

04 – Principes et paradigmes

Présenté par Yann Caron
skyguide

ENSG Géomatique

Plan du cours

Compilation vs
Interpretation

Typage

Portée

Les différents paradigmes



But

- ✓ Faire un inventaire des différentes caractéristiques des langages
- ✓ Typage ?
- ✓ Portée ?
- ✓ Paradigmes :
 - ✓ Impératif ? Structuré ?
 - ✓ Fonctionnel ?
 - ✓ Objet ? MOP ? AOP ? Duck Typing ?
 - ✓ Par contrat ?
 - ✓ Logique ?

Compilateurs vs Interpreters

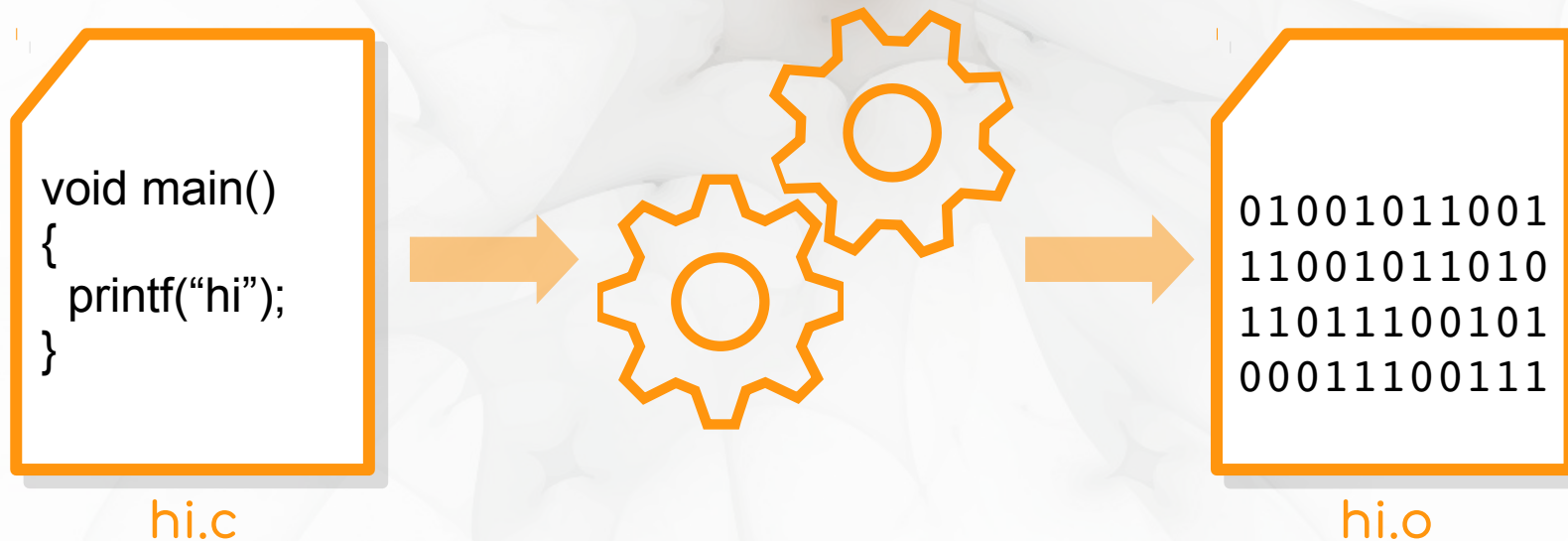


Constatation

- ✓ Différence d'implémentation du CodeGen
- ✓ Les deux sont très similaires, seul la "sortie" est différente
- ✓ Le compilateur génère du code machine dans un fichier binaire
- ✓ L'interpreteur exécute le programme en mémoire

Compilateur - définition

Le programme est transformé en code machine dans un fichier



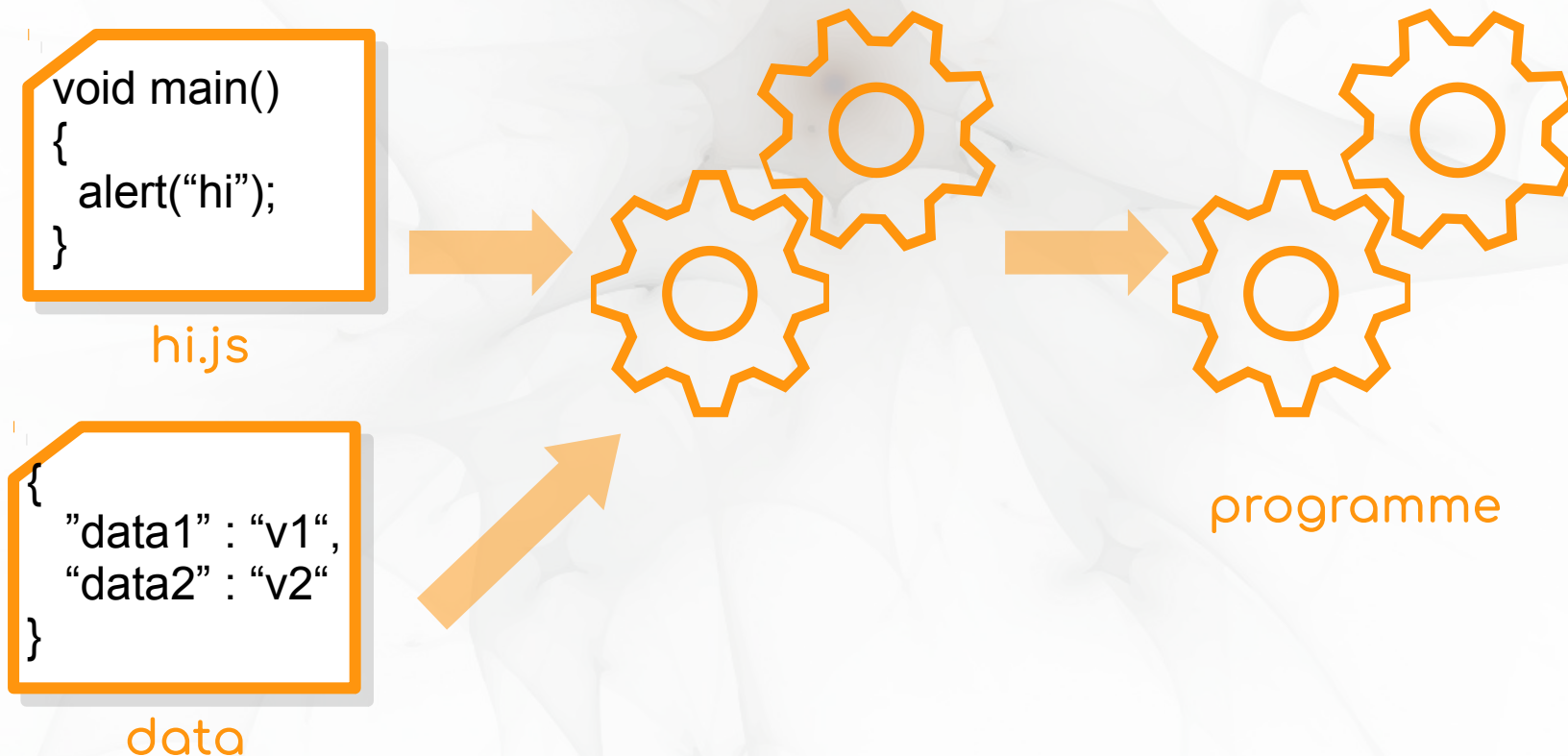
Compilateur - hybride

Le programme est transformé en code machine à destination d'une machine virtuelle



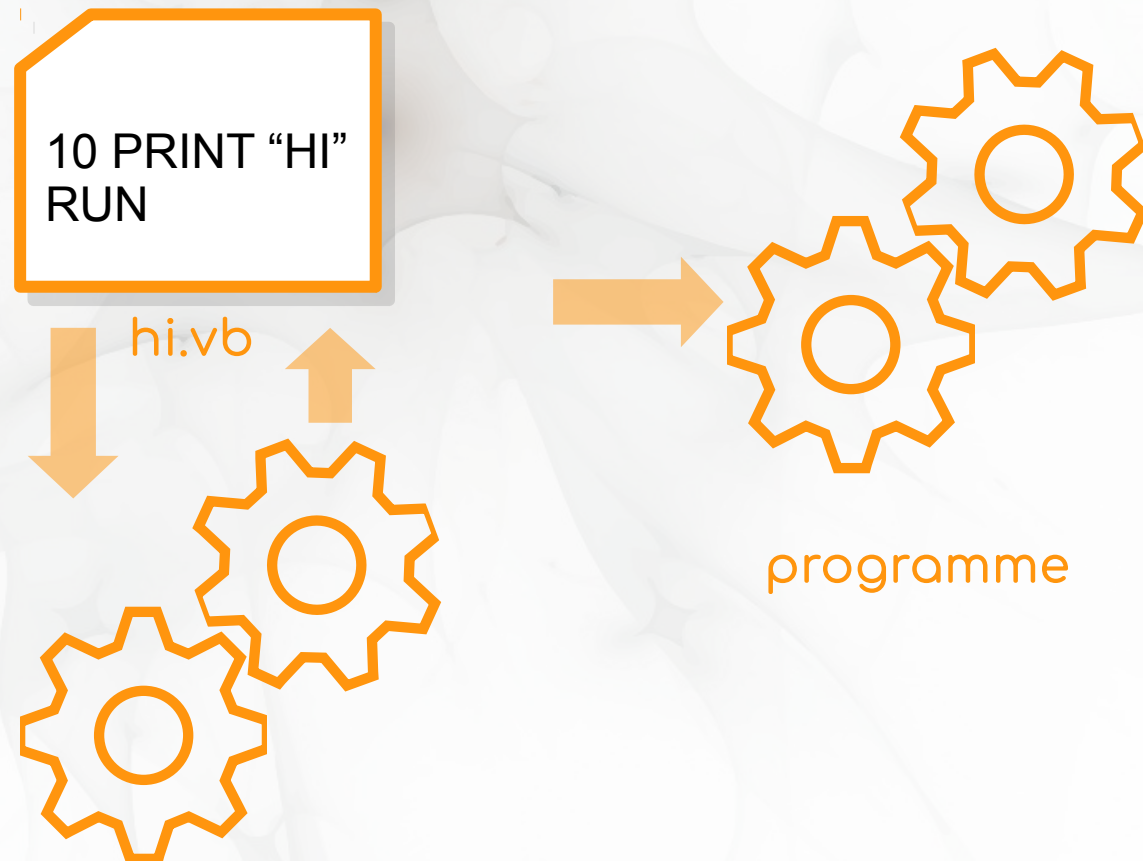
Interpréteur - définition

Le programme et les données sont transformés directement en résultat



Interprétation directe

Les première version de Basic. Le programme est exécuté à mesure qu'il est analysé



