future proof (1)

En production, les comptes seront gérés dans une base de donnée...

```
class Account {
  private final String ID;

  //Librairie hypothétique
  private final DatabaseConnection conn;

  private Account(
        String ID,
        DatabaseConnection conn ) {
    this.ID = ID;
        this.conn = conn;
  }
```

Future proof (2)

```
class Account {
  //...
  public String getID() { return ID; }
  public long getCredit() { /* requête DB */ }
  public long getDebit() {  /* requête DB */ }
  public long position() {  /* requête DB */ }
  public void withdraw( long amount ) {
    /* requête DB */
  }
}
```

Concept: Encaspsulation

On appelle encapsulation l'approche consistant à cacher l'état et l'implémentation des données, derrière de fonctions de plus haut niveau.

Considéré comme un des concepts fondamentaux de la POO.

Encapsulation

Avantages

- Plus évolutif
- Plus réutilisable
- Plus simple à apprendre
- Plus testable

Inconvénient

- Plus difficile à optimiser (en mémoire et en temps)

Encapsulation non OO

Le système UNIX utilise abondamment l'encapsulation. Par exemple:

- Gestion du temps: la représentation finale est cachée.
- Gestion des fichiers: les *inodes* sont accessibles mais pas les données.
- Gestion des IO: descripteur de fichier vs. données

D'une manière générale, en C:

- fonction `static` (modificateur d'accès)
- types opaques

Packages

Package Java (paquet ?)

Les packages permette de regrouper des classes.

Ils forment un espace de nom (*name space*) pour éviter les collisions de noms (*name clash*)

Ils permettent de contrôler la visibilité des classes.

L'espace des packages est hierarchique.

Déclaration de package

```java
//début du fichier
package ch.unige.cui.fake.example;

//La classe Account dans est le package
class Account {

  /*...*/

}
```

Import de package

Pas besoin d'importer les classes du même package.

Pour les autres:

```
//Importer directement une classe
import ch.unige.cui.fake.example.Account;

//Importer toutes les classes d'un package
import ch.unige.cui.fake.example.*;

//Attention l'asterisque est non récursif
//N'importe pas Account
import ch.unige.cui.fake.*;
```

Nom complet

Au lieu d'importer on peut utiliser le package en préfixe:

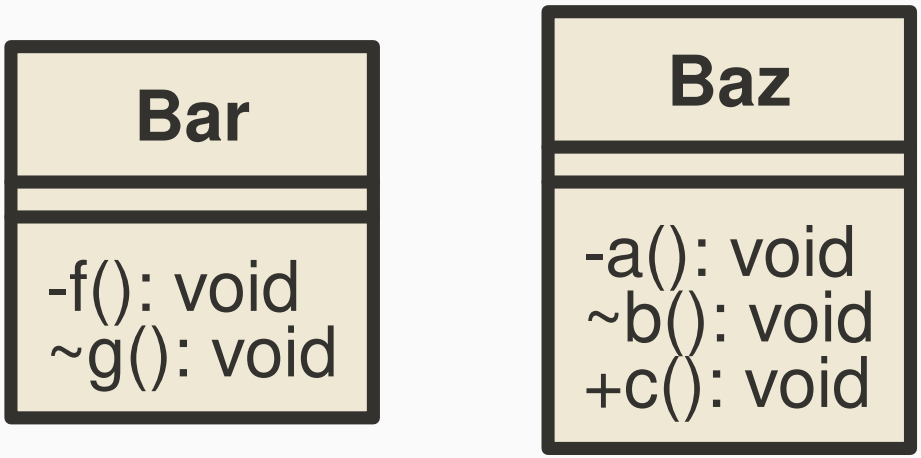
```
ch.unige.cui.fake.example.Acconut myAccount =
  new ch.unige.cui.fake.example.Account(
     "1234-56",120);

//Utiliser l'inférence avec profit:
var myAccount =
  new ch.unige.cui.fake.example.Account(
    "1234-56",120);
```

Modificateur d'accès: *package*

Le modificateur d'accès *package* est défini par l'absence d'autre modifcateur d'accès:

```
package foo;

class Bar { //Classe accessible dans foo
  private void f(); //Accessible à la classe
  void g(); //Accessible par autre classes de foo
}

public class Baz { //Accessible partout
  private void a(); //Accessible à la classe
  void b();  //Accessible par autre classes de foo
  public c(); //Accessible partout
}
```

package représentation UML



Import static

On peut aussi importer les membres statiques d'une classe:

```
import static Math.pow;
import static Math.sqrt;

class Point {
  //..
  double distance( Point that ) {
    return sqrt(
          pow( this.x - that.x, 2 ),
          .pow( this.y - that.y, 2 )
        );
  }
}
```

Diagrammes UML

Unified Modelling Language (UML)

Projet (1994)

- Modéliser complétement les systèmes informatiques par l'usage de diagrammes
  - Activité et processus
  - Structure du code
  - Spécification
  - Déploiement
- Standardiser et unifier les pratiques de *software design*

Evolution

- Programmation visuelle: échec
- Génération automatique: échec
- Dérive bureaucratique
- Haute complexité (version 2.0)

Usage dans ce cours

UML sera introduit progressivement.

Seule une partie (simplifiée) du standard sera utilisée.

L'accent sera mis sur UML en tant qu'outil de communication.

Objets, Interfaces et polymorphisme

Jean-Luc Falcone

8 mars 2022 (v2)

Classes dérivées

Dérivation de classe (1)

En Java toutes les classes dérivent d'une autre classe (sauf une).

Par défaut, toutes les classes dérivent de la class `java.lang.Object`.

La classe `Object` ne dérive d'aucune autre classe.

Dérivation de classe (2)

En java, la dérivation a deux effets simultanés, mais distincts:

1. Le sous-typage
2. L'héritage

Ces deux concepts sont souvent confondus.

Sous-typage

Un type T est un sous-type d'un type U si et seulement si toutes les valeurs de T sont aussi des valeurs de U. Notation:

$$T <: U \quad \iff \quad \forall t,\, t : T \to t : U$$

Si $T <: U$, alors on peut assigner une valeur du type `T` à une référence du type `U`:

```
T t = new T();
U u = t;        //Compile seulement si T<:U
```

Sous-typage: transitivité

La relation de sous-typage est transitive:

$$T <: U \text{ et } S <: T \implies S <: U$$

Type à la compilation vs. exécution (1)

Comme toutes les classes en Java sont des sous-type de `Object`, on peut toujours assigner une référence de type `Object`.

Le compilateur perd la trace du type sous-jacent.

```
Point p = new Point( 2.5, 1.2 );
Point q = new Point( 15, -1.0 );


Object o = p;
o.distance(q); //Erreur compilation
q.distance(o); //Erreur compilation
```

Type à la compilation vs. exécution (2)

On peut tester le type sous-jacent d'une instance à l'exécution avec l'opérateur `instanceof`.

```
Object o1 = new Point(2.3, 1.2);
Object o2 = "Hello, world !";

o1 instanceof Point;  //true
o1 instanceof Object; //true (toujours vrai)
o2 instanceof Point;  //false
o2 instanceof String; //true
```

Type à la compilation vs. exécution (3)

On peut forcer la conversion d'un type à l'exécution. Une erreur d'exécution se produit lorsque les types sont incompatibles:

```
Object o1 = new Point(2.3, 1.2);
Object o2 = "Hello, world !";

Point p = (Point) o1;
Point q = (Point) o2; //Erreur d'exécution
```

Type à la compilation vs. exécution: Remarques

Le *type cast* n'affecte que la référence. Cela n'affecte pas l'intance elle-même.

Pour éviter les surprises, il faudrait le protéger dans un `if`:

```
if( o instanceof Point ) {
  Point p = (Point) o;
  return p.distance(q)
} else {
  //Ne pas oublier de gérer ce cas
}
```

A partir de java 14: un meilleur `instanceof`

Il est désormais possible de combiner `instanceof` et *type-cast*.

```
if( o instanceof Point p ) {
  return p.distance(q);
} else {
  //Ne pas oublier de gérer ce cas
}
```

Héritage

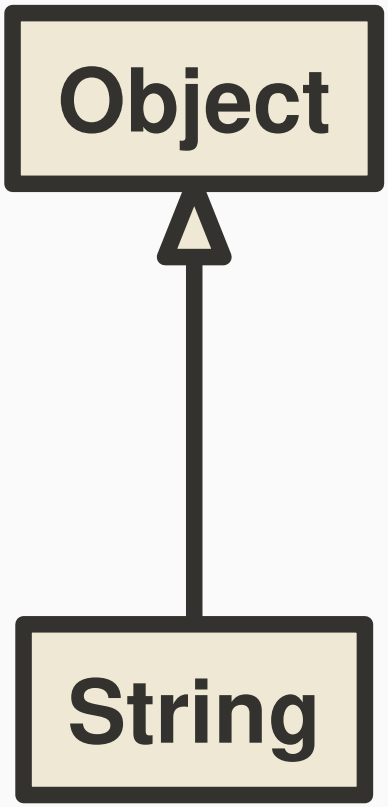
L'héritage est un mécanisme qui permet à une classe d'obtenir la structure et le comportement de la classe dont elle dérive (champs et méthodes).

Ce mecanisme est complexe, souvent mal compris, et très souvent mal utilisé.

En Java toute classe ne peut hériter que d'une seule classe, `Object` par défaut.

Donc, par transitivité toute classe est un sous-type d'`Object`.

Héritage: Représentation UML



En UML on note l'héritage par une flêche à la point blanche.
On dit que la classe `String` spécialise la classe `Object`
Réciproquement la classe `Object` généralise la classe `String`.

Héritage de `Object`

Comme toutes les classes dérivent d'`Object`, toutes les classes ont une implémentation par défaut de plusieurs méthodes. Notamment:

```java
public String toString(); //Instance en String

public boolean equals(Object o); //Egalité

public int hashCode(); //Code de hashage
```

On peut cependant redéfinir ces méthodes dans la classe dérivée (*override*).

Identité (==)

L'opérateur `==` retourne vrai si, et seulement si, deux références pointent sur la même instance:

```
Point p = new Point(1,2);
Point q = new Point(1,2);
Object o = p;

p == p; //vrai
p == q; //faux instances différentes
p == o; //vrai
```

Egalité (`equals`)

Par défaut la méthode `equals` fonctionne comme l'opérateur `==`:

```
Point p = new Point(1,2);
Point q = new Point(1,2);
Object o = p;

p.equals(p); //vrai
p.equals(q); //faux instances différentes
p.equals(o); //vrai
```

Egalité: redéfinition (1)

Exemple selon les bonnes pratiques:

```
class Point {
  private final double x;
  private final double y;
  /*...*/
  public boolean equals( Object o ) {
    if( o == this ) return true;
    if( o == null ) return false;
    if( ! (o instanceof Point) ) return false;
    Point that = (Point) o;
    return that.x == x && that.y == y;
  }
}
```

Egalité: redéfinition (2)

```
Point p = new Point(1,2);
Point q = new Point(1,2);
Object o = p;

p.equals(p); //vrai
p.equals(q); //vrai, cf redéfinition
p.equals(o); //vrai
```

Egalité: redéfinition (3)

En redéfinissant l'égalité, il faut respecter les règles suivantes:

```
//Pour tout x,y et z
x.equals(x) == true;
x.equals(y) == y.equals(x);
if( x.equals(y) && y.equals(z)  ) x.equals(z);
if( x != null ) ! x.equals(null);
x.equals(y) == x.equals(y);
```

Code de hachage

Le code de hachage (*hash code*) est une valeur numérique calculée pour chaque instance.

Utilisée dans la librairie standard (par exemple `HashMap`).

Règle: Si `x.equals(y)` alors `x.hashCode()` est égal à `y.hashCode()`.

La réciproque n'est pas nécéssaire.

Corrolaire: Si on redéfinit `equals`, il faut redéfinit `hashCode`.

Redéfinition: `hashCode` (1)

Critères à respecter (en plus de la règle), pour la fonction `hashCode`:

- aussi rapide que possible
- se base sur les mêmes champs qu'`equals`
- consistente d'un appel à l'autre
- retourne des valeurs bien réparties sur l'espaces des nombres entiers (`int`).

Redéfinition: `hashCode` (2)

Voici quelques pistes pour redéfinir un `hashCode`:

- Pour les membres qui sont des objets, utiliser la fonctions statiques `Objects.hashCode()`
- Pour les nombres entiers utiliser leur valeur telle quelle.
- Pour les nombre à virgule flottante, utiliser les fonctions statiques `Double.hashCode()` ou `Float.hashCode()`.
- Combiner les résultats obtenus en multipliant par des nombres premiers et en additionnant (arbitraires).

Redéfinition: hashCode (2)

```
class Point {
  private final double x;
  private final double y;
  /*...*/
  public int hashCode() {
    int h = 17;
        h = h*31 + Double.hashCode(x);
        h = h*31 + Double.hashCode(y);
        return h;
  }
}
```

Value Object vs. Entity

Deux types principaux d'objets:

1. Les objets-valeurs (*value objects*)
2. Les entités (*entity*)

Tiré de *Domain Driven Design*, E. Evans (2004)

Value Objects

Définis uniquement par la valeur de leurs champs.

Si les valeurs sont les mêmes, on considère que c'est le même objet.

Exemples:

- Couleurs
- Dates
- URL
- Point dans l'espace

Entité

N’est pas déinis par la valeur de ses champs.

Un même objet peut avoir des valeurs différentes au cours du temps:

Exemples:

- Personne
- Reservation
- Compte en banque

Recommandations

	Value object	*Entity*
Immutabilité	toujours	souvent
Redéfinir `equals`	toujours	à éviter
Identifiant unique	inutile	encouragé
Gérer le cycle de vie	inutile	nécéssaire

Interface

Polymorphisme

En informatique, on appelle polymorphisme le fait d'avoir plusieurs comportement rattaché à seul symbol.

On verra ce semestre trois types de polymorphisme:

1. Polymorphisme ad-hoc
2. Polymorphisme par sous-typage
3. Polymorphisme par paramétricité

Polymorphisme ad-hoc: Surcharge de méthode

```
/* Lit sur le système de fichier */
static Image load( File file ) { /*...*/ }

/* Lit sur le réseau */
static Image load( URL url ) { /*...*/ }
```

Polymorphisme ad-hoc: Surcharge d'opérateurs

```
/* addition d'entiers */
2 + 1;

/* concaténation de strings */
"deux" + "un";
```

Polymorphisme par sous-typage

Différents sous-types d'un même type peuvent avoir différentes implémentations pour les mêmes méthodes.

Deux manières de faire en Java:

- Héritage de classe
- Implémentation d'Interface

Interfaces

Abstraction permettant d'exprimer des contraintes sur la présence de méthodes et leur signature.

Définit une forme de contrat, analogue à un protocole de communication.

Mécanisme de sous-typage.

Notification d'alertes (1)

But: intégrer un système d'envoi d'alertes en cas de problèmes.

- Deux niveaux d'alerte: `WARNING` et `CRITICAL`.
- Doit contenir la date et l'heure.
- Plusieurs sorties envisagés:
  - Ecrire sur la console (`System.err.println`).
  - Sauver dans un fichier de log.
  - Envoyer un mail.
  - Utiliser l'API de *Signal* (ou *Slack*, ou un bot *Discord*)
  - Combinaison des options ci-dessus

Notification d'alertes (2a)

```
import java.time.LocalDateTime;
public class Alert {
  private String format(String level, String msg) {
    var t = LocalDateTime.now();
    return t + " " + level + " " + msg;
  }
  public void warning(String msg) {
    System.err.println(
      format("WARNING", msg));
  }
  public void critical(String msg) {
    System.err.println(
      format("CRITICAL", msg));
  }
}
```

Notification d'alertes (2b)

```
import java.io.File;
public void checkFiles(File[] files, Alert alert){
  for( int i=0; i<files.length; i++ ) {
    final var f = files[i];
    if( !f.exists() )
          alert.critical("File "
        + f + " missing.");
    else if( ! f.canRead() )
          alert.warning("File " + f
        + " not readable.");
  }
}
```

Notification d'alertes (3a)

```
import my.library.MailService;
public class Alert {

  private final String[] warnAddress;
  private final String[] critAddress;
  private final MailService mailer;

  public Alert( MailService mailer;
      String[] warnAddress,
      String[] critAddress ) {
          this.mailer = mailer;
          this.warnAddress = warnAddress;
          this.critAddress = critAddress;
  }
  /*...*/
```

Notification d'alertes (3b)

```
public class Alert {
  /* ... */
  public void warning(String msg) {
    for( int i=0; i<warnAddress.size; i++ ) {
          mailer.send( warnAddress[i], //dest
              "WARNING", //subjet
              msg );     //body
    }
  }
  /*...*/
}
```

Notification d'alertes (3c)

```
import java.io.File;
public void checkFiles(File[] files, Alert alert){
  for( int i=0; i<files.length; i++ ) {
    final var f = files[i];
    if( !f.exists() )
          alert.critical("File "
        + f + " missing.");
    else if( ! f.canRead() )
          alert.warning("File " + f
        + " not readable.");
  }
}
```

Notification d'alerte: interface

Définition des méthodes commune dans une interface:

```
public interface Alert {
  //Pas d'implémentation,
  //  les méthodes sont abstraites
  public void warning( String msg );
  public void critical( String msg );
}
```

Les interfaces

Syntaxe de la déclaration similaire à celles d'une classe:

- Méthodes abstraites (non implémentées)
- Pas de champs
- Pas de membres privés
- Fonctions et constantes statiques possibles
- Depuis JSE8, implémentations par défaut.

Les intefaces: implémentation

Classe peut implémenter une ou plusieurs interfaces:

- Mot clé: `implements`
- Doit implémenter toutes les méthodes abstraites

Introduit une relation de sous typage.

```
/* String <: CharSequence */
class String implements CharSequence {
  /*...*/
}
```

Notification d'alerte: implémentation (1)

```
import java.time.LocalDateTime;
public class ConsoleAlert implements Alert {
  private String format(String level, String msg) {
    var t = LocalDateTime.now();
    return t + " " + level + " " + msg;
  }
  public void warning(String msg) {
    System.err.println(
      format("WARNING", msg));
  }
  public void critical(String msg) {
    System.err.println(
      format("CRITICAL", msg));
  }
}
```

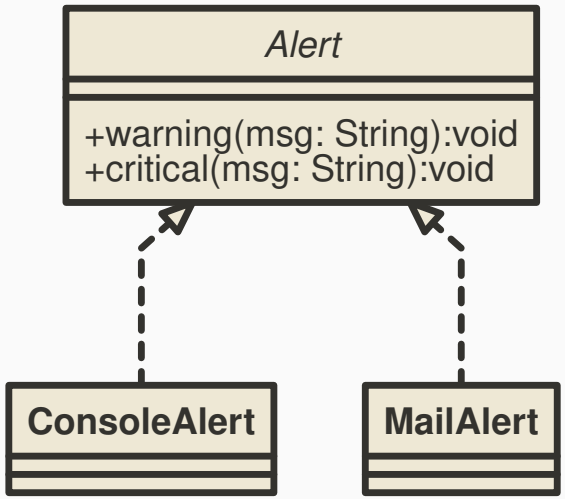
Notification d'alerte: implémentation (2)

```
import my.library.MailService;
public class MailAlert implements Alert {

  private final String[] warnAddress;
  private final String[] critAddress;
  private final MailService mailer;

  public Alert( MailService mailer;
      String[] warnAddress,
      String[] critAddress ) {
          this.mailer = mailer;
          this.warnAddress = warnAddress;
          this.critAddress = critAddress;
  }
  /*...*/
```

Notification d'alerte: utilisation

```
import java.io.File;
public void checkFiles(File[] files, Alert alert){
  for( int i=0; i<files.length; i++ ) {
    final var f = files[i];
    if( !f.exists() )
          alert.critical("File "
        + f + " missing.");
    else if( ! f.canRead() )
          alert.warning("File " + f
        + " not readable.");
  }
}
```

Interface: Représentation UML



On UML on dénote une interface par l'utilisation d'italique dans le nom.

L'implémentation est représentée par une flêche à la pointe blanche, en traitillé.

En jargon UML on dit `ConsoleAlert` réalise l'interface `Alert`.

Notification d'alertes: alertes multiples (1)

```
public class MultiAlert implements Alert {
  private final Alert[] warnAlerts;
  private final Alert[] critAlerts;
  public MultiAlert(
    Alert[] warnAlerts,
        Alert[] critAlerts
  ) {
     this.warnAlerts = warnAlerts;
     this.critAlerts = critAlerts;
  }
  /*...*/
}
```

Notification d'alertes: alertes multiples (2)

```
public class MultiAlert implements Alert {
  /*...*/
  public void warning( String msg ) {
    for( int i=0; i<warnAlerts; i++ ) {
          warnAlerts[i].warning(msg);
  }
  public void critical( String msg ) {
    for( int i=0; i<critAlerts; i++ ) {
          crittAlerts[i].critical(msg);
  }
}
```

Notification d'alertes: alertes multiples (3)

Envoyer les *warnings* sur la console, et les *critical* sur la console et par email.

```
var console = new ConsoleAlert();
var email = new MailAlert(ms,
  new String[0],
  new String[] {"alice@truc.com","bob@machin.com"}
);
Alert a = new MultiAlert(
   new Alert[] {console},
   new Alert[] {console,email}
);
```

Fake it till you make it!

