

# Programmation multi-paradigme

---

# Consignes et vie des cours / TD

- S'applique à tous les cours / TD Ménier (1<sup>er</sup> et 2<sup>nd</sup> semestre + M2)
- Entrée non acceptée si retard
- Téléphone éteint et ramassé / nourriture boissons interdite (& ramassées)
- Montres connectées
- Annulation note TD si
  - Boisson ou nourriture visible
  - Téléphone visible / utilisé
  - TD rendu en retard
    - Moodle : date/heure limite = rendre AVANT l'heure et la date PAS à l'heure et à la date. Avant
    - Pas de rendu par mail
  - Absences TD
- Exclusion des cours / TD / Moodle
  - Papote cours
  - Retards répétés
- Pas de prise de note PC / etc..
  - Sauf raison médicale + envoyer les notes prises dans la journée
- Interdiction stricte photo / vidéo

# Consignes et vie des cours / TD

- Les cours sont à comprendre, apprendre et à connaître
  - Contrôles sur l'apprentissage et compréhension des cours
- Les TDs sont à préparer
  - Contrôles sur la préparation du TD
- TD exclusivement (~~facebook~~ etc..)
- Les consignes données pour les TD sont à suivre **obligatoirement**
- **Tout le TD est à faire tel quel**

• **Cf Moodle : ent.univ-ubs.fr**

• **Clé d'inscription menierscala**

• Prise de notes

- $\frac{1}{4}$  des informations ne sont pas dans les transparents
- Ecrire 'imprime' également les informations
- Papier / crayon

• **Utilisation de Google**

• **Orthographe (pénalité en examen : jqa 2 points)**

## INF2162 Programmation multi-paradigme (Gildas Ménier)

[Tableau de bord](#) / [Mes cours](#) / [Faculté des Sciences](#) / [Campus de Tohannic - Départements MIS & SMV](#)  
/ [Département Mathématiques Informatique Statistique](#) / [INF2162](#)

# Programmation multi-paradigme

---

# Prérequis

- Java 8 (+ jar et écosystème Java)
  - Algorithmique
  - Sans Google et Sans ordinateur
  - GIT
- 
- IDE : Eclipse, Idea, etc **SAUF** si c'est un handicap (!)
  - Notepad + javac etc..

# Prérequis

```
List<Integer> numbers = Arrays.asList(2, 3, 6, 19, 120);
System.out.println(
    numbers.stream()
        .peek(e -> System.out.println())
        .filter(e -> e > 10)
        .filter(e -> e % 2 == 0)
        .map(e -> e * 2)
        .findFirst()
        .map(e -> "La valeur est " + e)
        .orElse("No value found"));
```

## Java de base

# Présentation des cours

- Manière de considérer l'exécution du programme
- Contrainte imposée par le langage de programmation
- Manière de penser la programmation
- Expressivité du langage
  - Écrire moins pour faire plus
  - Manière de penser adaptée au problème à résoudre
- Certains langages sont plus adaptés que d'autre à la résolution de certains problèmes
- Base de données : locale, distribuée, tolérante aux fautes
- Logistique : ET / optimisation
- Interaction homme / machine

# Présentation

- Multi-paradigme
- Le langage contient ce qu'il faut pour aborder le problème de manières différentes
- Maîtriser ces manières
- Java < 8: une manière de faire
  - Java 8 introduction (timide) de fonctionnel
- Algorithmique

# Présentation

- En fait, du bon sens
- Pas (plus) de bricolage
- Planification et étude du problème
- Faire évoluer le logiciel
- L'objectif n'est pas seulement de faire un programme qui 'tourne'
- Extensible
- Compréhensible
- Confiance
- Être capable d'expliquer ce que vous avez fait

# Présentation

- Langages
- **Java**
- Haskell
- **Scala**
- **Javascript**
- Autres
  - Inventor
  - Prolog
  - Erlang

# Paradigmes de programmation

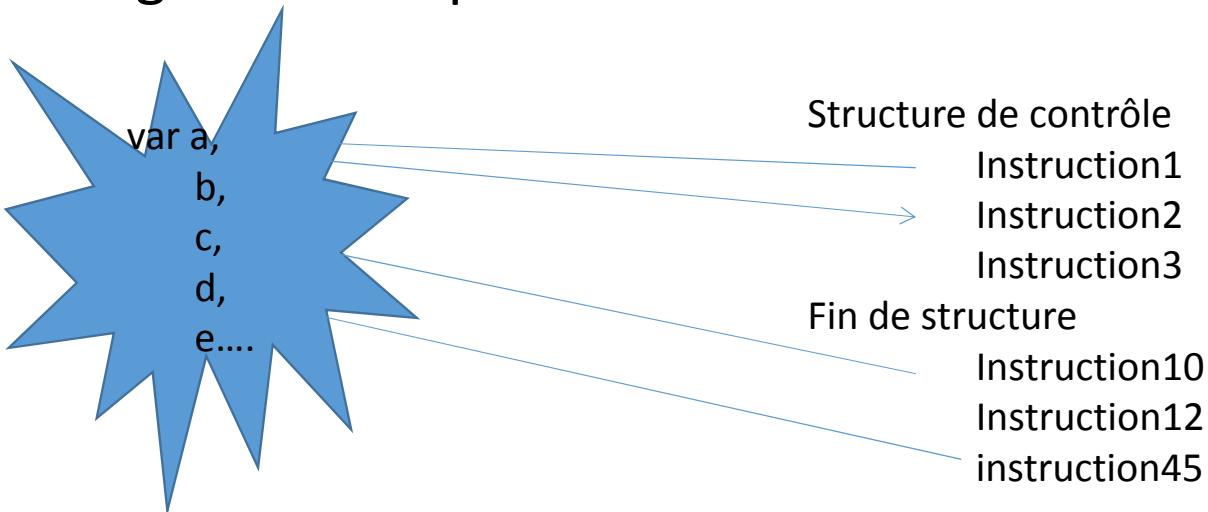
- Programmation impérative
  - Une séquence d'instruction change un état général
  - Détailler une suite d'instruction qui indique comment modifier des variables
  - Les variables contiennent les conditions de départ et leur évolution donne le résultat du programme
  - Démarche : trouver les structures de données et trouver une séquence d'instruction qui modifie l'état
  - Pas à pas -> diagnostique
  - Essai / erreur

# Paradigmes de programmation

- $A = A + 1$
- $A \leftarrow A + 1$
- $A + 1 \rightarrow A$
- $A // A + 1$
- $A + 1 \% A$
- $A, A + 1$

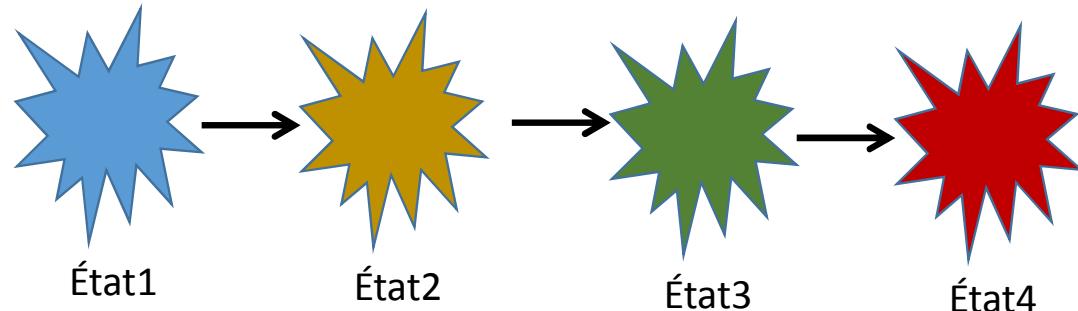
# Paradigmes de programmation

Cadre général : impératif



# Paradigmes de programmation

Cadre général : impératif



Chaque instruction modifie l'état de la mémoire

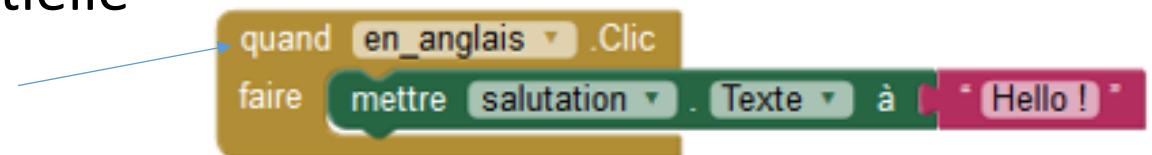
# Paradigmes de programmation

- Programmation structurée
  - Instructions de contrôle
  - Blocs de code
- Programmation procédurale
  - Procédures
- Programmation modulaire
- C++ / C / Java / PHP / Python / Ruby

# Paradigmes de programmation

- Programmation évènementielle

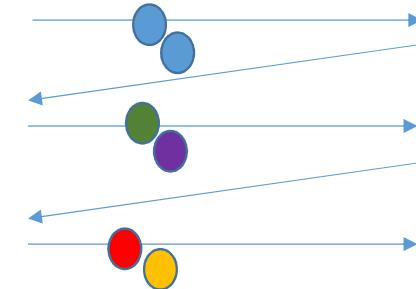
- Quand il se produit... Faire
    - fin



- Programmation séquentielle

- Notion de bloc d'exécution
  - Région spécifique
  - Le sens de l'exécution est important
  - De gauche à droite et de haut en bas

- Trouver une erreur revient à faire des hypothèses
  - À les vérifier



# Paradigmes de programmation

- Programmation orientée objet
  - Une collection d'objets en interaction via méthodes
  - Chaque objet est responsable de code
  - Chaque objet possède un état
- Programmation objet / orientée objet
  - Java / C++ : orienté objet
  - Smalltalk : programmation objet
    - Méthode = message envoyé à un objet
    - Paradigme : programmation par messages

Organisation  
Responsabilité  
(délégation)

# Paradigmes de programmation

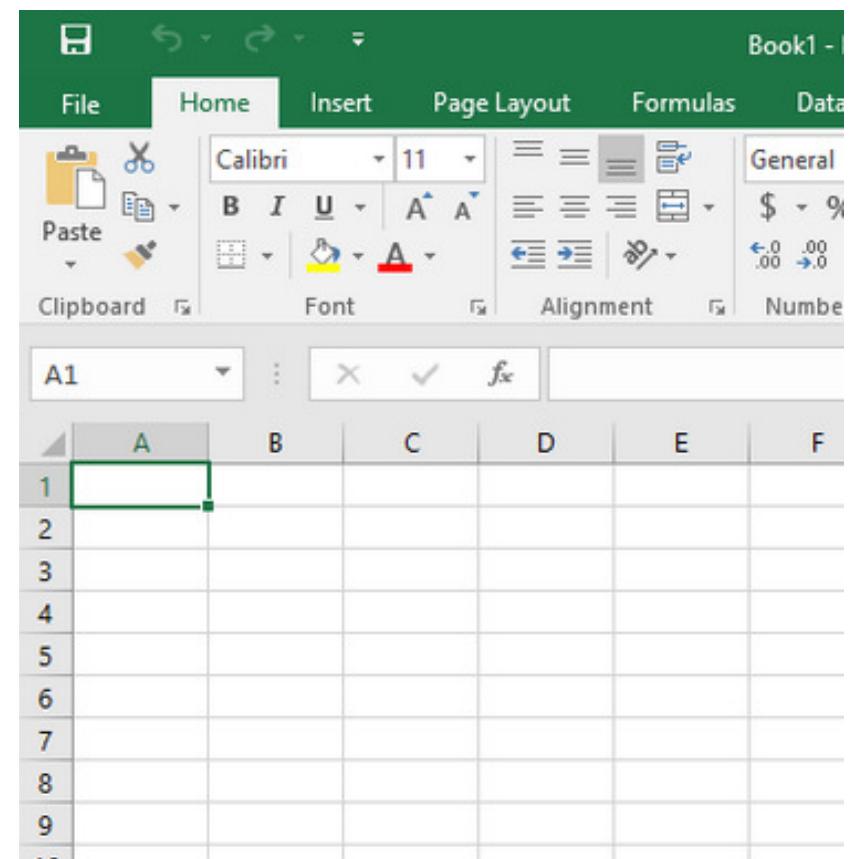
- Programmation par prototypes
  - Self / Javascript
  - Objet sans (nécessairement) classes
- Programmation chimique
  - Gamma
  - Transformation de multi ensembles
  - Exemples
  - Map/reduce
  - On ne s'intéresse pas au *comment*, mais au *quoi faire* !

# Paradigmes de programmation

- Programmation déclarative
  - Prolog
    - exemple
  - SQL
  - OWL
  - SPARQL
  - Expressions régulières

# Paradigmes de programmation

- Programmation par flots de données
  - Définition de dépendances entre données
  - $A = B + 1$ 
    - Programmation réactive
  - Excel
  - Angular (cf cours second semestre)
    - Modèle Vue Contrôleur
    - Reactif

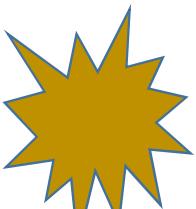


# Fonctionnel

- Pas d'état ?!
  - Le résultat d'un programme c'est la transformation des données de départ

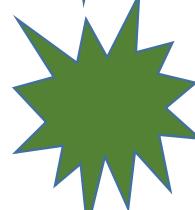


Données d'entrées

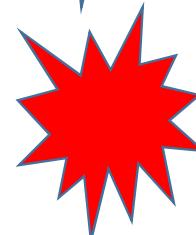


Données entrée

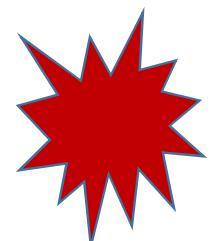
$$= f1(\text{blue starburst})$$



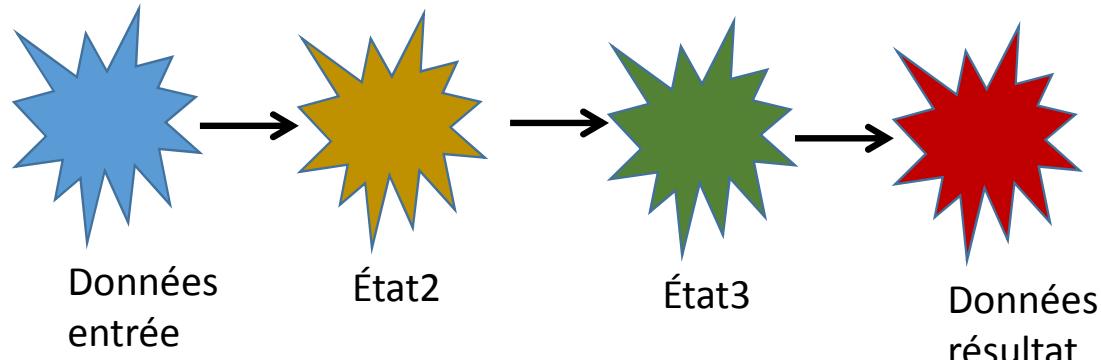
$$= f2(\text{yellow starburst})$$



$$= f3(\text{green starburst})$$

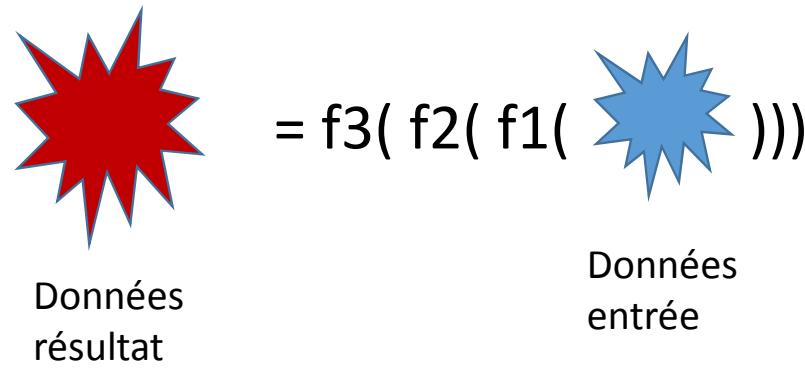


Données résultat



# Fonctionnel

- Pas d'état ?!
  - Le résultat d'un programme c'est la transformation des données de départ



- Pas besoin de variables
- Pas besoin de mémoire
- Distribution
- Fiabilité

# Paradigme récursif

- Boucle impérative
  - Parcourir explicitement chaque portion
  - Pas besoin de fonction
- Boucle récursive
  - Utiliser une fonction
  - Faire une petite partie, puis recommencer avec le reste

```
void compterJqa10APartirDe(int i) {  
    if (i<10) {  
        System.out.println(i);  
        compterJqa10APartirDe(i+1);  
    }  
}
```

compterJqa10APartirDe(0);

```
for( int i =0; i< 10; i++) {  
    System.out.println(i);  
}
```

# Paradigme récursif

- Récursivité

- Un exemple simple :

```
Fonction résoudreLeProblèmeSurEspace( espace ) {  
    siEncorePossible(espace) {  
        on utilise une partie U;  
        resoudreLeProblèmeSurEspace(espace -U)  
    }  
}
```

Résolution partielle  
Et on relance une résolution sur ce qui reste  
(on est certain de l'arrêt)



```
Fonction distribuerLeGateau( gateau ) {  
    siIlResteDu(gateau) {  
        on distribue une part  
        distribuerLeGateau(gateau-part)  
    }  
}
```

# Paradigmes de programmation

- Programmation récursive
- Programmation par multi agents
- Programmation par acteurs
- Programmation par contrainte
- Programmation non déterministe
- Programmation réflexive
- Programmation scalaire
- Programmation systolique
- Programmation par contrats
- Programmation orientée composants
- Programmation génétique
- Etc..

# Programmation multi-paradigme

- Développer

- préoccupations

méthode

Pas de méthode

- 1. ca marche ?

- tests
    - complexité

- 2. Extensibilité

- si je veux rajouter une fonctionnalité, est-ce que je dois tout modifier ?
        - proportion ?
        - Erreurs introduites

# Programmation multi-paradigme

- Développer
  - préoccupations
  - 3. Modularité
    - si je modifie un fichier, est-ce que ca a une conséquence pour les autres fichiers ?
      - réutilisation
      - travail en équipe
      - limiter la diffusion des erreurs
      - cascades

# Programmation multi-paradigme

- Développer
  - préoccupations
    - 4. Réutilisabilité
      - Beaucoup de duplication de code ?
      - Documentation ?
      - Qualité ?

# Programmation multi-paradigme

- Développer
  - préoccupations
    - 5. Testabilité
      - Facile à tester ?
      - Certitude ?
      - Couverture de test ?

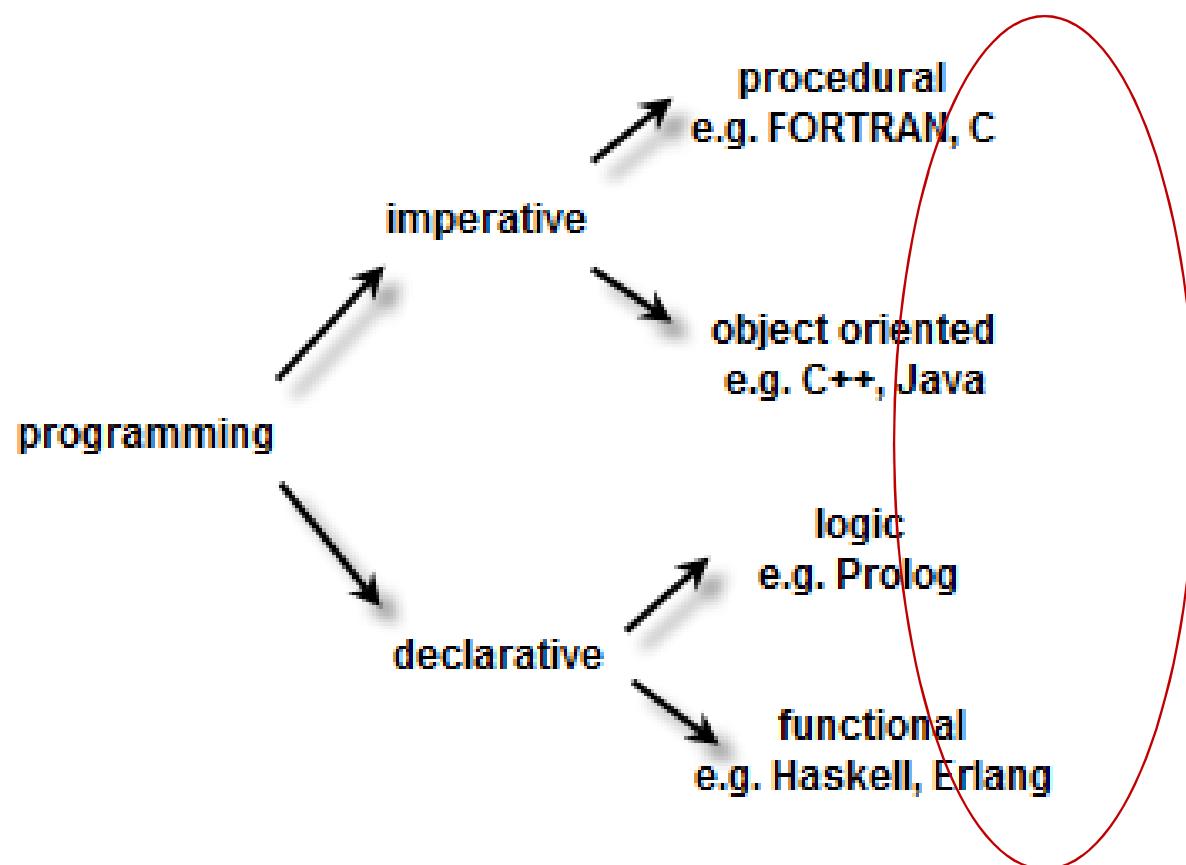
# Programmation multi-paradigme

- Développer
  - préoccupations
    - 6. Clarté
      - La structure est facile à comprendre ?
      - facile à expliquer ?
      - simple à documenter ?