Comparatif - Compilation

- ✓ Pour :
 - ✓ Meilleures performances
 - ✓ Opportunités d'appliquer des optimisations (analyse sémantique)
- ✓ Contre:
 - ✓ Rigide et plus complexe (ex : Généricité, Templates)
 - ✓ Faible portabilité

Comparatif - Interprétation

- ✓ Pour :
 - ✓ Flexible et plus simple à mettre en œuvre
 - ✓ Langage dynamiques (MOP)
 - ✓ Exécution à la volée
- ✓ Contre:
 - ✓ Coût sur les performances
 - ✓ Coût mémoire
 - ✓ Analyse minimale → détection d'erreur à posteriori

Compromis

- ✓ Une machine virtuelle permet de tirer partis des deux mondes :
- ✓ Portabilité (code once run anywhere de java)
- ✓ Optimisations (JIT)
- ✓ Analyse sémantique poussée
- ✓ Typage hybrides (Java object & cast)

Typage ?

- ✓ Le typage (statique vs dynamique) est il inhérent à la méthode d'exécution (compilé vs interprété) ?
- ✓ Python ?
- ✓ Common LISP ?

Typage



Enjeu

- ✓ Compromis entre la flexibilité et la robustesse
- ✓ Robustesse : Détecter le maximum d'erreurs lors de la phase de compilation
- ✓ Cela induit de la rigidité : exemple des génériques de Java 5
- ✓ Typage fort vs faible est trop flou

Typage statique vs dynamique

- ✓ Statique lorsque la vérification s'effectue lors de la compilation
- ✓ Dynamique lorsque celle ci est effectuée lors de l'exécution

Typage explicite vs implicite

- ✓ Explicite, est exprimée dans le programme

```
int i = 0;
```

- ✓ Implicite (ou inférence) ; est déduite par analyse ou lors de l'exécution

```
let i = 0;
```

Types composés - tableaux

- ✓ Les tableaux
- ✓ Ensemble non fini d'éléments du même type

```
char t[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
```

- ✓ La liste en est cas particulier (redimensionnable dynamiquement)

Types composés – structure (C#)



- ✓ Les structures
- ✓ Ensemble fini d'éléments de type différents

```
public struct Point
{
    public float X;
    public float Y;
    public string Name;
}
```

Types composés – tuples (C#)

- ✓ Les tuples
- ✓ Cas particulier de structures immutables (programmation fonctionnelle)

```
public class Unit<T1> {  
  
    private final T1 value1;  
  
    public T1 getValue1() {  
        return value1;  
    }  
  
    public Unit(T1 value1) {  
        this.value1 = value1;  
    }  
}
```

```
public class Pair<T1, T2> extends Unit<T1> {  
  
    private final T2 value2;  
  
    public T2 getValue2() {  
        return value2;  
    }  
  
    public Pair(T1 value1, T2 value2) {  
        super(value1);  
        this.value2 = value2;  
    }  
}
```


Types composés – union (C)

- ✓ Peut alternativement être de plusieurs types différents



```
public union Jour
{
    public char Lettre;
    public char Numéro;
}
```

Types récurifs

- ✓ Un type de donnée qui peu contenir des données de son type
- ✓ En Haskell

```
data List a = Nil | Cons a (List a)
```

- ✓ En Java

```
class List<E> {  
    E value;  
    List<E> next;  
}
```

Types récurifs

- ✓ S'applique à la création des listes chaînés
- ✓ S'applique à la création des arbres
 - ✓ Cf : Gof Composite
- ✓ Observation : une liste chaînée est un cas particulier de l'arbre. Un arbre dont tous les parent n'auraient qu'un seul enfant.

Types énumérés

- ✓ Ensemble fini de valeurs possibles pour une variable

```
public enum PrimaryColor {  
    RED, BLUE, YELLOW;  
}
```

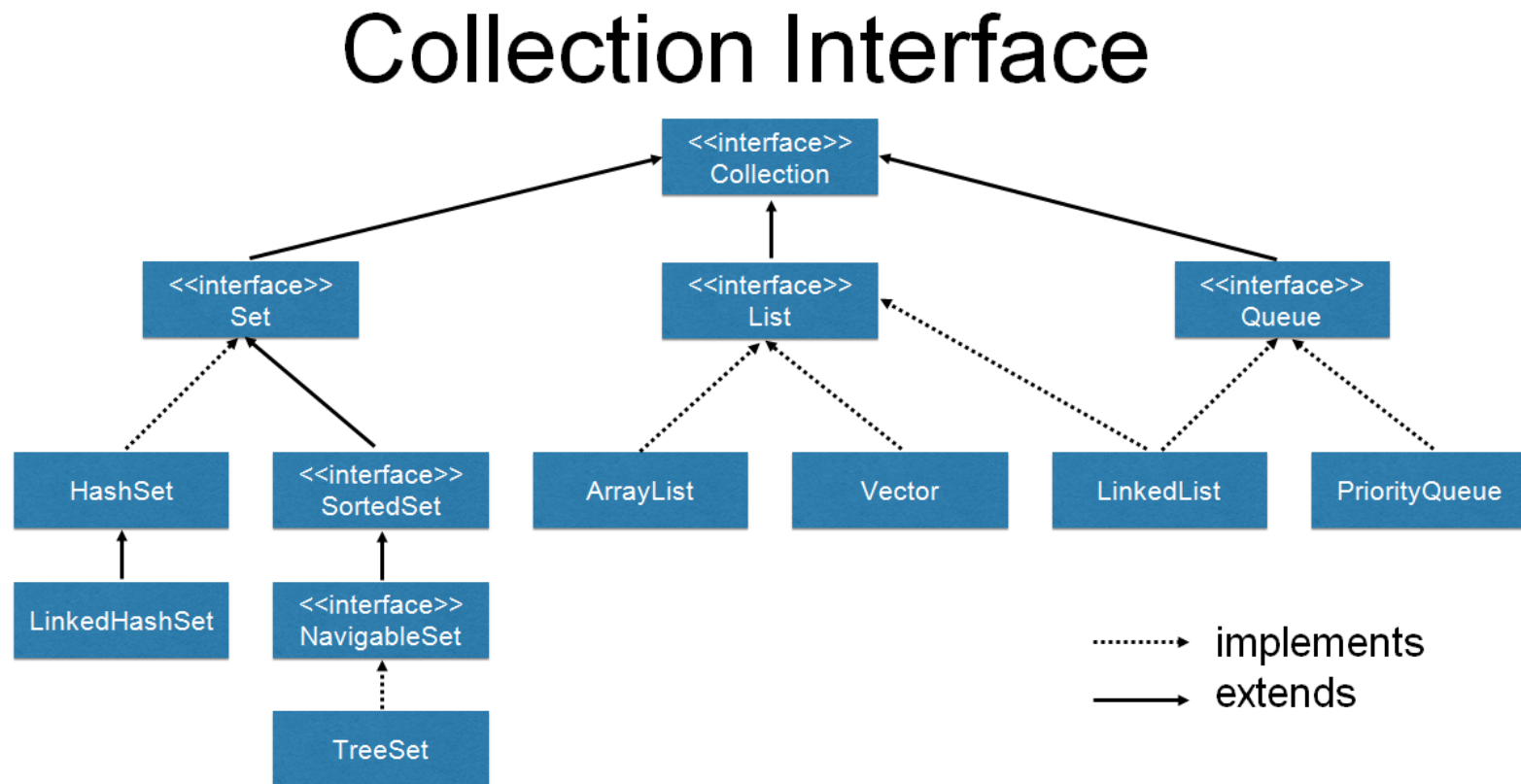

Types paramétrés

- ✓ Car des “generics” en Java et C#
- ✓ Templates en C++

```
class List<E> {  
    E value;  
    List<E> next;  
}
```