Fake it till you make it ! (1)

On peut utiliser des interface pour remplacer un composant qui n'est pas encore écrit.

On peut produire un semblant d'implémentation (en anglais *mock implementation* ou *dummy object*), pour les tests.

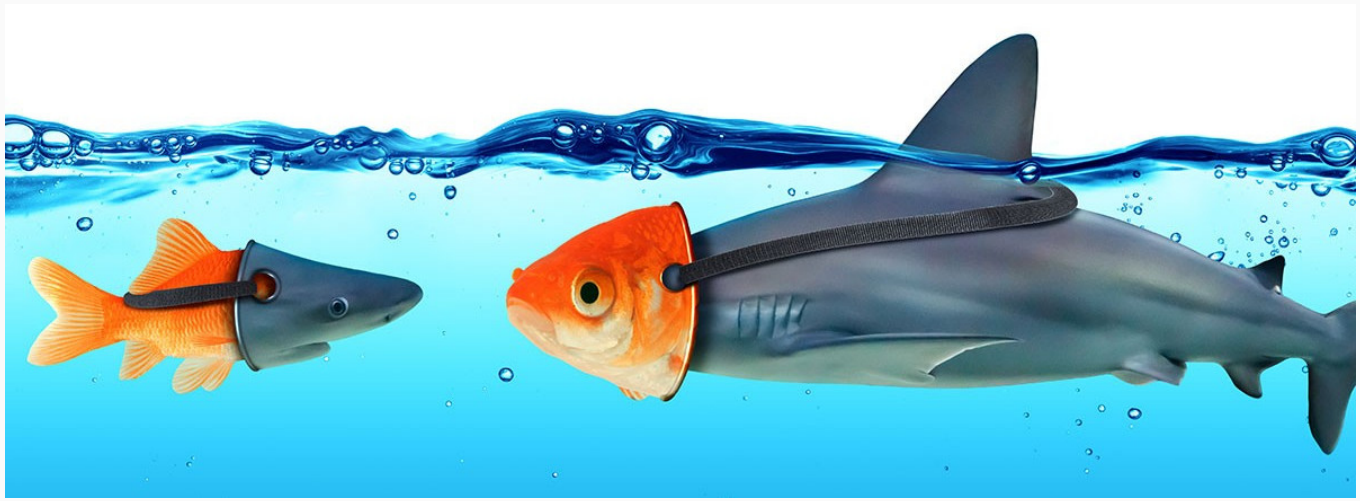


Image tirée du site *ABA For Law Student* (2017).

Fake it till you make it! (2)

```
public interface MailService {
  public void send( String recipient,
    String subject, String body );
}

public interface Authenticator {
  public boolean check( String userid,
    String password );
}
```

Fake it till you make it! (3)

```
public class TestMailer implements MailService {
  private String lastRecipient = "";
  private String lastSubject = "";
  private String lastBody = "";
  /* ... field getters ommitted */
  public void send( String recipient,
    String subject, String body ) {
    last.recipient = recipient;
    last.subject = subject;
    last.body = body;
  }
}
```

Fake it till you make it! (4)

```
public class FakeAuth implements Authenticator {
  private final String password;
  public FakeAuth( pw ) { password = pw; }
  public boolean check( String userid,
    String password ) {
          return this.password.equals( password );
  }
}
```

Quelques remarques

Extension d'interfaces (1)

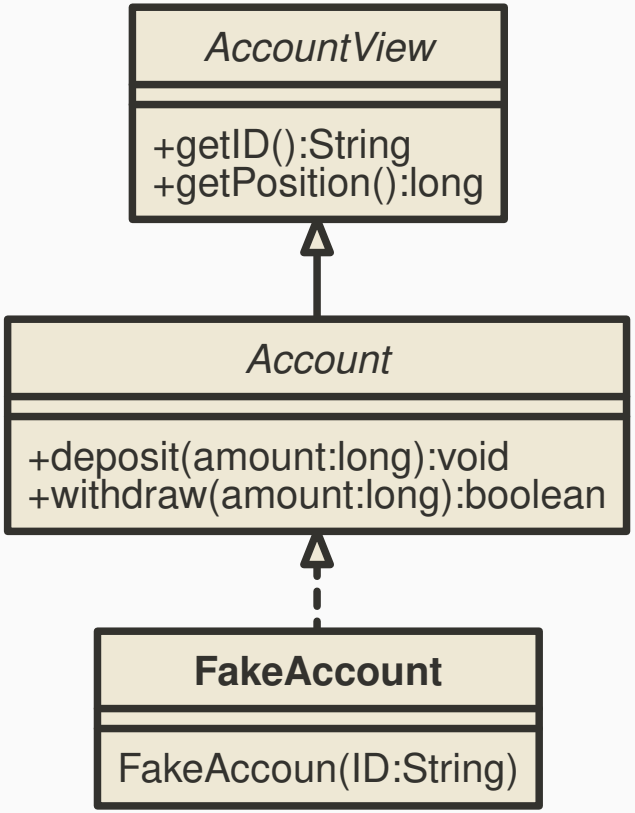
```
public interface AccountView {
  public String getID();
  public long getPosition();
}
```

```
public interface Account
  extends AccountView {
  public void deposit(long amount);
  public boolean widthdraw(long amount);
}
```

Extension d'interfaces (2)

```
class FakeAccount implements Account {
  private long amount = 0;
  private final String ID;
  FakeAccount(ID String) { this.ID=ID; }
  public String getID() { return ID; }
  public long getPosition() { return amount; }
  public void deposit(long amount) {
    this.amount += amount; }
  public boolean widthdraw(long amount) {
    if( this.amount >= amount ) {
          this.amount -= amount;
      return true;
    } else return false;
  }
}
```

Interface: Représentation UML



On a les relations de sous-typage suivantes:

- `Account <: AccountView`
- `FakeAccount <: Account`
- `FakeAccount <: AccountView`

Extension d'interfaces (3a)

```
void prettyPrint( AccountView acc ) {
  System.out.println(
    acc.getID() + "=" + acc.getPosition() );
}

AccountView accountByID( String ID ) {...};

prettyPrint( accountByID("123-456") );
prettyPrint( accountByID("123-456") );
```

Extension d'interfaces (3b)

```
boolean transfer( Account from, Account to,
  long amount ) {
  if( from.withdraw(amount) ) {
    to.deposit(amount);
    return true;
  } else return false;
}

//Ne compile pas
prettyPrint( accountByID("123-456"),
  accountByID("123-456") );
```

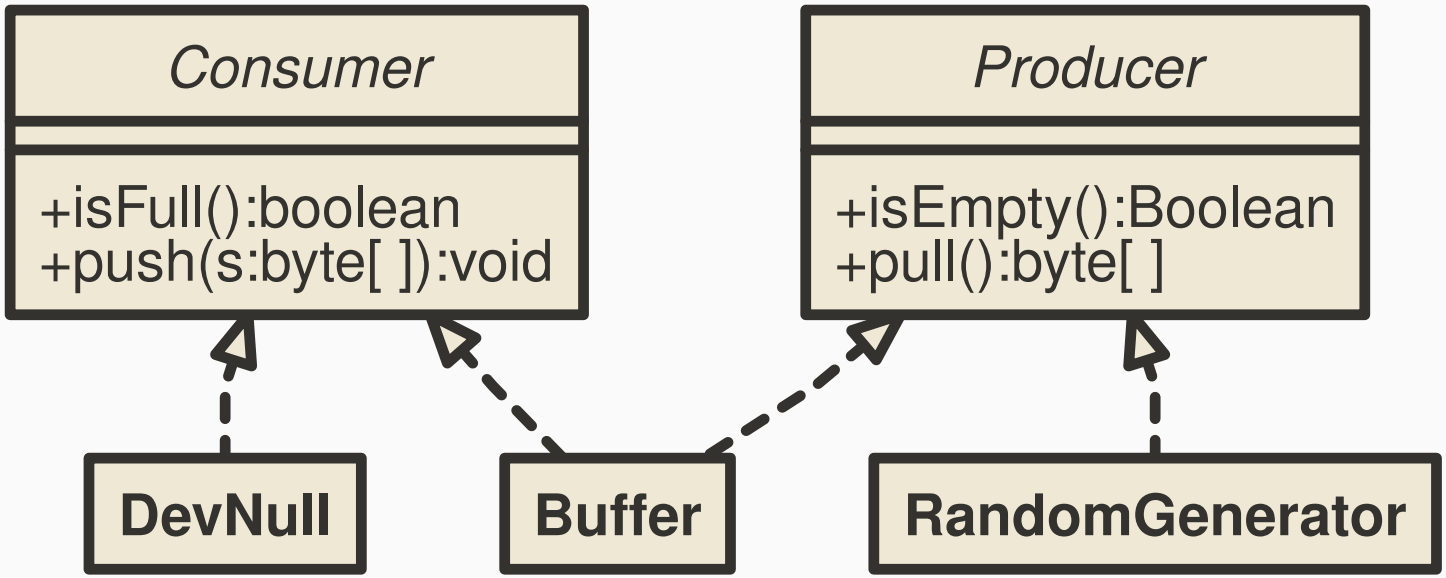
Types plutôt que des commentaire

```
//Ne modifie pas 'acc'
void prettyPrint( AccountView acc ) {
  //...
}

//Peut modifier 'from' et 'to'
boolean transfer( Account from, Account to,
    long amount ) { /*...*/ }

//Peut modifier 'x' mais pas 'y'
long f( Account x, AccountView y ) { //...
```

Attention aux petits malins (et aux idiots)

```
void prettyPrint( AccountView acc ) {
  if( acc instanceof Account ) {
    ( (Account) acc ).deposit(1_000_000);
  }
  System.out.println(
    acc.getID() + "=" + acc.getPosition() );
}
```

Autre exemple: Poducteur-Consommateur



Limitations des interfaces

Pas de constructeurs abstraits:

- Remplacé par les *abstract factory* plus tard dans le cours.

Pas de champs abstraits:

- Raison supplémentaire pour garder tous les champs privés.

Polymorphisme hors de la POO

Systèmes POSIX, en C:

- opérations sur les descripteurs de fichiers
- gestion des adresses pour les sockets: IPv4, IPv6, Unix Sockets

```
struct sockaddr_in server; //Adresse IPv4
/* ... */

//'struct sockaddr' est la "structure parent"
bind(fd, (struct sockaddr *) &server,
  sizeof(server));
```

Attention aux excès

L'abus de polymorphisme et d'encapsulation peut rendre le code:

- Lent
- Gourmand en mémoire
- Illisible
- Indébuggable

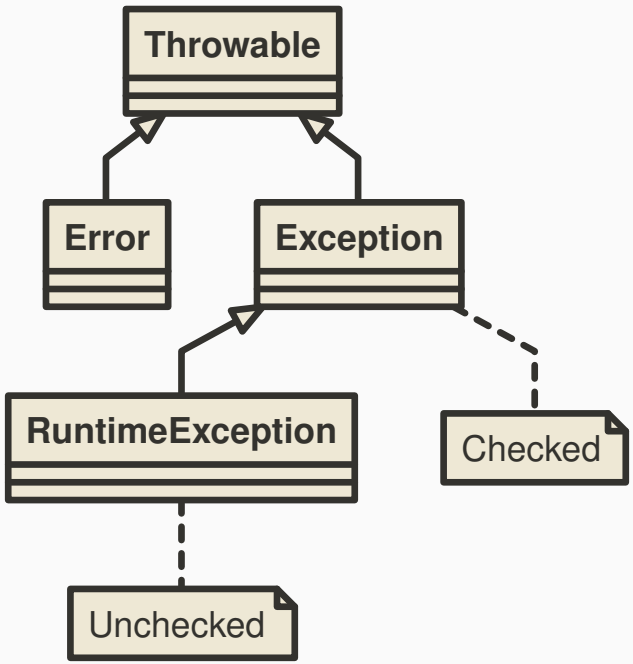
Objets, Interfaces et polymorphisme

Jean-Luc Falcone

15 mars 2022

Exceptions

Hierarchie des erreurs et exceptions



Sous-types de:

- `Error`: ne doit pas être gérée (`OutOfMemoryException`)
- `Exception`: doit être gérée (`IOException`)
- `RuntimeException`: peut être gérée (`IndexOutOfBoundsException`)

Distinction

`Throwable` est le super types de toutes les exceptions et erreurs.

Les `Error` sont des problèmes graves dus à l'environement: manque de mémoire, bug dans la machine virtuelle, etc.

Les `Exception` sont des problèmes imprévisibles à gérer: problème d'IO, *time-out*, *thread* interrompu (*Checked exceptions*)

Les `RuntimeExcepion` sont liés à des erreurs de programmation: dépassement de tableau, argument invalide, erreur de conversion, etc. (*Unchecked exception*)

Méthodes communes

Définies dans `Throwable`:

```
//Retourne le message d'erreur
String  getMessage() {/*...*/}

//Affiche la pile d'appel de méthodes
void printStackTrace(){/*...*/}

//Retourne la pile d'appel de méthodes
StackTraceElement[] getStackTrace() {/*...*/}

//Retourne la cause première ou null
Throwable getCause() {/*...*/}
```

Provoquons une exceptions (1)

```
1  class ExceptionDemo1 {
2    private static int go() {
3      int[] ii = {11,12};
4          return ii[3];
5    }
6    public static void main(String[] args) {
7      System.out.println("Result: " + go());
8    }
9  }
```

Provoquons une exceptions (2)

```
$ javac ExceptionDemo1.java
$ java ExceptionDemo1

Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException:
  Index 3 out of bounds for length 2
at ExceptionDemo1.go(Machin.java:4)
at ExceptionDemo1.main(Machin.java:7)
```

Lever une exception (1)

On peut lever une exception avec le mot clé **throw** suivi d'une instanciation de l'exception:

```
2  public static double compute( double p ) {
3    if( p < 0.0 || p > 1.0 )
4      throw new IllegalArgumentException(
5            "Invalid probability. Got: " + p);
6    return p*p;
7  }
8  public static void main(String[] args) {
9    System.out.println("x=" + compute(0.95));
10   System.out.println("y=" + compute(1.05));
11 }
```

Lever une exception (2)

```
x=0.9025
Exception in thread "main"
  java.lang.IllegalArgumentException:
  Invalid probability. Got: 1.05
at ExceptionDemo2.compute(Machin.java:4)
at ExceptionDemo2.main(Machin.java:10)
```

Lever une *checked* exception (1)

```
1  import java.io.File;
2  import java.io.IOException;
3  /*...*/
4  public static void save( File file ) {
5    if( ! file.canWrite() )
6      throw new IOException(
7        "File not writable: " + file );
8      /*...*/
9  }
```

Lever une *checked* exception (2)

```
$ javac ExceptionDemo3.java
ExceptionDemo3.java:6: error:
  unreported exception IOException;
  must be caught or declared to be thrown
throw new IOException("File not writable: " + file
^
1 error
```

Spécifier une exception (1)

On spécifie (déclare) une exception avec le mot clé `throws` à la fin de la signature de la méthode

```
4  public static void save( File file )
5    throws IOException  //Spécification
6  {
7    if( ! file.canWrite() )
8      throw new IOException(
9        "File not writable: " + file );
10     /*...*/
11 }
```

Spécifier une exception (2)

Toute méthode peut respécifier une *checked* exception:

```
public static void save( File file )
  throws IOException  //Spécification

//Ne compile pas, car ne gère pas l'exc. de
//save(...)
public static void foo() {
  save( new File("/tmp/foo.txt") );
}

//Compile
public static void bar() throws IOException {
  save( new File("/tmp/bar.txt") );
}
```

Gérer une exception

Décider du comportement à adopter:

- Terminer l'application
- Journaliser ou notifier l'exception
- Utiliser une valeur par défaut
- Ressayer (evt. après attente)
- Essayer un plan B
- Lever une autre exception
- Ignorer l'exception (mauvaise idée en général)

Gestionnaire d'exception (1)

Composé de trois parties:

1. Tentative d'exécuter du code (`try`)
2. Capture d'exceptions produite (`catch`)
3. Optionnellement, code exécuté dans tous les cas à la fin (`finally`)

Gestionnaire d'exception (2)

```
String filename = "/tmp/machin.txt";
try {
  File f = new File(filename);
  save( f );
} catch( IOException e ) {
  System.err.println("File: " + filename
     + " could not be saved" );
  System.err.println("Because: "
     + e.getMessage() );
}
```

Gestionnaire d'exception (3)

On peut capturer plusieurs exceptions:

```
try {
  foo();
  bar();
  baz();
} catch( FooException e ) {
  //Capture FooException
} catch( BarException|BazException e ) {
  //Capture BarException ou BazException
}
```

Gestionnaire d'exception (4)

Le bloc `finally` est exécuté dans toutes les circonstances:

- Pas d'exception
- Exception gérée
- Exception non gérée

```
try {
  foo();
} catch( FooException e ) {
  //Capture FooException
} finally {
  //Bloc toujours exécuté
}
```

Remarques

L'utilisation d'exceptions doit rester exceptionnelle:

- Anticiper les problème
- Utiliser le type de retour
- Ne pas utiliser pour la logique

Vous pouvez créer vos propres exceptions:

- Seulement si ça apporte quelque chose
- Mécanisme d'héritage

Anticiper les problèmes

```java
//Exception si le tableau est vide
static int max( int[] values ) { /*...*/ }

//Plus besoin d'exception
static int max( int value, int[] values ) {
  /*...*/
}
```

Abomination: exceptions en tant que contrôle de flot

```
static int sum( int[] values ) {
  int s = 0;
  try {
    for( int i=0; ; i++ ) {
          s += values[i];
        }
  } catch( IndexOutOfBoundsException e ) {
    return s;
  }
}
```

Entrées/Sorties

IO vs. NIO vs. NIO2 (1)

Java IO (Java 1.0, 1996)

- Package `java.io.*`
- Gestion basique de fichiers `File`
- E/S basées sur des flots `InputStream` et `OutputStream`

Java NIO (Java 1.4, 2002)

- Package `java.nio.*`
- IO non bloquantes et non déterministes
- Abstraction de plus bas niveau: `Buffer` et `Channels`

Java NIO2 (Java 7, 2011)

- Meilleures abstractions fichiers (`java.nio.file.*`)
- IO asynchrones (`AsynchronousChannel`)

IO vs. NIO vs. NIO2 (2)

Pour les E/S, utiliser:

- `java.nio.*` si les performances sont cruciales;
- `java.nio.*` dans un système concurrent;
- `java.nio.*` s'il est nécéssaire d'accéder à des opération de plus bas niveau;
- `java.io.*` dans tous les autres cas.

Pour les fichiers:

- `java.io.*` pour les petites demos;
- `java.nio.file.*` dans tous les autres cas.

Nous donnerons ici quelques pistes pour débuter.

PrintWriter: Ecritures, fichiers textes

```
import java.io.*;

String filename = "demo.txt"
try {
  var out = new PrintWriter(filename);
  out.print( "Hello, ");
  out.println("World !");
  out.close();
} catch( IOException e ) {
  System.err.println("Could not write, because:");
  System.err.println(e.getMessage());
}
```

Gestion des ressources manuelle

Il est essentiel d'appeler `close()` après avoir terminé:

- Pour forcer l'écriture du *buffer*;
- Pour libérer des resources;
- Pour éviter que le fichier ne soit bloqué.