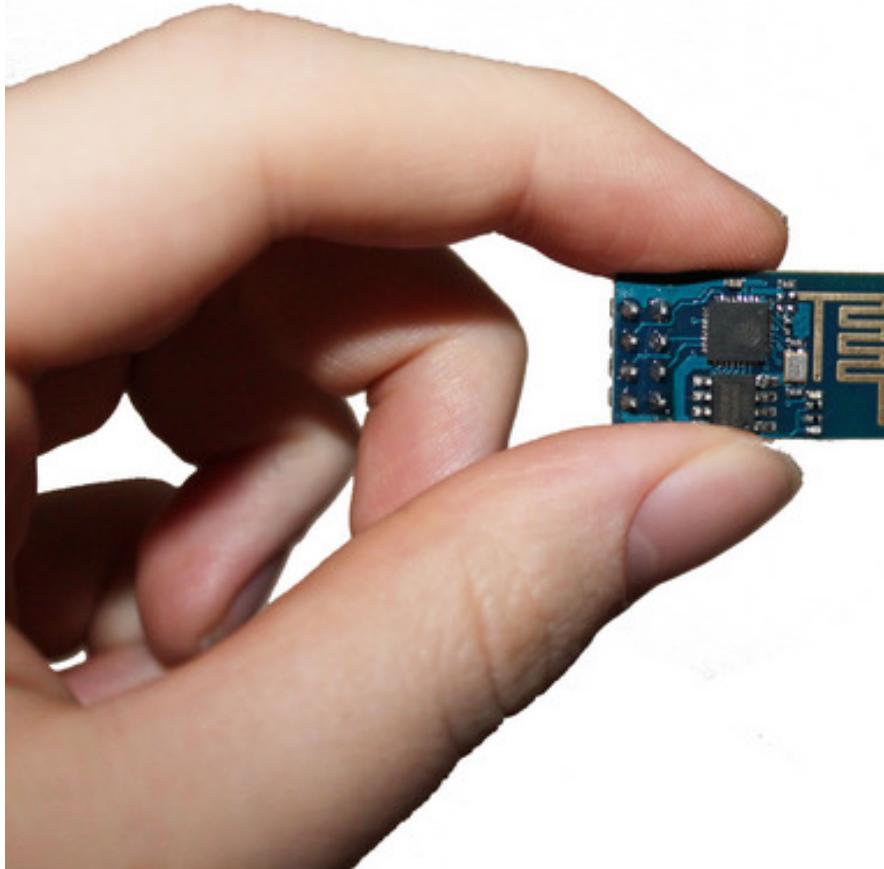
# Programmation multi-paradigme

- Développer
  - préoccupations
    - Extensible ?
    - Modulaire ?
    - Réutilisable ?
    - Testable ?
    - Clair ?
  - Programmation fonctionnelle
  - Programmation impérative
  - Programmation déclarative

... multiparadigme



# Paradigmes de programmation



```

cfg={}
cfg.ssid="ESP_MENIER"; cfg.pwd="12345678";
wifi.ap.config(cfg)

cfg={}
cfg.ip="192.168.1.1"; cfg.netmask="255.255.255.0"; cfg.gateway="192.168.1.1";
wifi.ap.setip(cfg);
wifi.setmode(wifi.SOFTAP)

srv=net.createServer(net.TCP)

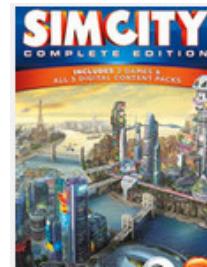
srv:listen(80,
    function(conn)
        conn:on("receive",
            function(conn)
                local res=<h1>Salut les M1</h1><br/><H2>Cette page est envoyée par l'ESP8266</H2>
                conn:send(res) end)
        conn:on("sent",
            function(conn) conn:close() end)
    end)

```



# Langage multi-paradigme

- Problème de conception
- Java
- JVM
- Réutilisation du code
- ... formation des développeurs



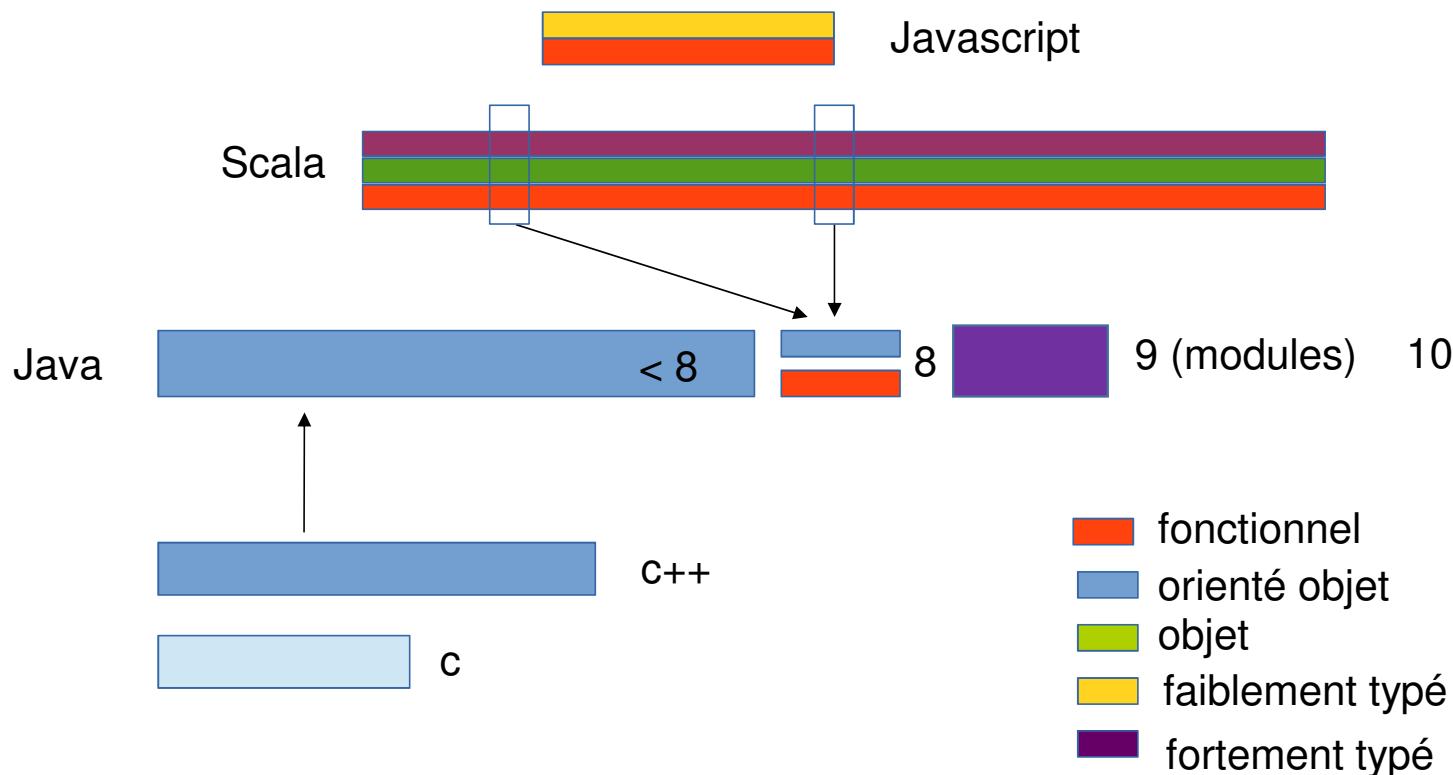


# Java rappels

(Java 5, Java 8, Java 9, Java 10)

**Le niveau d'entrée requis au niveau Master est Java 8**

# Java et Java 8



# Java 5 : généricité



- Généricité
  - À partir de Java 5
  - Méthode qui s'applique sur
    - *n'importe quel type*
  - Une méthode par type
  - Idem pour les collections
    - Tableau, liste etc..

# Java 5 : généricité



- Généricité

- Types primitifs

```
void afficheInt(int i) { System.out.println(i) ; }  
void afficheInt(double i) { System.out.println(i) ; }
```

..

- Objets

```
void afficheClasse1(Classe1 i) { ... }  
void afficheClasse2(Classe2 i) { ... }
```

- Polymorphisme

```
void affiche(Object i) { ... }
```

# Java 5 : généricité



- Collections

- ArrayList contient des Objects

```
List list = new ArrayList();
```

```
list.add(new Integer(2));  
list.add("hello");
```

- Retrouver le type par cast

```
Integer a = (Integer) list.get(0);
```

```
String chaine = (String) list.get(1);
```

# Java 5 : généricité



List  
↓  
ArrayList

List<String>  
↓  
ArrayList<String>

- Généricité
  - ArrayList de Strings :

```
List<String> chaines = new ArrayList<String>();  
chaines.add("bonjour");  
String uneChaine = chaines.get(0);
```

List est un type paramétré

- Un seul type autorisé pour la collection
- Héritage

# Java 5 : généricité



- Généricité
  - Itération

```
List<String> chaines = new ArrayList<String>();  
  
chaines.add("bonjour");  
  
for(String uneChaine : chaines){  
    System.out.println(uneChaine);  
}
```

- while, for etc..
- for 'connaît' la taille de la liste
- PAS D'INDICE

# Java 5 : généricité



- Généricité

- Itérateurs :

- Outil (méthode) pour parcourir les éléments d'une collection
    - On n'indique pas d'indice
      - Indépendant de la représentation interne mémoire
      - **Ordre conservé ou pas**
      - **Local ou pas**
      - Parallèle ou pas
    - Très différent d'un parcours élément par élément

Sucre syntaxique

# Java 5 : généricité



```
List<String> chaines = new ArrayList<String>();  
  
chaines.add("bonjour");  
  
for(String uneChaine : chaines){  
    System.out.println(uneChaine);  
}
```

Ou bien :

```
List<String> chaines = new ArrayList<String>();  
chaines.add("bonjour") ;  
  
Iterator<String> iterateur = list.iterator();  
  
while(iterateur.hasNext()){  
    String uneChaine = iterateur.next();  
    ...  
}
```

fichier ?

# Java 5 : généricité



- Généricité et polymorphisme
- Ou encore :

```
Set<String> ens = new HashSet<String>();
for(String uneChaine : ens) {
    System.out.println(uneChaine);
}
```
- Quand on utilise une collection plutôt qu'un tableau, les itérateurs permettent de ne pas faire d'hypothèse sur la manière d'accéder aux éléments
- Le code devient plus générique, il dépend moins du type
- Plus facile à maintenir ou à faire évoluer
- Laissez définitivement tomber les tableaux et les boucles à indice*

```
Set<String> ens = new HashSet<String>();
Iterator<String> iterateur = ens.iterator();

while(iterateur.hasNext()){
    String uneChaine = iterateur.next();
}
```

# Java 5 : généricité



–Langage de haut niveau :

- On se préoccupe moins de la représentation des données
  - Indication, espace mémoire, désallocation, locale ou pas etc..
- On s'intéresse plus à ce qu'on veut faire avec les données
- La déclaration des données impose un type
- Les opérateurs savent comment gérer ce type
- Le développeur se concentre sur ce qu'il veut faire, par sur la manière de le gérer en mémoire, accès etc.

# Java 5 : généricité



## –Plusieurs paramètres de type

```
Map<Integer, String> table = new HashMap<Integer, String>();

Integer clé = new Integer(123);
String valeur = "soleil";

table.put(clé, valeur);

String résultat = table.get(clé);

Iterator<Integer> iterateurSurClé = table.keySet().iterator();

while(iterateurSurClé.hasNext()){
    Integer uneClé = iterateurSurClé.next();
    String uneValeur = map.get(uneClé);
}
```

# Java 5 : généricité



## –Ou encore

```
Map<Integer, String> table = new HashMap<Integer, String>;  
  
Integer clé = new Integer(123);  
String valeur = "soleil";  
  
table.put(clé, valeur);  
  
for(Integer uneClé : table.keySet()) {  
    String uneValeur = table.get(uneClé);  
    System.out.println("'" + uneClé + ":" + uneValeur);  
}  
  
for(String uneValeur : table.values()) {  
    System.out.println(uneValeur);  
}
```

# Java 5 : généricité



## –Créer une classe générique

```
public interface Paire<K, V> {  
    public K getClé();  
    public V getValeur();  
}  
  
public class PaireOrdonnée<K, V> implements Paire<K, V> {  
  
    private K clé;  
    private V valeur;  
  
    public PaireOrdonnée(K clé, V valeur) {  
        this.clé = clé;  
        this.valeur = valeur;  
    }  
  
    public K getKey() { return clé; }  
    public V getValue() { return valeur; }  
}
```

# Java 5 : généricité



## –Créer une méthode générique

```
public static <T> T addAndReturn(T element, Collection<T> collection){  
    collection.add(element);  
    return element;  
}
```

- Type de retour : T
- Ajoute un élément de type T à une collection (de type T) et donne cet élément en résultat

Commencez à raisonner en terme de collection et plus de tableau ou de liste

# Java 5 : généricité



## –Inférences des types

```
String stringElement = "stringElement";  
List<String> stringList = new ArrayList<String>();
```

```
String unElement = addAndReturn(stringElement, stringList);
```

**Pas toujours possible**

Le param type n'est pas indiqué

```
Integer integerElement = new Integer(123);  
List<Integer> integerList = new ArrayList<Integer>();
```

```
Integer autreElement = addAndReturn(integerElement, integerList);
```

```
Integer autreElement = addAndReturn<Integer>(integerElement, integerList);
```

# Java 5 : généricité



–Rajouter à votre classe le mécanisme Itérateur

```
public class MaCollection<E> implements Iterable<E>{  
  
    public Iterator<E> iterator() {  
        return new MonIterateur<E>();  
    }  
}  
  
public class MonIterateur <T> implements Iterator<T> {  
  
    public boolean hasNext() {  
        ...  
    }  
  
    public T next() {  
        ...  
    }  
}
```

# Java 5 : généricité



```
public static void main(String[] args) {  
    MaCollection<Integer> entiersCollection = new MaCollection<Integer>();  
  
    for(Integer entier : entiersCollection){  
        ...  
    }  
}
```

## –Remarque :

```
public static void main(String[] args) {  
    MaCollection<MaCollection<String>> cCollection  
        = new MaCollection<MaCollection<String>>();  
  
    for(MaCollection<String> uneCollection : cCollection){  
        ...  
    }  
}
```

# Java 5 : généricité



## –Types ? dans les paramètres

```
void methode(List<? Extends MaClasse> lst)
```

```
HashMap<?,?> maMap = getMapFromDisk(...)
```

```
void methode(List<? Super autreClasse> lst) { ... }
```

## –Capture de type

## –Seulement dans les déclarations ou paramètres méthodes

# Java 8



- Mise à jour conséquente de Java
- (très) influencée par Scala
- Tentative de fonctionnel (mais pas fonctionnel)
- Limitations de la syntaxe
- Limitations de la JVM
- Sucre syntaxique
- Les sécurités intégrées au langage sont des obstacles pour l'évolution du langage...

ref. Ben Witterbe

# Java 8



- Méthodes *default* pour les interfaces
- normalement
  - interface = classe abstraite Java
    - complètement abstraite
    - *mais* Java 8 : définitions par défaut
  - s'inspire des *traits*

# Java 8



## • Exemples

```
interface Formule {  
    double calculer(int a);  
  
    default double sqrt(int a) {  
        return Math.sqrt(a);  
    }  
}
```

```
Formule maFormule = new Formule() {  
    @Override  
    public double calculer(int a) {  
        return sqrt(a * 100);  
    }  
};
```

```
maFormule.calculer(100); // 100.0  
maFormule.sqrt(16); // 4.0
```

# Java 8



- Lambda expressions

- 1930 Alonzo Church
- Langage de programmation théorique
- Manipuler des fonctions (exclusivement)

- $(\lambda x.E) P$

- *E est une expression*

- $(\lambda x.E) P$  est la même que  $E$  dans laquelle on a remplacé les occurrences de  $x$  par  $P$

- $(\lambda x.E)$  est une fonction

- $(\lambda x.(x+1))$  est une fonction telle que  $(\lambda x.(x+1)) P \Rightarrow (P+1)$  (appliquée à  $P$ )

- $(\lambda x.E).apply(P)$  ou  $(\lambda x.E) P$  (voir Scala)

# Java 8



- Lambda expressions

- $(\lambda x.x)$
- $(\lambda x.y)$
- $(\lambda x.(x*8))\ 5 \Rightarrow (5*8)$
- $(\lambda x.(x*8))\ (\lambda x.(x+1)) \Rightarrow ((\lambda x.(x+1)) *8) \Rightarrow (\lambda x.((x+1)*8))$

- Composition des fonctions :

- On peut passer une valeur à une fonction pour obtenir un résultat
- On peut passer une fonction à une fonction pour obtenir un résultat
  - Le résultat est une fonction

- En fait, si on prend en compte les fonctions constantes

- Le résultat est toujours une fonction

- Currying (impossible en Java / possible en Scala) : voir Wikipedia

# Java 8



## •Lambda expressions et fonctions

- $\lambda x.(x^*2)$       typé  
int foisDeux(int x) {  
    int res = x \*2 ;  
    return res ;  
}

- Normalement, une substitution :
- $x$  est évalué et remplacé par sa valeur

1 2

### • *appel par valeur*

```
foisDeux(4) {  
    int res = 4 *2 ;  
    return res ;  
}
```

- *x* n'est pas une variable (mais peut être implémenté comme tel)

# Java 8



- Attention aux effets de bord

```
int foisDeux(int x) {  
    x = x*2; // bêtise  
    return x ;  
}
```

x est remplacé par sa valeur

```
foisDeux(4) {  
    4 = 4 *2 ;  
    return 4 ;  
}
```

- **NE PAS UTILISER LES ARGUMENTS COMME VARIABLE !!!**
- **LES ARGUMENTS NE SONT PAS DES VARIABLES**
- **TRES GROSSE BETISE !**

- *Par rapport au lambda calcul, en Java x est un objet ou de type intrinsèque*
- *x n'est pas une fonction*
- *Pour pouvoir passer une fonction, on transforme x en une instance qui contient la fonction*
- **EN JAVA SEULEMENT !!! (Java n'est pas un langage fonctionnel)**

# Java 8



- Pseudo lambda expressions
  - Fonctionnel
- Exemple de tri classique Java (< 8)
- Passer une fonction en argument (impossible)

```
List<String> noms = Arrays.asList("pierre", "bob", "jim", "carla");
```

```
Collections.sort(noms,  
    new Comparator<String>() {  
        @Override  
        public int compare(String a, String b) {  
            return b.compareTo(a);  
        }  
    }  
);
```

*(modifié en place)*



java.util

**Interface Comparator<T>**

Type Parameters:

T - the type of objects that may be compared by this comparator

# Java 8



- Exemple de tri

```
Collections.sort(noms,
```

```
(String a, String b) -> {  
    return b.compareTo(a);  
}
```

```
);
```

Quel est le type de ?

```
Collections.sort(noms,
```

```
new Comparator<String>() {  
    @Override  
    public int compare(String a, String b) {  
        return b.compareTo(a);  
    }  
};
```

```
);
```

... *bricolage* syntaxique et réécriture de code par le compilateur

# Java 8



## ● Contraction

**À cause de sort, le compilateur peut déterminer  
les types de la déclaration**

```
Collections.sort(noms, (String a, String b) -> b.compareTo(a));
```

**Dans certains cas, le compilateur peut déterminer  
les types des paramètres**

```
Collections.sort(noms, (a, b) -> b.compareTo(a));
```

**Comment ?**

# Java 8



- Mécanisme
  - interface + *default*
  - une seule méthode abstraite
  - remplissage automatique par le compilateur
  - En gros, on a introduit le ‘default’ dans l’interface pour permettre le pseudo lambda calcul

# Java 8



## • Exemple

*annotation pour que le compilateur vérifie bien qu'il n'y a qu'une seule méthode abstraite*  
**IMPORTANT !**

**@FunctionalInterface**

```
interface Convertisseur<F, E> {  
    E convertir(F valeur); // méthode abstraite (convertir F -> E)  
}
```

```
Convertisseur<Float, Float> euro_francs = (de) -> de*6.35;
```

```
Float res = euro_francs.convertir(1.50);
```

```
System.out.println(res);
```

```
! Error:(10, 61) java: incompatible types: bad return type in lambda expression  
        double cannot be converted to java.lang.Float  
!  
! Error:(12, 43) java: incompatible types: double cannot be converted to java.lang.Float
```

# Java 8



## • Exemple

@FunctionalInterface

```
interface Convertisseur<F, E> {  
    E convertir(F valeur); // méthode abstraite (convertir F -> E)  
}
```

*annotation pour que le compilateur vérifie bien qu'il n'y a qu'une seule méthode abstraite*  
**IMPORTANT !**

```
Convertisseur<Float, Float> euro_francs = (de) -> de*(float)6.35;
```

```
Float res = euro_francs.convertir((float)1.50);
```

```
System.out.println(res);
```

# Java 8



## ● Exemple

ok si 'oublié' mais pas de vérification du compilateur  
NE JAMAIS OUBLIER annotations



```
@FunctionalInterface  
interface Convertisseur<F, E> {  
    E convertir(F valeur); // méthode abstraite  
}
```

```
Convertisseur<String, Integer> stringVersI = (de) -> Integer.valueOf(de);
```

```
Integer res = stringVersI.convertir("25");
```

```
System.out.println(res);
```

# Java 8



- Référence de méthodes statiques

```
Convertisseur<String, Integer> stringVersI = (de) -> Integer.valueOf(de);  
Integer res = stringVersI.convertir("25");  
System.out.println(res);
```

```
Convertisseur<String, Integer> stringVersI = Integer::valueOf ;  
Integer res = stringVersI.convertir("25");  
System.out.println(res);
```

référence statique : nom d'une classe

# Java 8



- Référence de méthodes d'objet

```
class SObjet {  
    String enMajuscules(String s) {  
        return String.toUpperCase(s);  
    }  
}
```

```
SObjet o = new SObjet();
```

```
Convertisseur<String, String> convertisseur = o::enMajuscules;  
String res = convertisseur.convertir("Java");  
System.out.println(res); // "JAVA"
```

@FunctionalInterface

```
interface Convertisseur<F, E> {  
    E convertir(F valeur); // méthode abstraite  
}
```

# Java 8



## • Référence de méthodes de constructeurs

```
class Robot {  
    String prenom;  
    String nom;  
  
    Robot() {}  
  
    Robot(String prenom, String nom) {  
        this.prenom = prenom;  
        this.nom = nom;  
    }  
}
```

```
@FunctionalInterface  
interface UsineRobot<R extends Robot> {  
    R create(String firstName, String lastName);  
}
```

```
UsineRobot<Robot> cree = Robot::new;
```

```
Robot r2d2 = cree.create("r2", "d2")
```

2 paramètres

# Java 8



- Clôture référentielle limitée
  - . final
  - . Scala
- Prédéfinis Lambda

```
Predicate<String> predcat = (s) -> s.length() > 0;
```

```
predcat.test("ok");           // true  
predcat.negate().test("ok"); // false
```

fonctionnel en cascade

```
Predicat<Boolean> nonNull = Objects::nonNull;  
Predicat<Boolean> isNull = Objects::isNull;
```

```
Predicat<String> isEmpty = String::isEmpty;  
Predicat<String> isNotEmpty = isEmpty.negate();
```

# Java 8



- Prédéfinis Lambda Functions

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"
```

## Suppliers

```
Supplier<Robot> fournisseurRobot = Robot::new;
fournisseurRobot.get(); // new Robot
```

# Java 8



- Prédéfinis Lambda Consumers