Types hiérarchiques

- ✓ Exemple la hiérarchie de classe en Java



Portée



Définition

- ✓ La portée est la portion de programme où la variable nommée est atteignable

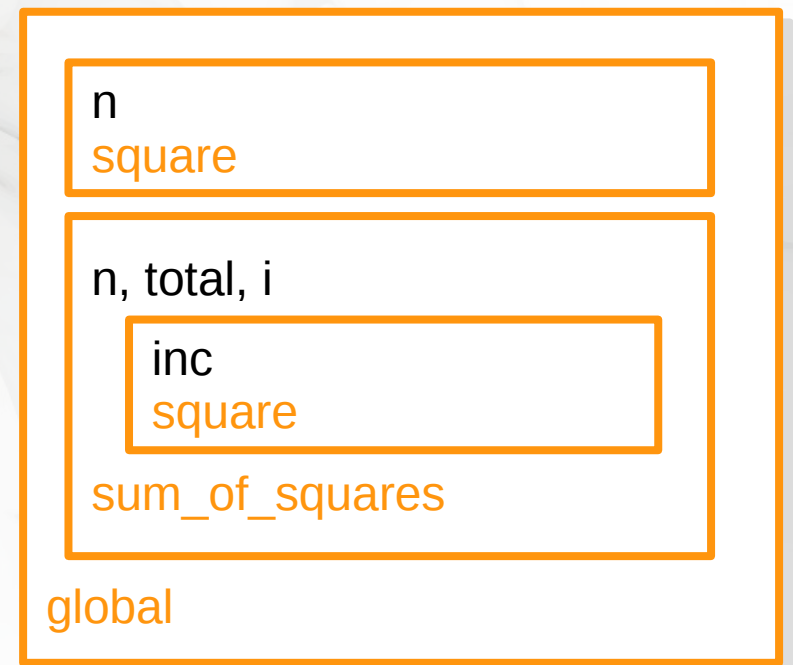
Portée globale

- ✓ Toutes les variables déclarées sont ajoutées dans la mémoire globale du programme
- ✓ Elles sont de fait accessible de n'importe quelle portion du programme
- ✓ Cas du Basic à ses débuts
- ✓ Inconvénient : Collision de noms et masquage in-intentionnel
- ✓ Mots clés : **local**, **global**
- ✓ Mauvaise pratique (effets de bord)

Portée lexicale (statique)

- ✓ La portée des variables est relative à l'emplacement où elles sont déclarées dans le code

```
def square(n):  
    return n * n  
  
def sum_of_squares(n):  
    total = 0  
    i = 0  
    while i <= n:  
        inc = 1  
        total += square(i)  
        i += inc  
    return total
```



Portée dynamique

- ✓ La portée des variables est relative au flux d'exécution

Lisp

```
(defun foo1 ()  
  (message "%s" a)) // 1  
  
(defun foo2 ()  
  (let ((a 2))  
    (message "%s" a))) // 2  
  
(defun foo3 ()  
  (let ((a 1))  
    (foo1)  
    (foo2)))
```



Portée implicite vs explicite

- ✓ Problématique de création des variables
- ✓ Explicite ; un mot clé détermine ou est créer la variable



JavaScript

```
function makeCounter() {  
  var counter = 0; // new variable  
  return function inc() {  
    counter = counter + 1; // reassign higher level  
    return counter;  
  }  
}
```


Portée implicite vs explicite

- ✓ Implicite ; si la variable est inconnue, elle est créée. Le programme réassigne en remontant dans les portées jusqu'à trouver la variable concernée



CoffeeScript

```
makeCounter = ->  
  counter = 0 # new variable  
  return ->  
    counter = counter + 1 # reassign higher level  
    return counter
```

Portée implicite vs explicite

- ✓ Intermédiaire ; Les variables sont créées de façon implicite, c'est la réaffectation qui est explicite



Python

```
def make_counter():  
    counter = 0 # new variable  
    def inc():  
        nonlocal counter  
        counter = counter + 1 # reassign higher level  
        return counter  
    return inc
```

Mutabilité des variables

- ✓ Variable définit une seule fois et accessible en lecture seule
- ✓ Mot clé : **final**, **const**
- ✓ Bonne pratique de programmation : immutabilité pour limiter les effets de bords
- ✓ Langages fonctionnels purs : réfutent la mutabilité des données

Variant

- ✓ Programmation réactive

```
int a = 7;  
int b = 7;  
int c = a + b;  
a = 10;
```

- ✓ Que vaut c à présent ?
- ✓ 14 ou 17 ?

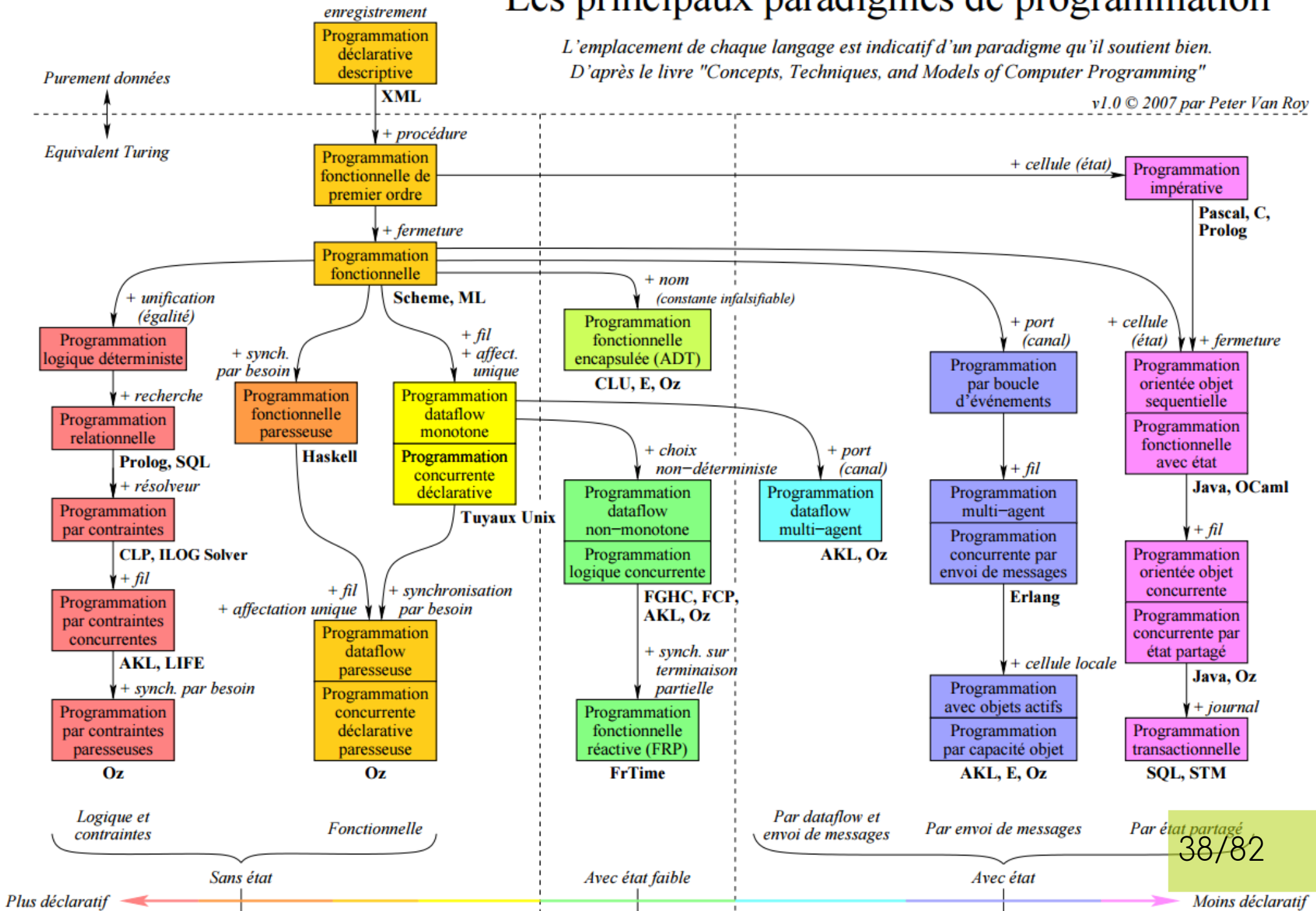
Paradigmes



Les principaux paradigmes de programmation

L'emplacement de chaque langage est indicatif d'un paradigme qu'il soutient bien.
D'après le livre "Concepts, Techniques, and Models of Computer Programming"

v1.0 © 2007 par Peter Van Roy



Programmation impérative

```
L0
  ICONST_0
  ISTORE 1
L1
  GOTO L2
L3
  IINC 1 1
L2
  ILOAD 1
  BIPUSH 10
  IF_ICMPLT L3
L4
  RETURN
L5
```

✓ Langage machine
(load / jump)

```
int i = 0;

while (i < 10) {
    i++;
}
```

Impératif non structuré



Dijkstra 1930

- ✓ IF / GOTO – BASIC
- ✓ Edsger Wybe Dijkstra : "Go To Statement Considered Harmful"

```
10 LET I = 0
20 IF I >= 10 THEN GOTO 50
30 I = I + 1
40 GOTO 20
50 ...
```

Impératif structuré

- ✓ Séquence d'instructions
- ✓ Assignment
- ✓ Conditionnelle : If then else elseif
- ✓ Boucles : for, while, until
- ✓ Branchements : Goto ou appel de procédures

```
LET I = 0
DO
    I = I + 1
LOOP WHILE I < 10
```

Orienté Objet

- ✓ (Classes ou prototype) : Définition de types complexes
- ✓ Objets : Instances de ces types

```
public class Character {  
    private final String name;  
    public Character(String name) {  
        this.name = name;  
    }  
}
```

```
public static void main(String[] args) {  
    Character instance1 = new Character("Dupont");  
    Character instance2 = new Character("Tintin");  
}
```



Orienté Objet

- ✓ Propriétés : états de l'objet (attribut est privé)
- ✓ Méthodes : comportements de l'objet
- ✓ Évènements : signal que peut émettre l'objet (callback)
- ✓ Portée : private, package, protected, public
- ✓ Immutabilité : final
- ✓ Class / Instance : static

Orienté Objet : Concepts

- ✓ Encapsulation : Les données et les méthodes qui les manipulent sont cachés du reste de l'application
- ✓ Composition : Un objet peut en contenir un autre dans ses attributs
- ✓ Délégation : Un objet peut en manipuler un autre au travers sa variable d'instance