Gestion des ressources automatique

Depuis Java 7 (2011), on peut utiliser cette syntaxe:

```
try( var out = new PrintWriter(filename) ) {
  out.print( "Hello, ");
  out.println("World !");
  out.close();
} catch( IOException e ) {
  System.err.println("Could not write, because:");
  System.err.println(e.getMessage());
}
```

`FileReader`, lire un fichier texte

```
import java.io.*;

try( var in = new BufferedReader(
     new FileReader(filename)) {
  String line = "";
  int i = 1;
  while ((line = in.readLine()) != null) {
    System.out.println(i + ": " + line );
    i++;
  }
} catch( IOException e ) {
  System.err.println("Couldn't read, because:");
  System.err.println(e.getMessage());
}
```

Character encoding

Pour vous simplifier la vie, choisissez l'**UTF-8** !

```
import static java.nio.charset.
        StandardCharsets.UTF_8;
import java.io.*;

//En écriture
var out = new PrintWriter("foo.txt", UTF_8);

//En lecture
var in = new BufferedReader(
  new FileReader("foo.txt", UTF_8) );
```

E/S binaires dans `java.io.*`

Fichiers binaires:

- `FileInputStream`
- `FileOutputStream`

Tableaux de bytes

- `ByteArrayInputStream`
- `ByteArrayOutputStream`

Génériques, Introduction

Une pile de Strings (1)

```
public class Stack {
  //Capacité initiale
  private static final int INIT_CAP = 20;
  private String[] data;
  private int next;
  public Stack() {
    data = new String[INIT_CAP];
    next = 0;
  }
  public boolean isEmpty() { //...
  public String pop() { //...
  public void push( String s ) //...
}
```

Une pile de Strings (2)

```
public void push( String s ) {
  if( next == data.length )
    increaseCapacity();
  data[next] = s;
  next++;
}
private void increaseCapacity() {
  int newSize = data.length * 2;
  var newData = new String[newSize];
  System.arraycopy(data,0,newData,0,data.length);
  data = newData;
}
```

Une pile de Strings (3)

```
public boolean isEmpty() {
  return next == 0;
}
public String pop() {
  if( isEmpty() )
    throw new NoSuchElementException(
        "Empty stack");
  var s = data[next-1];
  data[next-1] = null;
  next--;
  return s;
}
```

Une pile de Strings (4)

```
static void topToUpperCase( Stack stack ) {
  if( stack.isEmpty() ) return;
  String s = stack.pop();
  String s2 = s.toUpperCase();
  stack.push( s2 );
}

Stack stack = new Stack();
stack.push("hello");
stack.push("world");
topToUpperCase(stack);
```

Une pile d'objets (1)

```
public class Stack {
  //Capacité initiale
  private static final int INIT_CAP = 20;
  private Object[] data;
  private int next;
  public Stack() {
    data = new Objec[INIT_CAP];
    next = 0;
  }
  public boolean isEmpty() { //...
  public Object pop() { //...
  public void push( Object s ) //...
}
```

Une pile d'objets (2)

```
static void topToUpperCase( Stack stack ) {
  if( stack.isEmpty() ) return;
  Object o = stack.pop();
  if( o instanceof String ) {
    String s = (String) o;
    String s2 = s.toUpperCase();
    stack.push( s2 );
  } else  stack.push(o);
}

Stack stack = new Stack();
stack.push("hello");
stack.push(new Point(2.3, 4.5));
topToUpperCase(stack);
```

Une pile Générique (1)

```
public class Stack<T> {
  private static final int INIT_CAP = 20;
  private Object[] data;
  private int next;
  public Stack3() {
    data = new Object[INIT_CAP];
    next = 0;
  }
  public boolean isEmpty() { //...
  public void push( T t ) { //...
  public T pop() //...
}
```

Une pile Générique (2)

```
public boolean isEmpty() {
  return next == 0;
}

public void push( T t ) {
  if( next == data.length )
    increaseCapacity();
  data[next] = t;
  next++;
}
```

Une pile Générique (3)

```java
private void increaseCapacity() {
  int newSize = data.length * 2;
  var newData = new Object[newSize];
  System.arraycopy(data,0,newData,0,data.length);
  data = newData;
}
@SuppressWarnings("unchecked")
public T pop() {
  if( isEmpty() )
    throw new NoSuchElementException("Empty stack");
  var t = data[next-1];
  data[next-1] = null;
  next--;
  return (T) t; //Pas besoin de vérifier
}
```

Une pile générique (4)

```
static void topToUpperCase( Stack<String> stack ) {
  if( stack.isEmpty() ) return;
  String s = stack.pop();
  String s2 = s.toUpperCase();
  stack.push( s2 );
}

Stack<String> stack = new Stack<String>();
stack.push("hello");
topToUpperCase(stack);

//Erreur de compilation
stack.push(new Point(2.3, 4.5));
```

Déclarations

Les déclarations suivantes sont équivalentes:

```
//Redondant, mais explicite (JSE5 2004)
Stack<String> stack = new Stack<String>();

//Opérateur diamant (JSE7 2011)
Stack<String> stack = new Stack<>();

//Inférence de types (JSE10 2018)
var stack = new Stack<String>();
```

Génériques et primitives

Les types primitifs ne sont pas des objets.

```
//Erreur de compilation
Stack<int> stack = new Stack<>();
```

```
|  Error:
|  unexpected type
|    required: reference
|    found:    int
|  Stack<int> stack = new Stack<>();
|        ^-^
```

Boxing (1)

Il existe cependant des classes qui permettent d'emballer des primitives (*boxing*).

Primitive	Classe correspondante
`boolean`	`Boolean`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer` (!!?!)
`float`	`Float`
`double`	`Double`
`char`	`Character` (!!?!)

Boxing (2)

Ces classes sont des objets mais ne permettent pas de faire de l'arithmétique directement.

```
Integer i = Integer.valueOf(2);//Boxing
Integer j = Integer.valueOf(3);//Boxing

//k = i + j
Integer k = Integer.valueOf(  //Boxing
  i.intValue() + j.intValue() //Unboxing
);
```

Auto-Boxing (1)

Le compilateur peut automatiquement insérer le code pour de *boxing* et *unboxing* depuis JSE5:

```
Integer i = 2; //Auto-boxing
Integer j = 3; //Auto-boxing


//Auto-unboxing (2x), Auto-boxing
Integer k = i + j;
```

Auto-Boxing (2)

Cela permet d'utiliser les génériques pour des types primitifs:

```
static void addTop( Stack<Integer> ints ) {
  int i = ints.pop(); //Auto-unboxing
  int j = ints.pop(); //Auto-unboxing
  ints.push( i+j ); //Auto-boxing
}

var stack = new Stack<Integer>();
stack.push(2); //Auto-boxing
stack.push(3); //Auto-boxing
addTop(stack);
```

Auto-Boxing (3)

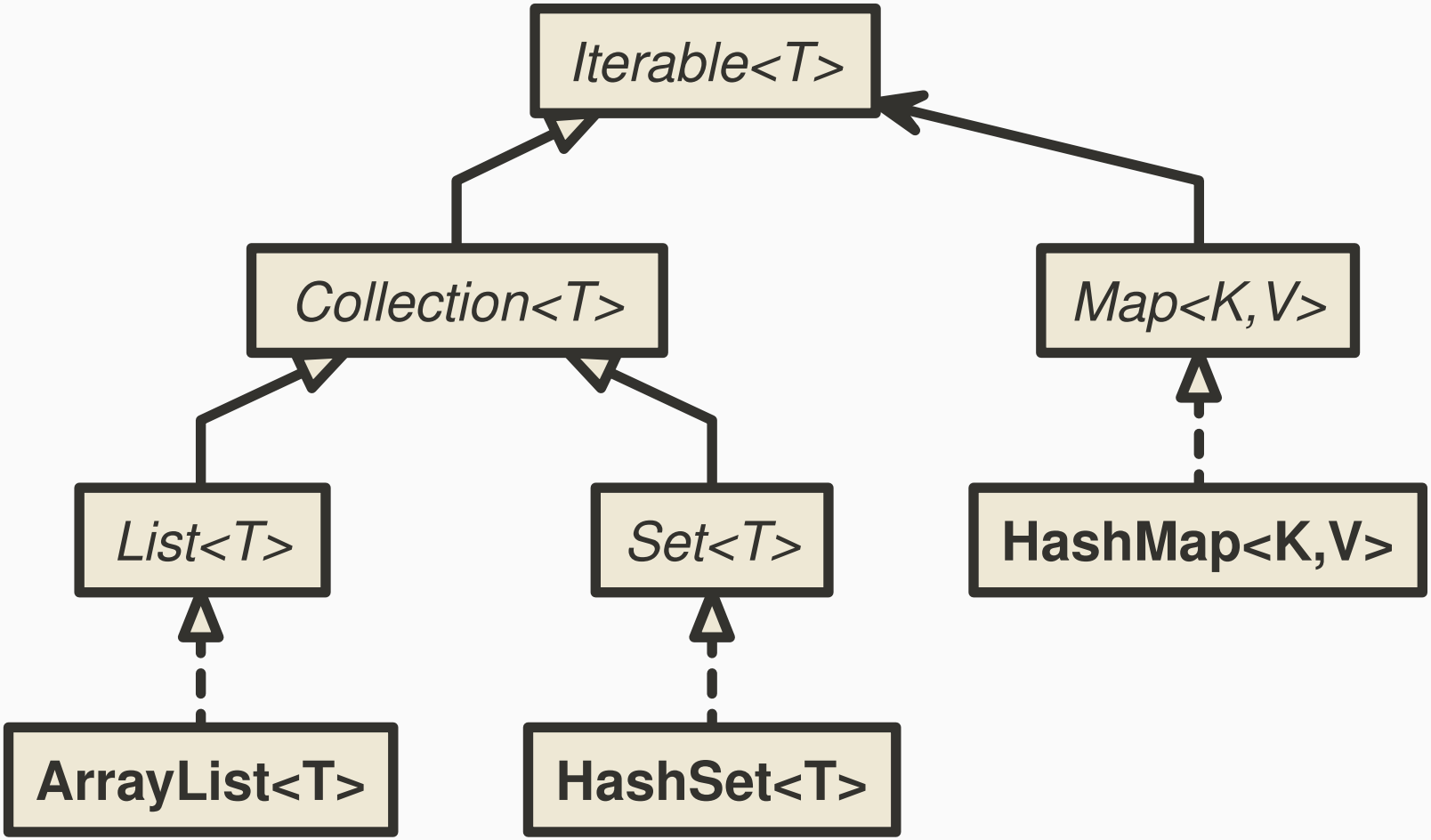
L'auto-boxing introduit une baisse de performance:

- Plusieurs cycles perdus à exécuter le code inséré
- Les instances prennent plus de mémoire, que les primitives.
- Les instances sont allouées sur le tas: plus lent, cache CPU inefficace

Collections

Organisation

Package `java.util`:



Interface `Collection<E>` (1)

Super-type de toutes les collections de la librairie standard. Exceptés les dictionnaires, les chaînes de caractères, et les tableaux.

Attention:

> *Certain methods are specified to be optional. If a collection implementation doesn't implement a particular operation, it should define the corresponding method to throw* `UnsupportedOperationException`. *Such methods are marked optional operation in method specifications of the collections interfaces.*
> *– Documentation JSE11*

Interface Collection<E> (2)

Quelques méthodes, non-optionnelles:

```
interface Collection<E> {
  //Vrai si 'o' est contenu (utilise equals)
  boolean contains(Object o);
  //Vrai si tous les éléments de 'c' sont contenus
  boolean containsAll(Collection<?> c)
  //Vrai si la collection est vide
  boolean isEmpty();
  //Retourne la taille
  int size();
  /*...*/
}
```

Interface Collection<E> (3)

Quelques méthodes optionnelles:

```
interface Collection<E> {
  //Ajoute un élément
  boolean add(E e);
  //Ajoute tous les éléments de 'c'
  boolean addAll(Collection<? extends E> c);
  //Supprime l'objet 'o'. Utilise 'equals'
  boolean remove(Object o);
  //Supprime tous les élémens de 'c'.
  //Utilise 'equals'
  boolean removeAll(Collection<?> c);
  /*...*/
}
```

Interface List<E>

Collection séquentielle ordonnée (index à partir de 0):

```
interface List<E> extends Collection<E> {
  //Retourne l'élément à la position 'index'
  E get(int index)
  //Retourne la position de la première occurence
  //de 'o'. Utilise 'equals'
  int indexOf(Object o);

  //Ajoute 'element' à la position 'index'
  //Décale les éléments vers la droite
  //(Impl Optionnelle)
  void add(int index, E element);
  /*...*/
}
```

La classe ArrayList<E>

Tableau dynamique. Liste à utiliser par défaut:

```
var l = new ArrayList<String>();

l.add("AAA");
l.add("BBB");
l.add("CCC");

var s = l.get(0);           //s == "AAA"
var i = l.indexOf("CCC"); //i == 2
```

Interface `Set<E>`

Ensembles, où chaque élément ne peut être contenu qu'une seule fois.

Mêmes méthodes que `Collection<E>`

Attention
Comportement *indeterminé* si un élément mutable est modifié.

La classe `HashSet<E>`

Ensemble non-ordonné, à utiliser par défaut. Les méthodes `equals` et `hashCode` doivent être correctement définies.

```
var s = new HashSet<String>();

s.add("AAA");
s.add("BBB");
s.add("CCC");

s.contains("AAA"); // true
s.contains("CcC"); // false
```

Interface `Map<K,V>`

Super-type des dictionnaires, *alias* tableaux associatifs.

Deux types génériques: les clés K et les valeurs V

Contient également des méthode optionnelles...

Attention
Comportement *indeterminé* si une clé mutable est modifiée.

La class HashMap<K,V>

Dictionnaires recommandés. Les méthodes **equals** et **hashCode** des clés doivent être correctement définies.

```
var count = new HashMap<String,Integer>();

count.put("AAA", 2);
count.put("BBB",3);

count.replace("AAA", 5);

var i = count.get("AAA") + count.get("BBB");
//i == 8
```

Iterables: *for each*

Il existe une forme de boucle `for` utilisable pour tous les sous-type de l'interface `Iterable<T>`.

```
var words = new ArrayList<String>();
words.add( "A" );
words.add( "BB" );
words.add( "CCC" );

int sum = 0;
for( String w : words ) {
  sum += w.length();
}
```

Iterables: *Map<K,V>*

Les dictionnaires ne sont pas directement itérables mais:

- La méthode `.keySet()` retourne les clés (`Set<K>`)
- La méthode `.values()` retourne les valeurs (`Collection<V>`)

Abstraction ou classe concrète ?

Arguments et types de retour:

- Préférez les classes concrètes pour les méthodes privées;
- Préférez les classes concrètes si l'implémentation est pertinente/nécéssaire;
- Préférez les interfaces dans les méthodes publiques (découplage)

```
public double averageLength(
    Collection<String> words ) { //...

private HashSet<String> unique(
  ArrayList<String> words ) {//...
```

Héritage

Jean-Luc Falcone

22 mars 2022

Héritage en Java

Héritage: Extension de classe

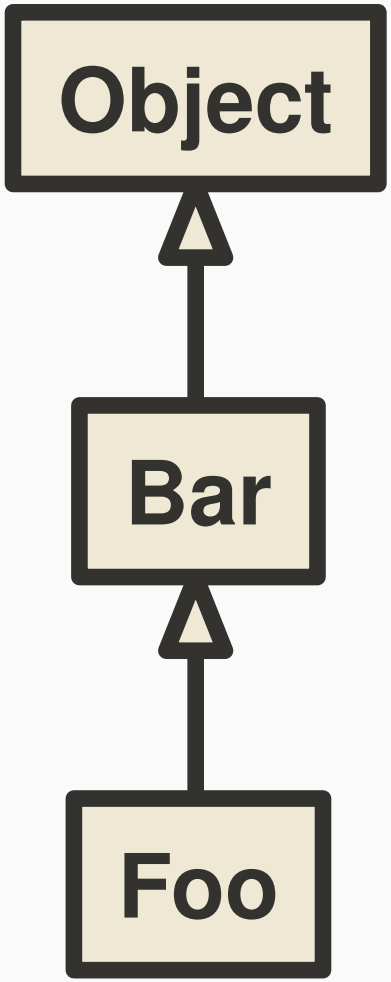
Définit par le mot clé `extends`, par exemple:

```
class Foo extends Bar { /*...*/ }
```

Deux conséquences distinctes:

1. `Foo` est un sous-type de `Bar` (relation *is-a*)
2. `Foo` récupère une partie des membres de `Bar`

Jargon et UML



On dit que:

- `Foo` est une *sous-classe* de `Bar`
- `Foo` est une *spécialisation* de `Bar`
- `Foo` est la *classe enfant*
- `Bar` est une *super-classe* de `Foo`
- `Bar` est une *généralisation* de `Foo`
- `Bar` est la *classe parent*

Héritage de membres

Une sous-classe hérite de tous les membres de sa super-classe qui ne sont pas privés.

```
class Foo {
  private int i = 0;
  int click() { i+=1; return i };
}
class Bar extends Foo {
  int doubleClick() {
    click();
    return click();
  }
  int get() { return i; } //Ne compile pas
}
```

Accesseur `protected`

L'accesseur **`protected`** est similaire à **`private`** mais accessible pour les sous-classes.

```
class Foo {
  protected int i = 0;
  int click() { i+=1; return i };
}
class Bar extends Foo {
  int doubleClick() {
    click();
    return click();
  }
  int get() { return i; } //Compile
}
```

Rédéfinition (*override*)

On peut redéfinit une méthode héritée. La référence `super` permet d'accéder aux méthodes de la super classe:

```
class Foo {
  private int i = 0;
  int click() { i+=1; return i };
}
class Bar extends Foo {
  int click() {
    System.out.println("Click!");
        super.click();
  }
}
```

Annotation `@Override` (1)

Cause une erreur de compilation si la méthode ne redéfinit rien:

```
class Bar extends Foo {
  @Override
  int click(boolean twice) {
        super.click();
        if( twice ) click();
  }
  //ERREUR DE COMPILATION:
  // error: method does not override or
  // implement a method from a supertype
}
```

Annotation `@Override` (2)

L'annotation `@Override` est optionnelle.

Son usage est:

- Recommandé dans le cadre d'une existension de classe
- Inutile dans le cas d'une implémentation d'interface

Construction

On peut/doit appeler un constructeur de la super-classe avec `super(...)`:

```
class Foo {
  private int i;
  Foo( int i ) { this.i = i; }
  //...}
class Bar extends Foo {
  private String  msg;
  Bar( int i, String msg ) {
    super(i); //Toujours en premier
    this.msg = msg,
  }
  //...
```

Méthode et classe `final` (1)

Dans le contexte de l'héritage, le mot-clé `final` a deux nouvelles significations:

- Une classe `final` ne peux pas être étendue
- Une méthode `final` ne peux pas être redéfinie

Méthode et classe `final` (2)

Une classe `final` ne peux pas être étendue

```
final class Foo { /*...*/ }


//Erreur de compilation
class Bar extends Foo  { /*...*/ }
```

Méthode et classe `final` (3)

Une méthode `final` ne peux pas être redéfinie

```
class Foo {
  final public String greetings() {
    return "Hello, from Foo";
  }
  //...
}
//Erreur de compilation
class Bar extends Foo  {
 public String greetings() {
    return "Hello, from Bar";
  }
  //...
}
```

Méthode et classe `abstract` (1)

Une classe avec le modificateur `abstract` ne peut être instanciée.

Une méthode déclarée avec le modificateur `abstract` n'est pas implémentée.

Toute classe qui contient des méthodes `abstract` doit être elle-même `abstract`.

Les sous-classes d'une classe `abtract` doivent:

- Soit implémenter les méthdes `abstract` de la super classe
- Soit être `abstact` à leur tour.

Méthode et classe `abstract` (2)

```
abstract class Point {
  protected final double x,
  protected final double y;

  Point( double x, double y) {
    this.x = x;
        this.y = y;
  }
  /*...*/
}
```

Méthode et classe `abstract` (3)

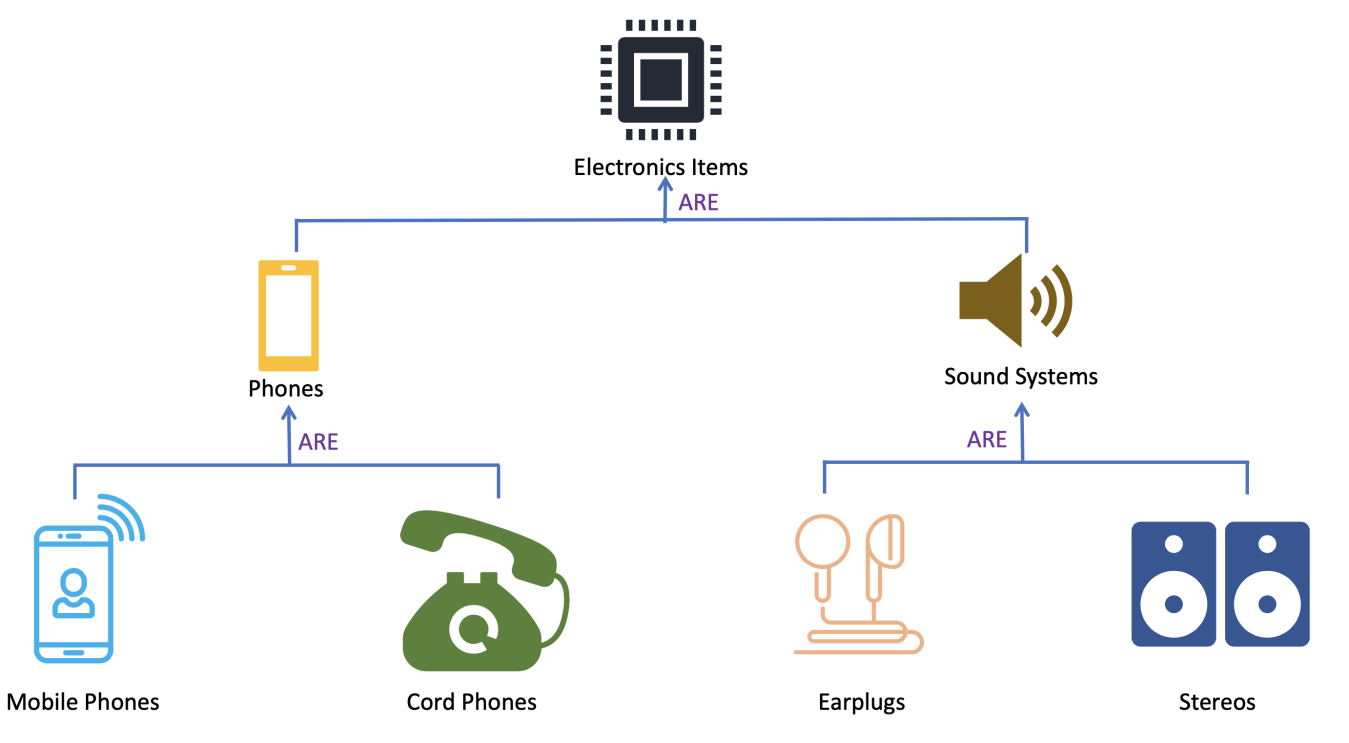
```
abstract class Point {
  /*...*/
  abstract double distance( Point that );
  Point closest( List<Point> ps ) {
    var chosen = ps.get(0);
    var best = distance(chosen);
    for( int i=1; i<ps.size(); i++ ) {
      var p = ps.get(i);
      var d = distance(p)
      if( d <  best ) {
            chosen = p; best = d;
      }
    }
        return chosen;
  }
```

Méthode et classe `abstract` (4)

```
class ManhattanPoint extends Point {
  //...
  double distance( Point that ) {
    return abs(x-that.x)+abs(y-that.y);
  }
}

class EuclidPoint extends Point {
  //...
  double distance( Point that ) {
    return Math.sqrt( Math.pow(x-that.x, 2) +
    Math.pow(y-that.y, 2) );
  }
}
```

Héritage: sous-typage

Exemple classique



Tiré de: Towards Datascience

Cercle-Ellipse (1)

Équation d'une ellipse, centrée sur l'origine:

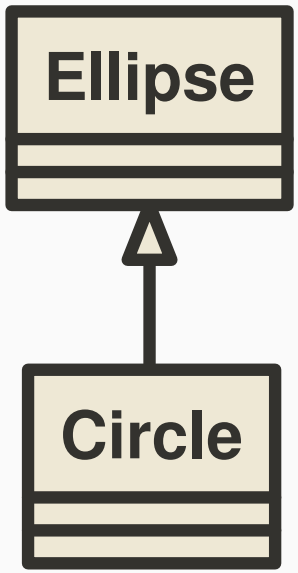
$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

Pour le cas particulier $a = b = r$:

$$\frac{x^2}{r^2} + \frac{y^2}{r^2} = 1 \quad \Leftrightarrow \quad x^2 + y^2 = r^2$$

Soit l'équation d'un cercle. centré sur l'origine.