```
Consumer<Robot> salut = (p) -> System.out.println("Hello, " + p.prenom);  
salut.accept(new Robot("R2", "D2"));
```

## Comparators

```
Comparator<Person> comparateur = (p1, p2) -> p1.firstName.compareTo(p2.firstName);
```

```
Person p1 = new Person("luke", "skywalker");  
Person p2 = new Person("alfred", "DarkVader");
```

```
comparateur.compare(p1, p2);  
comparateur.reversed().compare(p1, p2);
```

# Java 8



- Prédéfinis Lambda  
Optionals

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();          // true
optional.get();               // "bam"
optional.orElse("rien");     // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"
```

# Java 8



- Streams (collections) et fonctionnel stream

```
List<String> stringCollection = new ArrayList<>();  
  
stringCollection.add("ddd2");  
stringCollection.add("aaa2");  
stringCollection.add("bbb1");  
stringCollection.add("aaa1");  
stringCollection.add("bbb3");  
stringCollection.add("ccc");  
stringCollection.add("bbb2");  
stringCollection.add("ddd1");
```

# Java 8



- Streams (collections) et fonctionnel  
**filter**

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

# Java 8



- Streams (collections) et fonctionnel sorted

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);
```

```
// "aaa2", "aaa1"
```

```
// attention : pas de modif de la collection
d'origine
```

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Scala :  
mutable  
immutable

# Java 8



- Streams (collections) et fonctionnel  
**map**

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);
```

map crée une nouvelle collection (stream) en passant chaque élément par la fonction passée en argument

reduce

# Java 8



## • Streams (collections) et fonctionnel

### match

```
boolean anyStartsWithA =  
    stringCollection  
        .stream()  
        .anyMatch((s) -> s.startsWith("a"));  
  
System.out.println(anyStartsWithA); // true
```

```
boolean allStartsWithA =  
    stringCollection  
        .stream()  
        .allMatch((s) -> s.startsWith("a"));  
  
System.out.println(allStartsWithA); // false
```

```
boolean noneStartsWithZ =  
    stringCollection  
        .stream()  
        .noneMatch((s) -> s.startsWith("z"));  
  
System.out.println(noneStartsWithZ); // true
```

# Java 8



- Streams (collections) et fonctionnel  
**count**

```
long startsWithB =  
    stringCollection  
        .stream()  
        .filter((s) -> s.startsWith("b"))  
        .count();
```

```
System.out.println(startsWithB); // 3
```

# Java 8



- Streams (collections) et fonctionnel  
reduce

```
Optional<String> reduced =  
    stringCollection  
        .stream()  
        .sorted()  
        .reduce((s1, s2) -> s1 + "#" + s2);  
  
reduced.ifPresent(System.out::println);  
  
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

map +  
reduce  
Hadoop  
Spark  
MongoDB  
JavaScript  
Scala

# Java 8



- // Streams (collections) et fonctionnel  
**Traitement en // sur plusieurs Threads**

```
private static long countPrimes(int max) {  
    return IntStream.range(1, max).filter(TD2::isPrime).count();  
}  
  
private static boolean isPrime(int n) {  
    return (n > 1) && IntStream  
        .rangeClosed(2, (int) Math.sqrt(n))  
        .noneMatch(divisor -> ((n % divisor) == 0) );  
}
```

# Java 8



- // Streams (collections) et fonctionnel
- Traitement sur plusieurs Threads

```
System.out.println("Debut");
long startTime = System.nanoTime();
System.out.println( countPrimes(10000000) );
long endTime = System.nanoTime();
long duration = (endTime - startTime);
System.out.println("duree =" +(duration/1000000000)+"s");
```

```
Debut
664579
duree =153
```

Un seul cœur utilisé

# Java 8



- // Streams (collections) et fonctionnel

## Traitement sur plusieurs Threads

```
private static long countPrimes(int max) {  
    return IntStream.range(1, max).parallel().filter(TD2::isPrime).count();  
}
```

```
private static boolean isPrime(int n) {  
    return n > 1 && IntStream  
        .rangeClosed(2, (int) Math.sqrt(n))  
        .noneMatch(divisor -> n % divisor == 0);  
}
```

Debut  
664579  
duree =43

Tous les coeurs sont utilisés

Mais prudence avec *parallel* en Java...

Scala

# Java 8



- // Streams (collections) et fonctionnel  
**Traitement sur plusieurs Threads**

```
private static long countPrimes(int max) {  
    return IntStream.range(1, max).parallel().filter(TD2::isPrime).count();  
}  
  
private static boolean isPrime(int n) {  
    return (n > 1) && IntStream  
        .rangeClosed(2, (int) Math.sqrt(n))  
        .parallel()  
        .noneMatch(divisor -> ((n % divisor) == 0));  
}
```

Debut  
664579  
duree =32s

Tous les cœurs sont utilisés

Scala

85

# Java 8



- // Streams (collections) et fonctionnel  
traitement sur plusieurs Threads
- Java futurs & Co. (Java 5)
- Acteurs & Akka  
Scala

# Java 8



- Javascript

Rhino (mozilla) → Nashorn

Interpréteur / compilateur intégré  
+ rapide

jjs

appel depuis Java

appel de Java depuis Javascript

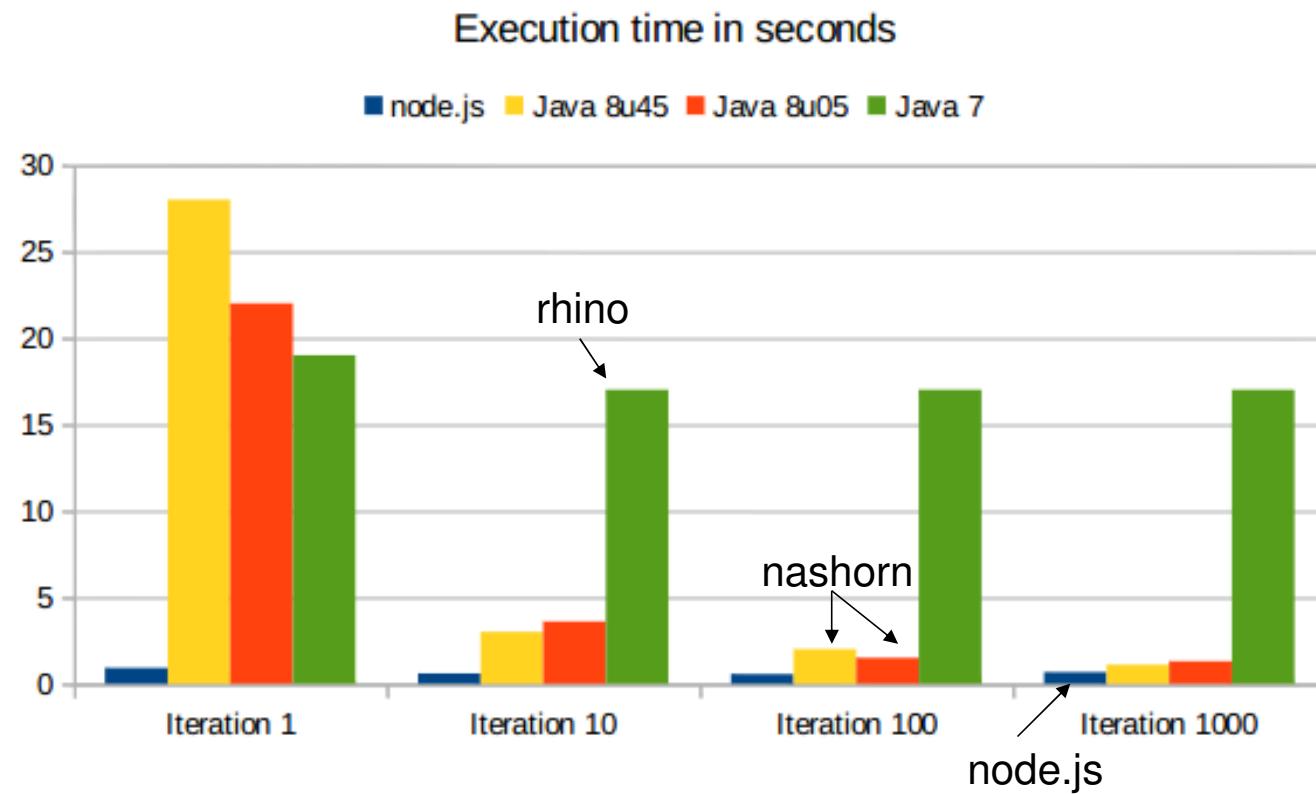
```
C:\Users\Gildas>jjs -version
nashorn 1.8.0_45
jjs> var maListe = ["am","stram","gram"]
jjs> for( i in maListe ) print(maListe[i])
am
stram
gram
jjs> -
```

node.js

# Java 8



- rhino vs nashorn vs node.js



# Java 8



## •Conclusion

Tous les langages de développements  
orientés objets  
maintenant orientés fonctionnels  
map, reduce, type fonction  
lambda calcul

Java à la traine

Java 8

bricolage limité

Scala

# Java



- Actuellement
  - Développement
  - Maintenance
- Gros projets
  - Association Java Scala
  - Scala
- Javascript
  - Compilateur Javascript → Java
  - Intégration Rhino → Nashorn
  - Cordova et PhoneGap (android)

# Java 9



- Jshell
  - REPL autours de Java / Interpreter / JIT
  - *Read Evaluate Print Loop*
  - Kulla
- API : mises à jour
  - Futures
  - Flow
  - Process
  - HTTP2 (voir cours second semestre)
  - Etc..
- Nouveau *Garbage Collector* G1 (Java possède 4 GC)
- Système de modules

# Java 10



- Inférence de type

Java 9 :

```
MyComplexType obj = new MyComplexType();
Map<String,List<MyComplexType>> map = new HashMap<String,List<MyComplexType>>();
```

Java 10 :

```
var obj = new MyComplexType();
var map = new HashMap<String,List<MyComplexType>>();
```

- Copie de collections

- **List.copyOf(), Set.copyOf(), et Map.copyOf()**

- **Gestion des collections immuables**

```
Collectors.toUnmodifiableList()
Collectors.toUnmodifiableSet()
Collectors.toUnmodifiableMap(keyFunc, valueFunc)
Collectors.toUnmodifiableMap(keyFunc, valueFunc, mergeFunc)
```

# Java 10



- Imports de certaines caractéristiques Scala
    - **Optional.orElseThrow()** etc..
  - Garbage collector
    - **G1 de Java 9**
    - Nouveau : **G1 multithread**
  - **Graal**
    - JVM en Java
- Voir <https://www.azul.com/109-new-features-in-jdk-10>

# Introduction à Scala

## How PayPal Scaled To Billions Of Transactions Daily Using Just 8VMs

MONDAY, AUGUST 15, 2016 AT 8:56AM

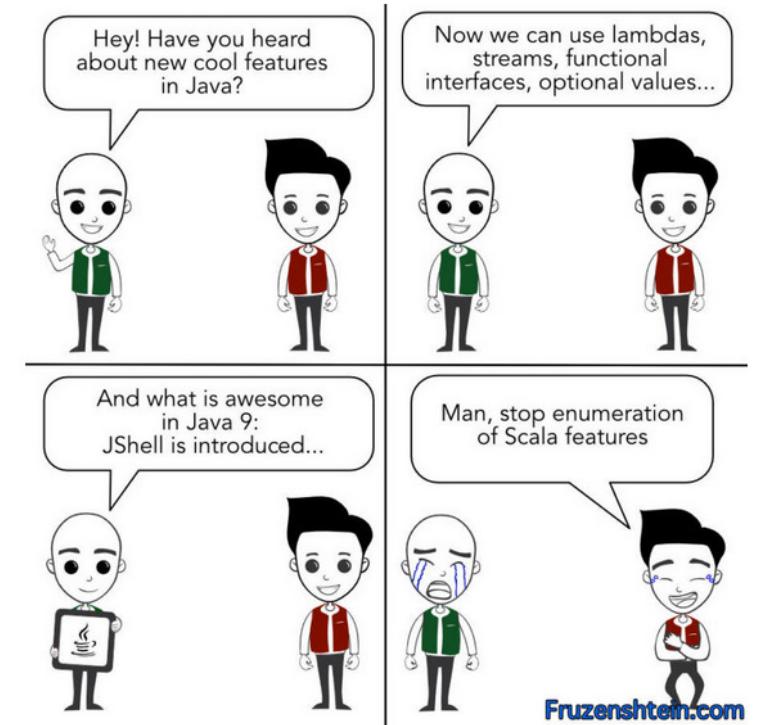
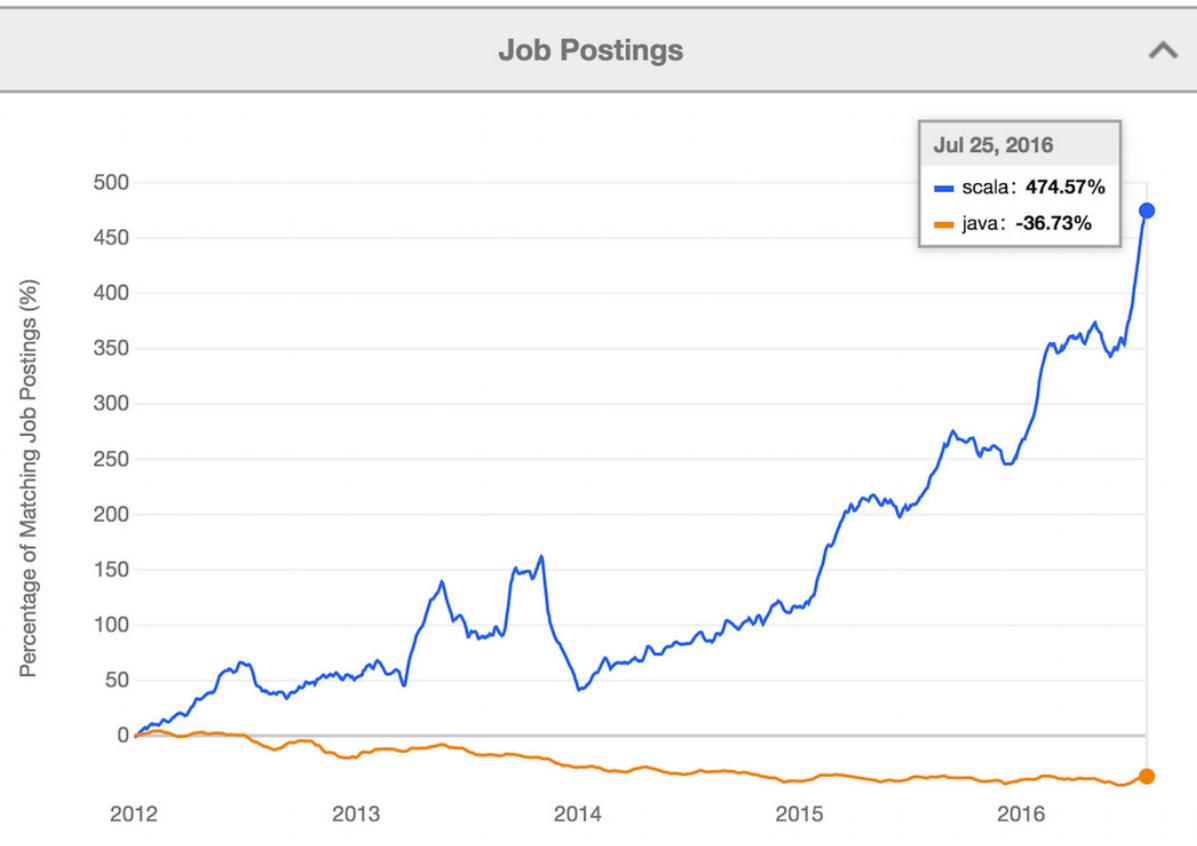
How did Paypal take a billion hits a day system that might traditionally run on a 100s of VMs and shrink it down to run on 8 VMs, stay responsive even at 90% CPU, at transaction densities Paypal has never seen before, with jobs that take 1/10th the time, while reducing costs and allowing for much better organizational growth without growing the compute infrastructure accordingly?



PayPal moved to an Actor model based on [Akka](#). PayPal told their story here: [squbs: A New, Reactive Way for PayPal to Build Applications](#). They open source squbs and you can find it here: [squbs on GitHub](#).

# Introduction à Scala

- Paradigmes
  - Procédural, modulaire, impératif, orienté objet, récursif
  - Fonctionnel
  - Acteur (Akka)
  - Déclaratif (pattern)
  - Réflexive
  - Meta programmation



# Scala



- Java
- Martin Odersky
  - EPFL (Ecole Polytechnique Lausanne)
  - Participe à Javac
    - Liens avec Java
  - Utilise la JVM
  - Interopérabilité
  - Fortement typé
  - Objet / orienté objet
  - Fonctionnel
  - Akka

Java repensé ?

- Currification
- Inférence de types
- Immuabilité
- Evaluation paresseuse
- *Pattern matching*
- Types algébriques (case classes)
- Co variance et contravariance
- Types anonymes et types d'ordres supérieurs
- Surcharge d'opérateurs
- Paramètres nommés
- Chaînes typées
- Fonctionnel
- Continuations délimitées
- DSL
- Conversions implicites
- Macro manipulations

# Scala

- Scalable + language
  - En fonction des connaissances / besoin utilisateur
    - Qui utilise Scala ?
      - Apple, Paypal, AOL, LinkedIn, Novell, Sony, Twitter, Tumblr, Vmware, Xerox, Amazon, Siemens, EDF ...
    - Ecrire moins de code (fonctionnel)
    - Fiabilité (inférence de type)
    - Distribution (Akka + Cloud)
    - Interopérabilité Java (librairies, formation)
    - Mais... formation difficile : discipline ?
    - Salaires++

# Scala



- Formation ?
  - Javascript
  - Haskell, clojure, Erlang, prolog ?
  - Programmation fonctionnelle
  
- Importance de l'analyse
  - Types et inférence
  - Collections
- Map/reduce/parallelisme
- Habitudes de Java

# Scala



- Quelques exemples

```
object HelloWorld extends App {  
    def main(args : Array[String]) {  
        println("Hello, World!")  
    }  
}
```

```
scalac HelloWorld.scala
```

```
scala HelloWorld
```

# Interpréteur Scala

- Installation

scala 2.11.8

- JDK
- Scala : [www.scala-lang.org](http://www.scala-lang.org)
- scaladoc, scalac et scala
- Read Eval Loop Print (REPL)

```
Welcome to Scala version 2.11.0
(Java HotSpot(TM) 64-Bit Server VM, Java 1.7. 0_02).
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

# Interpréteur Scala

```
scala> 5.0
res0: Double = 5.0
```

```
scala> "a"+"b"
res1: String = ab
```

```
scala> res1
res2: String = ab
```

# Interpréteur Scala

```
scala> :type "a" * 5
String
```

```
scala> "a" * 5
res3: String = aaaaa
```

# Interpréteur Scala

```
scala> val test_val = "yeah"  
test_val: String = yeah
```

```
scala> val test = 9  
test: Int = 9
```

```
scala> val test_val:Int = 34  
test_val: Int = 34
```

# Interpréteur Scala

```
scala> var test_var:Double = 3.4
test_var: Double = 3.4
```

```
scala> val a = 3 ; val b = 5 ; 6+1
a : Int = 3
b : Int = 5
res4 : Int = 7
```

# Interpréteur Scala

```
scala> 5.0
res5: Double = 5.0
```

```
scala> 5.0.plus(6)
<console>:8: error: value plus is not a member of Double
```

```
scala> 5.0.+ (6)
res6: Double = 11
```

# Interpréteur Scala

```
scala> 5.0 +(6)  
res7: Double = 11
```

```
scala> 5.0 + 6  
res8: Double = 11
```

```
scala> "comment ça va ?".indexOf("ça")  
res9: Int = 8
```

```
scala> "comment ça va ?" indexOf "ça"  
res10: Int = 8 ←
```

# Interpréteur Scala

```
"comment ça va ?" indexOf("ça")
```

ok

```
"comment ça va ?".indexOf "ça"
```

non

```
scala> 5.0.+ 6
<console>:1: error: ';' expected but integer literal found. 5.0.+ 6
```

# Interpréteur Scala

```
scala> "comment ça va ?".indexOf(("AV".reverse).toLowerCase)
res11: Int = 11
```

# Interpréteur Scala

```
scala> 0.to(5)
res12: scala.collection.immutable.Range.Inclusive
      = Range(0, 1, 2, 3, 4, 5)
```

```
scala> 0 to 5
res13: scala.collection.immutable.Range.Inclusive
      = Range(0, 1, 2, 3, 4, 5)
```

# Interpréteur Scala

```
scala> val r1 = 0 to 10
res15: scala.collection.immutable.Range.Inclusive
      = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
scala> r1 by 2
res16: scala.collection.immutable.Range
      = Range(0, 2, 4, 6, 8, 10)
```

# Interpréteur Scala

```
scala> 0 to 10 by 2 contains 4
res17: Boolean = true
```

```
scala> if (0 to 10 by 2 contains 4) print("4 existe bien entre ~
0 et 10")
4 existe bien entre 0 et 10
```

```
scala> if ("c'est un anglais" contains "anglais") "hello" else ~
"bonjour?"
res18: String = hello
```

# Interpréteur Scala

```
scala> ("c'est un anglais" contains "anglais") ? "hello": "bonjour?"  
<console>:1:error:identifier expected but string literal found.
```

Pas d'opérateur ternaire

```
scala> var salutation: String = if ("c'est un anglais" contains ↴  
"anglais") "hello" else "bonjour?"  
salutation: String = hello |
```