Orienté Objet : Dispatch

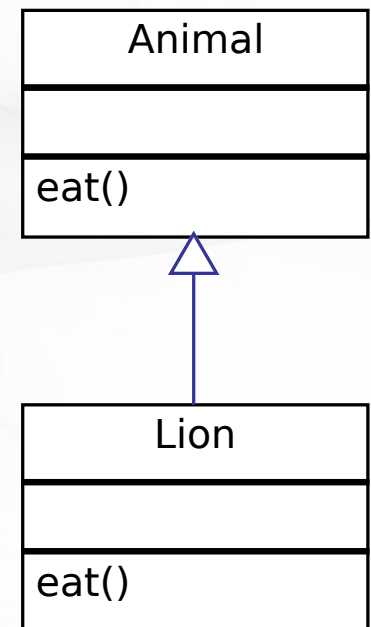
- ✓ L'objet est responsable de savoir quelle méthode appeler
- ✓ Phénomène de surcharge (overload)

```
public class Dispatch {  
  
    public void method(String value) {  
    }  
  
    public void method(int value) {  
    }  
  
}
```

Orienté Objet : Héritage

- ✓ Une classe peut hériter d'une autre classe dans une relation de type "est un"
- ✓ Phénomène de spécialisation (override)

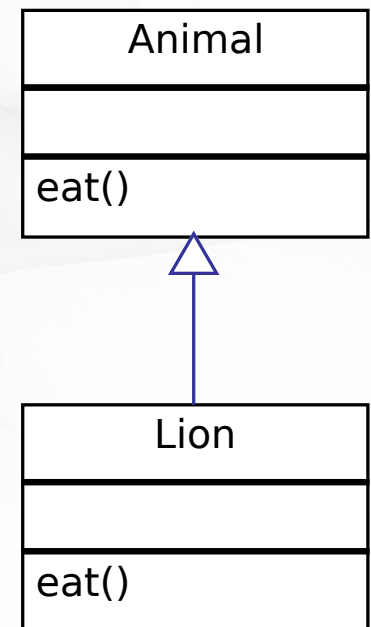
```
public class Animal {  
    public void eat() {}  
}  
  
public class Lion extends Animal {  
    public void eat() {  
        // eat meat !  
    }  
}
```



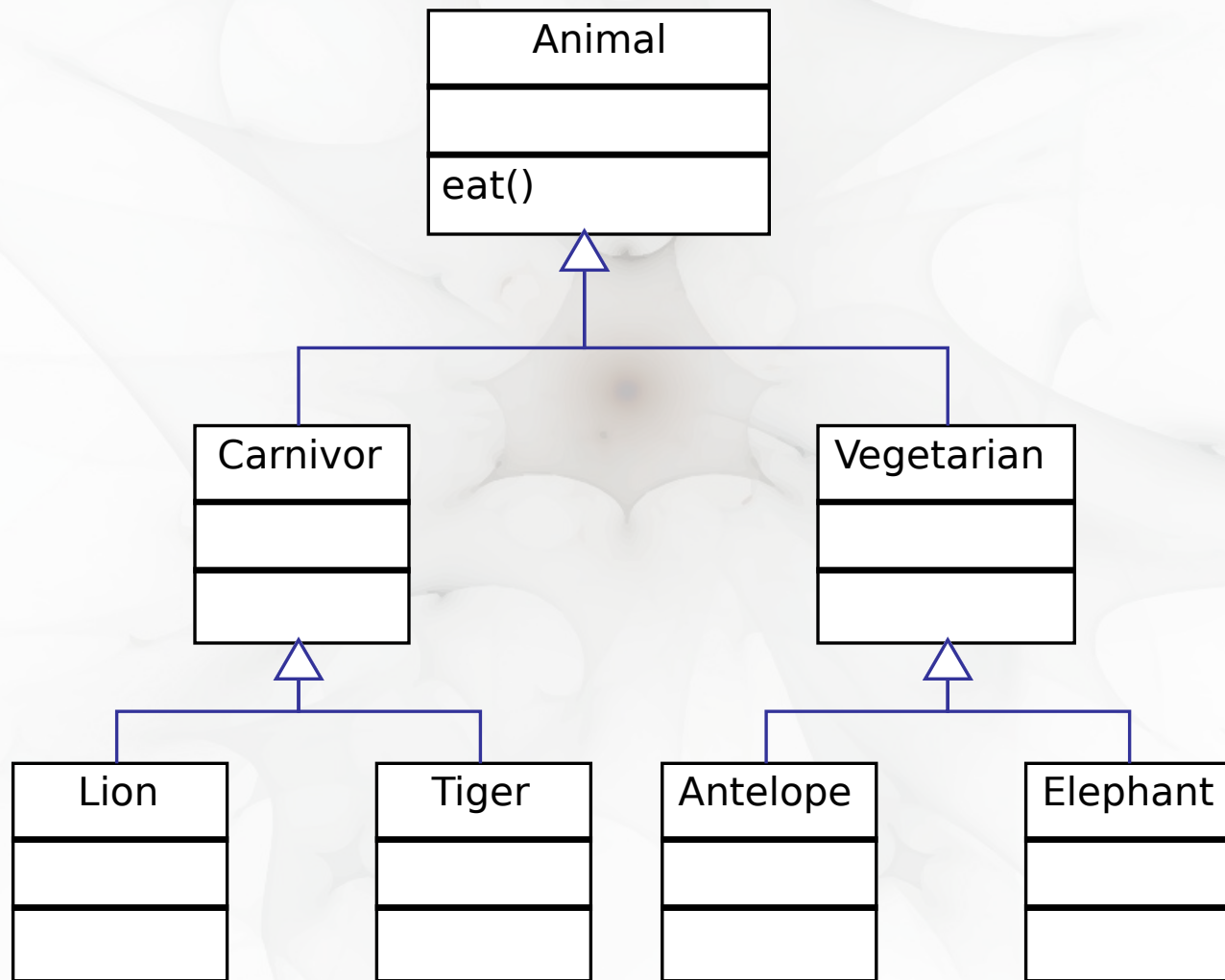
Orienté Objet : Héritage

- ✓ Une classe peut hériter d'une autre classe dans une relation de type "est un"
- ✓ Phénomène de spécialisation (override)

```
public class Animal {  
    public void eat() {}  
}  
  
public class Lion extends Animal {  
    public void eat() {  
        // eat meat !  
    }  
}
```

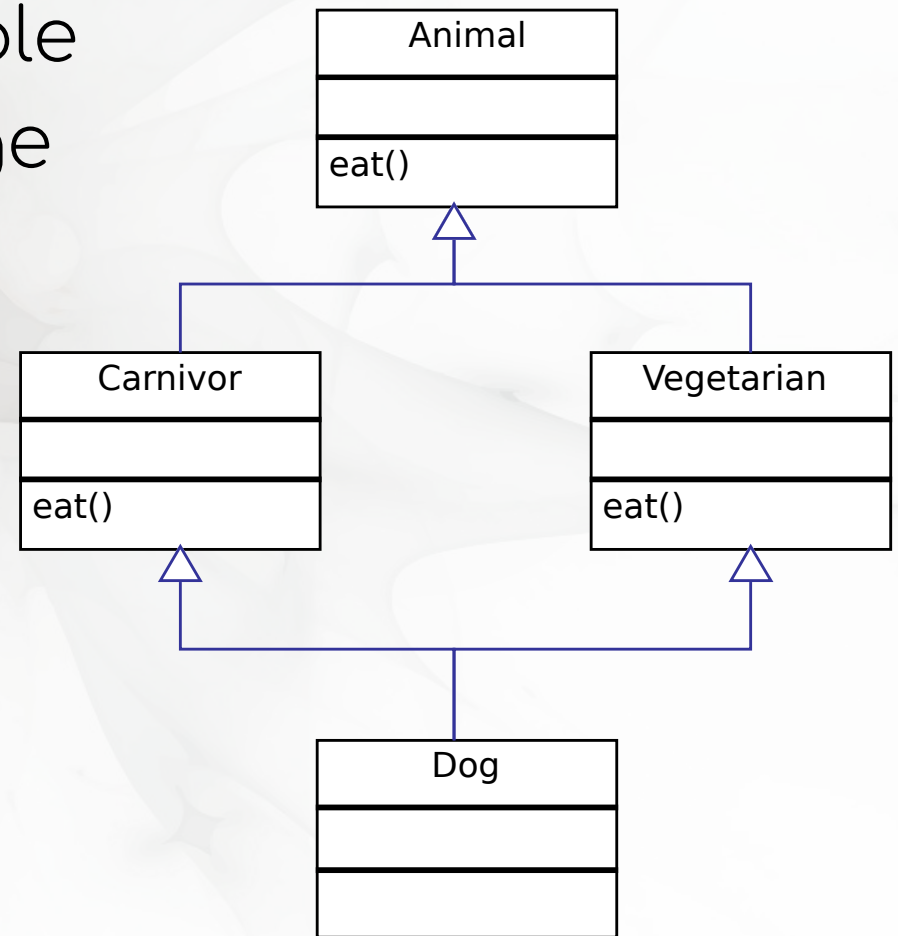


Hiérarchie de classes



Héritage multiple

- ✓ En C++ : héritage multiple
- ✓ Problèmes de masquage
- ✓ Les interfaces en Java contournent le problème :
- ✓ Pas d'états et pas de méthodes
- ✓ Classes abstraites sont un intermédiaire (mais toujours pas d'héritage multiple)



Duck typing

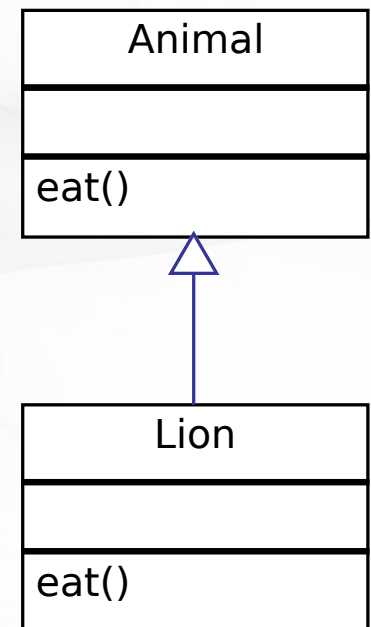
- ✓ Dans les langage au typage dynamique, forme de polymorphisme sans héritage

```
public class Dog {  
    public void eat() {}  
}  
  
public class Duck {  
    public void eat() {}  
}  
  
public void eat(Animal) {  
    animal.eat();  
}
```