Cercle-Ellipse (2)



En géométrie, les cercles sont des cas particuliers d'ellipse (relation *is-a*).

Il est donc naturel de penser qu'une classe `Circle` devrait être un sous type de la classe `Ellipse`.

Classe Ellipse

```
class Ellipse {
  private double width;
  private double height;
  Ellipse( double w, double h ) {
    width = w; height = h;
  }
  void stretchX( double k ) {
    width *= k;
  }
  void stretchY( double k ) {
    height *= k;
  }
  double area() {
    return Math.PI * width/2 * height/2;
  }
```

Classe Cercle

```
class Circle extends Ellipse {
  Circle( double radius ) {
    super( radius*2, radius*2 );
  }
  double getRadius() {
    return width/2; //or height/2
  }
  double circumference() {
    return 2*getRadius()*Math.PI;
  }
}
```

Problème

```
Circle c = new Circle(1.0);
System.out.println(c.circumference()):

c.stretchX(0.5);
c.stretchY(2.0);
System.out.println(c.getRadius());    // ???
System.out.println(c.circumference());// ???
```

Solution 1: Exceptions (1)

```
class Circle extends Ellipse {
  Circle( double radius ) {
    super( radius*2, radius*2 );
  }
  @Override
  void stretchX( double k ) {
    throw new NoSuchOperationException(
          "Cannot stretch a circle");
  }
  @Override
  void stretchY( double k ) {
    throw new NoSuchOperationException(
          "Cannot stretch a circle");
  }
```

Solution 1: Exceptions (2)

Avantage:

- L'invariant du cercle est respecté

Désavantage:

- Risque supplémentaire d'exceptions à l'exécution

Solution 2: Exceptions mais autrement (1)

```
class Circle extends Ellipse {
  Circle( double radius ) {
    super( radius*2, radius*2 );
  }
  double getRadius() {
    if( width != height )
          throw new IllegalStateException(
            "Undefined radius");
    return width/2; //or height/2
  }
  double circumference() {
    return 2*getRadius()*Math.PI;
  }
}
```

Solution 2: Exceptions mais autrement (2)

Avantage:

- Le cercle peut être temporairement invalide sans que cela ne pose problème.

Désavantage:

- Risque supplémentaire d'exceptions à l'exécution
- La survenue de l'exception peut être loin de la cause du problème.

Solution 2: Exceptions mais autrement (3)

```
Circle c = new Circle(1.0);
System.out.println(c.circumference()):

c.stretchX(0.5); //Circularité brisée
c.stretchY(0.5); //Circularité restorée

System.out.println(c.circumference());
```

Solution 3: Changer le comportement de `stretchX/Y` (1)

```
class Ellipse {
  protected double width;
  protected double height;
  /*...*/
}
class Circle extends Ellipse {
  //...
  @Override
  void stretchX( double k ) {
    width*=k; height*=k;
  }
  void stretchY( double k ) {
    width*=k; height*=k;
  }
}
```

Solution 3: Changer le comportement de `stretchX/Y` (2)

Avantages:

- Pas d'exception
- Circularité toujours respecté

Désavantages:

- `stretchX` agit sur deux dimensions, alors que son nom indique qu'il ne change d'une dimension (introduction de *surprise*).
- Moins d'encapsulation (`protected`)

Solution 3: Changer le comportement de `stretchX/Y` (3)

Selon la géométrie ces deux fonctions devrait avoir le même comportement:

```
static double wideArea0( Ellipse e ) {
  e.stretchX(2);
  double a = e.area();
  e.stretchX(0.5);
  return a;
}

static double wideArea1( Ellipse e ) {
  return 2 * e.area();
}
```

Solution 4: Pas de classe cercle (1)

```
class Ellipse {
  //...
  static Ellipse circular( double radius ) {
    return new Ellipse( radius*2, radius*2);
  }
  boolean isCircular() {
    return width == height;
  }
}
//Exemple d'utilisation
var c = Ellipse.circular( 2.0 );
if( c.isCircular() ) {
  var cicumference = //...
}
```

Solution 4: Pas de classe cercle (2)

Avantages:

- Plus simple
- Pas d'exceptions
- Pas de surprises

Désavantages:

- Plus de méthodes spécialisées pour les cercles
- Indistinguable à la compilation

Solution 5: Immutabilité (1)

```
class Ellipse {
  private final double width;
  private final double height;
  Ellipse( double w, double h ) {
    width = w; height = h;
  }
  Ellipse stretchX( double k ) {
    return new Ellipse( width*k, height );
  }
  Ellipse stretchY( double k ) {
    return new Ellipse( width, height*k );
  }
  //...
}
```

Solution 5: Immutabilité (2)

```
class Circle extends Ellipse {
  Circle( double radius ) {
    super( radius*2, radius*2 );
  }
  double getRadius() {
    return width/2; //or height/2
  }
  double circumference() {
    return 2*getRadius()*Math.PI;
  }
  Circle scale( double k ) {
    return new Circle( getRadius()*k );
  }
}
```

Solution 5: Immutabilité (3)

```
Circle c = new Circle(2.0);

Ellipse e = c.stretchX(0.5);
e.area() == 0.5*c.area();
e.circumference(); //Erreur compilatation
Circle c1 = c.stretchX(0.5); //Erreur compilation

Circle c2 = c.scale(2.0);
c2.circumferences() == 2*c.circumference();
```

Solution 5: Immutabilité (4)

Selon la géométrie ces deux fonctions devrait avoir le même comportement:

```
static double wideArea0( Ellipse e ) {
  return e.sretchX(2).area();
}
```

```
static double wideArea1( Ellispe e ) {
  return 2 * e.area();
}
```

Solution 5: Immutabilité (5)

Avantages:

- Invariants respectés
- Erreur de compilation au lieu d'exceptions

Désavantages:

- Immutable

Solution 6: Pas d'héritage (1)

```
interface Shape {
  double area();
  void scale( double k );
}
interface Stretchable {
  void stretchX( double k );
  void stretchY( double k );
}

class Ellipse implements Shape, Stretchable {
  //...}
class Circle implements Shape {
  //...}
```

Solution 6: Pas d'héritage (2)

Avantage:

- Invariants découplés
- Erreur de compilation au lieu d'exceptions
- Généralisation à d'autres formes facile

Désavantage:

- Pas de de code en commun: plus de lignes de code

Classement subjectif des solutions

Mes solutions préférées dans l'ordre décroissant:

1. Pas d'héritage (si possible avec immutabilité)
2. Immutabilité
3. Pas de cercle
4. Autres solutions *ex aequo*

Conséquences

Une relation *is-a* entre les deux classes est nécéssaire mais pas suffisante pour que l'utilisation d'héritage soit cohérente et sensée.

Principe de substitution de Liskov

Si S est une sous-classe de T, alors on doit pouvoir utiliser une instance de S chaque fois qu'une instance de T est attendue, *sans causer de problèmes*.

Programmation par contrat (1)

Pour chaque opération (appel de méthode), on peut définir des contraintes:

- Pré-conditions
- Post-conditions
- Invariants

Programmation par contrat (2)

Pré-conditions: Contraintes qui doivent être réalisées avant d'effectuer l'opération. Par exemple:

- Type des arguments
- Valeur des arguments
- Etat de l'objet courant

Programmation par contrat (3)

Post-conditions: Contraintes qui doivent être réaliés après avoir effectué l'opération:

- Type de retour
- Valeur de retour
- Etat de l'objet courant
- Etat des arguments

Programmation par contrat (4)

Invariants: Contraintes sont toujours réalisées. A la fois des pré- et des post-conditions.

- Valeur de l'objet courant

Programmation par contrat (5)

Exemple dans le language *Eiffel*:

```
withdraw (sum: INTEGER) is
  -- Withdraw sum from the account.
  require
    sum >= 0
    sum <= balance - minimum_balance
  do
    add (-sum)
  ensure
    balance = old balance - sum
  end
```

Principe de substitution de Liskov

Pour toute sous-classe:

- Les pré-conditions des méthodes de la super classe ne peuvent être durcies
- Les post-conditions des méthodes de la super classe ne peuevnt être assouplies.
- Les invariants de la super classe doivent être préservés tels quels.

Définition de Barbara Liskov & Jeanette Wing (1994):

> *Subtype Requirement*: *Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.*

Cercle-Ellipse (3)

En géométrie, les cercles sont des cas particuliers d'ellipse avec:

- Invariant: Les deux axes sont égaux (rayon)

Or la méthode `stretchX` de l'ellispe est définie avec:

- Pré-condition: Toujours applicable
- Post-condition: Seul l'axe horizontal est affecté

Ce qui est en contradiction avec l'invariant du cercle. Donc le Cercle ne peut pas sous classer l'Ellipse.

Conclusion

Contrairement à l'intuiation, la relation de sous-typage (*is-a*) n'implique pas une compatiblité de structure, mais de comportement.

Le comportement dépend du programme, donc la question de savoir si `Foo` est une sous-classe de `Bar` ne peut être tranchée sans avoir déterminé le comportement désiré.

Héritage: réutilisation de code

Don't Repeat Yourself (DRY principle)

Programmer est compliqué, débugger l'est plus encore.

Moins le code est répété, moins il y aura de code à débugger.

Parmi, les approches permettant de réduire la quantité de code, on peut utiliser l'héritage.

Mais le mécanisme est souvent plus complexe qu'attendu...

Exemple jouet (1)

```
class Bar {
  public int i() {
    return 2;
  }
}
```

```
class Foo extends Bar {
  public int i() {
    return
      super.i()+1;
  }
}
```

Qu'affiche le code suivant ?

```
Foo f = new Foo();
System.out.println(  f.i() );
```

Exemple jouet (2)

```java
class Bar {
  public int i() {
    return 2;
  }
}
```

```java
class Foo extends Bar {
  public int i() {
    return
      super.i()+1;
  }
}
```

Qu'affiche le code suivant ?

```java
Bar b = new Foo();
System.out.println(  b.i() );
```

Dynamic dispatch

Le langage Java utilise un mécanisme de *dynamic dispatch* pour choisir quelle méthode appeler.

Le choix est fait à l'exécution (donc dynamique).

C'est toujours la méthode définie par l'instance qui est appelée, indépendamment de la référence.

Exemple jouet (3)

```
class Bar {
  public int i() {
    return 2;
  }
  public int j() {
    return 3*i();
  }
}
```

```
class Foo extends Bar {
  public int i() {
    return
      super.i()+1;
  }
  public int j() {
    return
      super.j()*10;
  }
}
```

Qu'affiche le code suivant ?

```
Foo f = new Foo();
System.out.println(  f.j() );
```

Instrumentation de classe existente

On aimerait créer une variante de la class `Stack` vu au cours précédent pour étudier le nombre total d'objets stockés dans une pile, indépendemment de sa taille:

- fonctionne comme `Stack`
- garde un nombre d'éléments totaux
- incrémente ce nombre chaque fois qu'un élément est ajouté à la pile

Exemple inspiré de *Effective Java v3, Joshua Bloch, 2018, Item 18.*

Classe de base: Stack

```
public class Stack<T> {
  //...
  public boolean isEmpty() { //...
  public boolean size() { //...
  public void push( T t ) { //...
  public void pushAll( Stack<T> ts ) { //...
  public T pop() //...
}
```

Classe dérivée: InstrumentedStack

```
public class InstrumentedStack<T>
extends Stack<T> {
  private int total;
  public int getTotal() { return total; }
  public void push( T t ) {
    total += 1;
    super.push(t);
  }
  public void pushAll( Stack<T> ts ) {
    total += ts.size();
    super.pushAll(ts);
  }
}
```

Exemple d'exécution (1)

```
var stack = new InstrumentedStack<Integer>();
stack.push(2);
stack.push(3);
stack.pop();
stack.pushAll(stack);
stack.pop();

System.out.println( stack.size() );
System.out.println( stack.getTotal() );
```

Exemple d'exécution (2)

```
var stack = new InstrumentedStack<Integer>();
stack.push(2);
stack.push(3);
stack.pop();
stack.pushAll(stack);
stack.pop();

System.out.println( stack.size() ); //=>  1
System.out.println( stack.getTotal() ); //=> 6
```

Explication: implémentation de `pushAll`

```
public class Stack<T> {
  public void push( T t ) { //...
  public void pushAll( Stack<T> ts ) {
    for( int i=0; i<ts.next; i++ ) {
      push( data[i] );
    }
  }
}
```

Correction de la classe dérivée

```
public class InstrumentedStack<T>
extends Stack<T> {
  private int total;
  public int getTotal() { return total; }
  public void push( T t ) {
    total += 1;
    super.push(t);
  }
  /*public void pushAll( Stack<T> ts ) {
    total += ts.size();
    super.pushAll(ts);
  }*/
}
```

Exemple d'exécution (1)

```
var stack = new InstrumentedStack<Integer>();
stack.push(2);
stack.push(3);
stack.pop();
stack.pushAll(stack);
stack.pop();

System.out.println( stack.size() );  //1
System.out.println( stack.getTotal() ); //4
```

Exemple d'exéction (2)

Un an, et plusieurs mises à jour plus tard:

```
var stack = new InstrumentedStack<Integer>();
stack.push(2);
stack.push(3);
stack.pop();
stack.pushAll(stack);
stack.pop();

System.out.println( stack.size() );  //1
System.out.println( stack.getTotal() ); //2
```

Meilleur approche: Aggrégation

On peut souvent remplacer l'héritage par de l'aggrégation (ou composition).

Ceci implique de:

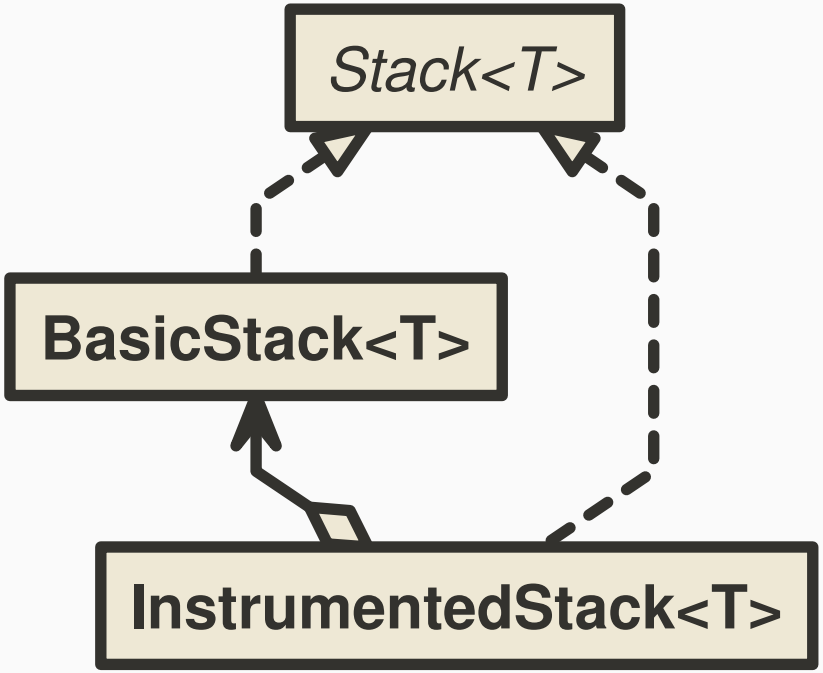
- Avoir une interface commune pour les différentes implémentations (polymorphisme).
- Encapsulter une instance de la classe dont on dérive le comportement aggrégation (ou composition).

Exemple jouet

```
interface IJ{
  int i();
  int j();
}

class Bar implements IJ{
  public int i() {
    return 2;
  }
  public int j() {
    return 3*i();
  }
}
```

```
class Foo implements IJ{
  final var bar =
    new Bar();
  public int i() {
    return
      bar.i()+1;
  }
  public int j() {
    return
      bar.j()*10;
  }
}
```

Aggrégation: `InstrumentedStack` (1)



Super-type: interface

```
interface Stack<T> {
  public boolean isEmpty();
  public boolean size();
  public void push( T t );
  public void pushAll( Stack<T> ts );
  public T pop();
}
```

Sous-types: classes concrètes

```
class BasicStack<T> implements Stack<T> {
  /*...*/
}

class InstrumentedStack<T>
implements Stack<T> {
  private int total;
  public int getTotal() { return total; }
  /*...*/
}
```

Aggrégation

```
class InstrumentedStack<T>
implements Stack<T> {
  /*...*/
  private var stack =
    new BasicStack<T>();
  public void push( T t ) {
    total += 1;
    stack.push(t);
  }
  public void pushAll( Stack<T> ts ) {
    total += ts.size();
    stack.pushAll(ts);
  }
  /*...*/
}
```

Délégation

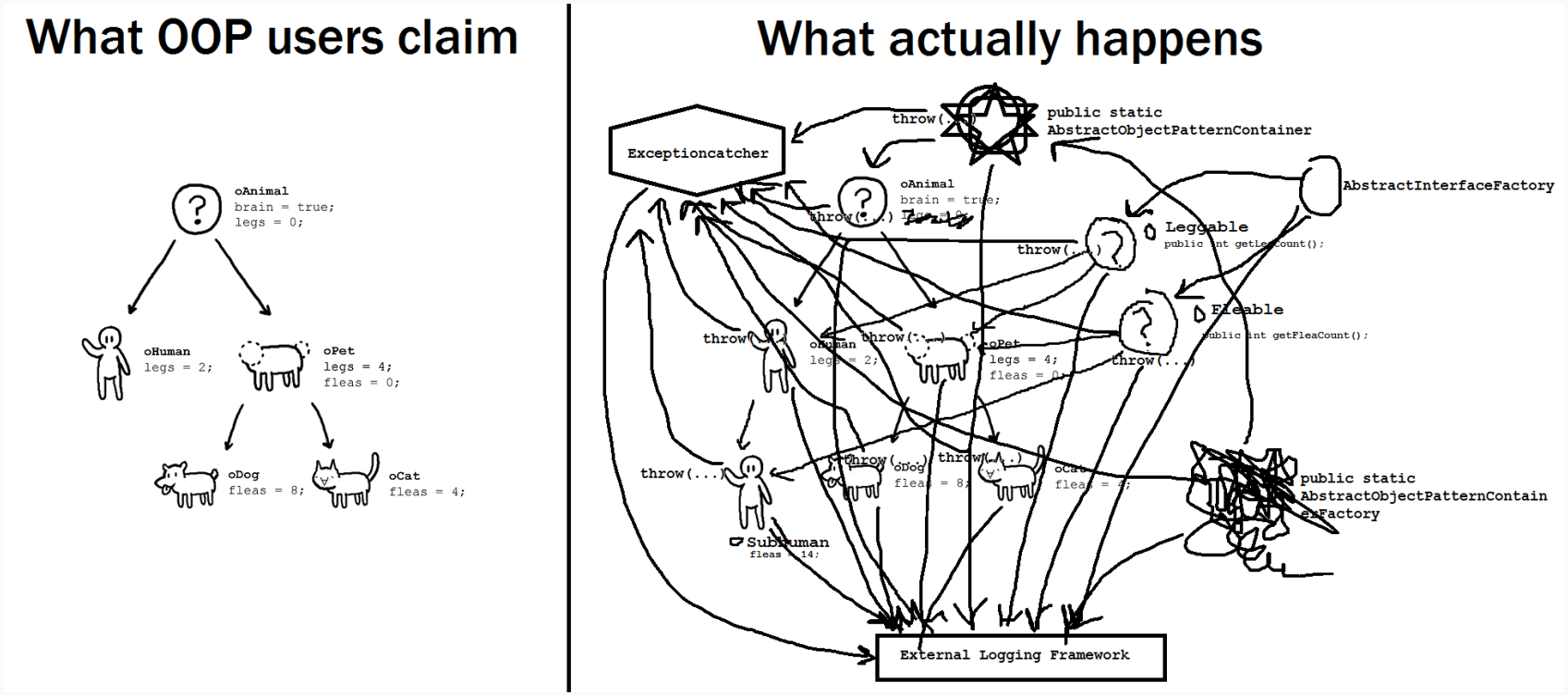
```
class InstrumentedStack<T>
implements Stack<T> {
  /*...*/
  public boolean isEmpty() {
    return stack.isEmpty();
  }
  public boolean size() {
    return stack.size();
  }
  public T pop() {
    return stack.pop();
  }
}
```

Conclusion (1)

Pour réutiliser du code: préférez l'aggrégation à l'héritage !