# Interpréteur Scala

```
scala> :type print("rencontre du 3ieme type:")
Unit
```

```
scala> val pasDeValeur: Unit = ()
pasDeValeur: Unit = () ↵

scala> pasDeValeur ←

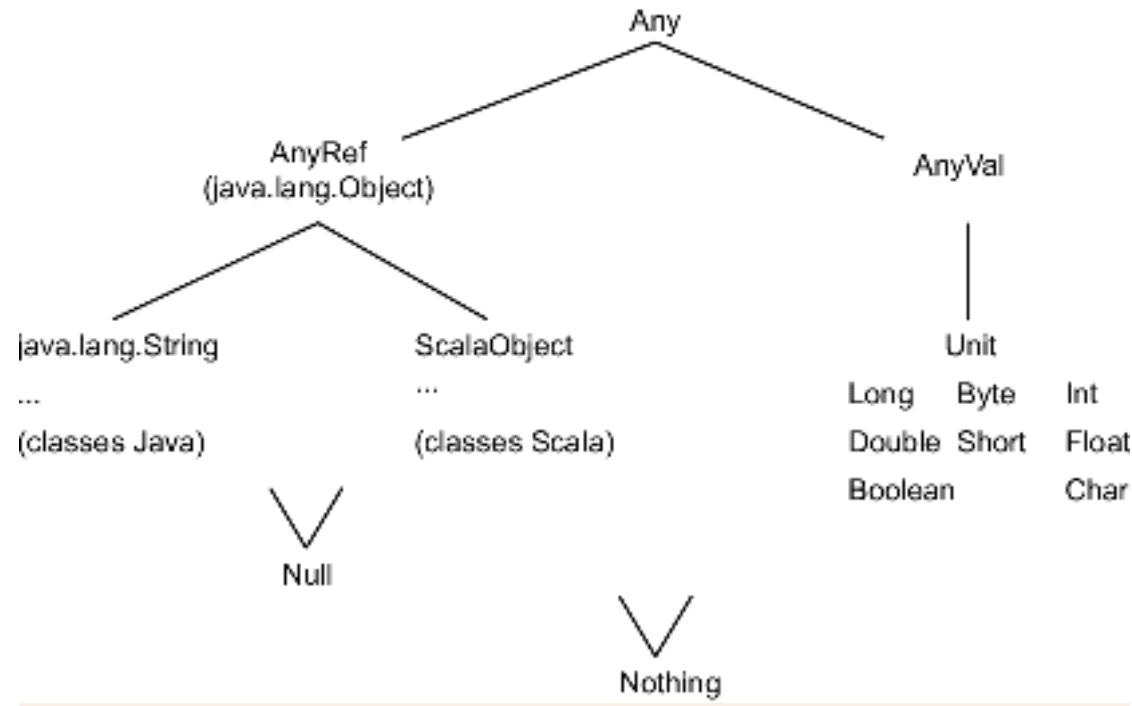
scala>
```

# Interpréteur Scala

```
scala> :type if (4 > 5) "c'est impossible" else 34
Any
```

```
:type if (4>5) true else 34
AnyVal
```

# Interpréteur Scala



# Interpréteur Scala

```
scala> { 5+4; 7+2; "a"+"b" }  
res19: String = ab
```

```
scala> val couleurDrapeau = { val c1="bleu"; val c2="blanc"; ↵  
    val c3="rouge"; c1+" "+c2+" "+c3 } ↵  
  
couleurDrapeau: String = bleu blanc rouge
```

# Interpréteur Scala

```
scala> val fCouleurDrapeau: (String, String, String) => String <-
      = (c1, c2, c3) => { c1+" "+c2+" "+c3 } <
      fCouleurDrapeau: (String, String, String)=>String = <function3>
```

val constante : type = valeur

Type de fCouleurDrapeau : (String, String, String) => String

Valeur : (c1,c2,c3) => { c1+" "+c2+" "+c3 }

# Interpréteur Scala

```
scala> fCouleurDrapeau ←  
  
res20: (String, String, String) => String = λ  
<function3>
```

```
scala> fCouleurDrapeau("bleu", "blanc", "rouge")  
  
res21: String = bleu blanc rouge
```

# Interpréteur Scala

```
scala> val fCouleurDrapeau: (String, String, String) => String <-
  = (c1, c2, c3) => { c1+" "+c2+" "+c3 } <
```

```
fCouleurDrapeau: (String, String, String)=>String = <function3>
```

```
val fCouleurDrapeau = new Function3[String, String, String, String] {
  def apply(a:String, b: String, c: String) = {
    ...
  }
}
```

```
scala> fCouleurDrapeau("bleu", "blanc", "rouge")
res22: String = bleu blanc rouge
```

```
scala> fCouleurDrapeau.apply("bleu", "blanc", "rouge")
res23: String = bleu blanc rouge
```

```

scala> val couleurHautBas: (String, String, String) => String =⇒
  (c1, c2, c3) => { c1 +"\n" + c2 +"\n" + c3 +"\n" } ↵
couleurHautBas: (String, String, String) => String = <function3> ↵

scala> val couleurGaucheDroite: (String, String, String) => String =⇒
  (c1, c2, c3) => { c1 + " " + c2 + " " + c3 +"\n" } ↵
couleurGaucheDroite: (String, String, String) => String = <function3>

scala> val couleurDrapeau: (String, String, String, ⇒
  (String, String, String) => String) => String =⇒
  (c1, c2, c3, f) => { f(c1, c2, c3) } ↵
couleurDrapeau: (String, String, String,
  (String, String, String) => String)
  => String = <function4> ↵

scala> val hongrie =⇒
  couleurDrapeau("rouge", "blanc", "vert", couleurHautBas) ↵
hongrie: String =
"rouge
blanc
vert
" ↵

scala> val france =⇒
  couleurDrapeau("bleu", "blanc", "rouge", couleurGaucheDroite) ↵
france: String =
"bleu blanc rouge
" ↵

```

# Interpréteur Scala

```
scala> val fCouleurDrapeau =(c1, c2, c3)=> {c1+" "+c2+" "+c3}
<console>:7: error: missing parameter type
```

```
scala> val fCouleurDrapeau = (c1:String, c2:String, c3:String) =>
    => {c1+" "+c2+" "+c3}
fCouleurDrapeau: (String, String, String) => String
= <function3>
```

# Interpréteur Scala

```
scala> def fCouleurDrapeau(c1: String, c2: String, c3: String) ←  
:String ={ c1+" "+c2+" "+c3 } ←  
  
fCouleurDrapeau: (c1: String, c2: String, c3: String)  
String ←  
  
scala> fCouleurDrapeau("bleu", "blanc", "rouge")  
res24: String = bleu blanc rouge ←
```

# Interpréteur Scala

```
scala> def salutation(qui: String): Unit = print("bonjour "+qui)
salutation: (qui: String)Unit <-

scala> salutation("victor")
bonjour victor
```

```
scala> def salutation(qui: String) { print("bonjour "+ qui) }
salutation: (qui: String)Unit
```

# Interpréteur Scala

```
scala> "bonjour".apply(2)
res25: Char = n
```

```
scala> "bonjour"(2)
res26: Char = n
```

```
scala> val f = "bonjour"
f: String = bonjour
scala> f(2)
res27: Char = n
```

# Interpréteur Scala

```
scala> var tab = new Array[Int](10)
tab: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

```
scala> tab(5) = 6
scala> print(tab(5))
6
```

```
scala> var tab = Array(5,4,3,2,1)
tab: Array[Int] = Array(5, 4, 3, 2, 1)
```

# Interpréteur Scala

`[Array(5,4,3,2,1)]` est la même chose que `[Array.apply(5,4,3,2,1)]`

Sans un new devant Array : new Array,  
la création de l'objet peut se faire dans la méthode apply :

```
def apply() = { new Array() }
```

# Interpréteur Scala

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer ←

scala> var ab = new ArrayBuffer[Char]()
ab: scala.collection.mutable.ArrayBuffer[Char] = ArrayBuffer()
```

```
ab += 'd' // rajoute un élément à la fin ←

ab += ('a', 'b', 'c') // rajoute plusieurs éléments à la fin ←

// insère les éléments 'x', 'y' et 'z' avant l'élément d'indice 3
ab insert(3, 'x', 'y', 'z') ←

ab remove 4 // enlève l'élément d'indice 4
```

# Interpréteur Scala

```
scala> var a = Array(1,2,3,4)
a: Array[Int] = Array(1, 2, 3, 4) ←
```

```
scala> a toBuffer
res28: scala.collection.mutable.Buffer[Int]
= ArrayBuffer(1, 2, 3, 4)
```

```
scala> 0 to 10 by 2 toArray
res29: Array[Int] = Array(0, 2, 4, 6, 8, 10)
```

# Interpréteur Scala

```
def fct(tab: Array[Int], i: Int) {  
    if (i < tab.length) {  
        println(tab(i)); fct(tab, i+1)  
    }  
}  
scala> fct(a, 0)  
1 2 3 4 ←
```

```
var i = 0  
while( i < a.length) { println(a(i)); i = i+1 }
```

# Interpréteur Scala

```
scala> for(v <- a) println(v)
```

```
Java for( final int v : tab)
```

```
scala> for(i <- 0 to 10) print(i+" ")  
0 1 2 3 4 5 6 7 8 9 10
```

```
scala> for (i <- 0 until a.length) print( a(i)+" ")  
1 2 3 4
```

```
scala> for( i <- 0 until 10 reverse) print(i+" ")  
9 8 7 6 5 4 3 2 1 0
```

# Interpréteur Scala

```
scala> val tab=Array(1, 5, 9,7, 11, 3)
tab: Array[Int] = Array(1, 5, 9, 7, 11, 3) ←

scala> tab sum // calcule la somme des valeurs du tableau
res30: Int = 36 ←

scala> tab max //trouve la valeur maximale du tableau
res31: Int = 11 ←

scala> tab min // valeur minimale
res32: Int = 1 ←

scala> tab product //produit des valeurs du tableau
res33: Int = 10395 ←

scala> tab sorted // tri du tableau par ordre croissant
res34: Array[Int] = Array(1, 3, 5, 7, 9, 11)
```

# Interpréteur Scala

```
scala> val carres = for( v <- tab) yield v*v
carres: Array[Int] = Array(1, 25, 81, 49, 121, 9)
```

```
scala> val tab = Array(-3, -2, 10, 5) ↵
tab: Array[Int] = Array(-3, -2, 10, 5) ↵

scala> val somme = (
    for( v <- tab if (v >= 0) ) yield scala.math.sqrt(v)
) sum ↵

somme: Double = 5.39834563766817
```

# Interpréteur Scala

```
Attention -3 < 0
Attention -2 < 0
La racine de 10 est 3.1622776601683795
La racine de 5 est 2.23606797749979
```

```
val tab2 =
    for(v <- tab) yield
        if (v<0) "Attention "+v+" < 0" else
            "La racine de "+v+" est "+
                scala.math.sqrt(v) +"
```

```
tab2: Array[String] = Array(Attention -3 < 0,
Attention -2 < 0, La racine de 10 est 3.1622776601683795,
La racine de 5 est 2.23606797749979)
```

# Interpréteur Scala

```
scala> for(message <- tab2) println(message)
```

```
scala> print( tab2 mkString("\n") )
```

# Interpréteur Scala

```
print (
  ( for( v <- Array(-3, -2, 10, 5) ) yield
    if (v<0) "Attention "+v+" < 0" else
      "La racine de "+v+" est "+scala.math.sqrt(v)
    ) mkString("\n")
) <-->
```

# L'essentiel Scala

```
object test { // ceci est un commentaire  
    def main(args: Array[String]) {  
        println("Scala : c'est parti !")  
    }  
    /* ceci est également un commentaire */  
}
```

import scala.Console.\_

Par défaut : java.lang, scala, scala.predef

# L'essentiel Scala

```
val nom = "joe"  
  
// affiche "bonjour joe ça va ?"  
println("bonjour "+nom+" ça va ?")
```

```
val nom ="joe"  
  
// affiche "bonjour joe ça va"  
println( s"bonjour $nom ça va ?" )
```

```
// affiche "aussi simple que 2 et 2 font 4"  
println( s"aussi simple que 2 et 2 font ${2+2}" )
```

# L'essentiel Scala

```
val test = raw"et\nhop"  
  
// affiche "et\nhop"  
// '\n' n'est pas remplacé par un retour de ligne  
println(test)
```

```
val pi = 3.1415  
val nom = "Archimède"  
  
// affiche "d'après Archimède, pi = 3,14"  
println( f"d'après $nom, pi = $pi%2.2f")
```

# L'essentiel Scala

```
// affiche : "L'ogre Jean pèse 200 kg et mesure 10,50m"
printf("L'ogre %s pèse %d kg et mesure %2.2fm\n", "Jean", 200, 10.5)
```

```
val chaine = """am
stram
gram"""\n
println(chaine)
```

am  
stram  
gram

```
val chaine = """am
|stram
|gram"""\n.stripMargin
```

```
val chaine = """am
#stram
#gram"""\n.stripMargin('#')
```

# L'essentiel Scala

```
// le bloc est évalué en séquence :  
//   println est évalué  
//   5 est évalué  
// la valeur du bloc est 5  
val a = { println("initialisation de a") ; 5 }  
  
println("début")  
  
val b = a + 1  
  
println("valeur de b = " + b)  
  
val c = a + 2  
  
println("valeur de c = " + c)  
println("fin")
```

*initialisation de a  
début  
valeur de b = 6  
valeur de c = 7  
fin*

# L'essentiel Scala

```
lazy val a = { println("initialisation de a") ; 5 }

// le bloc n'est pas évalué à ce stade

println("début")

val b = a+1

println("valeur de b = "+b)

val c = a+2
println("valeur de c = "+c)
println("fin")
```

*début*  
*initialisation de a*  
*valeur de b = 6*  
*valeur de c = 7*  
*fin*

# L'essentiel Scala

```
def a = { println("initialisation de a") ; 5 }  
  
// pas de val ici  
  
println("début")  
  
val b = a+1  
  
println("valeur de b = " + b)  
  
val c = a+2  
  
println("valeur de c = " + c)  
  
println("fin")
```

*début*  
*initialisation de a*  
*valeur de b = 6*  
*initialisation de a*  
*valeur de c = 7*  
*fin*

# L'essentiel Scala

```
def a = { println("initialisation de a") ; 5 }
```

```
def somme(v1: Int, v2: Int) = v1 + v2
```

```
def somme(v1: Int, v2: Int) : Int = { v1 + v2 }
```

# L'essentiel Scala

```
def somme(de:Int, jqa: Int):Int = {  
    if (de < jqa) (de + somme(de+1,jqa)) else de  
}
```

```
def texteAEncadrer(texte: String = "",  
                    avant: String = "<" ,  
                    apres: String = ">") =  avant + texte + apres  
  
def baliseHTML(texte: String) = texteAEncadrer(texte)
```

```
// affiche "[bien encadré]"  
println(texteAEncadrer(apres="]", avant="[, texte="bien encadré"]))
```

# L'essentiel Scala

```
def moyenne( valeurs: Int * ) : Double = {
    if (valeurs.length == 0) 0 else {
        var sommeTotale = 0
        for(v <- valeurs) sommeTotale += v
        sommeTotale / valeurs.length
    }
}
println(moyenne(1,2,3,4,5)) // affiche "3.0"
println(moyenne()) // affiche "0.0"
```

```
// même chose que println(moyenne(1,2,8,7,2))
println(moyenne( Array(1,2,8,7,2) : _*)) ←

// même chose que println(moyenne(1,2,3,4))
println(moyenne( 1 to 4 : _*))
```

# L'essentiel Scala

```
def afficheSomme(a:Int, b: Int) : Unit = {  
    println(a+b)  
}
```

```
def afficheSomme(a:Int, b: Int) {  
    println(a+b)  
}
```

# L'essentiel Scala

```
// Définition incorrecte pour Scala
def somme(a: Int, b: Int) {
    return a+b
}
```

```
def somme(a: Int, b: Int): Int = { // n'oubliez pas '='
    a + b
}
```

```
def somme(a: Int, b: Int) = { // n'oubliez pas '='
    a + b
}
```

# L'essentiel Scala

```
val saison="hiver"
val commentaire = if (saison=="hiver") "fait froid" else
                  if (saison=="printemps") "fait bon" else
                  if (saison=="été") "fait chaud" else
                  if (saison=="automne") "fait humide" else
                    saison+" n'est pas une saison valide"
println(commentaire)
```

```
switch(saison) { // exemple JAVA - impossible en Scala
    case "hiver": System.out.println("fait froid"); break;
    case "printemps" : System.out.println("fait bon") ; break;
    case "été" : System.out.println("fait chaud"); break;
    case "automne": System.out.println("fait humide"); break;
    default : System.out.println(" n'est pas une saison valide");
        break;
}
```

# L'essentiel Scala

```
saison match {  
    case "hiver" => println("fait froid")  
    case "printemps" => println("fait bon")  
    case "été" => println("fait chaud")  
    case "automne" => println("fait humide")  
    case _ => println(saison+" n'existe pas")  
}
```

```
saison match {  
    case "hiver" => {  
        println("brrrrrr")  
        println("fait froid")  
    }  
    case "printemps" => println("fait bon")  
    case "été" => println("fait chaud")  
    case "automne" => println("fait humide")  
    case _ => println(saison+" n'existe pas")  
}
```

# L'essentiel Scala

```
val monAvisSurLaSaison = saison match {  
    case "hiver" => "fait froid"  
    case "printemps" => "fait bon"  
    case "été" => "fait chaud"  
    case "automne" => "fait humide"  
    case _ => " n'existe pas"  
}
```

# L'essentiel Scala

## Exceptions

```
throw new MalformedURLException("URL incorrecte")
```

```
try {
    var input = connectTo(new URL("univ-ubs-"))
} catch {
    case pepin : MalformedURLException => println("mauvaise URL")
    case ex : IOException => ex.printStackTrace()
} finally {
    input.close()
}
```

# L'essentiel Scala

Collections et types paramétriques

```
val tableau = new Array[Int](3,4,7,6,7)
```

```
tableau[3] // ok en Java mais impossible en Scala
```

```
tableau(3) // ok en Scala
```

# L'essentiel Scala

## Fonctions et types paramétriques

```
def hey[T] (x: T) = "hey " + x.toString + " !"  
  
// affiche "hey joe !"  
println(hey[String] ("joe"))  
  
// affiche "hey 50 !"  
println(hey[Int] (50))
```

# L'essentiel Scala

## Fonctions et types paramétriques

```
import Ordering.Implicits._

def estSuperieur[T : Ordering](x:T, y:T): Boolean = x > y

// affiche "false"
println(estSuperieur(5,7))

// affiche "true"
println(estSuperieur("b","a"))
```

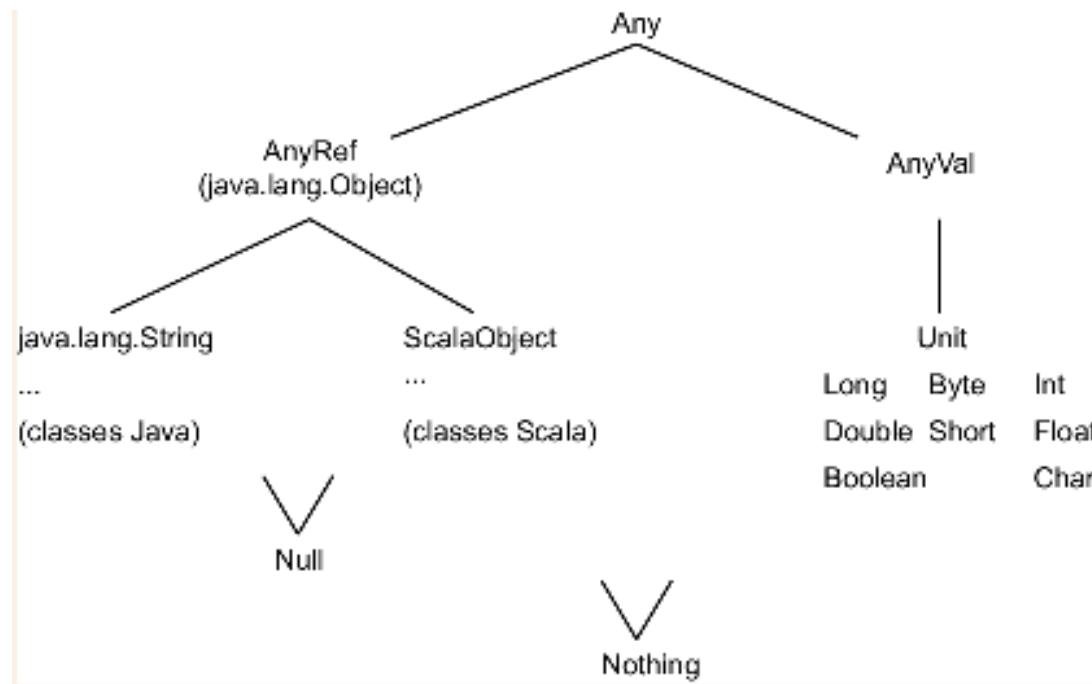
# L'essentiel Scala

## Alias et types

```
type tableauEntiers = Array[Int]  
  
def traitement(tab: tableauEntiers) { ... }
```

# L'essentiel Scala

## Programmation objet



# L'essentiel Scala

```
public class Test { // Programme Java - impossible en Scala
    static void main(String[] args) {
        System.out.println("Java : c'est parti !")
    }
}
```

```
object test { // Programme Scala
    def main(args: Array[String]) {
        println("Scala : c'est parti !")
    }
}
```

# L'essentiel Scala

```
class Tirelire {  
    private var contenu : Int = _  
  
    private var contenu : Int = 0  
    def +=(monnaie : Int) { contenu += monnaie }  
    def vider() = { val solde = contenu; contenu = 0; solde }  
    def combien = contenu  
    override def toString = "contient ("+combien+)"  
}  
override def toString(): String = "contient ("+combien+)"
```

```
val cochon = new Tirelire()  
println(cochon)  
cochon += 10  
println(cochon)  
println(cochon vider())
```

```
contient (0)  
contient (10)  
10
```

# L'essentiel Scala

```
object Tirelire {  
    def voiciUneMéthodeDeClasse() {  
        // ...  
    }  
}
```

Pas de *static*

Invocation

```
Tirelire.voiciUneMéthodeDeClasse()
```

# L'essentiel Scala

## Constructeurs

```
class Tirelire(argentDepart: Int) { // constructeur  
    private var contenu : Int = argentDepart // initialisation  
  
    def this() = this(0) // constructeur  
  
    def += (monnaie : Int) { contenu += monnaie }  
    def vider() = { val solde = contenu; contenu = 0; solde }  
    def combien = contenu  
    override def toString = "contient ("+combien+)"  
}  
val cochon = new Tirelire(100)  
val cochon2 = new Tirelire()
```

# L'essentiel Scala

Librairies Java à partir de Scala

.Jar accessibles + import

```
import java.rmi.* ;      java
```

```
import java.rmi._           scala
```

Types de Java directement compatibles

Iterator<Component> (java) correspond à Iterator[Component] (scala)

# Tableaux

## Tableaux

```
val monTabEntiers = new Array[Int] (30)
```

```
val monTabChaines = new Array[String] (30)
```

```
val monTabEntiersInit = Array(1, 2, 5, 6)←
```

```
val lesSaisons = Array("été", "automne", "printemps", "hiver")
```

# Tableaux

Scaladoc

< Back

Showing results for "array"

**Entity results**

- scala
- o c **Array**
- c **FallbackArrayBuilding**

**Member results**

- scala
- o c **Array**
  - def `toArray(): Array[A]`
  - def `copyToArray(xs: Array[A], start: Int, Int): Unit`

**Companion object Array**

`final class Array[T] extends java.io.Serializable with java.lang.Cloneable`

Arrays are mutable, indexed collections of values. `Array[T]` is Scala's representation for Java's `T[]`.

```
val numbers = Array(1, 2, 3, 4)
val first = numbers(0) // read the first element
numbers(3) = 100 // replace the 4th array element with 100
val biggerNumbers = numbers.map(_ * 2) // multiply all numbers by two
```

Arrays make use of two common pieces of Scala syntactic sugar, shown on lines 2 and 3 of the above example code. Line 2 is translated into a call to `apply(Int)`, while line 3 is translated into a call to `update(Int, T)`.