Orienté Objet : Polymorphisme

- ✓ Le sous typage est la faculté de référencer la superclasse et d'en ignorer l'implémentation

```
Animal animal;  
// sub class is not known  
animal = new Lion();  
animal.eat();
```



Orienté Objet : Open recursion

- ✓ Faculté d'une méthode à appeler les méthodes membre de l'objet (mot clé **this** ou **self**)

```
public static class Recursion {  
  
    public void loop10(int i) {  
        if (i == 10) return;  
        // critère de sortie  
        this.loop10 (i+1);  
    }  
}
```


Union and Intersection types

- ✓ Cas de Ceylon
- ✓ Union (est du type X ou Y)



```
void write(String|Integer|Float printable) { ... }
```

- ✓ Intersection (a les interfaces X et Y)

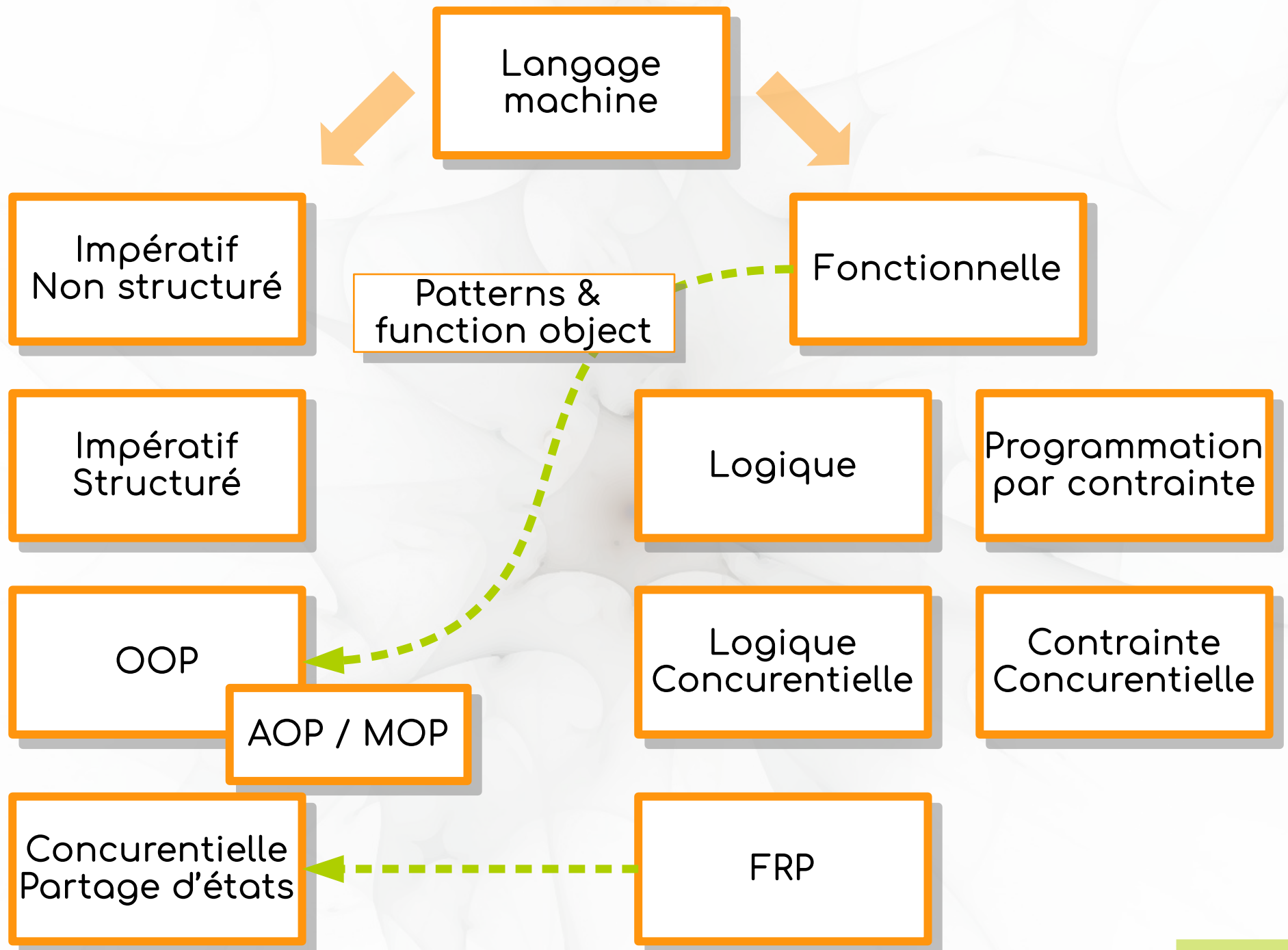
```
void store(Persistent&Printable&Identifiable obj)  
{ ... }
```

AOP, MOP

- ✓ MOP : Meta Object Protocol, possibilité de modifier la structure de la classe (ou du prototype) par programmation
- ✓ AOP : Séparation des responsabilités (on isole les problématiques transverses)
- ✓ AspectJ : Par injection de code à des points donnés (Tissage de greffons)
- ✓ $MOP + FP = AOP$

Impératif vs Fonctionnel

- ✓ Deux grandes familles :
 - ✓ Programmation impérative
 - ✓ Manipulation d'état (changement d'état effets de bord)
 - ✓ Décrit le "comment"
 - ✓ Programmation fonctionnelle
 - ✓ Manipule le comportement et réfute la mutation d'état
 - ✓ Programmation déclarative (le "quoi")



Programmation fonctionnelle

- ✓ Fonctions dites pures (sans états)
- ✓ Récursion

Lisp

```
(defun it (&optional (i 0))  
  (if (>= i 10)  
      i  
      (it (+ i 1))))
```


Programmation fonctionnelle

- ✓ Transparence révérencielle
 - ✓ pas de changement d'états

- ✓ Impératif :

```
x = x + 1
```

- ✓ Fonctionnel :

```
int plusone(int x) : return x + 1;
```

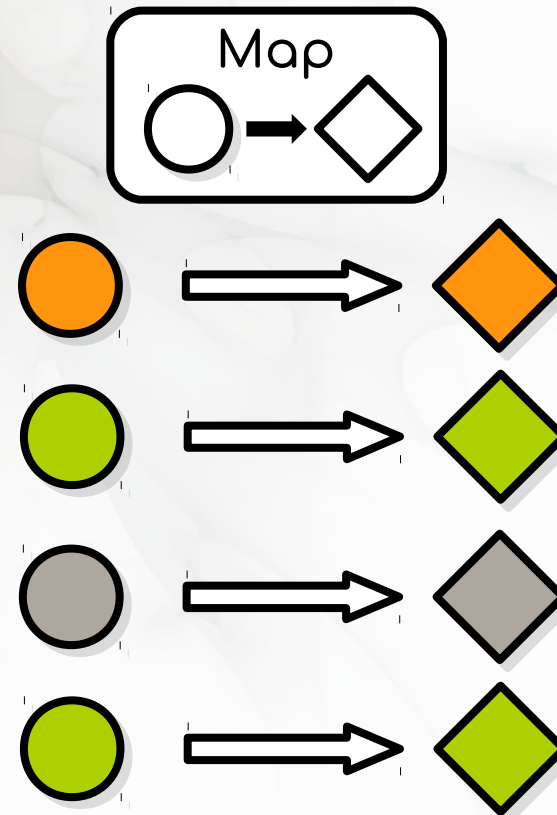
Programmation fonctionnelle

- ✓ Fonctions d'ordre supérieur :
 - ✓ Peuvent prendre une fonction en paramètre
 - ✓ Ou renvoyer une fonction comme résultat
 - ✓ Ou les deux
- ✓ Map – Filter – Fold (reduce) – Zip

Map

```
Stream.of(1, 2, 3, 4, 5)
    .map((item) -> "The number is : " + item)
    .forEach(System.out::println);
```

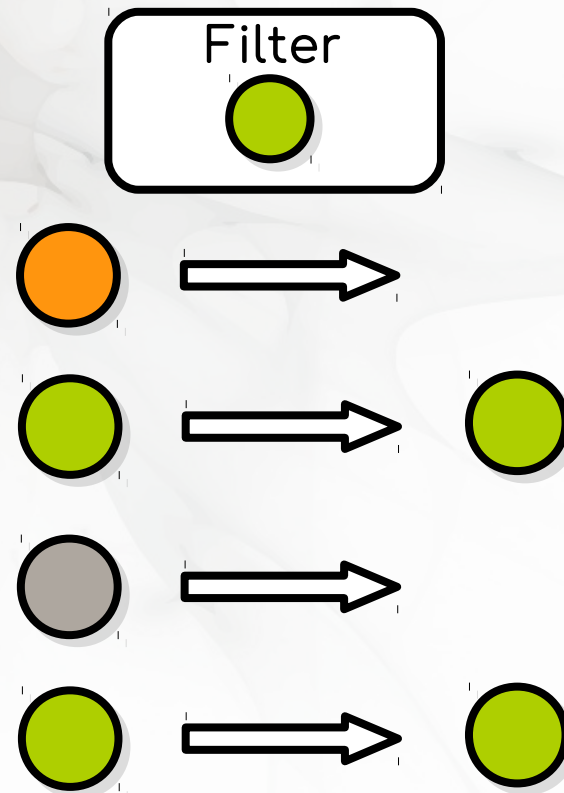
```
The number is : 1
The number is : 2
The number is : 3
The number is : 4
The number is : 5
```