> ***So What About Inheritance?***
> *At Facebook, we use React in thousands of components, and we haven't found any use cases where we would recommend creating component inheritance hierarchies.*
> *– Tiré de la Documentation de ReactJS*

Conclusion (2)



Auteur /u/yotamN/.

Quelques Langages Orientés-Objets

Jean-Luc Falcone

29 mars 2022

Langages Orientés Objets

Concepts de POO

Jusqu'ici nous avons vu:

- Classes
- Interfaces
- Encapsulation
- Polymorphisme
- Héritage

Comparaison (1)

	Elements de base POO
Java	`class interface (enum)`
Swift	`struct class enum protocol extension`
Python	`class`
Javascript	`object (class)`
Rust	`struct enum trait`
C++	`struct class`
Lua	`table`
Scala	`class object trait`

Comparaison

	Interfaces	Encapsulation	Polymorp.	Héritage
Java	+	+	+	+
Swift	+	+	+	+
Python	(-)	-	(+)	+
Javascript	-	(+)	(+)	+
Rust	+	+	+	-
C++	(-)	+	+	+
Lua	-	(+)	(+)	+
Scala	+	+	+	+

Swift

Description

- Usage générique:
  - Bas niveau: prog. système
  - Haut niveau: app. desktop et mobile
- Typage statique et fort
- Multi paradigme:
  - Procédural
  - Fonctionel
  - Orienté-Objet
- Multiplateforme: MacOS, GNU/Linux, Windows (mais surtout MacOS)

Historique et motivation

- Créé par des ingénieurs Apple dès 2010
- But: remplacer Objective-C
- Inspirations: Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, et autres.
- Première version: 2014 sur MacOS (support Linux dès 2016, Windows dès 2020).

Déclarations d'objets: `struct`

```
struct Point {
  let x: Double
  let y: Double
  func dist(with:Point)->Double {
    let d2 = pow(with.x-x,2) + pow(with.y-y,2)
    return sqrt(d2)
  }
}

let p = Point(x:2,y:3)
let q = Point(x:0,y:4)
let d = p.dist(with:q) //2.23606797749979
```

Déclaration d'objets: `class`

```
class Point {
  let x: Double
  let y: Double
  init(x:Double,y:Double) {
    self.x = x; self.y = y
  }
  func dist(with:Point)->Double {
    let d2 = pow(with.x-x,2) + pow(with.y-y,2)
    return sqrt(d2)
  }
}

let p = Point(x:2,y:3)
let q = Point(x:0,y:4)
let d = p.dist(with:q) //2.23606797749979
```

Encapsulation: contrôle d'accès

Classes, structures et leurs membres:

```
class Bar {
  //Accessible partout
  public var a: Int

  //Accessible dans le même module (défaut)
  internal var b: Int

  //Accessible depuis le même fichier
  fileprivate var c: Int

  //Accesseible depuis la même classe
  private var d: Int
}
```

`struct` vs `class`: sémantique (1)

```
struct BoxS {
  var i: Int
  var b: Bool
}

class BoxC {
  var i: Int
  var b: Bool
  init(i ii:Int, b bb: Bool) {
    i = ii; b = bb;
  }
}
```

`struct` vs `class`: sémantique (2)

```
let s1 = BoxS(i:2,b:false)
let c1 = BoxC(i:2,b:false)


c1.i = 100 //Change la valeur de 'i'


s1.i = 100 //Erreur de compilation:
           //error: cannot assign to property:
           // 's1' is a 'let' constant
```

`struct` vs `class`: sémantique (3)

```
var s1 = BoxS(i:2,b:false)
var c1 = BoxC(i:2,b:false)

var s2 = s1
var c2 = c1

s1.i = 100
c1.i = 100

print(s2.i) //Affiche 2
print(c2.i) //Affiche 100
```

`struct` vs `class`: sémantique (4)

Les instances de **`struct`** ont une sémantique basée sur la valeur, comme les primitives en Java.

Les instances de **`class`** ont une sémantique basée sur la référence, comme les objects en java.

struct vs class: mutation (1)

```
class Rectangle {
  private var width: Double
  private var height: Double
  init(w:Double,h:Double) {
    width = w; height = h
  }
  func area()->Double {
    return width*height;
  }
  func scale(factor:Double)->Void {
    width *= factor
    height *= factor
  }
}
```

struct vs class: mutation (2)

```
struct Rectangle {
  private var width: Double
  private var height: Double
  func area()->Double {
    return width*height;
  }
  //Ne compile pas !
  func scale(factor:Double)->Void {
    width *= factor
    height *= factor
  }
}
```

struct vs class: mutation (3)

```
struct Rectangle {
  private var width: Double
  private var height: Double
  func area()->Double {
    return width*height;
  }
  //Ne compile pas !
  func scale(factor:Double)->Void {
    self.width *= factor
    self.height *= factor
  }
}
```

struct vs class: mutation (4)

```
struct Rectangle {
  private var width: Double
  private var height: Double
  func area()->Double {
    return width*height;
  }
  mutating func scale(factor:Double)->Void {
    self.width *= factor
    self.height *= factor
  }
}
```

Protocoles

- Similaires aux interfaces Java
- Peuvent être implémentés par des `class` ou des `structs`
- Dans la jargon *Swift*, on dit qu'une classe ou une structure *se conforme à* un protocol.

Protocoles: `struct`

```
protocol Shape {
  func area()->Double
  //Nécéssaire qu'une struct puisse se conformer
  mutating func scale(factor:Double)->Void
}
struct Rectangle: Shape {
  private var width: Double
  private var height: Double
  func area()->Double {
    return width*height;
  }
  mutating func scale(factor:Double)->Void {
    width *= factor; height *= factor;
  }
}
```

Protocoles: `class`

```
class Rectangle: Shape {
  private var width: Double
  private var height: Double
  init(w:Double,h:Double) {
    width = w; height = h
  }
  func area()->Double {
    return width*height;
  }
  func scale(factor:Double)->Void {
    width *= factor; height *= factor;
  }
}
```

Protocoles, au delà des interfaces Java

Les protocoles *Swift* peuvent déclarer des propriétés, des constructeurs, et des membres statiques:

```
protocol SomeProtocol {
  //Propriétés
  var mustBeSettable: Int { get set }
  var doesNotNeedToBeSettable: Int { get }

  //Membres statiques
  static var someTypeProperty: Int { get set }
  static func someTypeMethod()

  //Constructeurs
  init(someParameter: Int)
}
```

Héritage

Seulement pour les `class`, pas pour les `struct`:

```
class RotatedRectangle: Rectangle {
  var angle: Double
  override init(w:Double,h:Double,a:Double) {
    angle = a
    super.init(w:w,h:h)
  }
  func rotate(a:Double)->void {
    angle += a
  }
}
```

Beaucoup de sujets non abordés

- Propriétés
- *Downcasting*
- Extensions
- Enumérations
- Déinitialisation

Python

Description

- Usage générique:
  - Langage de script
  - Plutôt haut niveau (*glue code*)
- Typage dynamique et fort
- Multi paradigme:
  - Procédural
  - Fonctionel
  - Orienté-Objet
- Multiplateforme et très répandu.

Historique

- Créé dès 1989 par Guido Van Rossum
- Langage académique à la base (CWI, Amsterdam)
- But: second langages pour développeurs de C et C++
- Inspirations: ABC (langage pour l'enseignement au CWI)
- Extrêmement populaire
- Depuis Python 3.5: *Gradual Typing* et typeur statique auxilliaire (`Mypy`)

Système de types

- Que des objets ! Mais pas de super-type commun.

```
>>> type(2)
<class 'int'>

>>> type("Hello")
<class 'str'>

>>> type(["Hello", 2])
<class 'list'>

>>> type(type(2))
<class 'type'>
```

Polymorphisme structural: *Duck typing* (1)

> *« If it walks like a duck and it quacks like a duck, then it must be a duck »*

Ce qui compte ce n'est pas le type, mais la présence des champs/méthodes.

Polymorphisme structural: *Duck typing* (2)

```
def twice( foo ):
  bar = foo.pop()
  foo.append(bar)
  foo.append(bar)
```

Quel doit être le type de `foo` ? Que fait cette méthode ?

Polymorphisme structural: *Duck typing* (2)

Quel doit être le type de `foo` ?
N'importe quel type qui a une méthode **`pop`** sans arguments et une méthode **`append`** qui prend un argument

Que fait cette méthode ?
Cela dépend du type passé en argument

Déclaration de classe

```
class Point:
  def __init__(self,x,y):
    self.x = x
    self.y = y
  def dist( self, that ):
    d2 = (that.x-self.x)**2 + (that.y-self.y)**2
    return sqrt(d2)

>>> p = Point(2,3)
>>> q = Point(0,4)
>>> p.distance(q)
2.23606797749979
```

Contrôle d'accès

Pas de vrais membres privés, mais deux conventions assez bien respectées:

- Les membres qui commencent par un *underscore* ne devraient pas être utilisés (~ privés)
- Les membres qui commencent par deux *underscores* ne devraient vraiment pas être utilisés (~ plus privés)

Pas de déclaration formelle des champs

```
>>> p = Point(2,3)
>>> p.x
2
>>> p.y
3
>>> p.prenom
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no
                                attribute 'prenom'
>>> p.prenom = "Peter"
>>> p.prenom
'Peter'
```

Liste des attributs

On peut lister les attributs d'une classe:

```
>>> dir(Point) ['__class__', '__delattr__', '__dict
'__eq__', '__format__', '__ge__', '__getattribute__
'__init__', '__init_subclass__', '__le__', '__lt__'
..., 'dist']
```

Ou d'une instance:

```
>>> dir(p) ['__class__', '__delattr__', '__dict__',
'__eq__', '__format__', '__ge__', '__getattribute__
'__init__', '__init_subclass__', '__le__', '__lt__'
..., 'dist', 'prenom', 'x', 'y']
```

Pas d'interfaces

Le typage dynamique rend d'éventuelles interfaces inutiles...

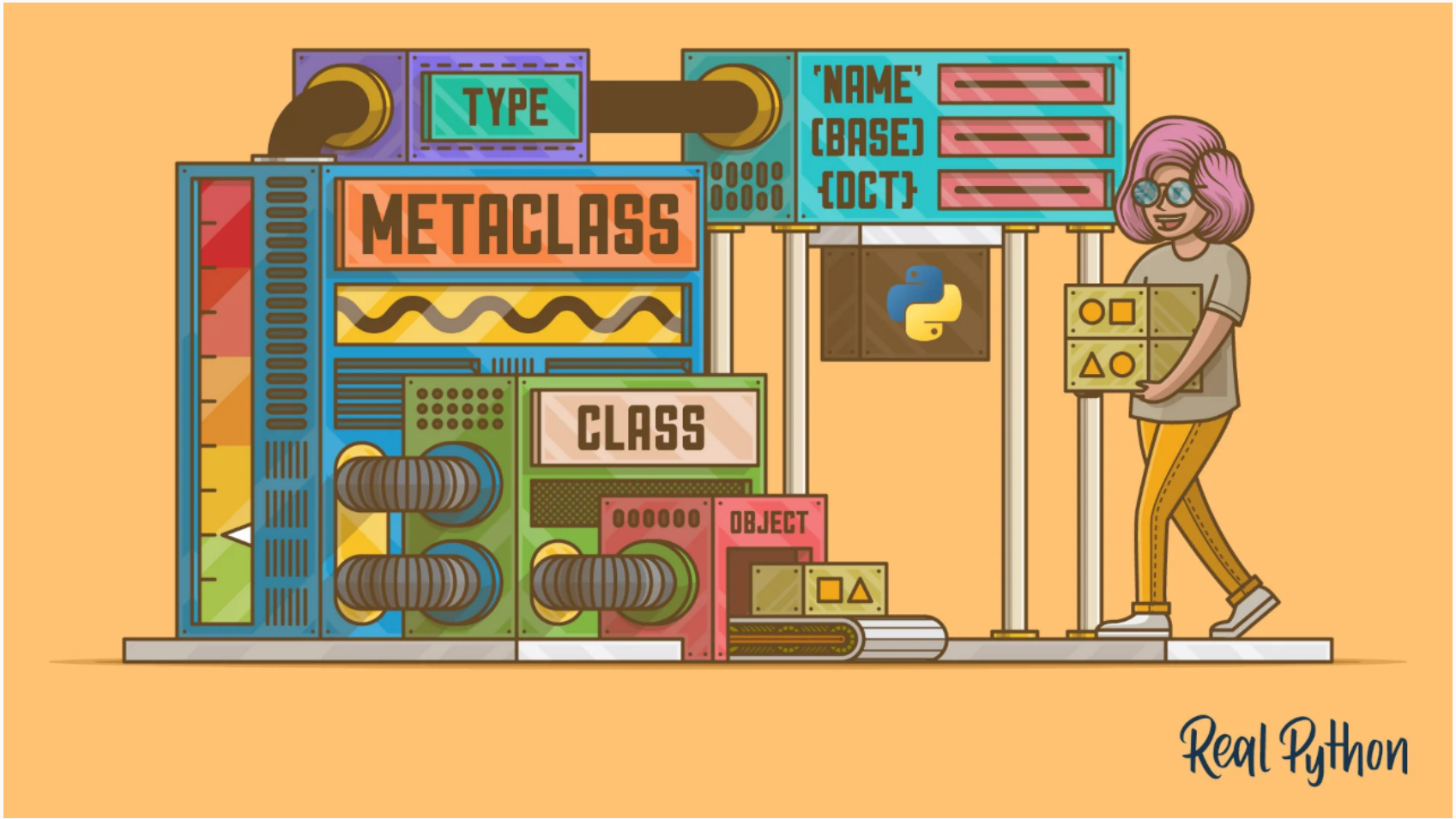
Si nécéssaire on peut tester la présence d'une méthode:

```
def sample( number, dest ):
  if "append" in dir(dest) and
                              callable(dest.append):
    f = dest.append
  elif "add" in dir(dest) and callable(dest.add):
    f = dest.add
  else:
    raise Exception("No append or add")
  for i in range(number):
    f( random.random() )
```

Classes et métaclasses

```
>>> Point.dist
<function Point.dist at 0x7f2cc60a89d0>
>>> Point.score = "0 partout, match nul"

>>> def foo(self):
      print("Hello", self.prenom)
>>> Point.greetings = foo

>>> p.greetings
<bound method foo of <__main__.Point object at 0x7f
>>> p.greetings()
Hello Peter
```

Métaclasses



Tiré de:

`https://realpython.com/python-metaclasses/`

Métaprogrammation

Largement en dehors du cours. Quelques liens pour les curieux:

- https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Metaprogramming.html
- https://stackabuse.com/python-metaclasses-and-metaprogramming/
- https://realpython.com/python-interface/
- https://realpython.com/python-metaclasses/

Python typé: un exemple en passant

```
@dataclass(frozen=True)
class Message:
  key: str
  mail: mailbox.MaildirMessage
  def subject(self)->str:
    return decode(self.mail['subject'])
  def sender(self)->str:
    return decode(
      email.utils
           .parseaddr( self.mail['From'] )[1]
   ).lower()
```

Javascript

Description

- Usage générique:
  - Client web
  - Lanagage embarqué
  - Serveur ou script (`node.js`)
- Typage dynamique et plutôt failble
- Multi paradigmes:
  - Procédural
  - Fonctionel
  - Orienté-Objet
- Multiplateforme et très répandu

Historique et motivation

- Créé en 10 jours par *Brendan Eich* pour le compte de Netscape en 1994
- But: petit langage sympa pour le web
- Inspirations: scheme et C
- Extrêmement populaire
- Première version standardisée 1997

Idée initiale: Scheme sur browser

```
(define fac
  (lambda (n)
    (if (= n 0)
      1
      (* n (fac (- n 1))))))

;;; Usage example
(fac 5)
```

Syntaxe adoptée: *curly-braces*

```
function fac(n) {
  if( n == 0 ) {
    return 1;
   } else {
    return n * fac( n - 1 );
  }
}

// Usage example
fac(5);
```

Types

Il n'existe que 9 types:

- `undefined`
- `Boolean`
- `Number`
- `String`
- `BigInt`
- `Symbol`
- `Object`
- `Function`

Objets

Le type `object` représente les objets. Il s'agit de tableaux associatifs.

```
> let a = { name: "Alice", age: 34 };
> a
{ name: 'Alice', age: 34 }
> a.name
'Alice'
> a.age = 35
35
> a
{ name: 'Alice', age: 35 }
```

Objets (2)

```
> a["name"] = "Alysse"
'Alysse'
> a
{ name: 'Alysse', age: 35 }
> a.address
undefined
> Object.keys(a)
[ 'name', 'age' ]
```

Données complexes

```
let a = {
  name: {
     firstnames: [ 'Alice', 'Pleasance' ],
     surname: 'Hargreaves',
     born: 'Lidell'
  },
  birthday: { date: "1852-05-04", place: "Oxford" }
  address: undefined
};
```

Fonctions membre des objets

Comme une fonction est une valeur comme les autres, on peut ajouter des fonctions à un objet:

```
> let Mathematique = {
  abs: function(x) {
    if( x < 0 ) {
      return -x;
    } else {
      return x;
    }
  },
  racineCarrée: Math.sqrt
};
```

Fonctions membres des objets (usage)

```
> Mathematique.racineCarrée( 100 );
10
> Mathematique.abs( -2 );
2
> Mathematique.abs
[Function]
```

Fonctions et construction d'objet

Un objet étant une valeur comme une autre, on peut les retourner depuis une fonction:

```
> function point2D( x, y ) {
  return {
    dim: 2,
    x: x,
    y: y
  }
}
> point2D( 2, -3 )
{ dim: 2, x: 2, y: -3 }
```

Orienté-Objet en JavaScript

- Le langage JavaScript est orienté-objet, mais sans classes !

Orienté-Objet en JavaScript

- Le langage JavaScript est orienté-objet, mais sans classes !
- Il existe des mots-clé `this` et `new` mais la sémantique est différente de celle de Java, C++, Scala, C#, etc.

Orienté-Objet en JavaScript

- Le langage JavaScript est orienté-objet, mais sans classes !
- Il existe des mots-clé `this` et `new` mais la sémantique est différente de celle de Java, C++, Scala, C#, etc.
- La version ES6 introduit une nouvelle syntaxe plus habituelle avec des classes. Elle est cependant déconseillée.

`https://www.toptal.com/javascript/es6-class-chaos-keeps-js-developer-up`

Le mot clé `this`

En Java, `this` représente l'instance à partir de laquelle la méthode est définie.

En JS, `this` représente l'objet dans lequel la méthode est appelée.