This implicit conversion exists in scala.Predef so that one frequently applies to convert a conversion to

Scala Standard Library 2.11.... +

À la une UBS Prog Graph Son Perso M D f S B

#ABCDEFHIJKLMNOPQRSTUVWXYZ

display packages only

scala hide focus

- c AnyVal
- t App
- o c **Array**
- o c Boolean
- o c Byte
- o c Char
- t Cloneable
- o Console

final class **Array**

Arrays are mutable, indexed collections of values.

```
val numbers = Array(1, 2, 3, 4)
val first = numbers(0) // read the first element
numbers(3) = 100 // replace the 4th array element with 100
val biggerNumbers = numbers.map(_ * 2) // multiply all numbers by two
```

Arrays make use of two common pieces of Scala syntactic sugar, shown on lines 2 and 3 of the above example code. Line 2 is translated into a call to `apply(Int)`, while line 3 is translated into a call to `update(Int, T)`.

## scala.Array

final class **Array**[T] extends java.io.Serializable with **java.la**

Arrays are mutable, indexed collections of values. **Array**[T] is Scala's representation for Java's T

```
val numbers = Array(1, 2, 3, 4)
val first = numbers(0) // read the first element
numbers(3) = 100 // replace the 4th array element with 100
val biggerNumbers = numbers.map(_ * 2) // multiply all numbers by two
```

Arrays make use of two common pieces of Scala syntactic sugar, shown on lines 2 and 3 of the ab

Two implicit conversions exist in [scala.Predef](#) that are frequently applied to arrays: a conversion to [ArrayOps](#) (a subtype of [scala.collection.Seq](#)). Both types make available many of the standard operations found in [Seq](#), while the conversion to [WrappedArray](#) is permanent as all operations return a [WrappedArray](#).

The conversion to [ArrayOps](#) takes priority over the conversion to [WrappedArray](#). For instance, c

```
val arr = Array(1, 2, 3)
val arrReversed = arr.reverse
val seqReversed : Seq[Int] = arr.reverse
```

Value `arrReversed` will be of type `Array[Int]`, with an implicit conversion to `ArrayOps` occurring before `WrappedArray` first and invoking the variant of `reverse` that returns another `WrappedArray`.

---

Source [Array.scala](#)

Version 1.0

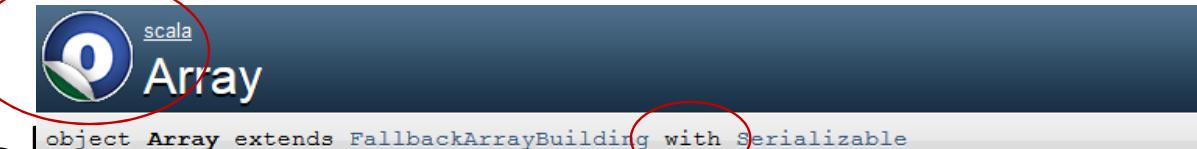
See also ["The Scala 2.8 Collections' API"](#) section on `Array` by Martin Odersky for more information. See also ["Scala 2.8 Arrays"](#) the Scala Improvement Document detailing arrays since ["Scala Language Specification"](#), for in-depth information on the transformation.

---

► [Linear Supertypes](#)

► [Type Hierarchy](#)

# L'essentiel



Utility methods for operating on arrays. For example:

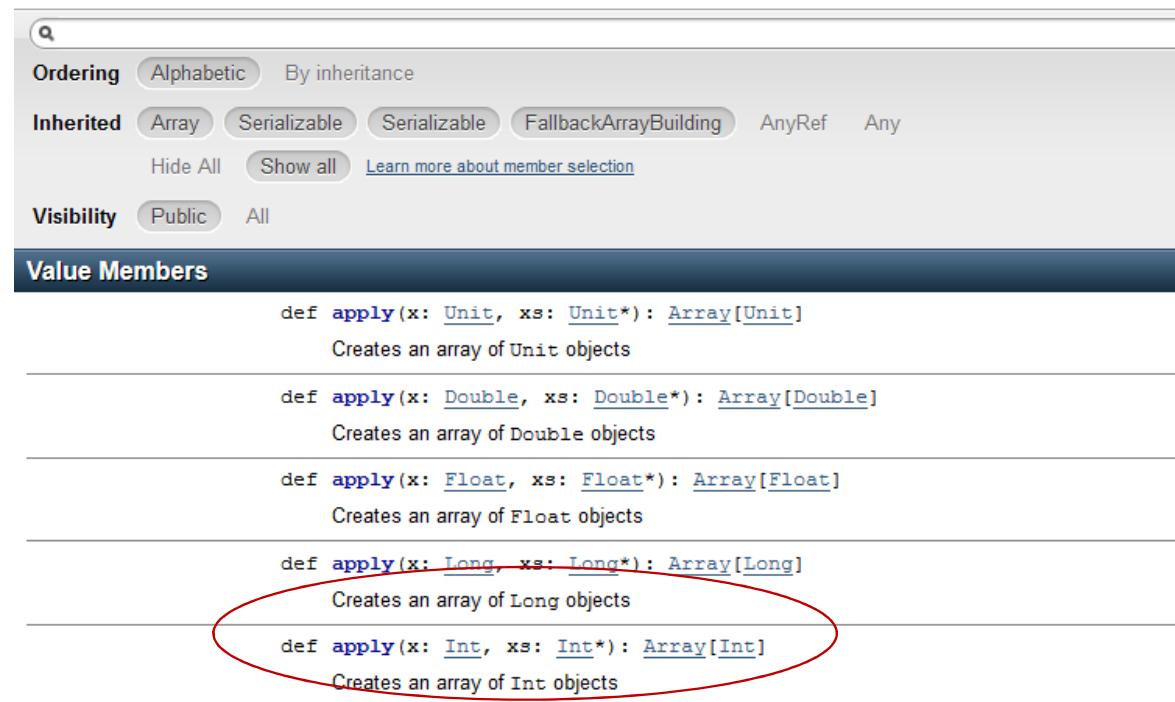
```
val a = Array(1, 2)
val b = Array.ofDim[Int](2)
val c = Array.concat(a, b)
```

where the array objects a, b and c have respectively the values `Array(1, 2)`, `Array(0, 0)` and `Array(1, 2, 0, 0)`.

Source [Array.scala](#)

Version 1.0

► Linear Supertypes



Ordering	Alphanumeric	By inheritance
Inherited	<a href="#">Array</a>	<a href="#">Serializable</a>
	<a href="#">Serializable</a>	<a href="#">FallbackArrayBuilding</a>
	<a href="#">AnyRef</a>	<a href="#">Any</a>
Visibility	<a href="#">Public</a>	<a href="#">All</a>
Value Members		
<b>def apply(x: Unit, xs: Unit*): Array[Unit]</b> Creates an array of Unit objects		
<b>def apply(x: Double, xs: Double*): Array[Double]</b> Creates an array of Double objects		
<b>def apply(x: Float, xs: Float*): Array[Float]</b> Creates an array of Float objects		
<b>def apply(x: Long, xs: Long*): Array[Long]</b> Creates an array of Long objects		
<b>def apply(x: Int, xs: Int*): Array[Int]</b> Creates an array of Int objects		

GitHub This repository Search Explore Features Log in

scala / scala

tag: v2.11.4 ▾ scala / src / library / scala / Array.scala

retronym on 5 Mar 2013 Name boolean arguments in src/library.

19 contributors

536 lines (491 sloc) | 20.92 kb

```
/*
 * _____ _ _   Scala API
 * / \_/_//_ | // /_ | (c) 2002-2013, LAMP/EPFL
 * \ \ / /_ | / /_ | http://scala-lang.org/
 * / \/_/_//_ |/_/_/_ | /
 * \_/_/_/_ | /_/_/_/_ | /
 */
package scala
import scala.collection.generic._
import scala.collection.{ mutable, immutable }
import mutable.{ ArrayBuffer, ArraySeq }
import scala.compat.Platform.arraycopy
import scala.reflect.ClassTag
import scala.runtime.ScalaRunTime.{ array_apply, array_update }

/** Contains a fallback builder for arrays when the element type
 *  does not have a class tag. In that case a generic array is built.
 */
class FallbackArrayBuilding {

  /** A builder factory that generates a generic array.
   *  Called instead of `Array.newBuilder` if the element type of an array
   *  does not have a class tag. Note that fallbackBuilder factory
   *  needs an implicit parameter (otherwise it would not be dominated in
   *  implicit search by `Array.newBuilder`). We make sure that
}
```



# Tableaux

```
// Fabrique un tableau d'objets Int  
def apply(x: Int, xs: Int*): Array[Int]
```

```
val monTabEntiersInit = Array.apply(1,2,5,6)
```

```
val monTabEntiersInit = Array(1, 2, 5, 6)←
```

```
val lesSaisons = Array("été", "automne", "printemps", "hiver")
```

# Tableaux

```
// fonction qui prend au moins un entier
// et une liste d'entiers en arguments
// et qui renvoie un tableau d'entiers←

def apply(x: Int, xs: Int*): Array[Int] = {

    // définition d'un tableau en fonction du nombre
    // d'arguments
    val array = new Array[Int](xs.length + 1)

    array(0) = x // premier indice = 0←

    // premier élément
    var i = 1 ←

    // boucle pour remplir le tableau
    // avec les arguments
    for (x <- xs.iterator) { array(i) = x; i += 1 } ←

    // la dernière expression est la valeur de retour
    array

}←
```

# Tableaux

```
// matrice de 4 lignes et 9 colonnes
val matrice4x9 = Array.ofDim[Double](4, 9)
```

# Tableaux

```
val element = monTabEntiers(2)
```

```
val element2 = monTabEntiers.apply(2)
```

```
monTabEntiers(2) = 34
```

```
monTabEntiers.update(2,34)
```

# Tableaux

```
var i = 0
while( i < lesSaisons.length ) {
    println( lesSaisons(i) )
    i = i+1
}
```

Forme impérative

```
for( i <- 0 until lesSaisons.length ) println( lesSaisons(i) )
```

# Tableaux

```
for( element <- lesSaisons ) println( element )
```

Fonction (anonyme) :

```
element => println( element )
```

Appliquer la fonction à chaque élément :

```
lesSaisons.foreach( element => println( element ) )
```

# Tableaux

```
// _ représente l'élément courant
lesSaisons.foreach( println( _ ) )

// est également acceptable ici
lesSaisons.foreach( println )
```

# Tableaux

Transformation :

```
val lesSaisonsEnMajuscules = for( element <- lesSaisons )  
                                yield element.toUpperCase
```

Filtrage et sélection :

```
val noms = Array("jean-claude","jean-michel","paul","jean","éric")  
  
val lesJEANS = for( nom <- noms if nom.contains("jean") )  
                            yield nom.toUpperCase
```

# Tableaux

```
val lesAnnées = Array(1900, 1901, 1910, 1983, 2000) ←  
  
val lesSaisons = Array("été", "automne", "hiver", "printemps") ←  
  
val laMode =  
    for( année <- lesAnnées if année % 2 == 0 ; saison <- lesSaisons)  
        yield saison + " " + année ←  
  
laMode.foreach(println)  
  
été 1900  
automne 1900  
hiver 1900  
printemps 1900  
été 1910  
automne 1910  
hiver 1910  
printemps 1910  
été 2000  
automne 2000  
hiver 2000  
printemps 2000
```

# Tableaux

Transformation en un autre type (collection ou non) :

```
def filter(p: (T) => Boolean): Array[T]  
    e => e%2 == 0
```

fonctionnel

```
// le résultat est Array(2,4,6)  
Array(1,2,3,4,5,6).filter( e => e%2 == 0 )
```

# Tableaux

```
def map[B] (f: (A) => B) : Array[B]
```

```
    e => e.toUpperCase
```

```
    e => e.toString
```

```
// création d'un nouveau tableau qui contient les saisons  
// en majuscule  
val lesSaisonsEnMajuscules = lesSaisons.map( e => e.toUpperCase )  
  
// création d'un tableau de chaînes de caractères  
// à partir d'un tableau d'entiers  
val chiffresEnLettres = Array(1,2,3,4).map( e => e.toString )
```

# Tableaux

```
val noms = Array("jean-claude", "jean-michel", "paul", "jean", "éric")  
  
val lesJEANS = for( nom <- noms if nom.contains("jean") )  
                  yield nom.toUpperCase
```

```
val noms = Array("jean-claude", "jean-michel", "paul", "jean", "éric")  
  
val lesJEANS =  
  noms.filter(n => n.contains("jean")).map(e => e.toUpperCase)
```

```
val lesJEANS = noms.filter(_ contains("jean")).map(_ toUpperCase)
```

# Tableaux

```
def sortWith(lt: (T, T) => Boolean): Array[T]
```

```
(e1,e2) => e1 < e2
```

```
Array(2,8,3,4,7,54).sortWith( (e1,e2) => e1 < e2 )
```

# Tableaux

```
Array(1,2,3,4,5).sum // 15
```

```
def reduce[A1 >: A] (op: (A1, A1) => A1): A1
```

A1 est un super type de A

```
(e1,e2) => e1+e2
```

```
// est équivalent à Array(1,2,3,4,5).sum  
Array(1,2,3,4,5).reduce( (e1,e2) => e1+e2 )
```

# Tableaux

**map** : transformation d'une collection en une autre  
en appliquant une opération sur chaque élément

**filter** : sélection d'éléments dans une collection

**reduce** : réduction d'une collection par opérations sur des tuples



The screenshot shows the Wikipedia page for "Hadoop". The page title is "Hadoop". The content area starts with a brief introduction: "Hadoop est un framework Java libre destiné à faciliter la création d'applications distribuées et échelonnables (scalables). Il permet aux applications de travailler avec des milliers de nœuds et des pétaoctets de données. Hadoop a été inspiré par les publications MapReduce, GoogleFS et BigTable de Google." Below this, it states that Hadoop was created by Doug Cutting and is part of the Apache Software Foundation since 2009. A sidebar on the left contains navigation links like Accueil, Portails thématiques, Article au hasard, Contact, Contribuer, and Imprimer / exporter. A sidebar on the right provides summary information about Hadoop, including its developer (Apache Software Foundation), environments (Multiples-formes, principalement POSIX), type (Architecture distribuée), license (Licence Apache), and website (hadoop.apache.org).

# Tableaux

Tableau de taille variable

```
import scala.collection.mutable.ArrayBuffer
```

```
val tab = ArrayBuffer[Int] ()  
  
val tableauVide = Array.empty[Int]  
  
// création d'instances de plusieurs manières différentes  
val tab2 = new ArrayBuffer[Int]
```

# Tableaux

```
// rajoute une nouvelle valeur à la fin du tableau  
tab += 25  
  
// rajoute plusieurs valeurs à la fin  
tab2 += (34, 75, 34, 90)  
  
// insère les 3 valeurs (34, 56, 78)  
// à partir de la position 2 du tableau  
tab2 insert(2, 34, 56, 78)  
  
// rajoute les arguments en fin de tableau  
tab2 ++= Array(4, 5, 6, 7)  
  
// enlève le 4ième élément du tableau  
tab2.remove(4)
```

# Liste

Liste : rappels

```
val liste1 = 1::Nil  
val liste2 = 1::2::3::Nil  
val liste3 = List(1,2,3,4)
```

```
val t::q = liste3 // deconstructeur
```

```
println(t) // 1  
println(q) // List(2,3,4)
```

```
println(liste1 :: liste3) // Liste(1,1,2,3,4)
```

```
liste3.head  
liste3.tail  
liste3(1) // commence à 0  
liste3.size  
liste3.isEmpty  
List() // Nil
```

# Liste

Liste

```
liste2 match {  
    case t:: q => "pas vide"  
    case Nil => "vide"  
}
```

# Case Class

Case class

```
class Personne(nom:String, prenom: String) {  
  ...  
}
```

```
object Personne {  
  def apply(n:String, p:String) = new Personne(n,p)  
}
```

+ equals  
+ toString  
+ hashCode  
+ déconstructeur

case class Personne(nom: String, prenom: String)

```
val jojo = Personne("revault","joel")  
val groupe = List( Personne("paul","eric"), Personne("revault","joel"))
```

Foncteurs Prolog

# Case Class

Case class

```
case class Personne(nom: String, prenom: String)
```

```
val jojo = Personne("revault", "joel")
val groupe = List( Personne("paul", "eric"), Personne("revault", "joel"))
```

```
println(jojo.nom)
```

```
val Personne(n,p) = jojo // deconstructeur
println(n)
```

# Case class

Case class

```
case class Personne(nom: String, prenom: String)
```

```
val groupe = List( Personne("paul","eric"), Personne("revault","joel"))
```

```
groupe.foreach( p => println(p.nom+" "+p.prenom))
```

Ou encore :

```
groupe.foreach{ case Personne(n,p) => println(n+" "+p) } // voir plus loin : fonction partielle
```

# Case class

## Case class

```
case class Personne(nom: String, prenom: String)
```

```
val jojo = Personne("revault","joel")
```

```
jojo match {  
    case Personne(n,p) => println(n+" "+p)  
}
```

# Les principales collections

**Mutables ou non ?**

```
import scala.collection._

val couleurs = mutable.Set("blanc", "rouge", "vert")
couleurs += "rose"
```

couleurs est toujours le même objet (donc val c'est ok, on n'a pas changé la valeur de couleurs)

couleurs est un objet mutable, donc son état peut changer

+ = rajoute quelque chose à couleurs

# Les principales collections

**Mutables ou non ?**

```
import scala.collection._

val couleurs = Set("blanc", "rouge", "vert")
couleurs += "rose"
```

Le compilateur refuse :

Par défaut, les collections sont immutables : on ne peut pas changer l'état d'une collection.

Pour rajouter un élément, il faut créer un nouvel ensemble (l'ancien + un élément)

```
val nouvellesCouleurs = nouvellesCouleurs + "rose"
```

# Les principales collections

Mutables ou non ?

```
import scala.collection._  
  
var couleurs = Set("blanc", "rouge", "vert")  
couleurs = couleurs + "rose" // pas la même chose que +=
```

Cette fois vaR : on a changé d'objet

Attention : ne pas confondre : val (on ne change pas d'objet – on peut changer d'état)  
var (on peut changer d'objet – on peut changer d'état)

Par défaut les collections sont immutables

L'indiquer explicitement si on veut une collection mutable.

# tuple

```
def sommeDiff(a: Int, b: Int) = Array(a+b, a-b)  
val res = sommeDiff(3,6)  
println(res(0)+" et "+res(1))
```

# tuple2

```
def sommeDiff(a: Int, b: Int) : Tuple2[Int, Int] = Tuple2(a+b, a-b)

def sommeDiff(a: Int, b: Int) = (a+b, a-b)

val res = sommeDiff(3,6)

println(res._1+” et ”+res._2 ) // attention, ce n'est pas un indice, c'est un attribut
```

Remarque : on peut avoir des types différentes : (“coucou”, 45)  
                  ( “identifiant”, (“Jean”, “paul”) )

# tuple2

Les Tuples sont des case class

Ils ont des déconstructeurs

```
val (s, d) = sommeDiff(3,6)
```

est la même chose que :

```
val res = sommeDiff(3,6)
val s = res._1
val d = res._2
```

# tuple2

Une propriété des déconstructeurs : l'argument 'je m'en tape' \_

```
val (s, _) = sommeDiff(3,6)
```

Ne fabriquera pas de constante pour la différence

Intérêt de :

```
val (_,_) = sommeDiff(3,6) ?
```

# tupleN

```
val enregistrement = ("jean", "lestienne", 45, "rue du pont", "Paris")
```

```
val (_, nom, _, _, ville) = enregistrement  
// fabrique les deux constantes nom et ville
```

Cf tables et enregistrement des bases de données

# tupleN

```
val nom = ("alfred", "dupond")
val adresse = (3, "rue du carré", "Paris")
val specialite = ("ninja","8ieme dan")

val enregistrement1 = (nom, adresse, specialite)
val enregistrement2 = (nom, adresse)

val enregistrement = enregistrement1

enregistrement match {
    case ( _,nom), ( _,_,ville), _ => println(nom+" habite "+ville)
    case ( _, nom), ( _,_,ville) => println(nom+" habite " + ville)
}
```

# tuple2

```
val nom = ("alfred", "dupond")
```

Peut aussi s'écrire

```
val nom = "alfred" -> "dupond"
```

Est utilisé dans Map

# Map

```
val a = tab(4)  
// tableau ou apply
```

Map : Ne pas confondre avec map  
Idem avec indicage par valeur

```
val mapNombres = Map("un"->1, "deux"->2, "trois"->3)
```

```
println(mapNombres("deux")) // affiche 2
```

```
println(mapNombres("quatre")) // exception !!!!
```

```
println(mapNombres.getOrElse("quatre", -1)) // donne -1 si quatre n'est pas dedans
```

# Map

```
println(mapNombres.getOrElse("quatre", -1) // donne -1 si quatre n'est pas dedans
```

Pas une bonne idée : il faut trouver une valeur qui ne sera pas dans la table. Il vaut mieux utiliser Option[Int] – voir plus loin

ATTENTION : par défaut, une map est immutable !

Pour créer une map **mutable** :

```
val maMap = scala.collection.mutable.Map("un" -> 1, "deux" -> 2)
maMap("cinq") = 5 // on rajoute ou on met à jour une valeur de la map
maMap += "cinq" -> 5
maMap += ("cinq", 5)

maMap -= "cinq"
```

# Map

Pour créer une map **immutable** :

```
val maMap = Map("un" -> 1, "deux" -> 2)
maMap = maMap + "cinq" -> 5 // une nouvelle collection
maMap = maMap + ("cinq", 5)

maMap = maMap - "cinq"
```

# Map

## Itérations

```
maMap.foreach( element => println(element) ) // on récupère des tuples2
```

```
(un,1)  
(deux, 2)  
(trois,3)
```

```
for(element <- maMap) println(element)
```

```
for( (cle, valeur) <- maMap) println(cle+" "+valeur) // oui on peut : for n'est pas une fonction,  
c'est une construction du langage
```

# Map

## Itérations

On ne peut pas écrire :

```
maMap.foreach( (cle, valeur) => println(cle++" "+valeur) )
```

Cela voudrait dire qu'on passe une fonction d'arité 2 à foreach – qui demande une fonction d'arité 1.

MAIS : on peut lui passer une fonction particulière – nommée fonction partielle (voir plus loin).

```
maMap.foreach( { case (cle,valeur) => println(cle++" "+valeur) } )
```

ou encore :

```
maMap.foreach { case (cle,valeur) => println(cle++" "+valeur) }
```

# Map

clé -> valeur et si on veut construire une nouvelle map avec valeur -> clé ?