Filter

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8)
      .filter((item) -> item % 2 == 0)
      .forEach(System.out::println);
```

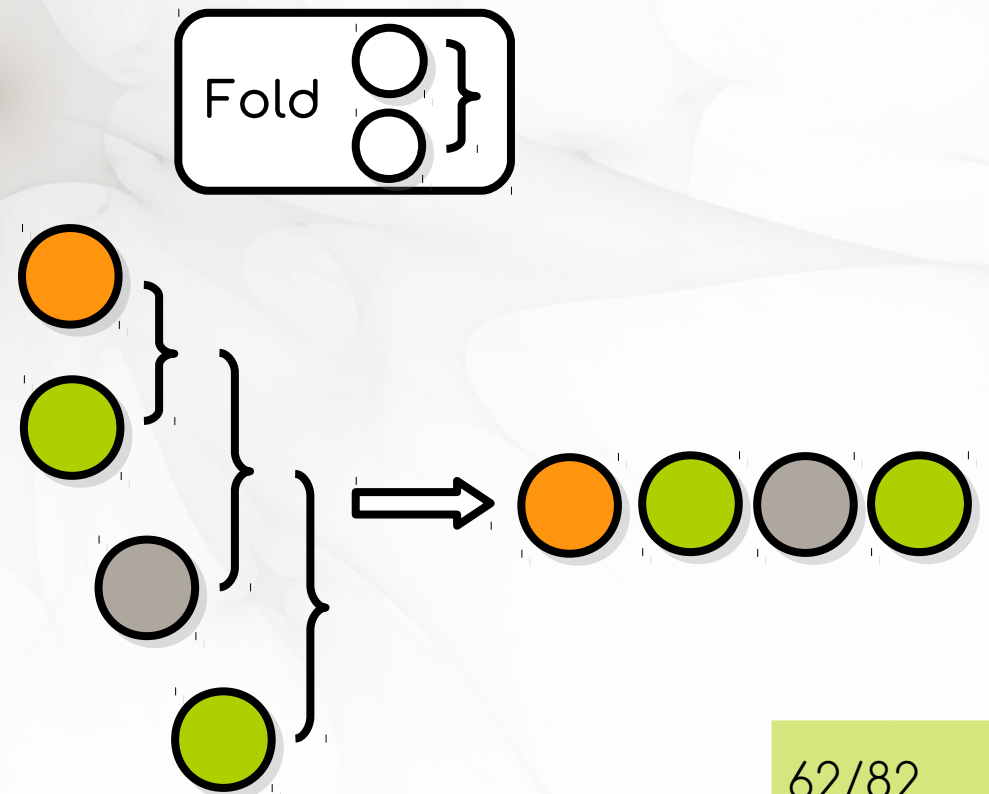
2
4
6
8



Reduce (fold)

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8)  
      .reduce((a, b) -> a + b).get()
```

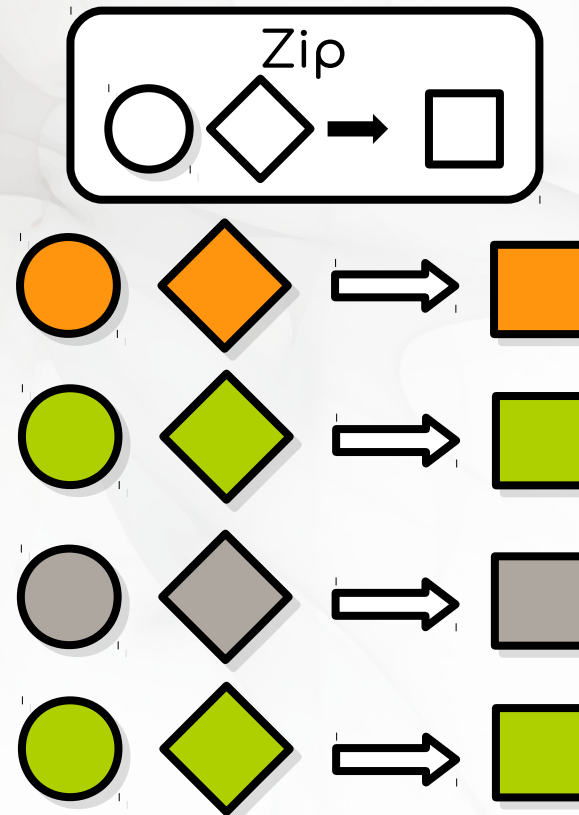
36



Zip

```
Stream.of(1, 2, 3, 4, 5)
    .map((item) -> "The number is : " + item)
    .forEach(System.out::println);
```

```
The number is : 1
The number is : 2
The number is : 3
The number is : 4
The number is : 5
```

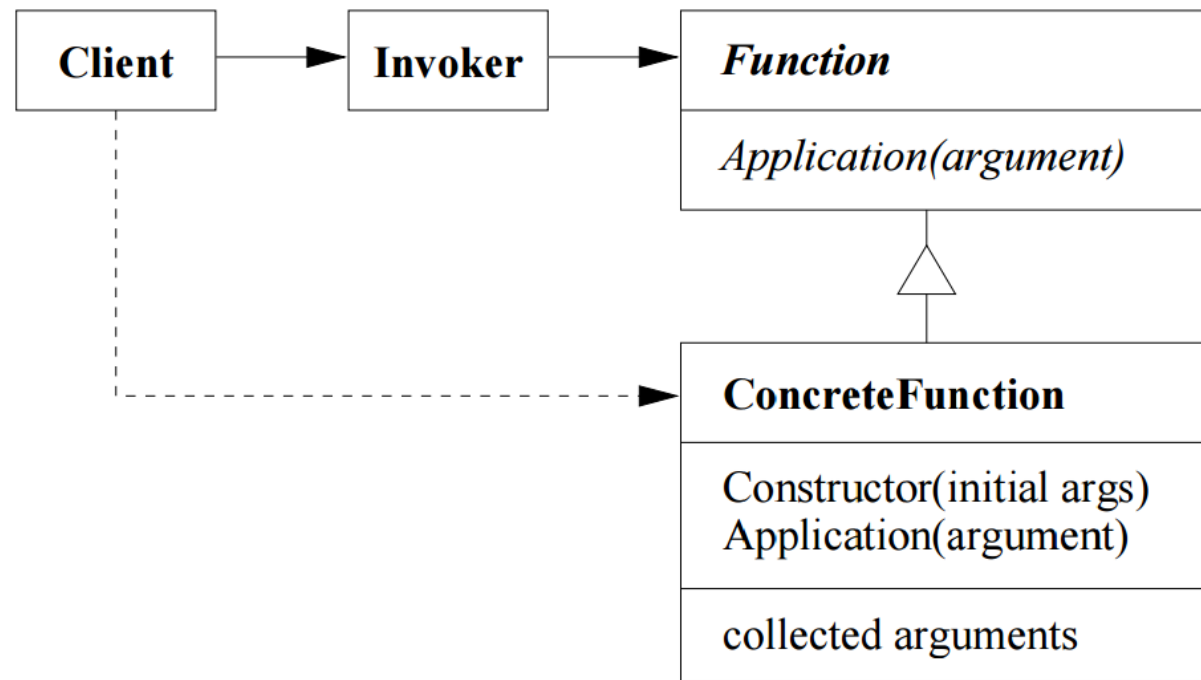


Function object

- ✓ But : Intégrer le paradigme fonctionnel dans les langages à Objets
- ✓ Généralisation du patron **Strategy** ou **Command**
- ✓ **Lambdas** = sucre syntaxique
- ✓ C# 3.0 (.NET Framework 3.5)
- ✓ Java 8

Function object

- ✓ Thomas Kühn - Patron de conception
- ✓ Un objet qui représente une fonction



En Java

```
List<String> names = Arrays.asList("Jean", "Paul", "Claude", "Charles");

Function<String, String> formater = new Function<String, String>() {

    @Override
    public String apply(String t) {
        return "The name is " + t + ".";
    }

};

names.stream().map(formater).forEach(new Consumer<String>() {
    @Override
    public void accept(String t) {
        System.out.println(t);
    }
});
```

Avec lambdas

- ✓ Concision
- ✓ Expressivité
- ✓ Lisibilité
- ✓ Programmation déclarative
- ✓ Debug difficile !

```
List<String> names = Arrays.asList("Jean", "Paul", "Claude", "Charles");  
Function<String, String> formater = (String t) -> "The name is " + t + ".";  
names.stream().map(formater).forEach(System.out::println);
```


Programmation par contrat

- ✓ Principe : ajouter des contraintes aux variables et méthodes
- ✓ Pré-condition : garantis la cohérence des valeurs d'entrée d'une fonction
- ✓ Post-condition : garantis la cohérence des valeurs de sortie d'une fonction
- ✓ Invariant : garantis que la valeur est cohérente tout au long de l'exécution (pour des variables)

Contrat en Java (Cofoja)

```
interface Time {  
    @Ensures({  
        "result >= 0",  
        "result <= 23"  
    })  
    int getHour();  
  
    @Requires({  
        "h >= 0",  
        "h <= 23"  
    })  
    @Ensures("getHour() == h")  
    void setHour(int h);  
}
```

Programmation logique par contrainte (PLC)

- ✓ Exemple : ProLog
- ✓ Énumération des contraintes du système
- ✓ Sous forme de règles logiques
- ✓ Idéal pour résoudre de problèmes de satisfaction : SAT-Solver
 - ✓ Comme le Sudoku
 - ✓ Problème des 8 dames
- ✓ Programmation déclarative par excellence