Le mot clé `this` (1)

```
let counter = {
  value: 0,
  incr: function() {
    this.value += 1;
  }
};
> counter
{ value: 0, incr: [Function] }
> counter.incr(); counter.incr(); counter.incr();
> counter
{ value: 3, incr: [Function] }
```

Le mot clé `this` (2)

```
function f() {
  this.x += 1;
}
let a = { x: 2, incr: f  };
let b = { x: "hello", incr: f };
> a.incr();
> b.incr();
> a
{ x: 3, incr: [Function: f] }
> b
{ x: 'hello1', incr: [Function: f] }
```

Mot clé new

Le mot clé **new** permet d'interpréter une fonction comme un constructeur:

```
function Counter( x ) {
  this.value = x;
  this.incr = function() {
    this.value += 1;
  }
}
> let c = new Counter(2);
> c
Counter { value: 2, incr: [Function] }
> c.incr();
> c
Counter { value: 3, incr: [Function] }
```

Prototype: fonctions

Chaque objet a un prototype:

```
function Counter(x) {
  this.value = x;
}
Counter.prototype.incr = function() {
    this.value += 1;
}
> let c = new Counter(0);
> c
Counter { value: 0 }
> c.incr()
undefined
> c
Counter { value: 1 }
```

Prototype: juste un objet

Le prototype est juste un objet qui contient les propriétés et méthode assignées.

```
> Counter.prototype
Counter { incr: [Function] }
```

Prototype: héritage (1)

On peut utiliser le chaînage des prototypes pour hériter d'un autre contructeur.

```
function Counter() {}
Counter.prototype.incr =
  function() { this.value += 1; }
Counter.prototype.value = 0;

function CounterDeluxe() { }
CounterDeluxe.prototype = new Counter();
CounterDeluxe.prototype.reset =
  function() { this.value = 0 };
```

Prototype: héritage (2)

```
> let c = new Counter();
> c.incr(); c.incr();
> c
Counter { value: 2 }
> let cd = new CounterDeluxe();
> cd.value
0
> cd.incr();
> cd
Counter { value: 1 }
```

Méthode alternative

On peut se passer facilement de `new` et de `this` en utilisant une autre approche:

```
function Counter( x ) {
  let value = x;
  return {
    incr: function() { value += 1; },
    get: function()  { return value; },
    reset: function() { value = 0; }
  }
}
```

Méthode alternative (démo)

```
> let c = Counter(2);
> c
{ incr: [Function], get: [Function],
                          reset: [Function] }
> c.get()
2
> c.incr()
> c.get()
3
```

Méthode alternative: Encapsulation

On peut également obtenir des méthodes privées avec cette approche:

```
function Counter( x ) {
  let value = x;
  function add( n ) {
    value += n;
  }
  return {
    incr: function() { add(1); },
    decr: function() { add(-1); },
    get: function()  { return value; },
    reset: function() { value = 0; }
  }
}
```

Méthode alternative: Héritage

On peut utiliser le chaînage des prototypes pour hériter d'un autre contructeur.

```
function CounterDeluxe() {
  let c = Counter();
  c.reset = function() { c.set(0); };
  c.incrN = function( n ) {
    for( let i=0; i += 1; i<n ) {
          c.incr();
    }
  }
  return c;
}
```

Méthode alternative: Héritage (démo)

```
> let cd = CounterDeluxe();
> cd.incr();
> cd.incrN(100);
> cd.get();
101
```

Les Génériques en Java

Jean-Luc Falcone

5 avril 2022

Programmation générique

Polymorphisme paramétrique

Le polymorphisme paramétrique permet d'écrire des structures de données et des fonctions qui s'appliquent à plusieurs types, sans perdre la sûreté de typage (*type-safety*).

En Java ce mécanisme est implémenté par des generics.

Déclaration dans les classes et interfaces (1)

Déclarer un générique crée un nouvel identifiant (comme une variable).

```
public Machin {
  final int i;
  //i est l'identifiant d'une variable
  //la valeur sera fixée à l'instanciation
  //durant l'évaluation
}

public Truc<i> {
  //i est l'identifiant d'un type
  //sa valeur (le type en question) sera fixé
  //à l'instanciation, à la compilation
}
```

Déclaration dans les classes et interfaces (2)

```java
//'T' est un nouvel identifiant de type
public class Foo<T> { /* ... */ }

//'bob' est un nouvel identifiant de type
public interface Bar<bob> {  /* ... */ }

//'String' est un nouvel identifiant de type
// ce n'est pas le type String (!)
public interface Baz<String> {
        String get(); //De quel type s'agit-il ?
        //...
}
```

Déclaration dans les classes et interfaces (3)

Instancier un générique est similaire à assigner une valeur à une variable (mais à la compilation).

```
public Machin {
  final int i;
  Machin( int i ) { this.i = i };
}

public Truc<i> { }

var m = new Machin(2); // i = 2
var t = new Truc<String>; // i := String
```

Déclaration dans les classes et interfaces (4)

Les cas suivant ne sont pas des déclarations de génériques, mais des assignations.

```
//Ici 'String' est le type String
public class Hoo implements Bar<String> { /*...*/}


//Ici 'T' est un type pré-existant (!)
public class Goo implements Bar<T> { /*...*/}
```

Exemples d'interface générique (1)

```
interface RandomGenerator<T> {
  T single(Random rng);
}

interface Serializer<T> {
  byte[] convert( T t );
}

interface Stack<T> {
        T pop();
        void push(T t);
        boolean isEmpty();
}
```

Exemples d'interface générique (2)

Dans `java.util`

```
public interface Comparable<T> {
  //Retourne -1 si 'this' est plus petit que 'o'
  //          1 si 'this' est plus grand que 'o'
  //          0 en cas d'égalité
  int compareTo(T o);
}

public interface Comparator<T> {
  //Retourne -1 si 'o1' est plus petit que 'o2'
  //          1 si 'o1' est plus grand que 'o2'
  //          0 en cas d'égalité
  int compare(T o1, T o2);
}
```

Exemples d'implémentations génériques (1)

```
class PasswordGenerator
                    extends RandomGenerator<String> {
  private final int size;
  public PasswordGenerator( int size ) {
    this.size = size;
  }
  String single( Random rng ) {
    var buffer = new StringBuilder(size);
    for (int i = 0; i < size; i++) {
      int next = '0' + rng.nextInt('z' - '0');
      buffer.append((char) next);
    }
    return buffer.toString();
  }
}
```

Exemples d'implémentations génériques (2)

```
class ListGenerator<T>
                  extends RandomGenerator<List<T>> {
  private final int maxSize;
  private final RandomGenerator<T> elementGen;

  ListGenerator( int maxSize,
                 RandomGenerator<T> elementGen ) {
      this.size = size;
      this.elementGen = elementGen;
  }
  //...
}
```

Exemples d'implémentations génériques (2)

```
class ListGenerator<T>
                    extends RandomGenerator<List<T>> {
  public List<T> single(Random rng) {
    final var n = rng.nextInt(maxSize);
    final var ts = new ArrayList<T>(n);
    for( int i=0; i<n; i++ ) {
      ts.add( elementGen.single(rng) )
    }
    return ts;
  }
}
```

Exemples d'implémentations génériques (4)

```
public class Item {
  /*...*/
  String name() //...
  String reference() //...
  double popularity() //...
  int price() //...
}
```

Exemples d'implémentations génériques (5)

```
public class Item implements Comparable<Item>{
  /*...*/
  String name() //...
  String reference() //...
  double popularity() //...
  int price() //...
  int compareTo( Item that ) {
          //...
  }
}
```

Exemples d'implémentations génériques (6)

```
public class PriceComparator
                        extends Comparator<Item> {
  private final int order;
  PriceComparator( boolean ascending ) {
    if( ascending ) order = 1;
    else order = -1;
  }
  int compare( Item lhs, Item rhs ) {
    if( lhs.price < rhs.price )
      return -order;
    else if( lhs.price > rhs.price )
      return order;
    else return 0;
  }
```

Exemples d'implémentations génériques (6)

```
public class PopularityComparator
                        extends Comparator<Item> {
  private final int order;
  PopularityComparator( boolean ascending ) {
    if( ascending ) order = 1;
    else order = -1;
  }
  int compare( Item lhs, Item rhs ) {
    if( lhs.popularity < rhs.popularity )
      return -order;
    else if( lhs.popularity > rhs.popularity )
      return order;
    else return 0;
  }
```

Exemples d'implémentations génériques (7)

```
public class MultiComparator<T>
                          extends Comparator<T> {
  final private Comparator<T> fst;
  final private Comparator<T> snd;
  MultiComparator( Comparator<T> fst,
                   Comparator<T> snd ) {
    this.fst = fst; this.snd = snd;
  }
  int compare( Item lhs, Item rhs ) {
    int i = fst.compare(lhs,rhs);
    if( i == 0 ) return snd.compare(lhs,rhs);
    else return i;
  }
}
```

Méthodes génériques (1)

On peut également définir des méthodes génériques:

```
//U est un nouveau type générique
public <U> U foo( Foo<U> f, U u );

//R est un nouveau type générique
public static <R> void bar( R r, Comparator<R> c );
```

Méthodes génériques (2)

```
public static <T> twice( List<T> data ) {
  data.addAll( data );
}

public static <T> last( List<T> data ) {
  int i = data.size()-1;
  if( i < 0 ) throw NoSuchElementException();
  else return data.get(i);
}
```

Méthodes génériques (3)

```
public static <T> T min( List<T> values,
                         Comparator<T> comp ) {
  if( values.size() == 0 )
    throw new IllegalArgumentException(
                         "No min for empty list");
  T t = values.get(0);
  for( int i=1; i < values.size(); i++ ) {
    if( comp.compare( values.get(i), t ) {
       t = values.get(i);
    }
  }
  return t;
}
```

Attention au rédéfinition intempestives

```
class Foo<T> {

  //Même 'T' que l'instance
  public int bar1( Foo<T> foo ) { /*...*/ }

  //Introduit un nouveau type 'S'
  public <S> int bar3( Foo<S> foo ) { /*...*/ }

  //Ici T est redéfini !
  public <T> int bar3( Foo<T> foo ) { /*...*/ }

  //Ici T n'est pas défini du tout !
  public static int bar4( Foo<T> foo ) { /*...*/ }

}
```

Erasure (1)

En java, les types sont définis par erasure (effacement).

Durant la compilation:

1. Les contraintes liées aux types sont imposées;
2. Si tout est en ordre les génériques sont effacés
3. Le code nécéssaire (*type cast*) est éventuellement ajouté

Erasure (2)

```
//Définition dans le code
public static <T> T last( List<T> data ) {
  int i = data.size()-1;
  if( i < 0 ) throw NoSuchElementException();
  else return data.get(i);
 }

//Après compilation
public static Object last( List data ) {
  int i = data.size()-1;
  if( i < 0 ) throw NoSuchElementException();
    else return data.get(i);
}
```

Erasure (3)

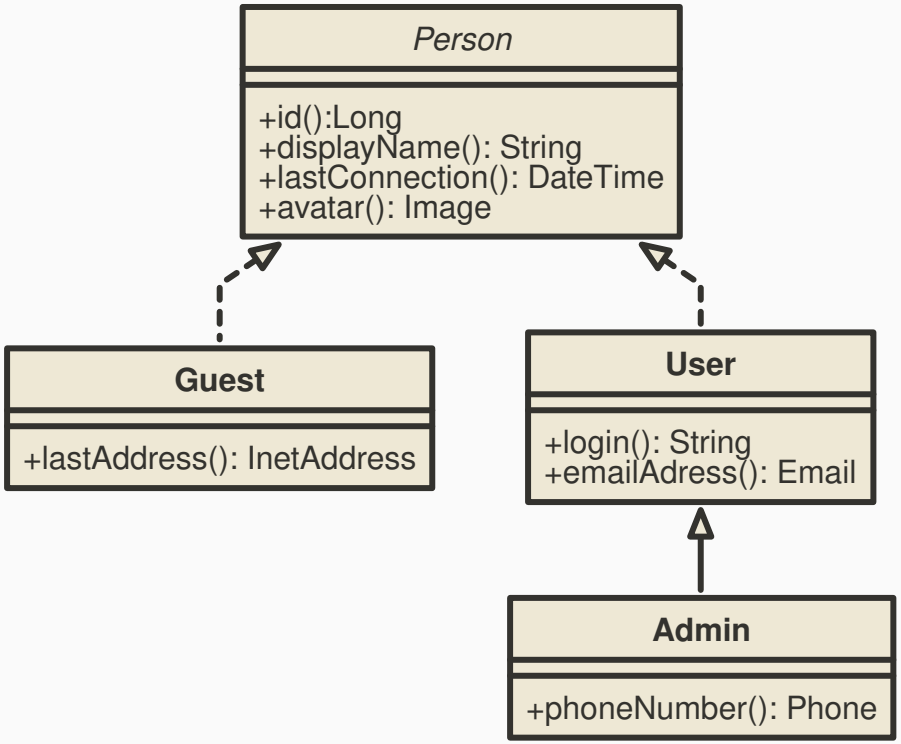
```
//Définition dans le code
var words = new ArrayList<String>();
words.add( "Hello" );
String lastWord = last( words );

//Après compilation
ArrayList words = new ArrayList();
words.add( (Object) "Hello" );
String lastWord = (String) last( words );
```

Variance et bornes

Dans la librairie standard…

```
package java.util;
class Collections {
  static <T> void
  fill(List<? super T> list, T obj) { /*...*/ }

  static <T> void
  copy(List<? super T> dest, List<? extends T> src)

  static <T> void
  sort(List<T> list, Comparator<? super T> c) {/*..

  static <T extends Comparable<? super T>> void
  sort(List<T> list)          { /*...*/ }
}
```

Gestion d’utilisateurs·rices



- Site de type blog multi-users
- Les traces de tous les users sont conservées et analysées
- Systèmes d’authentification et d’autorisation non-représentés

Petit Quizz

Si `User <: Person`, quelle est la bonne réponse:

1. `List<User> <: List<Person>`
2. `List<Person> <: List<User>`
3. `ArrayList<User> <: List<User>`
4. Toutes les réponses ci-dessus
5. Aucune des réponses ci-dessus

Tableaux en Java (1)

En Java les tableaux sont co-variants:

Si `A <: B` *alors* `A[] <: B[]`

Tableaux en Java (1)

En Java les tableaux sont co-variants:

Si `A <: B` *alors* `A[] <: B[]`

Cette décision est actuellement considérée comme catastrophique.

Tableaux en Java (2)

```java
//Aucune erreur de compilation

Admin[] admins = new Admin[1];

Person[] persons = admins;

persons[0] = new Guest();

Admin a = admins[0];
```

Tableaux en Java (3)

```
Admin[] admins = new Admin[1];

Person[] persons = admins;

persons[0] = new Guest();
//Erreur d'exécution: ArrayStoreException
```

Génériques en Java

En Java les génériques sont invariants:

Si `A <: B` *alors:*

- *ni* `F<A> <: F<B>`,
- *ni* `F<B> <: F<A>`,

ne sont vrais, pour tout type `F<?>`

Groupe d’utilisateurs·trices

```
class UserGroup {
  private final var members =
          new ArrayList<User>();
  public List<Email> getEmails() {
          var lst = new ArrayList<Email>();
          for( User u: members ) {
                  lst.add( u.email() );
          }
  }
}
```

Groupe d’administrateurs·trices

```
class AdminGroup {
  private final var members =
          new ArrayList<Admin>();
  /* ... */
  public List<Email> getEmails() {
          var lst = new ArrayList<Email>();
          for( Admin a: members ) {
                  lst.add( a.email() );
          }
  }
}
```

Groupe générique

```
class Group<U> {
  private final var members =
          new ArrayList<U>();
  /* ... */
  public List<Email> getEmails() {
          var lst = new ArrayList<Email>();
          for( U u: members ) {
                  if( u isinstanceof User ) {
                          User u_ = (User) u;
                          lst.add( (u_.email() );
                  }
          }
  }
}
```

Groupe générique borné (1)

```
class Group<U extends User> {
  private final var members =
          new ArrayList<Admin>();
  /* ... */
  public List<Email> getEmails() {
          var lst = new ArrayList<Email>();
          for( U u: members ) {
                  lst.add( (u.email() );
          }
  }
}
```

Groupe générique borné (2)

```
Group<User> ug = new Group<User>();

Group<Admin> ag = new Group<Admin>();

//Ne compile par car Guest enfreint la contrainte
Group<Guest> gg = new Group<Guest>();

//Ne compile pas car Group est invariant
Group<User> uag = new Group<Admin>();
```

Fonctions génériques bornées (1)

```
static <A extends User>
Group<A> mkGroup( List<A> members ) {
    /*...*/
}
```

Fonctions génériques bornées (2)

```
List<Admin> usr1 = /*...*/ ;
List<User> usr2 = /*...*/ ;
List<Guest> guests = /*...*/ ;

Group<Admin> group1 = mkGroup(usr1);

//Ne compile pas
Group<Admin> group2 = mkGroup(usr2);

//Ne compile pas
Group<Guest> group3 = mkGroup(guests);
```

Plusieurs bornes supérieures

On peut indiquer plusieurs contraintes grâce au symbole '&':

```
<T extends U & V & W>
// exprime les contraintes T <: U  et
//                          T <: V  et
//                          T <: W
```

Groupe générique borné: *type erasure*

```
class Group {
  private final var members =
          new ArrayList();
  /* ... */
  public List getEmails() {
          var lst = new ArrayList();
          for( User u: members ) {
                  lst.add( (u.email() );
          }
  }
}
```

Envoi de mail

```
public void sendEmail( User u,
                       String subject,
                       String body ) {/*...*/ }

var u = new User(...);
var a = new Admin(...);

sendEmail( u, "Test mail",
              "Ignore. Do not reply" );
sendEmail( a, "Test mail",
              "Ignore. Do not reply" );
```

Envoi de mails groupés (1)

```
public void sendEmail( List<User> users,
                       String subject,
                       String body ) {
  for( Users user: users )
          sendEmail(user,subject,body);
}

List<User> users = getAllUsers();

sendEmail( users, "Test mail",
                  "Ignore. Do not reply" );
```

Envoi de mails groupés (2)

```
List<Admin> admins = getAllAdmin();

//Ne compile pas
sendEmail( admins, "Test mail", "Ignore. Do not rep

//Ne compile pas, non plus
List<User> users = getAllAdmin();
```

Envoi de mails groupés (3)

```
public <U extends User>
void sendEmail( List<U> users,
                String subject,
                String body ) {
  for( U user: users )
          sendEmail(user,subject,body);
}

List<Admin> admins = getAllAdmin();
sendEmail(admins, "Test mail",
                          "Ignore. Do not reply" );
```

Envoi de mail groupé: *Wildcard*

```
public void sendEmail( List<? extends User> users,
                       String subject,
                       String body ) {
  for( User user: users )
          sendEmail(user,subject,body);
}

List<Admin> admins = getAllAdmin();
sendEmail(admins, "Test mail",
                          "Ignore. Do not reply" );
```

Sérialisation de profils utilisateurs (1)

```
interface Serializer<T> {
  byte[] convert( T t );
}
```

Sérialisation de profils utilisateurs (2)

Plusieurs implémentations possibles:

```
class AdminSerializer extends Serializer<Admin> {
  byte[] convert( Admin admin ) { /*...*/ }
}

class UserSerializer extends Serializer<User> {
  byte[] convert( User user ) { /*...*/ }
}

class PersonSerializer extends Serializer<Person> {
  byte[] convert( Person person ) { /*...*/ }
}
```

Sérialisation de profils utilisateurs (3)

Pour sérialiser `Admin`, on peut donc utiliser:

- sérialiseur de `Admin`;
- un sérialiseur de `User`,
- un sérialiseur de `Person`,
- un sérialiseur de `Object`

C'est à dire un sérialiseur de n'importe quel supertype. On note ceci:

```
Serializer<? super Admin>
```

Sérialisation de profils utilisateurs (4)

On ne peut pas sérialiser un `User` avec un `Serializer<Admin>`

Par contre on peut sérialiser un `Admin` avec un `Serializer<User>`

Donc:

```
Serializer<User> <: Serializer<? super Admin>
```

Sérialisation de profils utilisateurs (5)

```
import java.nio.file.Files;
void writeToFile( File file,
                  Admin admin,
                  Serializer<? super Admin>
                          serializer) {
        var data = serializer.serialize(admin);
        Files.write( file.toPath(), data );
}
```

Génération aléatoire d'Utilisateurs (1)

```
interface RandomGenerator<T> {
  T single(Random rng);
}

void testUserDisplay( RandomGenerator<User> user )
  /*...*/
}

RandomGenerator<Admin> rgAdm = //...;

//Ne compile pas
testUserDisplay( rgAdm );
```

Génération aléatoire d'Utilisateurs (2)

```
void testUserDisplay(
  RandomGenerator<? extends User> user ) {
  /*...*/
}

RandomGenerator<Admin> rgAdm = //...;

testUserDisplay( rgAdm );
```

Génération aléatoire d'Utilisateurs (3)

On peut générer un `User` aléatoire avec un `RandomGenerator<Admin>`

Par contre, on ne peut pas générer un `Admin` aléatoire avec un `RandomGenerator<User>`

Donc:

```
RandomGenerator<Admin> <: RandomGenerator<?
extends User>
```

Variance (1)

Etabli précédemment:

```
Admin <: User

Serializer<User> <: Serializer<? super Admin>

RandomGenerator<Admin> <:
                RandomGenerator<? extends User>
```

Mais pourquoi ?

Variance (2)

Comparons les interfaces.

```
interface RandomGenerator<T> {
  T single(Random rng);
}

interface Serializer<T> {
  byte[] convert( T t );
}
```

Quelle est différence fondamentale ?

Principe de substitution de Liskov (1)

Pour tout sous-type:

- Les pré-conditions des méthodes peuvent être assouplies, mais pas durcies
- Les post-conditions des méthodes peuvent être durcies, mais pas assouplies
- Les invariants du super type doivent être préservés tels quels.

- Le type des arguments est une précondition
- Le type de retour est une post-condition
- Un type spécialisé est plus dur qu'un type généralisé

Principe de substitution de Liskov (2)

Pour tout sous-type:

- Les types des arguments des méthodes peuvent être généralisés, mais pas spécialisés
- Les types de retour des méthodes peuvent être spécialisés, mais pas généralisés
- Si un type apparaît en argument et en retour, on doit le préserver tel quel.