**Manière Scala-à-la-manière-Java :**

```
var mapInverse = Map[String, Int]()
for( element <- maMap) mapInverse = mapInverse + (element._2, element._1)
```

**Manière Scala :**

```
mapInverse = for( (cle,valeur) <- maMap) yield (valeur, cle)
```

ou bien

```
mapInverse = maMap.map{ case(cle, valeur) => (valeur, cle) }
```

# Map

Pour récupérer seulement les clés (par exemple)

```
val lesCles = for( (cle, _) <- maMap ) yield cle
```

ou

```
val lesCles = maMap.map( element => element._1)
```

ou

```
val lesCles = maMap.map( _ => _._1)
```

ou

```
val lesCles = maMap.map(_._1)
```

```
val lesCles = maMap.keys // prédéfini
```

# Map

Tout un ensemble d'opérateurs sur les collections

Par exemple :

```
println( maMap.count( _.2 %2 == 0) )
```

```
val (mapPartiePaire, mapPartielImpaire) = maMap.partition( _.2 %2 == 0)
```

En fait tout une collection de Map : par défaut, Scala vous fournit une implémentation qui est raisonnablement bonne à tout faire, mais vous pouvez imposer une implémentation :

Immutables

HashMap, IntMap, LisyMap, LongMap, TreeMap

Mutable

HashMap, LinkedHashMap, ListMap, OpenHashMap, WeakHashMap

# Option[]

Peut prendre une valeur (Some[T]), ou pas de valeur (None)

```
val v: Option[Int] = Some(5)
val vr: Option[Int] = None

vr match {
  case Some(valeur) => println(valeur)
  case None => println("pas de valeur")
}
```

Dans le cas des maps :

```
maMap.get("cinq") match {
  case Some(valeur) => "on a trouve "+valeur
  case None => "pas de correspondance dans la table"
}
```

# Either[A,B]

Un type qui peut être soit de type A soit de type B (!)  
vaut left(v1: A) ou bien right(v2:B)

Exemple :

```
def divise(a: Double, b: Double) : Either[String, Double] = {  
    if (b == 0.0) Left("division par zero") else Right(a/b)  
}
```

```
val res = divise( 5, 0)  
println( res match {  
    case Left(s) => s  
    case Right(v) => " le resultat est "+v  
})
```

# Set

Ensemble de valeurs sans répétition (pas d'ordre particulier)

Attention implémentations différentes (mais un Set par défaut).

Immutable

BitSet, HashSet, TreeSet

Mutable

BitSet, HashSet, LinkedHashSet, TreeSet, LinkedHashSet

ParTreeMap, ParHashMap

Vector  
Stack  
Queue  
Range  
String  
ArrayBuffer

À voir...

# Accès

Attention, chaque collection a des propriétés qui permettent des accès (itération, ordre, parallélisme etc..)

Il est très très important d'étudier en détail les accès, sinon vous allez recréer des accès qui existent déjà !!!!!

L'étude des collections, ce n'est pas seulement l'étude des types, c'est surtout l'étude des méthodes communes !

Quelques exemple ici mais c'est à vous de consulter et d'analyser Scaladoc sur les collections !

# Accès

Conversion d'une collection en une autre :

- toArray
- toBuffer
- toList
- toMap
- toStream
- toString
- toVector

To[]

```
val collec = 2 to 10 by 2 (type Range)
val collecListe = collec.to[List]
```

# Accès

Rajouter / enlever

$c + e$

$c + (e_1, \dots, e_n)$

$c += e$

$c += (e_1, e_2, \dots, e_n)$

$c -= e$

$c -= (e_1, \dots, e_n)$

$c := e$

$e +: c$

$e :: c$

# Accès

Mettre à jour

$c(k) = e$

$c.update(k, e)$

Ensemblistes

$c ++ c2$

$c - c2$

$c2 ++=: c$

$c ::: c2$  liste rajout

$c | c2$  union

$c & c2$  intersection

$c &^~ c2$  différence

size  
count(*pred*)  
isEmpty  
nonEmpty  
sameElement c2  
forall(*pred*)  
exists(*pred*)  
sum  
product  
min  
max  
take(n)  
drop(n)  
splitAt(n)  
takeRight(n)  
dropRight(n)

takeWhile(*pred*)  
dropWhile(*pred*)  
filter(*pred*)  
filterNot(*pred*)  
partition(*pred*)

map  
reduce  
zip  
par  
seq  
fold  
aggregate  
view  
force

Etc...

# map

Ne pas confondre Map et map !

```
// lexique de traduction
val enAnglais = Map()
    "le" -> "the",
    "chat" -> "cat",
    "mange"->"eats",
    "souris"->"mouse",
    "la"->"the"
)

val texteATraduire = "le CHAT mange la souris"
// fabrique une liste en utilisant l'espace comme
// séparateur
val texteEnListe = texteATraduire split(" ")
```

```
val texteAnglaisListe = texteEnListe
    .toLowerCase
    .map(enAnglais(_ toLowerCase))
        // convertit le résultat en chaîne
val texteTraduit = texteAnglaisListe mkString(" ")
    // affiche "the cat eats the mouse"
    println(texteTraduit)
```

map: même nombre d'éléments, même type de collection, par forcément même type d'éléments

# map

```
val corpusFrancaisAnglais = texteATraduire

.toLowerCase
.split(" ")
.map(v => List(v,enAnglais(v)))
.mkString(" ")

// affiche "List(le, the) List(chat, cat)
//   List(mange, eat) List(la, the) List(souris, mouse)"
println(corpusFrancaisAnglais)
```

entrée dans la **Map**

Attention : une liste de listes !

*List(mange, eat) List(la, the) List(souris, mouse)*

# map

```
val corpusFrancaisAnglais = texteATraduire  
    .toLowerCase  
    .split(" ")  
    .flatMap(v => List(v, enAnglais(v)))  
    .mkString(" ")
```

flatmap permet d'aplatir une liste de liste en liste d'éléments

```
// affiche "le the chat cat mange eat la the souris mouse"  
println(corpusFrancaisAnglais)
```

# map

```
val corpusFrancaisAnglais = texteATraduire

.toLowerCase
.split(" ")
.flatMap(v => List(v,enAnglais(v)))
.toSet // un ensemble ne contient pas de doubles
.mkString(" ")

// affiche "le eat souris la mouse mange chat cat the"
println(corpusFrancaisAnglais)
```

Même si le résultat est toujours une chaîne (mkString), on passe par un ensemble pour éliminer ici les doublons (un corpus est un ensemble de mots)  
Du coup, on perd probablement l'ordre des mots (set est un ensemble donc sans ordre) – on ne peut pas trier (set n'a pas de méthode sort)

# map

Si on ne souhaite pas perdre l'ordre (et trier), il existe le type Seq (on peut aussi reconvertir en liste)

```
val corpusFrancaisAnglais = texteATraduire

    .toLowerCase // minuscules
    .split(" ") // convertir en liste
    .flatMap(v => List(v,enAnglais(v)))
    .toSet // on élimine les doublons
    .toSeq // on convertit en Seq ...
    .sorted // ... pour pouvoir trier
    .mkString(" ") // puis en chaîne de caractères

// affiche "cat chat eat la le mange mouse souris the"
println(corpusFrancaisAnglais)
```

```
.to[scala.collection.SortedSet]
```

# map / reduce

reduceLeft, reduceRight

```
def reduceRight[B >: A](op: (A, B) => B): B
```

La signature montre qu'on obtient forcément un sous-type des éléments de la collection !

```
val nombres = 1 to 5

// affiche :
//"(1,2) (2,3) (6,4) (24,5) 120"
println(
    nombres.reduceLeft(
        (a,b) => { print(("+" + a + "," + b + ") ")); a * b } )
)
```

# map / reduce

foldRight, foldLeft : reduce utilise les éléments de la collection : fold permet de rajouter un élément en plus

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

? Deux séries d'arguments ?    def reduceRight[B >: A](op: (A, B) => B): B

```
val maListe = List(1, 2, 4, 1, 3, 4)

// affiche :
// "Set()"
// Set(1)
// Set(1,2)
// Set(1,2,4)
// Set(1,2,4)
// Set(1,2,4,3)
// Set(1,2,4,3)"
println(
    maListe.foldLeft(Set[Int]())
        ((s,e) => { println(s); s + e } )
)
```

# map / reduce

foldRight, foldLeft : reduce utilise les éléments de la collection : fold permet de rajouter un élément en plus

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

? Deux séries d'arguments ?    def reduceRight[B >: A](op: (A, B) => B): B

```
val maListe = List(1, 2, 4, 1, 3, 4)

// affiche :
// "Set()"
// Set(1)
// Set(1,2)
// Set(1,2,4)
// Set(1,2,4)
// Set(1,2,4,3)
// Set(1,2,4,3)"
println(
    maListe.foldLeft(Set[Int]())  
        ((s,e) => { println(s); s + e } )
)
```

toSet

(s,e) à gauche, l'élément de départ s == z  
à droite, un élément de la collection e

à chaque étape , on rajoute e à s, puis le résultat  
(s augmenté de e) est réinjecté par (s, e) =>

en gros **s = s+e** pour tous les **e** et **s** de départ = Set()

## map / reduce

```
println(  
  List(1, 1, 2, 1, 3, 2). foldLeft(Map[Int,Int]())  
    ( (histo,e) => histo + (e -> (histo.getOrDefault(e,0) + 1)) )  
)
```

## map / reduce

```
println(  
    List(1, 1, 2, 1, 3, 2). foldLeft(Map[Int, Int]())  
        ( (histo,e) => histo + (e -> histo.getOrDefault(e,0) + 1)) )  
)
```

```
// affiche "Map(1 -> 3, 2 -> 2, 3 -> 1)"  
// 3 occurrences de 1, 2 de 2, une de 3
```

# map / reduce

Autre syntaxe pour foldLeft /:

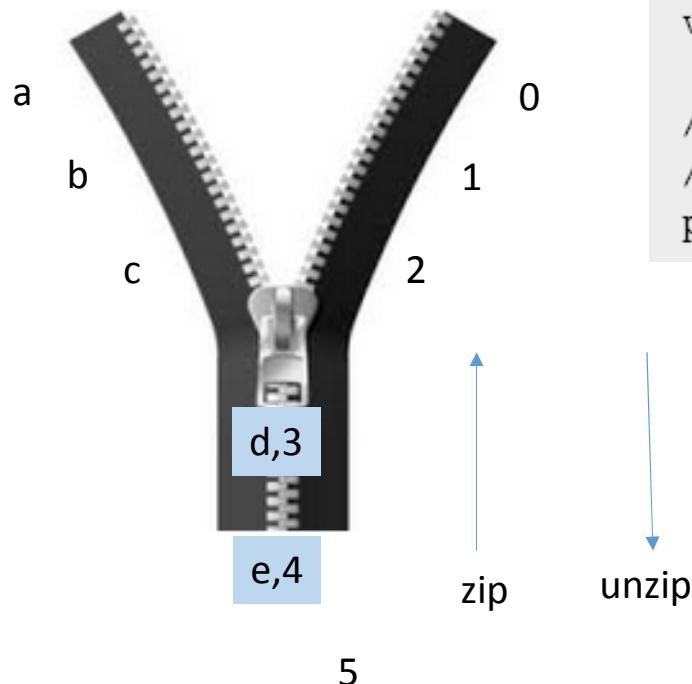
```
println(  
    (Map[Int,Int]() /: List(1, 1, 2, 1, 3, 2))  
        ( (histo,e) => histo + (e -> (histo.getOrDefault(e,0) + 1)) )  
)
```

Même chose que

```
println(  
    List(1, 1, 2, 1, 3, 2). foldLeft(Map[Int,Int]())  
        ( (histo,e) => histo + (e -> (histo.getOrDefault(e,0) + 1)) )  
)
```

# map / reduce / zip

```
// affiche "Vector((a,0), (b,1), (c,2), (d,3), (e,4))"  
println("abcde" zip (0 to 5))
```



```
val score = List( ("Tom",12), ("Jerry",2), ("Bob",9) )  
  
// affiche "(List(Tom, Jerry, Bob),List(12, 2, 9))"  
// le résultat est un couple de collections  
println(score unzip)
```

Voir aussi `zipWithIndex` et `zipAll`

# par

Quand le résultat ne dépend pas d'un ordre de parcours de collection, il est souvent possible de //

```
val debut = System.nanoTime

val r = 0 to 10000000

for (i <- (0 to 100)) {

    val cpt = r.count(_ % 3 == 0)
    // combien de multiples de 3

}

val micros = (System.nanoTime - debut) / 1000

// affiche "5556955 microsecondes"
println("%d microsecondes".format(micros))
```

C'est le cas de count ici

(par exemple)

# par

```
val debut = System.nanoTime

val r = 0 to 10000000

for (i <- (0 to 100)) {

    val cpt = r.par.count(_ % 3 == 0)
    // combien de multiples de 3

}

val micros = (System.nanoTime - debut) / 1000
// affiche "2611619 microsecondes"
println("%d microsecondes".format(micros))
```

(plus rapide – le gain dépend du type de la collection, du traitement etc.)

par



# par

Quel ordre ?

```
// affiche "bonjour"  
"bonjour".foreach(print)  
  
// 'peut' afficher : "onurjbo"  
"bonjour".par foreach(print)
```

Variables partagées  
(normalement à éviter en fonctionnel)

```
var taille = 0  
  
for (i <- "bonjour".par) taille = taille + 1  
  
// affiche "5" ou "6" ou "7" ou?  
println(taille)
```

Bon plan : **si le traitement ne dépend pas de l'ordre ou de variables partagées**, on peut tenter d'accélérer une boucle par :

```
for(i <- (1 to 10).par) { ... } // le contenu du bloc sera exécuté sur plusieurs Threads (si possible)
```

# par / fold/ reduce/ aggregate

reduceLeft, reduceRight, foldRight, foldLeft : ordre particulier : pas de *par*  
reduce et fold : versions *par*.

```
def reduce[U >: T](op: (U, U) => U): U
```

Attention : il faut que op soit associative ! *a minima* :

$$(a \text{ op } b) \text{ op } c == a \text{ op } (b \text{ op } c)$$

Attention, fold ne s'utilise pas de la même manière que foldLeft  
À voir !!!

# par

Pour faire plus ou moins la même chose que **foldLeft**, utiliser **aggregate**

```
maListe.foldLeft(Set[Int]())((s, e) => s + e )
```

Par pas possible

Deux opérations // : 1. on rajoute un élément à un ensemble 2. on fusionne deux ensembles

```
maListe.par.aggregate(Set[Int]())((s,e) => s+e, (s1,s2) => s1 ++ s2)
```

Par possible

# Lazy view / force

```
val v = (1 to 4) map(_ * 2) map(e => e*e) map( _ /3)
```

La collection 1 to 4 (range) est transformée en :

Collection(2, 4, 6, 8) (Vector)

Puis en

Collection(4, 16, 36, 64) (Vector)

Puis en Collection

Collection(4/3, 26/3, 36/3, 64/3) (Vector)

3 collections sont créées

Ca aurait été mieux de faire :

```
val v = (1 to 4) map ( e => ((e*2)*(e*2))/3 )
```

# Lazy view / force

Ca aurait été mieux de faire :

```
val v = (1 to 4) map ( e => ((e*2)*(e*2))/3 )
```

Plutôt que

```
val v = (1 to 4) map(_ * 2) map(e => e*e) map( _ /3)
```

Mais pb :

```
def f1(x : Int) = .....  
def f2....  
def f3(x : Int) = { if (g(x) > 4) x*2 else x*x ;
```

```
val v = (1 to 4) map( f1 ) map( f2 ) map( f3)
```

Composition des fonctions ?

# Lazy view / force

Ca aurait été mieux de faire :

```
val v = (1 to 4) map ( e => ((e*2)*(e*2))/3 )
```

Plutôt que

```
val v = (1 to 4) map(_ * 2) map(e => e*e) map( _ /3)
```

Mais pb :

```
def f1(x : Int) = ..... // super long  
def f2....           // hyper-trop-long  
def f3(x : Int) = ..... // une-usine-à-gaz-monstueuse-de-la-mort-qui-tue
```

```
val v = (1 to 40000000000000000000) map( f1 ) map( f2 ) map( f3 )
```

mais finalement, on n'utilisera que v(4567) et v(32019)

Espace mémoire + temps de calcul

# Lazy view / force

Transforme la collection en paresseuse

```
val v = (1 to 4).view map(_ * 2) map(e => e*e) map( _ /3)  
  
// affiche "SeqViewMMM(...)"  
// MMM: trois map 'gelés'  
println(v)  
  
// affiche "21"  
println(v(3)) Ici l'élément 3 est évalué (mais pas les autres)  
  
// affiche "Vector(1, 5, 12, 21)"  
println(v.force)
```

On force l'évaluation complète

Voir TensorFlow

# Lazy view / force

```
val resultat =  
    maCollection.traitemet1(...).traitemet2(...).traitemet3(...)
```

Appel fonctionnel classique

```
val resultat = maCollection.view  
  
.traitemet1(...).traitemet2(...).traitemet3(...)  
  
.force
```

Composition

# Collections Scala / Java

scala.collection	-> conversion implicite ->	java.util
Iterable	asJavaCollection	Collection
Iterable	asJavaIterable	Iterable
Iterator	asJavaIterator	Iterator
Iterator	asJavaEnumeration	Enumeration
Seq	seqAsJavaList	List
mutable.seq	mutableSeqAsJavaList	List
mutable.buffer	bufferAsJavaList	List
Set	setAsJavaSet	Set
mutable.Set	mutableSetAsJavaSet	Set
Map	mapAsJavaMap	Map
mutableMap	mutableMapAsJavaMap	Map
Map	asJavaDictionary	Dictionary
mutable.ConcurrentMap	asJavaConcurrentMap	concurrent.ConcurrentMap

# Collections Scala / Java

java.util	-> conversion implicite ->	scala.collection
Collection	collectionAsScalaIterable	Iterable
Iterable	iterableAsScalaIterable	Iterable
Iterator	asScalaIterator	Iterator
Enumeration	enumerationAsScalaIterator	Iterator
List	asScalaBuffer	mutable.Buffer
Set	asScalaSet	mutableSet
Map	mapAsScalaMap	mutable.Map
Dictionnary	dictionaryAsScalaMap	mutable.Map
Properties	propertiesAsScalaMap	mutable.Map
concurrent. ConcurrentMap	asScalaConcurrentMap	mutable. ConcurrentMap

# Fonctions et méthode

Équivalent d'une méthode Java : est définie dans un objet et a accès à this  
(contexte locale)

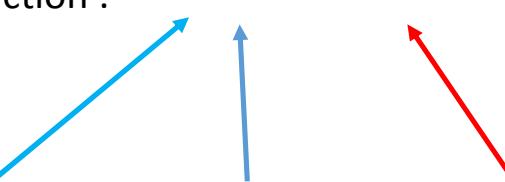
```
def sommeMethode(a: Double, b: Double) : Double = a+b  
  
// affiche "7.0"  
println(sommeMethode(3.0, 4.0))
```

# Fonctions et méthode

En Java, une fonction est une instance qui possède la méthode à invoquer pour évaluer la fonction) (pas de type *function* : cela dépend de l'utilisation – cf *comparator*)

En Scala : on peut créer une fonction comme instance de Function : arité 2 et résultat

```
class fSomme extends Function2[Double, Double, Double] {  
    def apply(a:Double, b: Double): Double = a+b  
}  
val sommeFonction = new fSomme  
  
// affiche "7.0"  
println( sommeFonction(3.0, 4.0) )
```



# Fonctions et méthode

```
class fSomme extends Function2[Double, Double, Double] {  
    def apply(a:Double, b: Double): Double = a+b  
}  
val sommeFonction = new fSomme  
  
// affiche "7.0"  
println( sommeFonction(3.0, 4.0) )
```

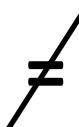
```
val sommeFonction = new Function2[Double, Double, Double] {  
    def apply(a:Double, b: Double): Double = a+b  
}  
  
// affiche "7.0"  
println( sommeFonction(3.0, 4.0) )
```

# Fonctions et méthode

```
def sommeMethode(a: Double, b: Double) : Double = a+b  
// affiche "7.0"  
println(sommeMethode(3.0, 4.0))
```

```
val sommeFonction = new Function2[Double, Double, Double] {  
    def apply(a:Double, b: Double): Double = a+b  
}  
  
// affiche "7.0"  
println( sommeFonction(3.0, 4.0) )
```

sommeMethode n'est pas l'instance de quoi que ce soit  
C'est une **méthode**



sommeFonction est un **objet**, c'est une instance qui peut être manipulée comme n'importe quel objet (stockage, passée en argument etc..)

```
val sommeFonction : (Double, Double) => Double  
= (a: Double, b: Double) => a+b
```

# Fonctions et méthode

```
val fonction1 = sommeFonction  
  
// affiche "7.0"  
println(fonction1(3.0, 4.0))
```

Pas d'évaluation de la fonction

```
// non !  
val fonction2 = sommeMethode  
  
println(fonction2(3.0, 4.0))
```

*sommeMethode()*  
Invoque la méthode

```
val fonction2 = sommeMethode _  
  
// affiche "7.0"  
println(fonction2(3.0, 4.0))
```



Demande au compilateur de créer une fonction qui fait un appel à la méthode

Par réellement une conversion

# Fonctions et méthode

Fonction anonyme    `(a: Double, b: Double) => a+b`

```
val sommeFonction = (a: Double, b: Double) => a+b
```

```
// affiche "3628800"  
println( (1 to 10).reduceLeft( (a:Int, b:Int) => a*b ) )
```

Indiquez les types pour les fonctions anonymes !!!

