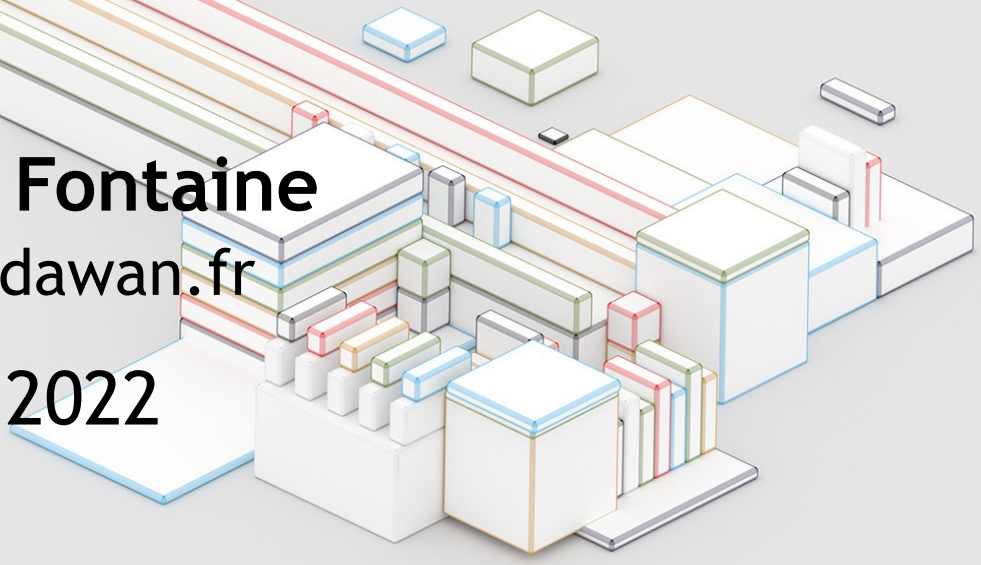
Dawan
Training is our way

Java SE Initiation :
Programmer en objet



Christophe Fontaine
cfontaine@dawan.fr

30/05/2022

Objectifs

- Découvrir la programmation orientée objet au travers du langage Java

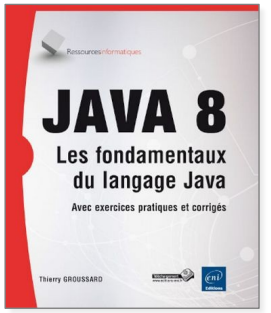
- Réaliser une première application complète

Pré-requis: connaissances informatiques de base, connaissances algorithmiques élémentaires

Durée: 5 jours

Bibliographie

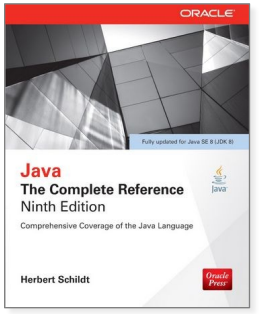
- **Java 8 Les fondamentaux du langage Java**
  Thierry Groussard
  Éditions ENI - Juillet 2014



- **Java Tête la première** (couvre Java 5.0)
  Kathy Sierra, Bert Bates
  Editions O'REILLY - Novembre 2006

- **Java The Complete Reference**
  Herbert Schildt
  Edition Oracle Press - 9th edition - Juin 2015

- **Java Platform Standard Edition 8 Documentation**
  https://docs.oracle.com/javase/8/docs/