Principe de substitution de Liskov (2)

Pour tout sous-type:

- Les types des arguments des méthodes peuvent être généralisés, mais pas spécialisés: contra-variance
- Les types de retour des méthodes peuvent être spécialisés, mais pas généralisés: co-variance
- Si un type apparaît en argument et en retour, on doit le préserver tel quel: invariance

Co-Variance

```
interface RandomGenerator<T> {
  T single(Random rng);
}
void testUserDisplay(
  RandomGenerator<? extends User> user
)  { /*...*/ }
```

- Type générique en retour de fonction uniquement.
- `RandomGenerator<T>` est un producteur

Contra-variance

```
interface Serializer<T> {
  byte[] convert( T t );
}
void writeToFile( File file, Admin admin,
  Serializer<? super Admin> serializer
) { /*...*/ }
```

- Type générique en argument uniquement
- `Serializer<T>` est un consommateur

Invariance

```
interface Stack<T> {
        T pop();
        void push(T t);
        boolean isEmpty();
}
void <T> swap( Stack<T> stack ) {
        var t1 = stack.pop();
        var t2 = stack.pop();
        stack.push(t1);
        stack.push(t2);
}
```

- Type générique en argument et retour
- `Stack<T>` est un producteur-consommateur

Truc mnémotechnique: PECS

Abbréviation de: *Producer Extends Consumer Super*

```
void writeToFile( File file, User user,
  Serializer<? super User> serializer)

void testUserDisplay(
  RandomGenerator<? extends User> user )
```

Dans la librairie standard (2)

```
package java.util;
class Collections {
  static <T> void
  fill(List<? super T> list, T obj) { /*...*/ }

  static <T> void
  copy(List<? super T> dest, List<? extends T> src)

  static <T> void
  sort(List<T> list, Comparator<? super T> c) {/*..

  static <T extends Comparable<? super T>> void
  sort(List<T> list)          { /*...*/ }
}
```

Copie conservative (1)

Lorsqu'on travaille avec des objets mutables, on a tendance à les copier pour éviter les effets de bord.

Cette copie doit être profonde.

Avec les objets immutables la copie n'est pas nécéssaire.

Interface Copiable (1)

```
interface Copiable {
        Object safeCopy();
}
class Point implements Copiable {
        private final double x;
        private final double y;
        /*...*/
        public Object safeCopy() { return this; }
}
```

Interface Copiable (2)

```
class Account implements Copiable {
        /*...*/
        Account( String ID, long amount ) {
                /*...*/
        }
        public Object safeCopy() {
                return new Account(ID, amount);
        }
}
```

Interface `Copiable` (3)

```
List<Object> deepCopy(
  List<? extends Copiable> objs
) {
        var out = new ArrayList<Object>();
        for( Copiable c: objs ) {
                out.add( c.safeCopy() );
        }
        return out;
}
```

Plus de typage (1)

```
interface Copiable<T> {
  T safeCopy();
}

//Problème: ça compile
class Point implements Copiable<String> {
        /*...*/
        public String safeCopy() {
                return "gna gna gna";
        }
}
```

Plus de typage (2)

```
interface Copiable<T> {
        T safeCopy();
}

class Point implements Copiable<Point> {
        /*...*/
        public Point safeCopy() {
                return this;
        }
}
```

Plus de typage (3)

```
interface Copiable<T> {
        T safeCopy;
}

//Problème: ça compile
class Point implements Copiable<String> {
        /*...*/
        public String safeCopy() {
                return "gna gna gna";
        }
}
```

Plus de typage (4)

```
interface Copiable<T extends Copyable<T>> {
        T safeCopy;
}

//ça ne compile plus !
class Point implements Copiable<String> {
        /*...*/
        public String safeCopy() {
                return "gna gna gna";
        }
}
```

Plus de typage (5)

```
class Point implements Copiable<Point> {
        /*...*/
        public Point safeCopy() {
                return this;
        }
}
```

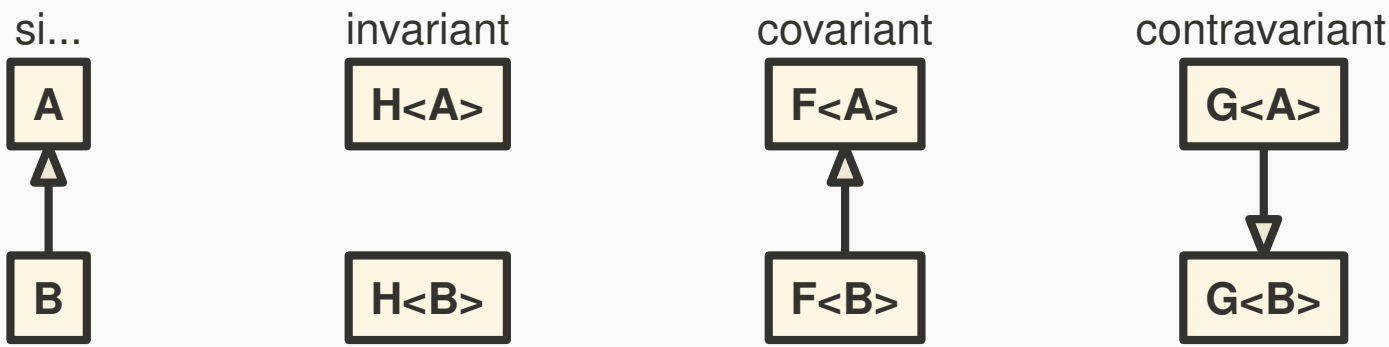
Plus de typage (6)

```
static <T extends Copiable<T>>
List<T> deepCopy( List<T> objs ) {
        final var out = new ArrayList<T>();
        for( T t: objs ) {
                out.add( t.safeCopy() );
        }
        return out;
}
```

Libraire standard (3)

```java
public interface Comparable<T> {
  int compareTo(T o);
}

static <T extends Comparable<? super T>>
void  sort(List<T> list) {
        /*...*/
}
```

La variance



Classes et expressions lambda

Jean-Luc Falcone

12 avril 2022

Classes Anonymes

GUI en Java

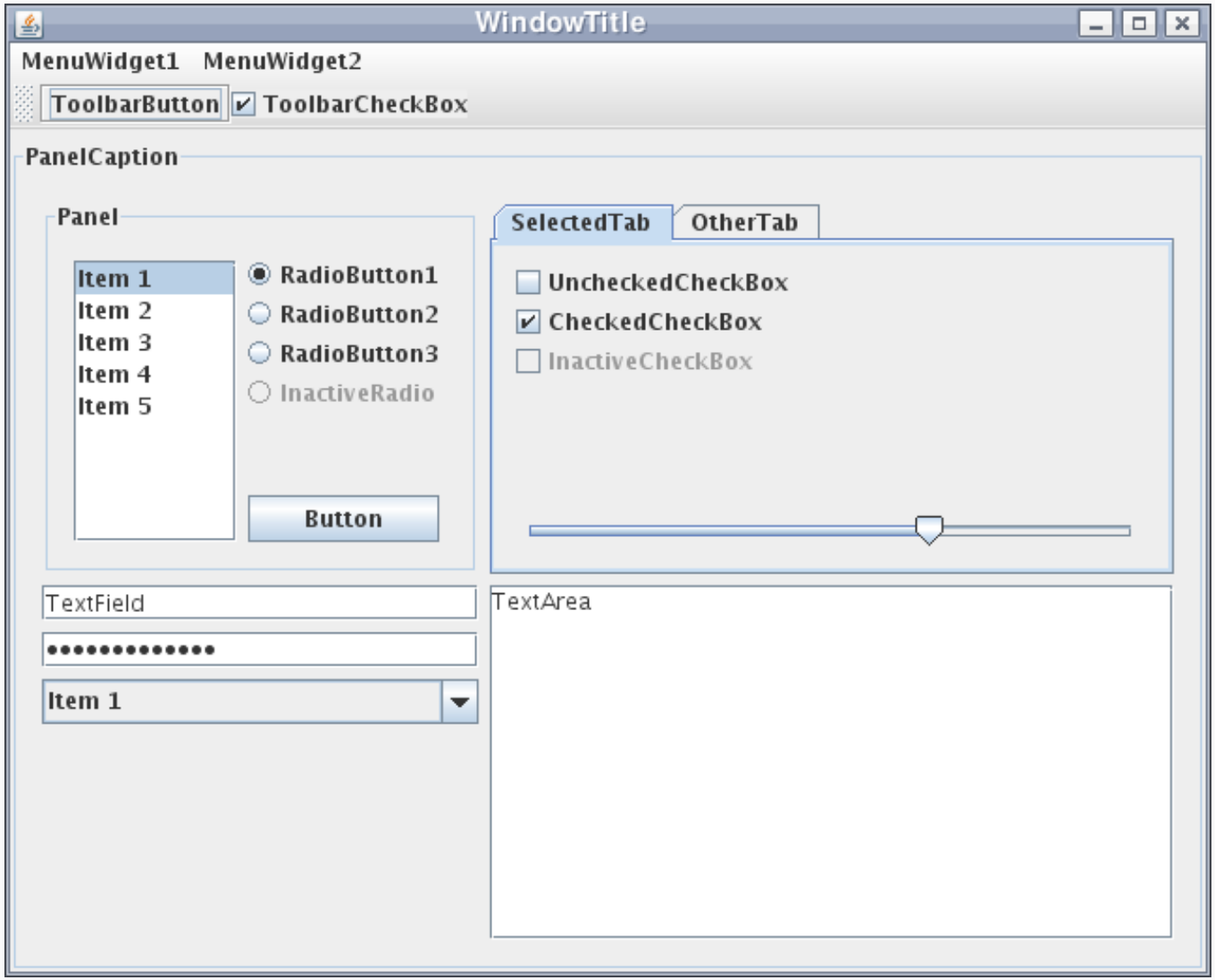
Plusieurs générations d'API:

- 1996: AWT
- 1999: Swing
- 2008: JavaFX

Problèmes:

- Multi-plateforme
- Performances
- Intégration

Exemple: Swing (1)



Exemple: Swing (2)

```
JButton b1 = new JButton("Greetings");
b1.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(frame,
      "Hello, World !");
  }
});
```

Exemple: Swing (3)

```
public abstract class AbstractButton /*...*/ {
  //...
  public void addActionListener(ActionListener l) {
    listenerList.add(ActionListener.class, l);
  }
  //...
}
```

Exemple: Swing (3)

```
public abstract class AbstractButton /*...*/ {
  //...
  public void addActionListener(ActionListener l) {
    listenerList.add(ActionListener.class, l);
  }
  //...
}

public interface ActionListener
extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

Exemple: Swing (4)

```
class HelloListener extends ActionListener {
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(frame,
      "Hello, World !");
  }
}

//Dans un autre fichier
JButton b1 = new JButton("Greetings");
b1.addActionListener( new HelloListener() );
```

Classe anonyme

```
ActionListener act = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(frame,
      "Hello, World !");
  }
};
```

Générateur aléatoire (1)

```
static <T>  RandomGenerator<T>
constant(final T value) {
  return new RandomGenerator<T>() {
    public T single(Random rng) {
      return value;
    }
  }
}

var random = constant("bravo");
```

Générateur aléatoire (2)

```
static <T>  RandomGenerator<T>
pick(final List<? extends T> data) {
  return new RandomGenerator<T>() {
    public T single(Random rng) {
      var i = rng.nextInt(data.size());
      return data.get(i);
    }
  }
}

List<String> names = /*...*/;
var randomNames = pick(names);
```

Comparaisons: exemple

```
public class Item {
  /*...*/
  String name() //...
  String reference() //...
  double popularity() //...
  int price() //...
}
```

Comparaisons (1)

```
var popularityComparator = new Comparator<Item>() {
  int compare( Item lhs, Item rhs ) {
    if( lhs.popularity < rhs.popularity )
      return -1;
    else if( lhs.popularity > rhs.popularity )
      return 1;
    else return 0;
  }
};
```

Comparaisons (2)

```
static Comparator<Item>
popularityComparator(boolean ascending ) {
  return new Comparator<Item>() {
    int order = -1;
    if( ascending ) order = 1;
    int compare( Item lhs, Item rhs ) {
      if( lhs.popularity() < rhs.popularity() )
        return -1;
      else if( lhs.popularity() > rhs.popularity()
        return 1;
      else return 0;
    }
  };
}
```

Comparaisons (3)

```
static Comparator<Item>
priceComparator(boolean ascending ) {
  return new Comparator<Item>() {
    int order = -1;
    if( ascending ) order = 1;
    int compare( Item lhs, Item rhs ) {
      if( lhs.price() < rhs.price() )
        return -1;
      else if( lhs.price() > rhs.price() )
        return 1;
      else return 0;
    }
  };
}
```

Itérateurs

cf. démo (à la rentrée…)

Expressions lambda

En Javascript (API fictive)

```
b1.addActionListener( function(e) {
  showMessageDialog("Hello, World !");
});


//Depuis ES6: syntaxe alternative
b1.addActionListener(
        e => showMessageDialog("Hello, World !")
);
```

En Python (API fictive)

```
b1.addActionListener(
  lambda e:
      showMessageDialog("Hello, World !")
)
```

En Swift (API fictive)

```
b1.addActionListener( { e in
  showMessageDialog("Hello, World !")
})
```

Expression lambda en Java

- Introduite dans Java 8 (2014).
- Synthèse programmation fonctionelle et orientée-objet.
- Très pratique pour manipuler les collections (cours suivant).

Swing revisité: avant

```
JButton b1 = new JButton("Greetings");

b1.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(frame,
      "Hello, World !");
  }
});
```

Swing revisité: après

```
JButton b1 = new JButton("Greetings");

b1.addActionListener( (ActionEvent e) -> {
  JOptionPane.showMessageDialog(frame,
    "Hello, World !");
});
```

Function interfaces & Single abstract methods

En java, une expression lambda peut être passée à la place de toute interface qui n'a qu'une seule méthode abstraite:

```
interface Foo<A,B> {
  int bar( A a, B, b );
}
```

Classe

```
class Diff implements Foo<String,String> {
  public int bar( String s1, String s2 ) {
    return s1.length() - s2.length();
  }
}
//Dans un autre fichier
Diff diff = new Diff();
int i = diff.bar( "ABCD", "EF" );
```

Classe Anonymes

```
var diff = new Foo<String,String>() {
    public int bar( String s1, String s2 ) {
      return s1.length() - s2.length();
    }
  };

int i = diff.bar( "ABCD", "EF" );
```

Expression lambda

```
Foo<String,String> diff = (s1,s2) -> {
  return s1.length() - s2.length();
};

int i = diff.bar( "ABCD", "EF" );
```

Limitation

Doit être assignée à un type cible:

```
var diff = (s1,s2) -> {
  return s1.length() - s2.length();
};
//error: cannot infer type for local variable diff
//(lambda expression needs an explicit target-type)

var fooss4 = (String s1,String s2) -> {
  return s1.length() - s2.length();
};
//error: cannot infer type for local variable diff
//(lambda expression needs an explicit target-type)
```

Générateur aléatoire (1)

```
static <T>  RandomGenerator<T>
constant(final T value) {
  return (Random rng) -> {
          return value;
  };
}

var random = constant("bravo");
```

Générateur aléatoire (2)

```
static <T>  RandomGenerator<T>
pick(final List<? extends T> data) {
  return (rng) -> {
    var i = rng.nextInt(data.size());
    return data.get(i);
  };
}

List<String> names = /*...*/;
var randomNames = pick(names);
```

Comparateurs (1)

```
public class Item {
  /*...*/
  String name() //...
  String reference() //...
  double popularity() //...
  int price() //...
}
```

Comparateurs (2)

```
interface Comparator<T> {
  static <T> Comparator<T> comparingInt(
    ToIntFunction<? super T> keyExtractor
  ) { /*...*/ }

  static <T> Comparator<T> comparingDouble(
    ToDoubleFunction<? super T> keyExtractor
  ) { /*...*/ }

  /*...*/
}
```

Types génériques: `java.util.function` (1)

```
interface Function<T,R> {
  R apply(T t);
}
interface Predicate<T> {
  boolean test(T t);
}
interface Supplier<T> {
  T get();
}
interface Consumer<T> {
  void accept(T t);
}
```

Types génériques: `java.util.function` (2)

Spécialisés pour les primitives (pour éviter le boxing):

```
interface ToIntFunction<T> {
  int applyAsInt(T value);
}

interface ToDoubleFunction<T> {
  double applyAsDouble(T value);
}
```

Comparateurs (3)

```
Comparator<Item> priceCmp =
  Comparator.comparingInt( item -> {
    return item.price();
  });

Comparator<Item> popularityCmp =
  Comparator.comparingDouble( item -> {
    return item.popularity();
  });
```

Syntaxe: Arguments

Formes valides:

```
//Types explicite
(int a, double b) -> //...

//Types inférés
(a,b) -> //...

//Pas d'arguments
() -> //...

//Si un seul argument, type inféré
a -> //...
```

Syntaxe: corps

Formes valides:

```
(a,b) -> {
  return a + b;
};

//Si expression:
(a,b) -> a+b
```

Syntaxe: corps

```
Comparator<Item> priceCmp =
  Comparator.comparingInt( item ->
    item.price()
  );

Comparator<Item> popularityCmp =
  Comparator.comparingDouble( item ->
    item.popularity()
  );
```

Référence de fonction

On peut référencer une fonction d'une instance en utilisant la notation `::`:

```
Comparator<Item> priceCmp =
  Comparator.comparingInt(Item::price);

Comparator<Item> popularityCmp =
  Comparator.comparingDouble(Item::popularity);
```

Référence de fonction statique

```java
//Définie dans java.util.function
public interface Consumer<T> {
  void accept(T t);
}

Consumer<String> display = System.out::println;
```

Autre références de fonctions

Référence de constructeur:

```
Foo:: new
```

Référence de méthode d'une instance:

```
(range.iterator())::hasNext
```

Les Streams

Jean-Luc Falcone

26 avril 2022

Streams

Définition

«Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections.»
– in Javadoc: `java.util.streams`

«A sequence of elements supporting sequential and parallel aggregate operations.»
– in Javadoc: `java.util.streams.Stream`

Exemple

- Lire un fichier ligne par ligne (`numbers.txt`)
- Convertir la ligne en nombre décimal
- Sommer les nombres

Exemple: sans Streams

```
import java.nio.file.*;
Path file =
  FileSystems.getDefault().getPath("numbers.txt");

try( var reader = Files.newBufferedReader(file) ) {
  var line = "";
  double sum = 0;
  while ((line = reader.readLine()) != null) {
    var num = Double.parseDouble(line);
    sum += num;
  }
} catch (/*...*/) { /*...*/ }
```

Exemple: avec Streams

```
import java.nio.file.*;
Path file =
  FileSystems.getDefault().getPath("numbers.txt");

try ( var lines = Files.lines(file) ) {
  double sum = lines
    .mapToDouble( Double::parseDouble )
    .sum();
} catch (/*...*/) { /*...*/ }
```

Exemple 2

- Lire un fichier ligne par ligne (`numbers.txt`)
- Convertir la ligne en nombre décimal
- Sommer les nombres, mais pas les nombres négatifs

Exemple 2: sans Streams

```
import java.nio.file.*;
Path file =
  FileSystems.getDefault().getPath("numbers.txt");

try ( var reader = Files.newBufferedReader(file) )
  var line = "";
  double sum = 0;
  while ((line = reader.readLine()) != null) {
    var num = Double.parseDouble(line);
    if( num > 0 )
      sum += num;
  }
} catch (/*...*/) { /*...*/ }
```

Exemple 2: avec Streams

```
import java.nio.file.*;
Path file =
  FileSystems.getDefault().getPath("numbers.txt");

try ( var lines = Files.lines(file) ) {
  double sum = lines
    .mapToDouble( Double::parseDouble )
    .filter( x -> x > 0.0 )
    .sum();
} catch (/*...*/) { /*...*/ }
```

Création de streams (1)

A partir d'une collection, `List`, `Set`, `Queue`, etc.:

```
interface Collection<E> {
  Stream<E> stream();
}

//Par exemple
ArrayList<Item> bag = loadItems();
Stream<Item> items = bag.stream();
```

Création de streams (2)

A partir d'un tableau:

```
import java.util.Arrays;
String[] words = openDict();
Stream<String> wordStream = Arrays.stream(words);
```

A partir des éléments, directement:

```
Stream<String> words =
  Stream.of( "foo", "bar", "baz" );
```

Création de stream (3)

Pour lire les lignes d'un fichiers:

- `BufferedReader.lines()`
- `Files.lines()`

Pour fragmenter une chaîne de caractères:

- `Pattern.splitAsStream()`

Création spécialisée (1)

Pour éviter le boxing, il existe des versions spécialisées de `Streams`:

- `IntStream`
- `LongStream`
- `DoubleStream`

Pour les créer, à partir de tableaux:

```
double[] xs = compute();
DoubleStream numbers = Arrays.stream(xs);
```

Création spécialisée (2)

Intervale de nombres:

```
IntStream is = IntStream.range(10,25);
```

Nombres aléatoires:

```
import java.util.Random;
LongStream is = Random.longs(
  10_000_000L, -5_000_000L, 5_000_000L
);
//10 mios de longs aléatoires entre -5mios et +5mio
```

Opérations

Types d'opérations (1)

Pipeline composés de:

- 0..n opérations intermédiaires
- 1 opération terminale

```
double sum = lines
  .mapToDouble( Double::parseDouble ) //Interm.
  .filter( x -> x > 0.0 )             //Interm.
  .sum();                             //Terminale
```

Types d'opérations (2)

Les opérations intermédiaires:

- Produisent un `Stream`
- Généralement paresseuses (*lazy*)
- Généralement *stateless*
- Peuvent court-circuiter le parcours (*short-circuit*)

Les opérations terminales:

- Produise une valeur ou un effet de bord
- Strictes (inverse de *lazy*)
- Peuvent court-circuiter le parcours (*short-circuit*)

Opérations intermédiaires: quelques exemples

- `limit(long n)` : garde les `n` prochains éléments;
- `skip(long n)` : saute les `n` prochains éléments;
- `map( Function<? super T,? extends R> f )` applique la fonction `f`. Produit un `Stream<R>` à partir d'un `Stream<T>`;
- `filter(Predicate<? super T> p)` garde seulement les éléments pour lesquels le prédicat `p` est vrai.

Opérations intermédiaires: spécialisées

Permettent de créer des Stream spécialisés pour les primitives pour éviter le boxing:

```
DoubleStream
mapToDouble(ToDoubleFunction<? super T> mapper)

IntStream
mapToInt(ToIntFunction<? super T> mapper)

LongStream
mapToLong(ToLongFunction<? super T> mapper)
```

Opérations intermédiaires *stateful* et couteuses

- `distinct()` Le Stream ne contient plus que des éléments distints (du point de vue de `equals`).
- `sorted()` Le Stream est trié si les éléments sont `Comparables`
- `sorted(Comparator<? super T> comparator)` Le stream est trié avec le comparateur passé

Opérations intermédiaires: `flatMap`

L'opération `flatMap` permet d'appliquer une fonction qui transforme chaque élément en Stream et concatène les résultats.