PLC en Java (Choco solver)

```
// 1. Create a Model
Model model = new Model("my first problem");

// 2. Create variables
IntVar x = model.intVar("X", 0, 5);           // x in [0,5]
IntVar y = model.intVar("Y", new int[]{2, 3, 8}); // y in {2, 3, 8}

// 3. Post constraints
model.arithm(x, "+", y, "<", 5).post(); // x + y < 5
model.times(x, y, 4).post();           // x * y = 4

// 4. Solve the problem
model.getSolver().solve();

// 5. Print the solution
System.out.println(x); // Prints X = 2
System.out.println(y); // Prints Y = 2
```

Programmation Réactive

- ✓ Maintenant que Java et C# ont introduit le paradigme fonctionnel
- ✓ Paradigme en cours de développement
- ✓ Rx.Net, Rx Java, Reactor, React4J
- ✓ <https://github.com/YannCaron/React4J/wiki>



RxJava



FRP - Idée

- ✓ Un programme impératif s'exécute de façon linéaire
- ✓ Les déclarations ne sont pas toujours juste dans le temps

```
Integer a = 0;  
Integer b = 0;  
Integer sum = a + b;
```

```
System.out.println("Sum = " + sum); // Sum = 0  
a = 7;  
b = 8;  
System.out.println("Sum = " + sum); // Sum = 0
```

FRP - React4J

```
final Var<Integer> a = new Var<>(0);  
final Var<Integer> b = new Var<>(0);  
Operation<Integer> sum = Operation  
    .mergeOperation(() -> a.getValue() + b.getValue(), a, b);  
// reactive sum = reactive a + reactive b
```

```
System.out.println("Sum = " + sum.getValue()); // Sum = 0
```

```
a.setValue(7);  
b.setValue(8);
```

Actualisé 2 x

```
System.out.println("Sum = " + sum.getValue()); // Sum = 15
```

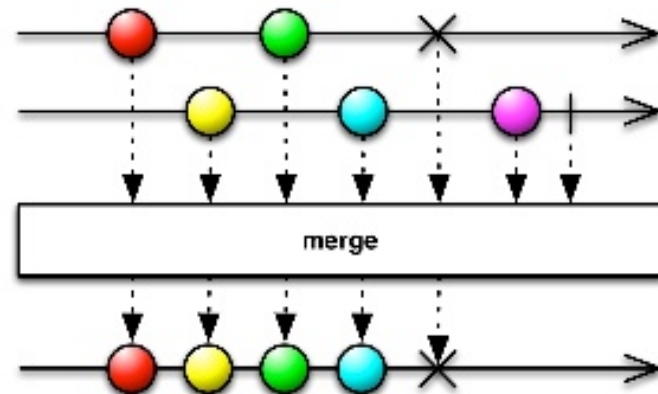
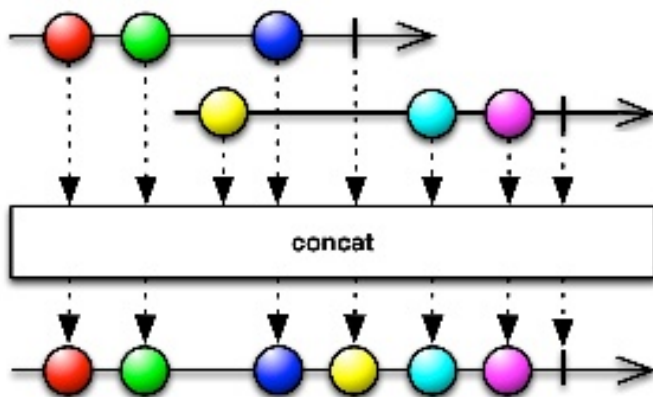
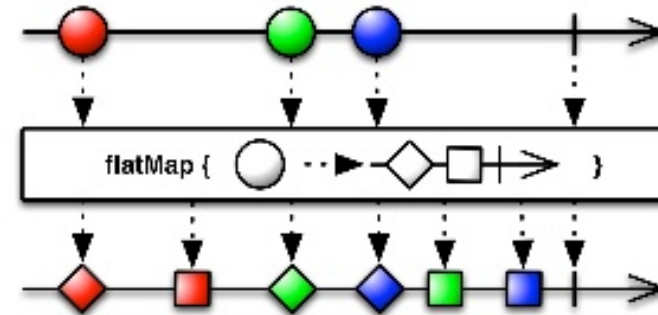
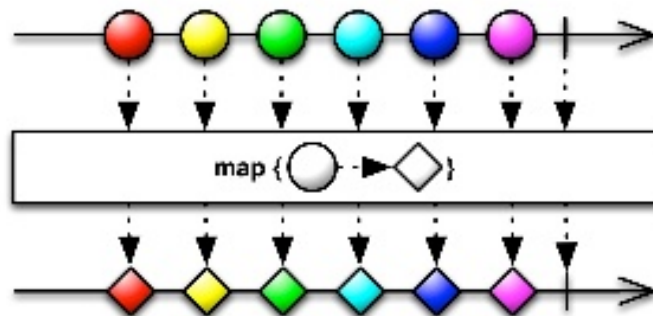
Toujours vérifiable

FRP - Concepts

- ✓ Réaction continue : un processus centralisé actualise périodiquement les signaux
- ✓ Réaction discrète : basé sur un modèle évènementiel. Les foncteurs sont chaînés. Chaque foncteur renvoie un nouveau réactif basé sur le précédent.

FRP - RxJava

RxJava operations as marble diagrams

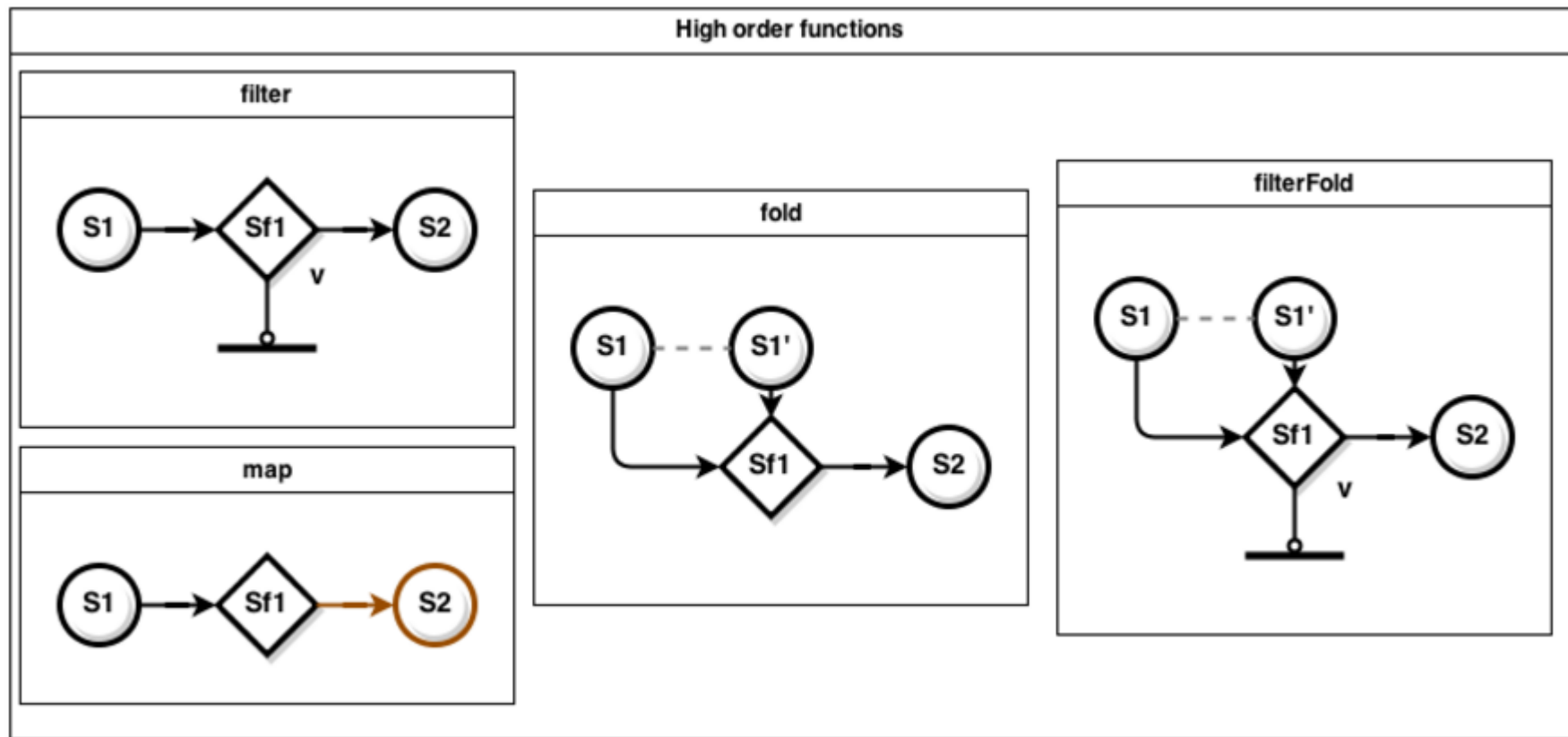


React4J - Exemple

```
Var<String> mouseAndTime = MouseReact.button1()  
    .map(arg1 -> arg1 ? "pressed" : "released")  
    .toVar("no button yet !")  
    .merge(MouseReact.positionX().toVar(0),  
        (arg1, arg2) -> arg1 + " ( x=" + arg2)  
    .merge(MouseReact.positionY().toVar(0),  
        (arg1, arg2) -> arg1 + ", y=" + arg2 + " )");  
  
label1.setText(mouseAndTime);
```