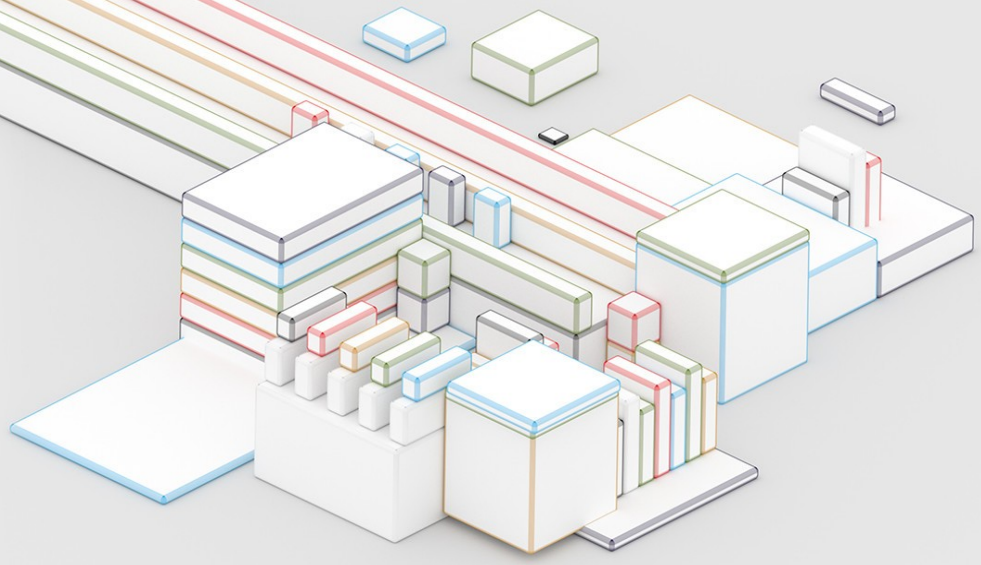
- **Développons en Java**
  https://www.jmdoudoux.fr/java/dej/index.htm

Plan

- Présentation
- Syntaxe du langage
- Programmation orientée objet
- Classes essentielles
- Exceptions
- Collections
- Entrées/Sorties
- Les fonctions de base de Git dans Eclipse