```
<R> Stream<R>
flatMap( Function<? super T,
         ? extends Stream<? extends R>> mapper)
```

Gestion de playlist (1)

```
interface Artist {
  List<Album> albums();
}
interface Album {
  List<Track> tracks();
}
interface Track { /*...*/ }
```

Gestion de playlist (2)

Tous les tracks:

```
List<Track> allTracks( List<Artists> artists ) {
  var tracks = new ArrayList<Tracks>();
  for( Artist artist: artists ) {
    for( Album album: artist.albums() ) {
      tracks.addAll( album.tracks() );
    }
  }
  return tracks;
}
```

Gestion de playlist (3)

Tous les tracks, avec les Streams:

```
List<Track> allTracks( List<Artists> artists ) {
  return artists.stream()
    .flatMap(art -> art.albums().stream() )
    .flatMap(alb -> alb.tracks().stream() )
    .collect( Collectors.toList() );
}
```

Opérations terminales: Prédicats

```java
//Tous les éléments satifsont le prédicat
boolean allMatch(Predicate<? super T> predicate)

//Au moins un élément satisfait le prédicat
boolean anyMatch(Predicate<? super T> predicate)
```

Opérations terminales: Réductions

```
T reduce(T identity, BinaryOperator<T> accumulator)

<U> U
reduce(U identity,
       BiFunction<U,? super T,U> accumulator,
       BinaryOperator<U> combiner);
```

Opérations terminales: Créer une collection

```
<R,A> R
collect(Collector<? super T,A,R> collector);

//Exemples
stream.collect(Collectors.toList());

stream.collect(Collectors.toSet());

stream.collect(
  Collectors.toCollection(()-> new ArrayList<T>)
);
```

Opérations terminales: Appliquer un effet de bord

```java
void forEach(Consumer<? super T> action);

//Par exemple
stream.forEach( System.out::println );
```

Exemple plus complexe

Gestion de playlist (4)

```
interface Artist {
  List<Album> albums();
  Set<String> genres();
}
interface Album {
  List<Track> tracks();
  YearMonth release();
}
interface Track {
  String title();
  Duration duration();
}
```

Gestion de playlist (5)

On aimerait selectioner toutes les tracks sauf:

- des artistes considérés *Country*
- des albums sortis durant les années 80,
- de plus de 10 minutes

Gestion de playlist, sans Streams (6)

```
final var TenMin = Duration.ofMinutes(10);
var tracks = new ArrayList<Tracks>();
for( Artist artist: artists ) {
  if(! artists.genres().contains("Country") ) {
    for( Album album: artist.albums() ) {
      int year = album.release().getYear();
      if( year < 1980 || year >= 1990 ) {
        for( Track track: album.tracks() ) {
          if( track.duration().compareTo( TenMin )
              <= 0 ) {
            tracks.add( track );
          }
        }
      }
    } } } }
```

Gestion de playlist: avec Streams (7)

```
final var TenMin = Duration.ofMinutes(10);
Stream<Track> tracks = artists.stream()
   .filter( r -> ! r.genres().contains("Country") )
   .flatMap(r -> r.albums().stream() )
   .filter( l -> {
       int year = l.release().getYear();
       return year < 1980 || year >= 1990;
    })
   .flatMap(l -> l.tracks().stream() )
   .filter( t ->
       track.duration().compareTo( TenMin ) <= 0 );
```

Gestion de playlist (8)

On aimerait selectioner toutes les tracks sauf:

- des artistes considérés *Country*
- des albums sortis durant les années 80,
- de plus de 10 minutes

... on veut juste afficher les 10 premiers morçeaux trouvés.

Gestion de playlist: sans streams (9)

```
final int n = Math.min( 10, tracks.size() );
for( int i=0; i < n; i++ ) {
  System.out.println(tracks.get(i).title());
}
```

Gestion de playlist: avec streams (10)

```
tracks
  .limit(10)
  .map(Track::title)
  .forEach(System.out::println);
```

Optionnelles

Valeur absente

```
//Retourne le profile utilisateur
//correspondant à un email
public User getUserByEmail( Email email ) {
  /*...*/
}
```

Valeur absente

```
//Retourne le profile utilisateur
//correspondant à un email
public User getUserByEmail( Email email ) {
  /*...*/
}
```

Et si aucun `User` ne correspond à `email` ?

Valeur absente: Exception ?

```
boolean isEmailRegistered( Email email ) {
  try {
    User user = getUserByEmail(email);
    return true;
   } catch( NoSuchUserException e ) {
    return false;
  }
}
```

Valeur absente: Exception ?

```
List<User> getUsers( List<Email> emails ) {
  try {
    return emails.stream()
      .map( e -> getUserByEmail(e) )
      .collect( Collector.toList() );
  } catch( NoSuchUserException e ) {
    return new ArrayList<User>();
  }
}
```

Valeur absente: Exception ?

```
List<User> getUsers( List<Email> emails ) {
    return emails.stream()
      .map( e -> {
          try{
            return getUserByEmail(e);
          } catch( NoSuchUserException e ) {
            return /* !!?! */;
          }
        })
      .collect( Collector.toList() );
}
```

Valeur absente: `null` ?

```
boolean isEmailRegistered( Email email ) {
  return getUserByEmail(email) != null;
}
```

Valeur absente: `null` ?

```
List<User> getUsers( List<Email> emails ) {
  return emails.stream()
    .map( e -> getUserByEmail(e) )
    .collect( Collectors.toList() );
}
```

Petit problème...

Valeur absente: `null` ?

```
List<User> getUsers( List<Email> emails ) {
  return emails.stream()
    .map( e -> getUserByEmail(e) )
    .filter( u -> u != null )
    .collect( Collectors.toList() );
}
```

Valeur absente: Optionnels

```
public Optional<User>
getUserByEmail( Email email ) {
  /*...*/
}
```

Optionnels

Un optionnel `Optional<T>` peut être:

- soit vide: pas de valeur
- soit présent: contient une valeur de type `T`, non `null`

On peut voir ça comme une collection qui contient 0 ou 1 élément.

Création (1)

```java
//Optionnelle vide
Optional<String> email1 = Optional.empty();

//Optionnelles non vide
Optional<String> email2 =
  Optional.of("foo@example.com");
Optional<String> email3 =
  Optional.ofNullable("foo@example.com");
```

Création (2)

```
Optional<String> email2 = Optional.of(null);
//NullPointerException !

Optional<String> email3 =
  Optional.ofNullable(null);
//retourne un optionnel vide
```

Utilisation: tests et extraction de la valeur

```java
var email = Optional.ofNullable("foo@example.com");

email.isPresent(); //true
email.isEmpty();   //false

//Déconseillée
email.get(); //"foo@example.com"
             //NoSuchElementException si vide
```

Utilisation: valeur par défaut

```
Optional<String> noMail = Optional.empty();
var email = Optional.ofNullable("foo@example.com");

email.orElse("no-reply@example.com");
  //"foo@example.com"
noMail.orElse("no-reply@example.com");
  //"no-reply@example.com"

email.orElse( expansiveDBaccess() );
  //LENT !
email.orElseGet( ()->expansiveDBaccess() );
  //RAPIDE !
```

Utilisation: passage d'effet de bord (1)

Au lieu de:

```
if( email.isPresent() ) {
  var mail = email.get();
  sendMail( mail, "Alert", "blah blah blah" );
}
```

Utiliser:

```
email.ifPresent( mail -> {
  sendMail( mail, "Alert", "blah blah blah" );
});
```

Utilisation: passage d'effet de bord (2)

Au lieu de:

```
if( email.isPresent() ) {
  var mail = email.get();
  sendMail( mail, "Alert", "blah blah blah" );
} else {
  System.err.println("Undefined email...");
}
```

Utiliser:

```
email.ifPresent( mail -> {
  sendMail( mail, "Alert", "blah blah blah" );
}, () -> {
  System.err.println("Undefined email...");
});
```

Transformation: map

Au lieu de:

```
if( email.isPresent() ) {
  var mail = email.get();
  return Optional.of( mail.toLowerCase() );
} else {
  return Optiona.empty();
}
```

Utiliser:

```
return email.map( mail -> {
   mail.toLowerCase() );
});
//ou mieux
return email.map( String::toLowerCase );
```

Filtrer: `filter`

Au lieu de:

```
if( email.isPresent() ) {
  var mail = email.get();
  if( validate(mail) )
    return Optional.of( mail );
  else
    return Option.empty();
}
```

Utiliser:

```
return email.filter( mail -> {
    validate(mail)
});
```

Enchaîner les valeurs manquantes (1)

Soit:

```
import java.nio.file.*;

//Retourne le répertoire parent s'il existe
//retourne null si non défini
Path parent0( Path p );

Optional<Path> parent( Path p );
```

Enchaîner les valeurs manquantes (2)

Remonter de trois niveaux (Posix `../../..`):

```
Path foo0( Path p ) {
  var px = parent0(p);
  if( px != null ) {
    var py = parent0(px);
    if( py != null ) {
      var pz = parent0(py);
      return pz;
    }
  }
  return null;
}
```

Enchaîner les valeurs manquantes (3)

Remonter de trois niveaux (Posix `../../..`):

```
Optional<Path> foo( Path p ) {
  var px = parent(p);
  if( px.isPresent() ) {
    var py = parent(px.get());
    if( py.isPresent() ) {
      var pz = parent0(py.get());
      return pz;
    }
  }
  return Optional.isEmpty();
}
```

Enchaîner les valeurs manquantes: `flatMap` (4)

Remonter de trois niveaux (Posix `../../..`):

```
Optional<Path> foo( Path p ) {
  return parent(p)
    .flatMap( px -> parent(px) )
    .flatMap( py -> parent(py) );
}

//Ou mieux
Optional<Path> foo( Path p ) {
  return parent(p)
    .flatMap( this::parent )
    .flatMap( this::parent );
}
```

Optionels en `Stream`

```
public Optional<User>
getUserByEmail( Email email ) {
  /*...*/
}

List<User> getUsers( List<Email> emails ) {
  return emails.stream()
    .flatMap( e -> getUserByEmail(e).stream() )
    .collect( Collectors.toList() );
}
```

Du bon usage

Les optionels ont été ajouté pour décrire une valeur de retour manquante.

Déconseillés dans plusieurs cas:

- Champs d'une classe
- Arguments de fonctions

A éviter si les performances sont cruciales, car overhead en mémoire.

Java: quelques éléments en vrac

Jean-Luc Falcone

3 mai 2022

Méthodes par défaut

Méthodes par défaut dans les interfaces (1)

```
interface Group<K extends Group<K>> {
  K add( K that );
  K negate();
  default K subtract( K that ) {
    return add( that.negate() );
  }
}
```

Méthodes par défaut dans les interfaces (2)

```
class ClockHour implements Group<Clock> {
  private final int i;
  ClockHour( int i ) {
    this.i = i % 12;
  }
  K add( K that ) {
    return new ClockHour( i + that.i );
  }
  K negate() {
    return new ClockHout(12-i);
  }
}
```

Méthodes par défaut dans les interfaces (3)

```
class ClockHour implements Group<Clock> {
  /*...*/
  K add( K that ) {
    return new ClockHour( i + that.i );
  }
  K negate() {
    return new ClockHour(12-i);
  }
  K subtract( K that ) {
    int j = i - that.i;
    j = (j < 0 ) ? 12-j : j;
    return new ClockHour(j);
  }
}
```

Enumérations

Enumérations

- Type énumérés: ensemble prédifini de constantes
- Par exemple:
  - mois dans l'années: janvier, février, ... , décembre
  - état d'une porte: ouverte, fermée, verouillée
  - couleurs aux échecs: noir ou blanc
  - couleurs des cartes: piques, carreaux, coeurs, trèfles
  - valeur des cartes: As, 2, 3, ..., Valet, Dame, Roi

Définitions (1)

```
public enum DoorState {
  OPEN, CLOSED, LOCKED
}

public enum TrafficLightColor {
  RED, YELLOW, GREEN
}
```

Définitions (2)

```
public enum FrenchSuite {
  SPADE,HEART,DIAMMOND,CLUBS
}

public enum FrenchValue {
  AS,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,
  NINE,TEN,JACK,QUEEN,KING
}
```

Utilisation

```
DoorState door1 = DoorState.LOCKED;

class Card {
  private final FrenchSuite suite;
  private final FrenchValue value;
  Card( FrenchSuite suite, FrencValue value ) {
    this.suite = suite; this.value = value;
  }
}

var kingOfSpade =
  new Card(FrenchSuite.SPADE, FrenchValue.KING );
```

Imports statiques

Permettent d'alléger la syntaxe.

```
import static DoorState.*;
import static FrenchSuite.*;
import static FrenchValue.*;

DoorState door1 = LOCKED;
var kingOfSpade = new Card(SPADE, KING );
```

Egalité

Comme les instances sont prédéfinies, on peut utiliser tester l'identité sans problème:

```
import static DoorState.*;

DoorState open( DoorState current ) {
  if( current == CLOSED )
    return OPEN;
  else if( current == LOCKED )
    throw new IllegalStateException(
     "Cannot open a locked door");
  else throw new IllegaleStateException(
     "Door is already open !");
}
```

Switch...case...

```
import static DoorState.*;

DoorState open( DoorState current ) {
  switch(current) {
  case CLOSED:
    return OPEN;
  case LOCKED:
    throw new IllegalStateException(
     "Cannot open a locked door");
  case OPEN:
    throw new IllegaleStateException(
     "Door is already open !");
  }
}
```

Switch...case... expression

A partir de Java 12:

```
import static FrenchSuite.*;

String color = switch(suite) {
  case SPADE, CLUBS -> "#000000";
  case DIAMMOND, HEARTS -> "#FF0000";
}
```

Classes enum (1)

En Java les enums sont des classes.

Les enums peuvent:

- Définir des méthodes
- Implémenter des interfaces

Classes enum (2)

```
public enum TrafficLightColor {
  RED("#cc3232"),
  YELLOW("#e7b416"),
  GREEN("#2dc937");

  private final String htmlColor;
  TrafficLightColor(String col) {
    htmlColor = col;
  }
  String asHTML() {
    return htmlColor;
  }
}
```

Classes enum (3)

```
var go = TrafficLightColor.GREEN;

System.println( go.asHTML() );
```

Classes enum (4)

```
public enum FrenchSuite
  implements Comparable<FrenchSuite>{
  SPADE(0),HEART(1),DIAMMOND(2),CLUBS(3);

  private final int order;
  FrenchSuite( int i ) { order = i; }
  public int compareTo( FrenchSuite that ) {
    return this.order - that.order;
  }
}
```

Parcourir les instances

Tous les enums ont une méthode statique `values()` retournant un itérable sur les valeurs:

```
for( var suite: FrenchSuite.values() ) {
  System.out.println(suite);
}
```

Classes imbriquées

Définition: *Inner classes*

```
class Foo {
  class Bar {

  }
  static class Baz {

  }
}
```

Non-statiques vs. statiques

- Une instance d'une classe imbriquée non-statique a accès à tous les membres (même privés) de la classe qui la contient
- Une classe imbriquée statique ne peut accéder qu'aux membres statiques de la classe qui la contient

Contrôle d’accès

Alors qu’une classe ne peut être que `public` ou *default*, une classe imbriquée peut être `private` ou `protected`

Pile (1)

```
class Stack<T> {
  private class Elem {
    final T value;
    final Elem next;
    Elem(T v, Elem n ) {
      value=v; next=n;
    }
  }
  private Elem top == null;
  public boolean isEmpty() {
    return top = null;
  }
  /*...*/
```

Pile (2)

```
class Stack<T> {
  private class Elem {
    final T value;
    final Elem next;
        //...
  }
  private Elem top = null;
  public void push( T value ) {
    top = new Elem(value,top);
  }
  /*...*/
}
```

Pile (3)

```
class Stack<T> {
  private class Elem {
    final T value;
    final Elem next;
        //...
  }
  /*...*/
  public T pop() {
    if( top.value == null )
      throw new NoSuchElementException();
    var t = top.value;
    top = top.next;
    return top;
  }
}
```

Pile (4)

```
class Stack<T> {
  private class Elem {
    final T value;
    final Elem next;
        //...
  }
  /*...*/
  public T peek() {
    if( top.value == null )
      throw new NoSuchElementException();
    return top.value;
  }
}
```

Pile (5)

```
class Stack<T> {
  /*...*/
  public int size() {
    int n = 0;
    Elem e = top;
    while( e != null ) {
      n += 1;
      e = e.next;
    }
    return n;
  }
}
```

Attention aus génériques

```
class Stack<T> {
  //Type différent avec le même nom
  private class Elem<T> {
    final T value;
    final Elem next;
    Elem(T v, Elem n ) {
      value=v; next=n;
    }
  }
  //...
}
```

Tableau de Classes ou Classe de tableaux ?

En Java:

- Chaque instance est allouée sur le tas.
- Un tableau d'instances est un tableau de référence.
- Les valeurs d'un tableau d'instances ne sont pas contiguës en mémoire
- Les données contigües en RAM sont lues dans la même ligne de cache
- Chaque instance a un sur-coût de 16B en mémoire

Donc un tableau d'objet est lent à parcourir et prend plus de place en RAM

Tableau de Classes (1)

```
class Particule {
  private double x_;
  private double y_;
  /*...*/
  public double x() { return x_; }
  public double y() { return y_; }
}
```

Tableau de Classes (2)

```
class Particules {
  private Particule[] ps;
  /*...*/
  public Particule centerOfMass() {
    double xx;
    double yy;
    for( int i=0; i < ps.length; i++ ) {
      xx += ps[i].x(); //LENT !
      yy += ps[i].y(); //LENT !
    }
    return new Particule(
     xx/ps.length, yy/ps.lengtj);
  }
  public Particule get(int i){return ps[i];}
}
```

Classe de tableaux (1)

```
class Particules {
  private double[] x_;
  private double[] y_;
  /*...*/
  public Particule centerOfMass() {
    double xx;
    double yy;
    for( int i=0; i < ps.length; i++ ) {
      xx += x_[i];
      yy += y_[i];
    }
    return new Particule(
     xx/ps.length, yy/ps.length);
  }
}
```

Classe de tableaux (2)

```
class Particules {
  private double[] x_;
  private double[] y_;
  /*...*/
  public Particule get(int i) {
    return new Particule( //Copie
     x_[i], y_[i]
    );
  }
}
```

Classe de tableaux avec Vue (1)

```
interface Particule {
  public double x();
  public double y();
  public void setPos( double x, double y );
}
```

Classe de tableaux avec Vue (2)

```
class Particules {
  private final double[] x_;
  private final double[] y_;
  /*...*/
  static class ConcreteParticule
    extends Particule {/*...*/ }
  private class ParticleView extends Particule{
    private final int i;
    ParticuleView( int i ) { this.i = i; }
    public double x() { return x_[i]; }
    public double y() { return y_[i]; }
    public void setPos( double x, double y) {
      x_[i] = x; y_[y] = y;
    }
  }
```

Classe de tableaux avec Vue (3)

```
class Particules {
  /*...*/
  public Particule centerOfMass() {
    /*...*/
    return new ConreteParticule(
     xx/ps.length, yy/ps.length);
  }
  public Particule get(int i) {
    return new ParticuleView(i);
}
```

Autres notions

- Interfaces imbriquées
- Enums imbriqués
- Classes locales