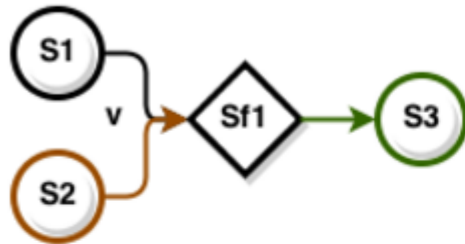
Toujours vérifiable

FRP - React4J

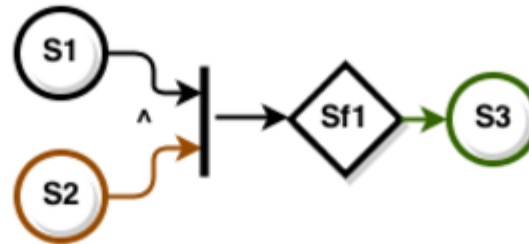


Combinations

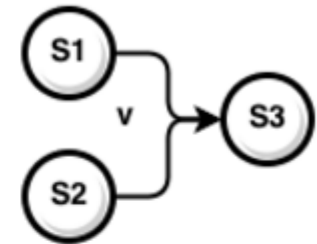
merge



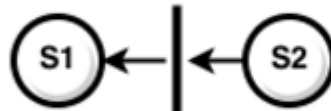
sync



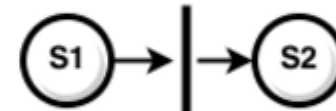
mergeSame



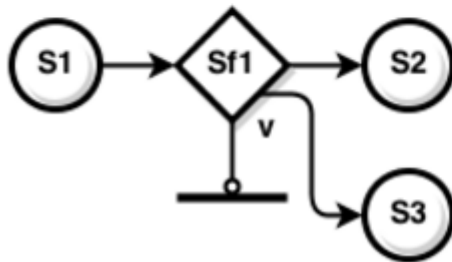
when



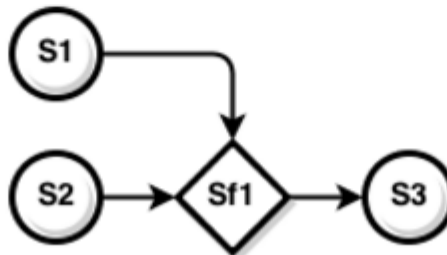
then



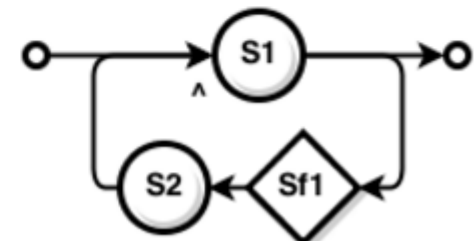
switchMap



edge

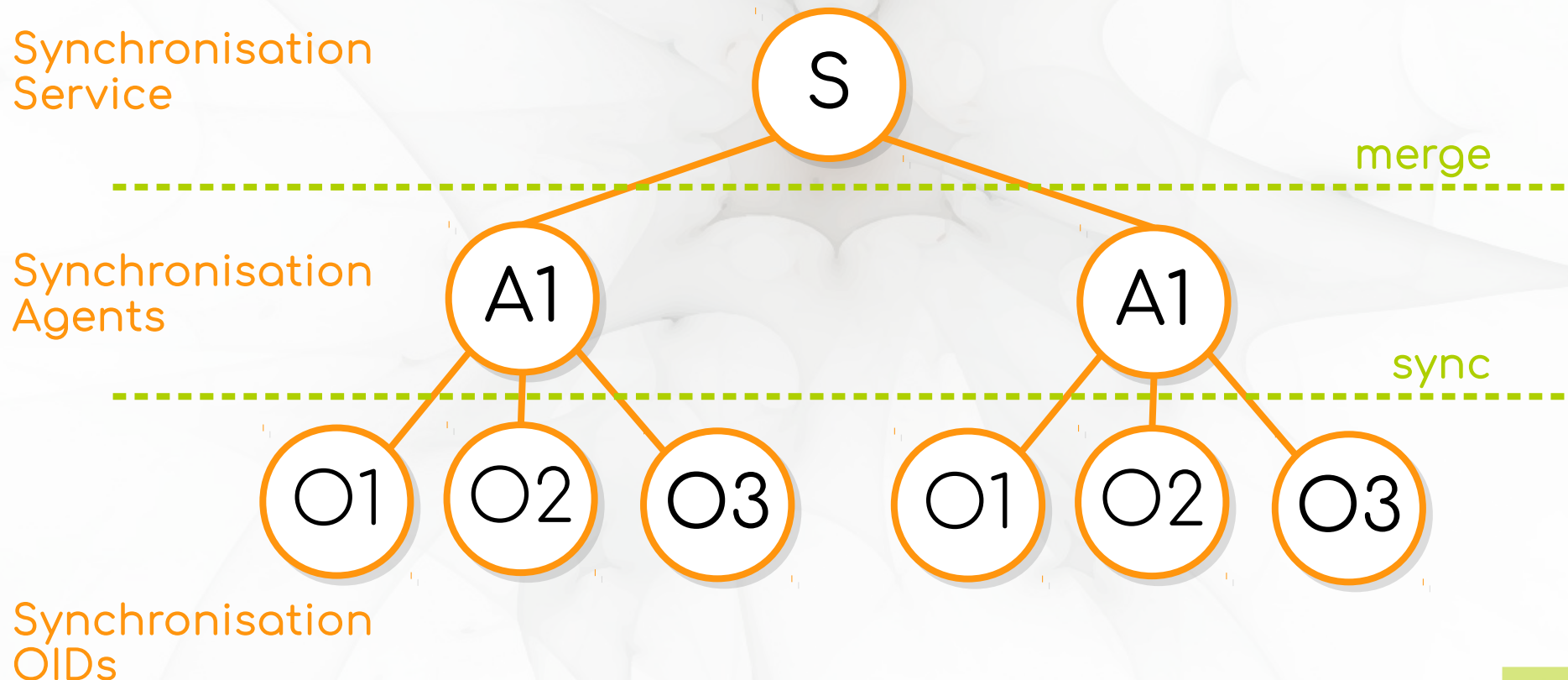


feedbackLoop

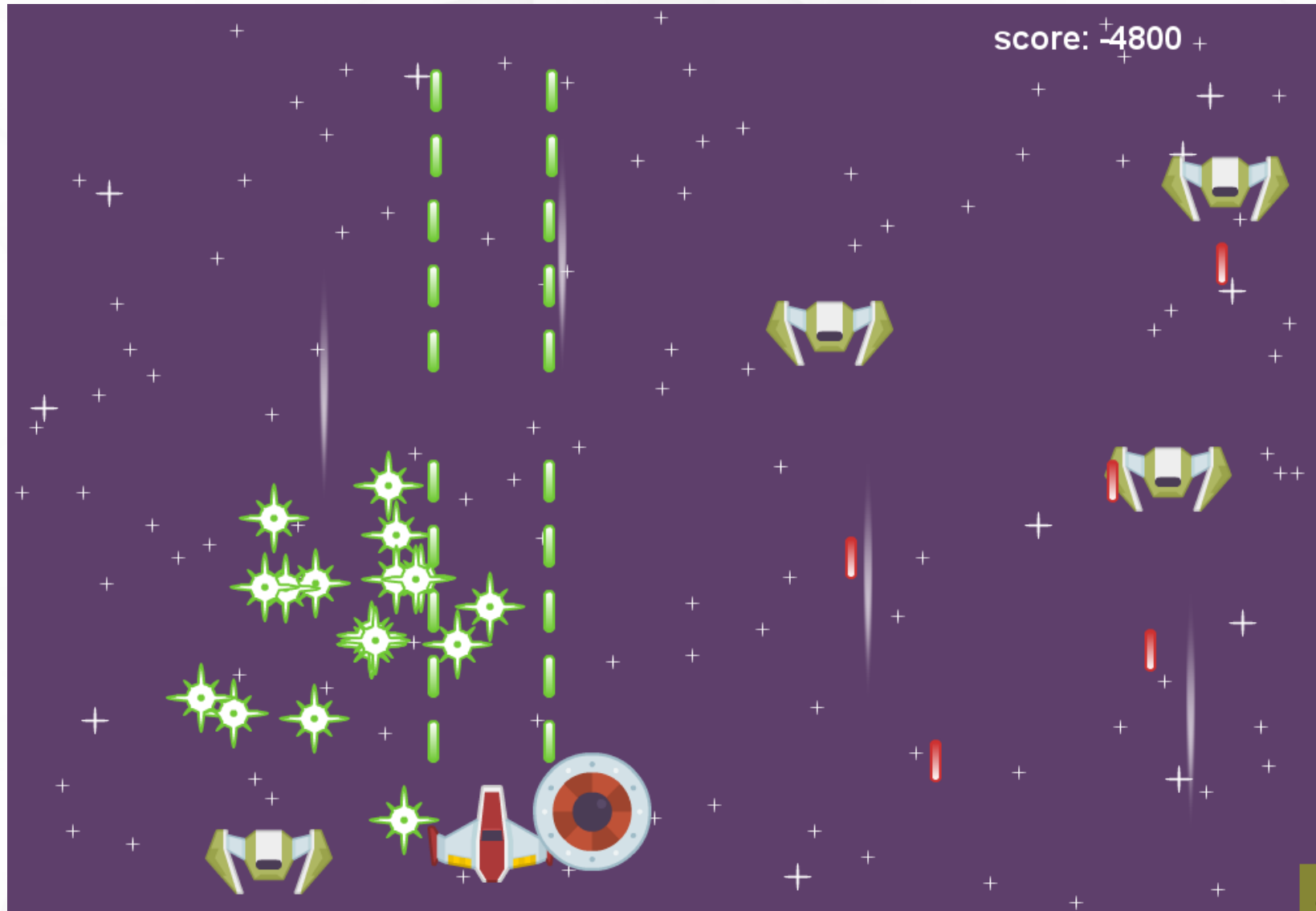


FRP – Cas d'utilisation

- ✓ Synchroniser des messages de serveurs SNMP distants



React4J - Démonstration



button pressed (x=372, y=515)

Number of circles: 136, Number of react: 7621

Merci de votre attention