Dawan
Training is our way

Présentation



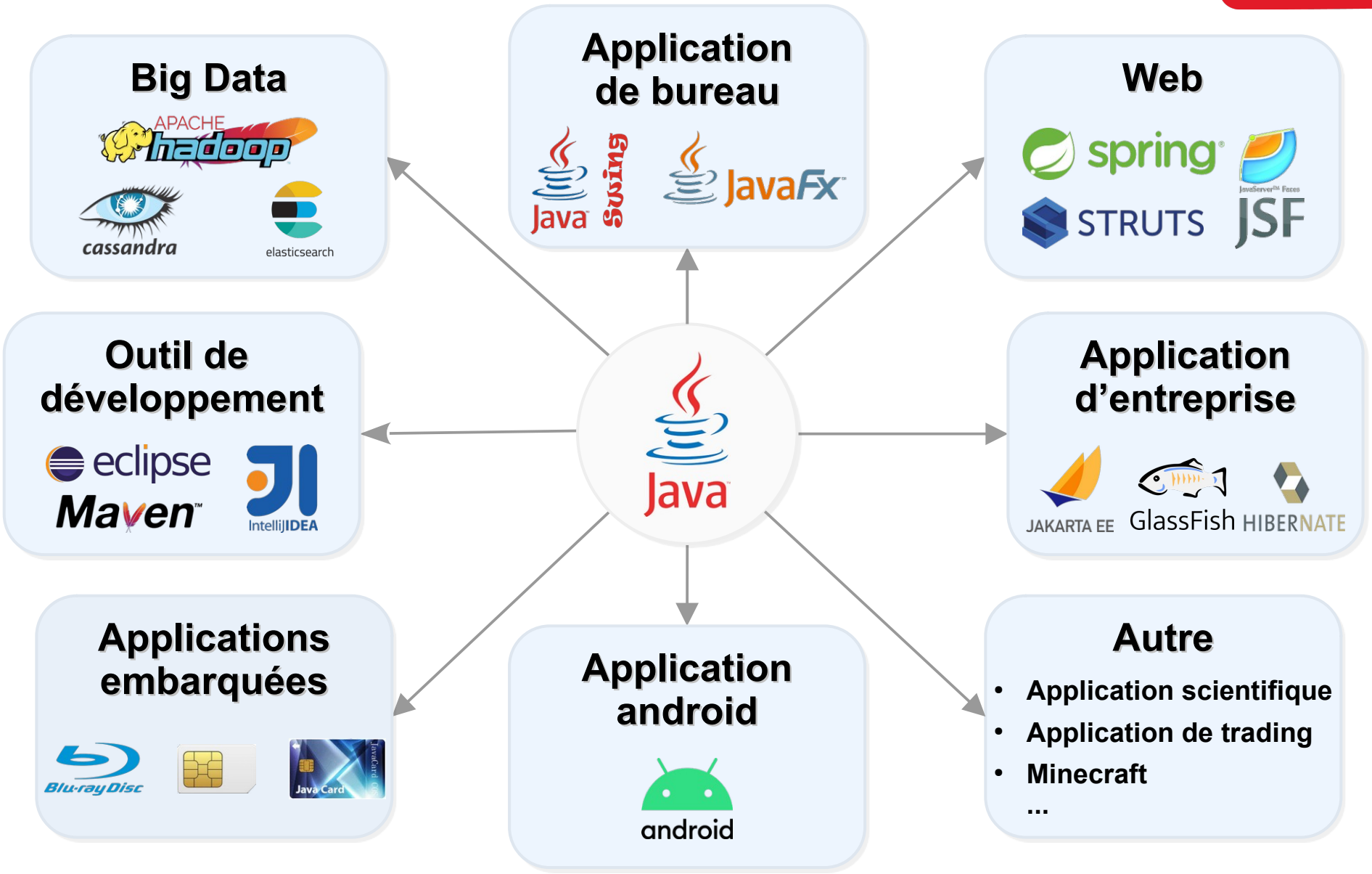
Historique

1991	Lancement du **Green Project**
1992	Première version : langage **OAK** (star seven)
1995	Lancement public de Java
1997	**Java 1.1** (Java beans, JDBC, Jar …)
1998	**Java 1.2** Plateforme Java (J2SE, J2EE et J2ME) Création du **JCP** (Java Community Process)
2000	**Java 1.3** (JVM Hotspot, Collections …)
2002	**Java 1.4** (NIO, API de log …)
2004	**Java SE 5** (généricité, types énumérés …)
2006	**Java SE 6** Java devient open source
2010	Sun Microsystem est racheté par Oracle
2011	**Java SE 7**

Historique

2014		**Java SE 8**	**LTS** support jusqu'en 2030		
		↪ (expression lambda, stream, api java time ...)			
2017		**Java SE 9**	(modularité ...)		
2018	03	**Java SE 10**	(inférence de type de variable locale ...)		
	09	**Java SE 11**	**LTS** support jusqu'en 2026		
		↪ changement de licence: on ne peut plus utiliser Oracle JDK gratuitement en production, alternative → OpenJDK			
2019	03	**Java SE 12**		09	**Java SE 13**
2020	03	**Java SE 14**	(text blocks ...)	09	**Java SE 15**
2021	03	**Java SE 16**	(records ...)		
	09	**Java SE 17**	**LTS** support jusqu'en 2024		
		↪ retour à une licence gratuite en production pour Oracle JDK			
2022	03	**Java SE 18**	(serveur web simple ...)	09	**Java SE 19**
2023	03	**Java SE 20**			

Domaine d'utilisation de Java



Caractéristiques

- Simple, orienté objet et familier
- Interprété
- Portable et indépendant des plateformes
- Robuste et sûr
- Distribué
- Dynamique et multi-thread