# Fonctions et méthode

## Fonction partielle

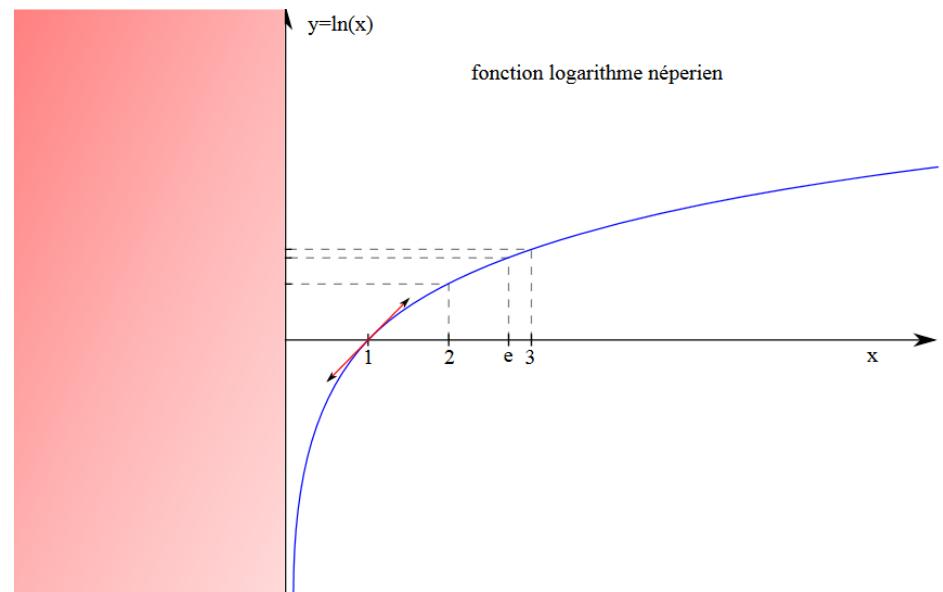
Une fonction partielle est une fonction partielle définie sur ses arguments par exemple `divise(x,y)` n'est pas définie pour  $y == 0$

La racine carrée n'est pas définie pour des valeurs négatives

Le Log (Logn) n'est pas défini pour des valeurs négatives

Ce sont des fonctions partielles

```
def sqrt(x:Double if x>=0) {  
} // impossible en scala
```



# Fonctions et méthode

## Fonction complètes

val racineCarree

= (x: Double) => { Math.sqrt(x) }

ou encore :

val racineCarree: (Double) => Double = (x: Double) => { Math.sqrt(x) }

Fonction définie sur tous les Doubles (y compris < 0)

Ce sont ici des fonctions complètes ! (pas des partielles)

Un calcul avec une valeur négative donne NaN (Not a Number)

Arrêt net

# Fonctions et méthode

Fonction partielle :

```
val racineCarree: (Double) => Double = { case x: Double if (x>=0) => Math.sqrt(x) }
```

Le **case** fabrique une fonction **partielle** qui teste si le paramètre est **Double** et  $\geq 0$

racineCarre(x) n'est **définie** que pour x:Double et  $x \geq 0$

*Empêche l'évaluation de Math.sqrt pour un x <0*

**Préconditions en algorithmique**

# Fonctions et méthode

Fonction partielle :

Pas tout à fait

val racineCarree: (Double) => Double = { case x: Double if (x>=0) => Math.sqrt(x) }

val racineCarree: Function[Double, Double] = ...

val racineCarree: **PartialFunction[Double, Double]** = { case x: Double if (x>=0) => Math.sqrt(x) }

# Fonctions et méthode

Exemple : réaliser une function qui prend en argument un entier ou une String

```
calculeDouble( x: Int ou_bien String) : Int ou_bien String = {  
    x *2  
}
```

# Fonctions et méthode

Exemple : réaliser une function qui prend en argument un entier ou une String

```
val calculeDouble : PartialFunction[Any, Any] = {  
    case i:Int => i * 2  
    case s:String => s * 2  
}  
// affiche "5"  
println( calculeDouble(5) )  
// affiche "**"  
println( calculeDouble("**") )
```

Fonction qui prend un Int ou bien une String et qui renvoie un Int ou une String

# Fonctions et méthode

```
// affiche "true"
println(calculeDouble.isDefinedAt("yes"))
```

# Fonctions et méthode

```
val CinqDivisePar: PartialFunction[Int, Int] = {  
    case i: Int => 5 / i      ←————— case i : Int if (i != 0) => 5 / i  
}  
// affiche "true"  
println(CinqDivisePar.isDefinedAt(0))          → false  
  
// exception!! java.lang.ArithmetricException  
// division par zéro  
println(CinqDivisePar(0))
```

# Fonctions et méthode

```
maMap.foreach ( c: Tuple2[Int, Int] => println(c._1 + " " + c._2) )
```

maMap est une Map de couples d'entiers

foreach ne prend qu'un paramètre (c, ici)

On peut décider de passer une fonction partielle qui prend des tuples

# Fonctions et méthode

```
maMap.foreach( c: Tuple2[Int, Int] => println(c._1 + " "+c._2) )
```

```
maMap.foreach( { case c: Tuple2[Int, Int] => println(c._1 + " "+c._2) } )
```

En utilisant un déconstructeur (puisque c'est un Tuple – donc une case class)

```
maMap.foreach( { case (x,y) : Tuple2[Int, Int] => println(x + " "+y) } )
```

```
maMap.foreach( { case (x,y) => println(x + " "+y) } )
```

Comme l'arité de foreach est de 1 (un seul paramètre) : on peut enlever les parenthèses

```
maMap.foreach { case (x,y) => println(x + " "+y) }
```

# Fonctions

## Fonction d'ordres supérieurs

Une fonction d'ordre supérieur est une fonction qui peut prendre une autre fonction dans ses paramètres

```
def composeFonctionsSurV( f1: Double => Double,  
                           f2: Double => Double,  
                           v: Double ) : Double = f1(f2(v))
```

```
// affiche "17"  
println(composeFonctionsSurV( 2 + _ , 3 * _ , 5))
```

# Fonctions

## Fonction d'ordres supérieurs

Quel est le type de cette fonction :

```
def composeFonctions(  
    f1: Double => Double,  
    f2: Double => Double ) : Double => Double =  
{ x => f1(f2(x)) }
```

# Fonctions

(Double => Double, Double => Double) => Double => Double

```
def composeFonctions(  
    f1: Double => Double,  
    f2: Double => Double ) : Double => Double =  
{ x => f1(f2(x)) }
```

# Fonctions

(Double => Double, Double => Double) => Double => Double

composeFonctions( 2 + \_, 3 \* \_) est une fonction qui prend un Double et retourne un Double :

```
(composeFonctions( 2 + _, 3 * _)) (5)  
// 17
```

# Fonctions

(Double => Double) (5)

(composeFonctions( 2 + \_, 3 \* \_)) (5)

composeFonctions( 2 + \_, 3 \* \_)(5)

composeFonctions( 2 + \_, 3 \* \_ ) (5)

# Fonctions



[Plus d'images](#)

## Haskell Curry



Logicien

Haskell Brooks Curry était un logicien et mathématicien américain. Ses travaux ont posé les bases de la programmation fonctionnelle. [Wikipédia](#)

**Naissance :** 12 septembre 1900, Millis, Massachusetts, États-Unis

**Décès :** 1 septembre 1982, State College, Pennsylvanie, États-Unis

**Père :** Samuel Silas Curry

**Livres :** Foundations of mathematical logic, plus...

**Formation :** Université de Göttingen (1930), Université Harvard

**Influences :** Moses Schönfinkel, David Hilbert, Alfred North Whitehead

wikipedia

# Haskell

An advanced, purely functional programming language

Declarative, statically typed code.

```
primes = filterPrime [2...]
where filterPrime (p:xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

## Try it!

Type Haskell expressions in here.

$\lambda$

## Got 5 minutes?

Type `help` to start the tutorial.

Or try typing these out and see what happens (click to insert):

```
23 * 36 or reverse "hello" or foldr (:) []
[1,2,3] or do line <- getLine; putStrLn line or
readFile "/welcome"
```

These IO actions are supported in this sandbox.

# Fonctions

## Moses Schönfinkel

Mathématicien



Moses Schönfinkel, né le à 4 septembre 1889 Ekaterinoslav – 1942, Moscou est un logicien et mathématicien juif soviétique. [Wikipédia](#)

**Naissance :** 4 septembre 1889, Dnipro, Ukraine

**Décès :** 1942, Moscou, Russie

**Formation :** Novorossiysk University of Odessa

# Fonctions

Curryfication  
Schönfinkelisation

```
def ajouteA( a: Int ) = ( b: Int ) => a + b
```

ajouteA(5) est la même chose que :

$(b: \text{Int}) \Rightarrow 5 + b$

# Fonctions

curryfication

```
println( ajouteA(5)(6) )
```

on peut écrire :

```
def ajouteCurryfie(x:Int)(y:Int) : Int = x + y
```

et

```
val ajoute5 = ajouteCurryfie(5) _
```

```
println(ajoute5(6))
```

# Fonctions

curryfication

```
def composeFonctionsSurV(  
    f1: Double => Double,  
    f2: Double => Double,  
    v: Double ) : Double = f1(f2(v))
```

# Fonctions

```
def composeFonctionsSurV(
```

curryfication

```
f1: Double => Double,
```

```
f2: Double => Double) (v: Double) : Double
```

```
= f1(f2(v))
```

```
val composeFonctions = composeFonctionsSurV( 2 + _, 3 * _ ) _
```

```
println(composeFonctions(5))
```

# Fonctions

curryfication

```
def baliseHTML( balise: String)(texte: String)  
= s"<${balise}>${texte}</${balise}>"
```

```
def br = baliseHTML("br") _
```

```
def h1 = baliseHTML("h1") _
```

```
def span = baliseHTML("span") _
```

```
// affiche "<br>hello</br>"  
println(br("hello"))
```

```
// affiche "<h1><span>bonjour</span></h1>"  
println(h1 span "bonjour")
```

# Fonctions

Structure de contrôle

```
def repete(nFois: Int, block: () => Unit) {  
    for(i <- 1 to nFois) block()  
}
```

Sans curryfication

```
repete(3, () => print("hip ") )  
  
// affiche "hip hip hip hourra"  
  
println("hourra")
```

# Fonctions

Structure de contrôle

```
repete(3, print("hip ") ) // sans arguments
```

```
repete(3, { print("hip "); print("!") } )
```

Et si on utilise en plus la curryfication :

# Fonctions

Structure de contrôle

Avant

```
def repete (nFois: Int, block: => Unit) {  
    for(i <- 1 to nFois) block  
}
```

Après

```
def repete (nFois: Int) ( block: => Unit ) {  
    for(i <- 1 to nFois) block  
}  
  
repete (3) ( { print("hip "); print("! ") } )
```

# Fonctions

Structure de contrôle

```
def repete (nFois: Int) ( block: => Unit ) {  
    for(i <- 1 to nFois) block  
}  
  
repete (3) ( { print("hip "); print("! ") } )  
  
repete (3) { print("hip "); print("! ") }  
  
repete (3) {  
    print("hip ");  
    print("! ")  
}
```

# Fonctions

Structure de contrôle

```
def while2 ( condition: => Boolean) (bloc: => Unit) {  
    if (condition == true) {  
        bloc  
        while2(condition)(bloc)  
    }  
}
```

```
var a = 0  
  
// affiche "01234"  
while2( a < 5 ) {  
    print(a)  
    a = a+1  
}
```

# Fonctions

Structure de contrôle

```
var a = 0  
  
    // affiche "01234"  
repeat {  
    print(a)  
    a = a +1  
} until (a > 5)
```

# Fonctions

Structure de contrôle

```
repeat { print(a) ; a = a +1 } until (a > 5)
```

# Fonctions

Structure de contrôle

```
repeat ({ print(a) ; a = a +1 }) until (a > 5)
```

# Fonctions

Structure de contrôle

```
repeat ({ print(a) ; a = a +1 }) until (a > 5)
```

```
until (a > 5)
```

```
.until(a > 5)
```

# Fonctions

Structure de contrôle

```
def repeat (bloc: => Unit) = new {  
    def until (c: => Boolean) {  
        bloc; while (! c) bloc  
    }  
}
```

```
var a = 0  
  
// affiche "01234"  
repeat {  
    print(a)  
    a = a +1  
} until (a > 5)
```

# Fonctions

exemple

```
def combienDeTempsDure( code : => Unit ) = {  
    val debut = System.currentTimeMillis()  
    code  
    val dif = System.currentTimeMillis() - debut  
    dif  
}
```

# Fonctions

utilisation

```
val duree = combienDeTempsDure {  
    var a = 5  
    var b = function(a, 56)  
    b.map( g => g.toString.....  
        .....  
        .....  
    }  
    println(duree)
```

# Pattern Matching

Filtrage par motif

```
// Java seulement
switch(entier) {

    case 1 : System.out.println("un"); break;
    case 2 : System.out.println("deux"); break;
    case 3 : System.out.println("trois"); break;
    default : System.out.println("soleil")
}

// en Scala
entier match {

    case 1 => println("un")
    case 2 => println("deux")
    case 3 => println("trois")
    case _ => println("soleil")

}
```

# Pattern Matching

Filtrage par motif

```
def analyseArgument(arg: String) = arg match {  
    // -h ou bien --help  
    case "-h" | "--help" => afficheAide  
  
    // -v ou bien --version  
    case "-v" | "--version" => afficheVersion  
  
    case erreur => afficheInconnu(erreur)  
}
```

# Pattern Matching

Filtrage par motif

Détection de type

```
obj match {
    case s : String => s.toUpperCase
    case entier : Int => s"entier($entier)"
    case _ : Array[Int] => "tableau d'entiers"
    case _ : Array[String] => "tableau de chaînes"
    case _ : List[Int] => "Liste d'entiers" // attention !
    case _ : List[String] => "Liste de chaînes" // attention !
    case _ => obj.toString
}
```

# Pattern Matching

Filtrage par motif

Détection de type

```
def tester(obj: Any): String = {  
    obj match {  
        // NON ! obj est de type Any donc pas nécessairement String  
        // Erreur à la compilation!  
        case s: String => obj.toUpperCase  
        case _ => obj.toString  
    }  
}
```

# Pattern Matching

Filtrage par motif

Détection de type

```
case _ : Array[Int] => "tableau d'entiers"  
case _ : Array[String] => "tableau de chaînes"
```

Récupère tous les types de liste : *erasure* type

```
case _ : List[Int] => "Liste d'entiers" // attention!  
case _ : List[String] => "Liste de chaînes" // attention!
```

```
case _ : List[_] => "Liste"
```

# Pattern Matching

Filtrage par motif

Détection de type

Conditionnelle de clause (ou de garde)

```
nom match {  
  
    case s: String if (s.startsWith("M.")) =>  
    case s: String if (s.startsWith("Mme")) => demanderNomJeuneFille()  
    case _ => ""  
  
}
```

# Pattern Matching

Filtrage par motif

Détection de type

Attention à ce que vous mettez à droite de **case** :

Minuscule : ok, ca devient une variable à créer par le déconstructeur

```
case s: String =>
```

Majuscule : on prend la valeur de la variable :

```
val Admin = "administrateur:0"
val Anon = "anonymous"

nom match {
    case Admin => println("accès autorisé")
    case Anon => println("accès limité")
    case utilisateur => println(s"$utilisateur identifiez-vous")
}
```

```
case "aministrateur:0" =>
```

# Pattern Matching

Filtrage par motif

```
val admin = "administrateur:0"  
val anon = "anonymous"  
nom match {  
    case 'admin' => println("accès autorisé")  
    case 'anon' => println("accès limité")  
    case utilisateur => println(s"$utilisateur identifiez-vous")  
}
```

Également possible en utilisant la quote inversée (dans ce cas, majuscule ou minuscule)

# Pattern Matching

Filtrage par motif

## Tuples

```
case (a:Int, _, b:String) =>
```

```
unObjet match {  
  
    case (1,_) => println("ce Tuple commence par 1")  
    case (a,b) => println(s"une paire ($a,$b)")  
    case (a,_,b) => println(s"Tuple3: $a en premier et $b en dernier")  
    case (_,_,_,_) => println("un tuple de 4 valeurs")  
    case _: Tuple5 => println("un tuple de 5 valeurs")  
    case _ => println("n'importe quoi d'autre")  
}
```

# Pattern Matching

Filtrage par motif

## Tuples

```
// affiche "2 et 4"
(1, (2, (3, 4))) match {
    case (1, (a, (3, b))) => println("$a et $b")
    case _ => println("")
}
```

# Pattern Matching

Filtrage par motif

## Tuples

```
// affiche "à droite de 2 on trouve (3,4)"  
(1, (2, (3, 4))) match {  
  
    case (1, (a, interne @ (_, _)))  
        => println(s"à droite de $a on trouve $interne")  
    case _ => println("")  
}
```

*interne* est ici apparié au tuple

Si on met (b,c) on a 2 variables

*interne* : une seule variable : le couple

```
case (1, (a, interne)) =>
```

```
case (1, (a, interne @ (_, _)))
```

'de la forme de'

# Pattern Matching

Filtrage par motif

## Tuples

Rappel :

```
def analyseArgument(arg: String) = arg match {  
    // -h ou bien --help  
    case "-h" | "--help"    => afficheAide  
  
    // -v ou bien --version  
    case "-v" | "--version" => afficheVersion  
  
    case erreur => afficheInconnu(erreur)  
}
```

# Pattern Matching

Filtrage par motif

## Tuples

```
case (1 | 2 | 3, 4 | 7) =>
```

Première partie : 1 ou 2 ou 3

Seconde partie : 4 ou 7

# Pattern Matching

Filtrage par motif

## Tuples

Pour faire porter les ou sur des types :

```
case ( a @ ( _ : Int | _ : Double ), _ : Int | _ : String ) =>
```

# Pattern Matching

Filtrage par motif

## Tuples

Attention : pas d'unification réelle (cf Prolog)

```
case (a, a) =>  
    NON !
```

```
case (a, b) if (a == b) =>  
    OK !
```

# Pattern Matching

Filtrage par motif

## Collections (certaines collections)

```
maListe match {  
  
    case 0 :: Nil => "0"  
    case 0 :: _ => "liste 0, ...."  
  
    // reste représente une liste  
    case 7 :: _ :: reste => s"$reste"  
    case a :: b :: _ => s"liste $a,$b,..."  
    case _ => "un autre type de liste"  
  
}
```



Ou pas !

# Pattern Matching

Filtrage par motif

## Collections (certaines collections)

```
maListe match {  
  
    case List(0) => "0"  
    case List(0, _*) => "liste 0, ...."  
    case List(7, _, reste @ _) => s"$reste"  
    case List(a, b, _) => s"liste $a,$b,..."  
    case _ => "un autre type de liste"  
  
}
```

Cherchez unapplySeq dans Scaladoc

# Pattern Matching

Filtrage par motif

## Expressions régulières

```
val date1 = """(\d\d)-(\d\d)-(\d\d\d\d)""".r
```

""" sert à conserver les littéraux \d

\d : digit

.r fabrique une instance d'expression régulière

# Pattern Matching

Filtrage par motif

## Expressions régulières

```
// Déclaration d'une expression régulière
val date1 = """(\d\d)-(\d\d)-(\d\d\d\d)""".r

// Application de l'extraction
val date1(jour, mois, annee) = "25-12-2014"

// affiche "jour:10 mois:12 année:2014"
println(s"jour:$jour mois:$mois année:$annee")
```

# Pattern Matching

Filtrage par motif

## Expressions régulières

```
// affiche "Noël 2014"
"25-12-2014" match {
    case date1("25", "12", annee) => println(s"Noël $annee")
    case _ => println("aujourd'hui pas de cadeau!... ")
}
```

Toujours chaînes de caractères

# Pattern Matching

Filtrage par motif

## apply et unapply

```
class Fraction(val numerateur: Int, val denominateur: Int) {  
  
    override def toString = s"$numerateur/$denominateur"  
}  
  
object Fraction {  
  
    def apply(num : Int, denom : Int) = new Fraction(num, denom)  
}  
  
val deuxTiers = new Fraction(2,3) // construction classique  
  
val cinqDemi = Fraction(5,2) // en utilisant l'objet Fraction  
  
// affiche "5/2"  
println(cinqDemi)
```

# Pattern Matching

Filtrage par motif

## apply et unapply

```
class Fraction(val numerateur: Int, val denominateur: Int) {  
    override def toString = s"$numerateur/$denominateur"  
}  
  
object Fraction {  
    def apply(num : Int, denom : Int) = new Fraction(num, denom)  
}  
  
// affiche "false" !  
println( Frac(5,2) == Frac(5,2) )
```

# Pattern Matching

Filtrage par motif

```
class Fraction(val numerateur: Int, val denominateur: Int) {  
    override def equals(autre: Any) : Boolean = {  
  
        autre match {  
  
            case autreFrac: Fraction =>  
                (( autreFrac.canEqual(this)) &&  
                    (numerateur==autreFrac.numerateur) &&  
                    (denominateur==autreFrac.denominateur))  
            case _ => false  
        }  
    }  
    def canEqual(autre: Any) = autre.isInstanceOf[Fraction]  
    override def hashCode:Int = 2999*(denominateur + 2999*numerateur)  
    override def toString = s"$numerateur/$denominateur"  
}  
  
val s = scala.collection.immutable.HashSet(Fraction(5,2))  
  
// affiche "true" (ok !)  
println(s.contains(Fraction(5,2)))
```

Pas évident

# Pattern Matching

Filtrage par motif

Précondition qui empêche la construction d'un objet

```
class Fraction(n: Int, d: Int) {  
    // SURTOUT pas de dénominateur à 0  
    require(d != 0) // précondition
```

```

class Fraction(n: Int, d: Int) {
    // SURTOUT pas de dénominateur à 0
    require(d != 0) // précondition

    // calcul du pgcd
    private val pgcd = calcPgcd(n.abs, d.abs)

    val numerateur = (if (d < 0) -n else n) / pgcd

    val denominateur = d.abs / pgcd

    private def calcPgcd(a: Int, b: Int): Int =

```

Le calcul du PGCD permet de réduire la fraction  
+ simple pour le test d'égalité  
Du coup, les attributs stockés ne sont pas les  
attributs de construction !

### Calcul du PGCD (Algo d'Euclide)

```

        if (b == 0) a else calcPgcd(b, a % b)

    override def equals(autre: Any) : Boolean = {
        autre match {
            case autreFrac: Fraction =>
                ((autreFrac.canEqual(this)) &&
                 (numerateur==autreFrac.numerateur) &&
                 (denominateur==autreFrac.denominateur))
            case _ => false
        }
    }

    def canEqual(autre: Any) = autre.isInstanceOf[Fraction]
    override def hashCode:Int = 2999*(denominateur + 2999*numerateur)
    override def toString = s"$numerateur/$denominateur"
}

// affiche "5/2"
println(Fraction(10,4))

```

# Pattern Matching

Filtrage par motif

Un **constructeur** fabrique une instance à partir des **attributs de construction**

```
val Frac1 = Fraction(10,4)
```

Un **déconstructeur** retrouve DES **attributs de construction** à partir d'une instance

```
val (n1,d1) = Frac1.extracteur
```

Fraction.unapply( Fraction(3,2) )

Méthode à définir

```
def unapply(instance: Any): Option[T]
```

# Pattern Matching

Filtrage par motif

À définir dans l'objet compagnon

```
object Fraction {  
    def apply(num : Int, denom : Int) = new Fraction(num, denom)  
  
    def unapply(instance: Fraction): Option[(Int, Int)] = {  
        if (instance == null) None  
        else Some((instance.numerateur, instance.denominateur))  
    }  
}
```

# Pattern Matching

Filtrage par motif

```
// val cinqDemi = Fraction(5,2)
val cinqDemi = Fraction.apply(5,2)

// affiche "Some((5,2))"
println(Fraction.unapply(cinqDemi))
```

```
val Frac1 = Fraction(10,4)

// affiche "Frac1 = new Fraction(5,2)"
Frac1 match {

    case Fraction(n, d) => println(s"Frac1 = new Fraction($n,$d)")
    case _ => println("impossible d'extraire les arguments")

}
```

# Pattern Matching

Filtrage par motif

```
val frac = Fraction(10,30)

val Fraction(num, den) = frac

// affiche "1 et 3"
println(s"$num et $den")
```

# Pattern Matching

Filtrage par motif

Le déconstructeur peut également être utilisé dans les tests d'égalité :

```
val frac = Fraction(10, 30)

if (frac == Fraction(1,3)) {

    // affiche "frac se réduit en 1/3 !"
    println("frac se réduit en 1/3 !")