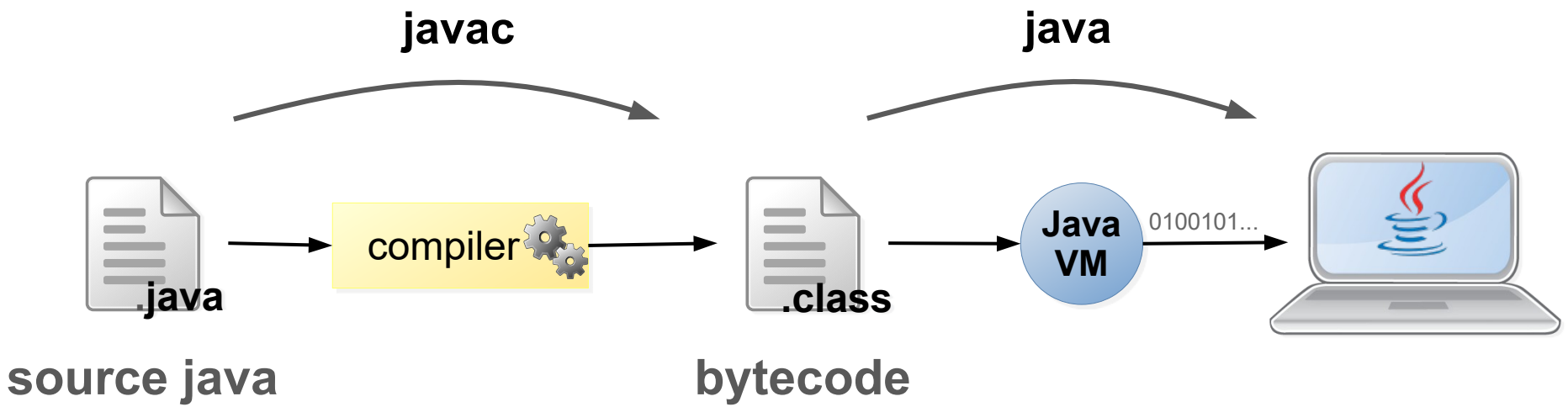
The Java Language Environment: A White Paper
1996 James Gosling, Henry McGilton

Développement java

- Écrire un programme Java ne nécessite pas d'outils sophistiqués :
  - un éditeur de texte : notepad suffit
  - un JDK (Java Development Kit)

Oracle JDK https://www.oracle.com/java/technologies/downloads/

OpenJDK https://openjdk.java.net/ (source)
https://adoptium.net/ (binaire pré-construit)



Kit de développement Java (JDK)

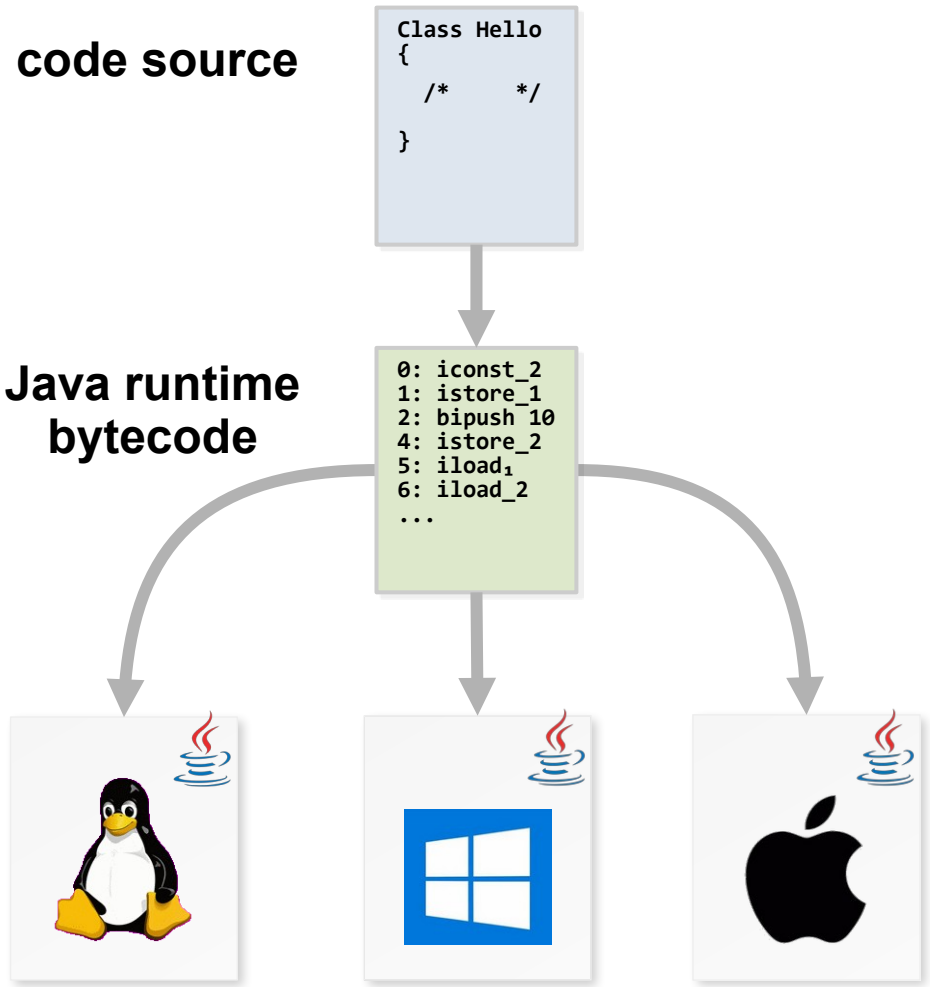
Le kit de développement comprend de nombreux outils :