}
```

# Pattern Matching

Filtrage par motif

Dans le cas général, **unapply** n'est pas simple à définir

Ce n'est pas toujours possible de retrouver les arguments du constructeur à partir de l'état de l'objet

Ces arguments doivent être compatibles avec le calcul du Hash et l'égalité

Dans le cas le plus simple SANS transformation des arguments, on utilise une **case class**

# Pattern Matching

Filtrage par motif

case class

**définition automatique de copy, toString, equals, hashCode, apply, unapply**

```
case class Personne( nom : String, anneeNaissance : Int )
```

```
val jane = Personne("Calamity", 1852)  
  
val sonNom = jane.nom  
val sonAnneeNaissance = jane.anneeNaissance
```

```
val Personne(qui, quand) = jane  
if (jane == Personne(qui, quand)) {  
  
    println(jane) // affiche "Personne("Calamity", 1852)"  
}
```

Immu(t)able !!!!

# Pattern Matching

Filtrage par motif

case class

**définition automatique de copy, toString, equals, hashCode, apply, unapply**

```
// notez le "var"  
case class Personne( var nom: String, var anneeNaissance: String)
```



→ Mu(t)able !!!!  
(pas franchement un bonne idée)

# Pattern Matching

Filtrage par motif

case class

**Attention !**

```
class Personne(nom: String)
val p = new Personne("joe")

// Erreur ! nom n'est pas accessible!
println(p.nom)
```

```
case class Personne/* val */ nom: String)
val p = new Personne("joe")

// affiche "joe"
println(p.nom)
```

# Pattern Matching

Filtrage par motif

case class

**Copy (quelle idée avec une classe immuable – en fait, si)**

```
case class Personne(prenom: String, nom: String, nation:String)

val pere = Personne("Jean","Dupond","France")
// affiche "Personne(Jean,Dupond,France)"
println(pere)

val fille = pere.copy(prenom="Julie")
// affiche "Personne(Julie,Dupond,France)"
println(fille)
```

# Pattern Matching

## Types Algébriques

 WIKIPÉDIA  
L'encyclopédie libre

- [Accueil](#)
- [Portails thématiques](#)
- [Article au hasard](#)
- [Contact](#)
- [Contribuer](#)
- [Débuter sur Wikipédia](#)
- [Aide](#)
- [Communauté](#)
- [Modifications récentes](#)
- [Faire un don](#)
- [Outils](#)
- [Pages liées](#)
- [Suivi des pages liées](#)
- [Importer un fichier](#)
- [Pages spéciales](#)
- [Lien permanent](#)
- [Informations sur la page](#)
- [Élément Wikidata](#)
- [Citer cette page](#)
- [Imprimer / exporter](#)
- [Créer un livre](#)
- [Télécharger comme PDF](#)

## Filtrage par motif

Non connecté Discussion Contributions Crée un compte Se connecter

Article Discussion Lire Modifier Plus Rechercher dans Wikipédia

## Type algébrique de données

 Cet article est une ébauche concernant l'informatique.

Vous pouvez partager vos connaissances en l'améliorant ([comment ?](#)) selon les recommandations des projets correspondants.

Un **type algébrique** est une forme de **type de données composite**<sup>note 1</sup>, qui combine les fonctionnalités des **types produits** (*n*-uplets ou enregistrements) et des **types sommes** (union disjointe). Combinée à la **récursivité**, elle permet d'exprimer les données structurées telles que les listes et les arbres.

**Sommaire** [masquer]

- [1 Définitions](#)
  - [1.1 Type produit](#)
  - [1.2 Type somme](#)
  - [1.3 Type énuméré](#)
  - [1.4 Type algébrique](#)
- [2 Polymorphisme](#)
  - [2.1 Type algébrique généralisé](#)
- [3 Récursivité](#)
  - [3.1 Listes](#)
  - [3.2 Arbres](#)
- [4 Abstraction](#)
- [5 Voir aussi](#)
- [6 Notes & références](#)

# Pattern Matching

Filtrage par motif

On souhaite pouvoir manipuler directement et par programmation  
des arbres de formules  
pour les évaluer, les dériver, les simplifier

On souhaite rajouter au langage les structures de données nommées capables  
de générer des fonctions (par exemple)

Exemple

```
uneFonction = ????
uneAutreFonction = derivée(uneFonction)
println( uneAutreFonction(45) )
```

# Pattern Matching

Filtrage par motif

## case class et pattern matching

Toutes les sous-classes de Expr sont dans ce fichier (interdiction d'en créer d'autres)

```
sealed abstract class Expr
case class X() extends Expr
case class C(v : Double = 0) extends Expr

// e1 + e2
case class Plus(e1 : Expr, e2 : Expr) extends Expr

// e1 * e2
case class Mult(e1 : Expr, e2 : Expr ) extends Expr
```

# *Pattern Matching*

case class et pattern matching

```
sealed abstract class Expr
case class X() extends Expr
case class C(v : Double = 0) extends Expr

// e1 + e2
case class Plus(e1 : Expr, e2 : Expr) extends Expr

// e1 * e2
case class Mult(e1 : Expr, e2 : Expr ) extends Expr
```

```
val e = Mult( X(), Plus( X(), C(5) ))
```

# Pattern Matching

Type algébrique

```
def enChaine(e: Expr): String = {
    e match {
        case X() => "x"
        case C(v) => v.toString
        case Plus(e1, e2) => s"(${enChaine(e1)} + ${enChaine(e2)})"
        case Mult(e1, e2) => s"(${enChaine(e1)} * ${enChaine(e2)})"
    }
}
// affiche "(x * (x + 5.0))"
println(enChaine(e))
```

```
sealed abstract class Expr
case class X() extends Expr
case class C(v : Double = 0) extends Expr

// e1 + e2
case class Plus(e1 : Expr, e2 : Expr) extends Expr

// e1 * e2
case class Mult(e1 : Expr, e2 : Expr ) extends Expr
```

# Pattern Matching

## Type algébrique

```
def evaluate(e: Expr, x: Double): Double = {  
  
    e match {  
  
        case X() => x  
        case C(v) => v  
        case Plus(e1, e2) => evaluate(e1, x) + evaluate(e2, x)  
        case Mult(e1, e2) => evaluate(e1, x) * evaluate(e2, x)  
  
    }  
}  
// affiche "50.0"  
println( evaluate(e, 5.0) )
```

```
sealed abstract class Expr  
case class X() extends Expr  
case class C(v : Double = 0) extends Expr  
  
// e1 + e2  
case class Plus(e1 : Expr, e2 : Expr) extends Expr  
  
// e1 * e2  
case class Mult(e1 : Expr, e2 : Expr ) extends Expr
```

# Pattern Matching

Type algébrique

```
def evaluate(e: Expr)(x: Double): Double = {  
    e match {  
        case X() => x  
        case C(v) => v  
        case Plus(e1, e2) => evaluate(e1)(x) + evaluate(e2)(x)  
        case Mult(e1, e2) => evaluate(e1)(x) * evaluate(e2)(x)  
    }  
}  
  
// affiche "50.0"  
println( evaluate(e)(5.0) )
```

```
sealed abstract class Expr  
case class X() extends Expr  
case class C(v : Double = 0) extends Expr  
  
// e1 + e2  
case class Plus(e1 : Expr, e2 : Expr) extends Expr  
  
// e1 * e2  
case class Mult(e1 : Expr, e2 : Expr ) extends Expr
```

# *Pattern Matching*

Type algébrique

```
sealed abstract class Expr
case class X() extends Expr
case class C(v : Double = 0) extends Expr

// e1 + e2
case class Plus(e1 : Expr, e2 : Expr) extends Expr

// e1 * e2
case class Mult(e1 : Expr, e2 : Expr ) extends Expr
```

```
// _ indique les arguments à renseigner
val f = evaluate(e)_
```

```
// affiche "50.0"
println(f(5.0))
```

# Pattern Matching

```
def derive(e: Expr): Expr = {
  e match {
    // X' donne 1
    case X() => C(1)

    // dérivé d'une constante donne 0
    case C(_) => C(0)

    case Plus(e1, e2) => Plus( derive(e1), derive(e2) )
    case Mult(e1, e2) => Plus( Mult(e1, derive(e2)),
                                Mult(derive(e1), e2) )
  }
}

// affiche "((x * (1.0 + 0.0)) + (1.0 * (x + 5.0)))"
println(enChaine(derive(e)))
```

```
sealed abstract class Expr
case class X() extends Expr
case class C(v : Double = 0) extends Expr

// e1 + e2
case class Plus(e1 : Expr, e2 : Expr) extends Expr

// e1 * e2
case class Mult(e1 : Expr, e2 : Expr ) extends Expr
```

# Pattern Matching

```

def reduit(e: Expr): Expr = {
    e match {
        case Mult(C(0), _) | Mult(_, C(0)) => C(0)
            // 0*e ou e*0 se simplifie en 0
        case Plus(C(v1), C(v2)) => C(v1+v2)
            // v1 + v2
        case Mult(C(1), e) => reduit(e)
            // 1*e se simplifie e
        case Mult(e, C(1)) => reduit(e)
            // e*1 se simplifie e
        case Plus(e1, e2) => Plus(reduit(e1), reduit(e2))
        case Mult(e1, e2) => Mult(reduit(e1), reduit(e2))
        case e => e
    }
}

println(enChaine( reduit( reduit( derive(e)))))

// affiche "(x + (x + 5.0))"

```

```

sealed abstract class Expr
case class X() extends Expr
case class C(v : Double = 0) extends Expr

// e1 + e2
case class Plus(e1 : Expr, e2 : Expr) extends Expr

// e1 * e2
case class Mult(e1 : Expr, e2 : Expr ) extends Expr

```

```
val e = Mult( X(), Plus( X(), C(5) ))
```

# *Pattern Matching*

## Avancé :

La gestion des types algébriques permet de manipuler directement des structures de données

En fait, en scala, on a également accès à la structure du programme

Continuation délimitée (voir livre)

Arbre syntaxique du langage (voir macro Scala)

# Pattern Matching

## Pattern matching

déetecter une **configuration** dans les données d'entrée  
déTECTER un **patron** dans les données d'entrées  
en **extraire** des données et les transformer

Case détecteur      Personne("Jean", 45)  
                            structure    données

Semi structuré : mélange de structures et de données par exemple **XML**

# XML

## Exemple

```
scala> val helloxml = <p> hello </p>
helloxml: scala.xml.Elem = <p> hello </p>
```

```
scala> print(helloxml)
<p> hello </p>
```

# XML

## Exemple

```
scala> val img = 
img: scala.xml.Elem = 
```

```
scala> print(img)

scala>
```

scala.xml est une librairie programmée  
en scala  
Voir scaladoc spécifique (en dehors api de base)

# XML

```
C:\Windows\system32\cmd.exe - scala
scala> val mapage = <HTML>
    |   <HEAD>
    |   </HEAD>
    |
    |   <BODY>
    |       <HR/>
    |       <H1> HELLO ! </H1>
    |   </BODY>
    | </HTML>
mapage: scala.xml.Elem =
<HTML>
<HEAD>
</HEAD>

<BODY>
<HR/>
<H1> HELLO ! </H1>
</BODY>
</HTML>

scala> _
```



# XML

```
val fruits = List("pomme", "banane", "orange")
```

```
val ul = <ul> { fruits.map(i => <li> {i} </li>) } </ul>
```

```
println(ul)
```

```
<ul><li>pomme</li><li>banane</li><li>orange</li></ul>
```

# XML

## Recherche pseudo Xpath

`ul \ "li"`

## Filtre les li dans ul (retourne une collection NodeSeq)

- \ directement fils (ou fille)
- \\" tous les filles (fils)

```
val fruits = List("pomme", "banane", "orange")
```

```
val ul = <ul> { fruits.map(i => <li> {i} </li>) } </ul>
```

```
println(ul)
```

```
<ul><li>pomme</li><li>banane</li><li>orange</li></ul>
```

```
scala> val fruits = List("pomme", "banane", "orange")
fruits: List[String] = List(pomme, banane, orange)

scala>

scala> val ul = <ul>(fruits.map(i => <li>i</li>))</ul>
ul: scala.xml.Elem = <ul><li>pomme</li><li>banane</li><li>orange</li></ul>

scala>

scala> println(ul)
<ul><li>pomme</li><li>banane</li><li>orange</li></ul>

scala>

scala> ul \ "li"
res8: scala.xml.NodeSeq = NodeSeq(<li>pomme</li>, <li>banane</li>, <li>orange</li>)

scala>

scala> (ul \ "li").map(_.text)
res9: scala.collection.immutable.Seq[String] = List(pomme, banane, orange)

scala>
```

# XML

```
val people =  
<people>  
  <family>  
    <person>Mom</person>  
  </family>  
  <friends>  
    <person>Bill</person>  
    <person>Candy</person>  
  </friends>  
</people>
```

# XML

```
scala> val family = people \ "family" \ "person"
family: scala.xml.NodeSeq = NodeSeq(<person>Mom</person>)
```

```
scala> val friends = people \ "friends" \ "person"
friends: scala.xml.NodeSeq =
NodeSeq(<person>Bill</person>, <person>Candy</person>)
```

```
scala> val allPeople = people \ "_" \ "person"
allPeople: scala.xml.NodeSeq =
NodeSeq(<person>Mom</person>, <person>Bill</person>, <person>Candy</p ...
```

```
val people =
<people>
  <family>
    <person>Mom</person>
  </family>
  <friends>
    <person>Bill</person>
    <person>Candy</person>
  </friends>
</people>
```

# XML

```
scala> val allPeople = people \ "_" \ "person"
```

```
allPeople: scala.xml.NodeSeq =
```

```
NodeSeq(<person>Mom</person>, <person>Bill</person>, <person>Candy</p ...
```

```
scala> allPeople(0)
```

```
res0: scala.xml.Node = <person>Mom</person>
```

```
scala> allPeople(1)
```

```
res1: scala.xml.Node = <person>Bill</person>
```

```
val people =  
<people>  
  <family>  
    <person>Mom</person>  
  </family>  
  <friends>  
    <person>Bill</person>  
    <person>Candy</person>  
  </friends>  
</people>
```

# XML

```
scala> val allPeople = people \ "_" \ "person"
```

```
allPeople: scala.xml.NodeSeq =
```

```
NodeSeq(<person>Mom</person>, <person>Bill</person>, <person>Candy</p ...
```

```
scala> allPeople.foreach(println)
```

```
<person>Mom</person>
<person>Bill</person>
<person>Candy</person>
```

```
scala> for (person <- allPeople) println(person.text)
```

```
Mom
```

```
Bill
```

```
Candy
```

```
scala> allPeople.map(_.text)
```

```
res2: scala.collection.immutable.Seq[String] = List(Mom, Bill, Candy)
```

```
val people =
<people>
  <family>
    <person>Mom</person>
  </family>
  <friends>
    <person>Bill</person>
    <person>Candy</person>
  </friends>
<people>
```

# XML

```
val weather =  
<rss>  
  <channel>  
    <title>Yahoo! Weather - Boulder, CO</title>  
    <item>  
      <title>Conditions for Boulder, CO at 2:54 pm MST</title>  
      <forecast day="Thu" date="10 Nov 2011" low="37" high="58" text="Partly Cloudy"  
             code="29" />  
    </item>  
  </channel>  
</rss>
```

## XML

```
val weather =  
<rss>  
  <channel>  
    <title>Yahoo! Weather - Boulder, CO</title>  
    <item>  
      <title>Conditions for Boulder, CO at 2:54 pm MST</title>  
      <forecast day="Thu" date="10 Nov 2011" low="37" high="58" text="Partly Cloudy"  
               code="29" />  
    </item>  
  </channel>  
</rss>
```

```
scala> val forecast = weather \ "channel" \ "item" \ "forecast"  
  
forecast: scala.xml.NodeSeq = NodeSeq( <forecast day="Thu"  
                                              date="10 Nov 2011" low="37" high="58" text="Partly Cloudy" code="29"/> )
```

# XML

```
val weather =  
<rss>  
  <channel>  
    <title>Yahoo! Weather - Boulder, CO</title>  
    <item>  
      <title>Conditions for Boulder, CO at 2:54 pm MST</title>  
      <forecast day="Thu" date="10 Nov 2011" low="37" high="58" text="Partly Cloudy"  
               code="29" />  
    </item>  
  </channel>  
</rss>
```

```
val day = weather \ "channel" \ "item" \ "forecast" \ "@day"
```

```
val date = weather \ "channel" \ "item" \ "forecast" \ "@date"
```

# XML

```
val forecast = weather \ "channel" \ "item" \ "forecast"
```

```
scala> val day = forecast \ "@day"  
day: scala.xml.NodeSeq = Thu
```

```
scala> val day = (forecast \ "@day").text  
day: String = Thu
```

```
scala> val foo = forecast \ "@foo"  
foo: scala.xml.NodeSeq = NodeSeq()
```

```
val weather =  
<rss>  
  <channel>  
    <title>Yahoo! Weather - Boulder, CO</title>  
    <item>  
      <title>Conditions for Boulder, CO at 2:54 pm  
MST</title>  
      <forecast day="Thu" date="10 Nov 2011"  
low="37" high="58" text="Partly Cloudy"  
code="29" />  
    </item>  
  </channel>  
</rss>
```

## XML

```
val xml =  
  <order>  
    <item name="Pizza" price="12.00">  
      <pizza>  
        <crust type="thin" size="14" />  
        <topping>cheese</topping>  
        <topping>sausage</topping>  
      </pizza>  
    </item>  
    <item name="Breadsticks" price="4.00">  
      <breadsticks />  
    </item>  
    <tax type="federal">0.80</tax>  
    <tax type="state">0.80</tax>  
    <tax type="local">0.40</tax>  
  </order>
```

# XML

```
scala> val items = xml \ "item"
items: scala.xml.NodeSeq =
NodeSeq(<item name="Pizza" price="12.00">
  <pizza>
    <crust type="thin" size="14"/>
    <topping>cheese</topping>
    <topping>sausage</topping>
  </pizza>
</item>, <item name="Breadsticks" price="4.00">
  <breadsticks />
</item>)
```

```
// number of items in the order
scala> val numItems = items.length
numItems: Int = 2
```

```
val xml =
<order>
  <item name="Pizza" price="12.00">
    <pizza>
      <crust type="thin" size="14" />
      <topping>cheese</topping>
      <topping>sausage</topping>
    </pizza>
  </item>
  <item name="Breadsticks" price="4.00">
    <breadsticks />
  </item>
  <tax type="federal">0.80</tax>
  <tax type="state">0.80</tax>
  <tax type="local">0.40</tax>
</order>
```

# XML

// Liste des prix

```
scala> val prices = items.map(i => i \@ "price")
```

```
prices: scala.collection.immutable.Seq[scala.xml.NodeSeq] = List(12.00, 4.00)
```

// Sous-total

```
scala> val subtotal = items.map(i => (i \@ "price").text.toDouble).sum
```

```
subtotal: Double = 16.0
```

```
val xml =
<order>
  <item name="Pizza" price="12.00">
    <pizza>
      <crust type="thin" size="14" />
      <topping>cheese</topping>
      <topping>sausage</topping>
    </pizza>
  </item>
  <item name="Breadsticks" price="4.00">
    <breadsticks />
  </item>
  <tax type="federal">0.80</tax>
  <tax type="state">0.80</tax>
  <tax type="local">0.40</tax>
</order>
```

# XML

```
// les taxes
scala> val taxItems = xml \ "tax"
taxItems: scala.xml.NodeSeq = NodeSeq(<tax type="federal">0.80</tax>,
<tax type="state">0.80</tax>, <tax type="local">0.40</tax>)

// taxe totale
scala> val totalTax = taxItems.map(i => i.text.toDouble).sum
totalTax: Double = 2.0
```

```
val xml =
<order>
  <item name="Pizza" price="12.00">
    <pizza>
      <crust type="thin" size="14" />
      <topping>cheese</topping>
      <topping>sausage</topping>
    </pizza>
  </item>
  <item name="Breadsticks" price="4.00">
    <breadsticks />
  </item>
  <tax type="federal">0.80</tax>
  <tax type="state">0.80</tax>
  <tax type="local">0.40</tax>
</order>
```

# XML

```
val xml =  
  <order>  
    <item name="Pizza" price="12.00">  
      <pizza>  
        <crust type="thin" size="14" />  
        <topping>cheese</topping>  
        <topping>sausage</topping>  
      </pizza>  
    </item>  
    <item name="Breadsticks" price="4.00">  
      <breadsticks />  
    </item>  
    <tax type="federal">0.80</tax>  
    <tax type="state">0.80</tax>  
    <tax type="local">0.40</tax>  
  </order>  
  
val taxItems = xml \ "tax"  
  
// on récupère la liste des taxes fed  
  
val federalTax = for { item <- taxItems if (item \ "@type").text == "federal" } yield item.text
```

# *XML et fichiers*

**Comment convertir un fichier XML en objet Scala ?**

```
import scala.xml.XML
```

```
val xml = XML.loadFile("fichier.xml")
```

**Comment écrire un fichier XML sur disque ?**

1. java : printStream

2. Java 7:

```
import java.nio.file.{Paths, Files}  
import java.nio.charset.StandardCharsets  
Files.write(Paths.get("file.txt"), "file  
contents".getBytes(StandardCharsets.UTF_8))
```

# Fichier

## Scala

```
// lire un fichier texte dans une variable en une ligne  
val s = scala.io.Source.fromFile("file.txt").mkString
```

```
// écrire un fichier texte en une seule ligne  
scala.tools.nsc.io.File("file.txt").writeAll("hello world")
```

# *Programmation multi paradigme*

Impératif

Objet

Récursif

Fonctionnel (map / reduce etc..)

Déclaratif et par patterns (Match)

Connexionniste (Perceptron)

Choisir le bon paradigme

plus simple et rapide à écrire

possible à écrire

passage à l'échelle

Scala = langage multi paradigme