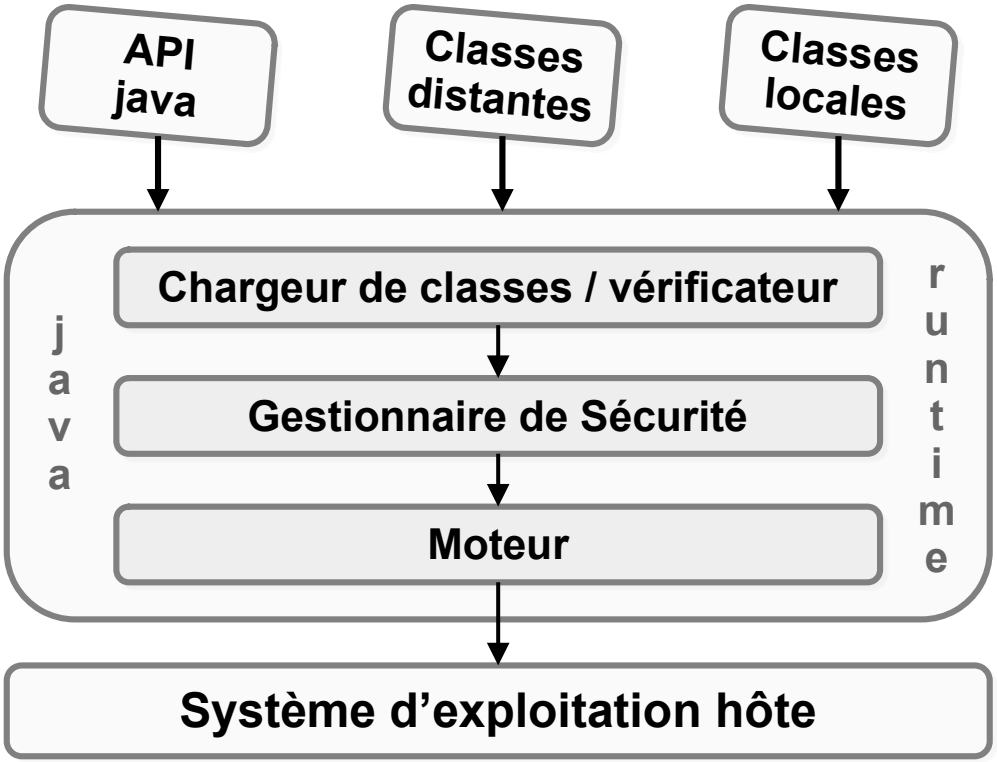
- **javac** Compilateur
- **java** Interpréteur d'application
  javaw
- **javadoc** Générateur de documentation
- **jdb** Débogueur
- **javap** Désassembleur
- **visualvm** Outils de monitoring
  jconsole

Packages Java (API)

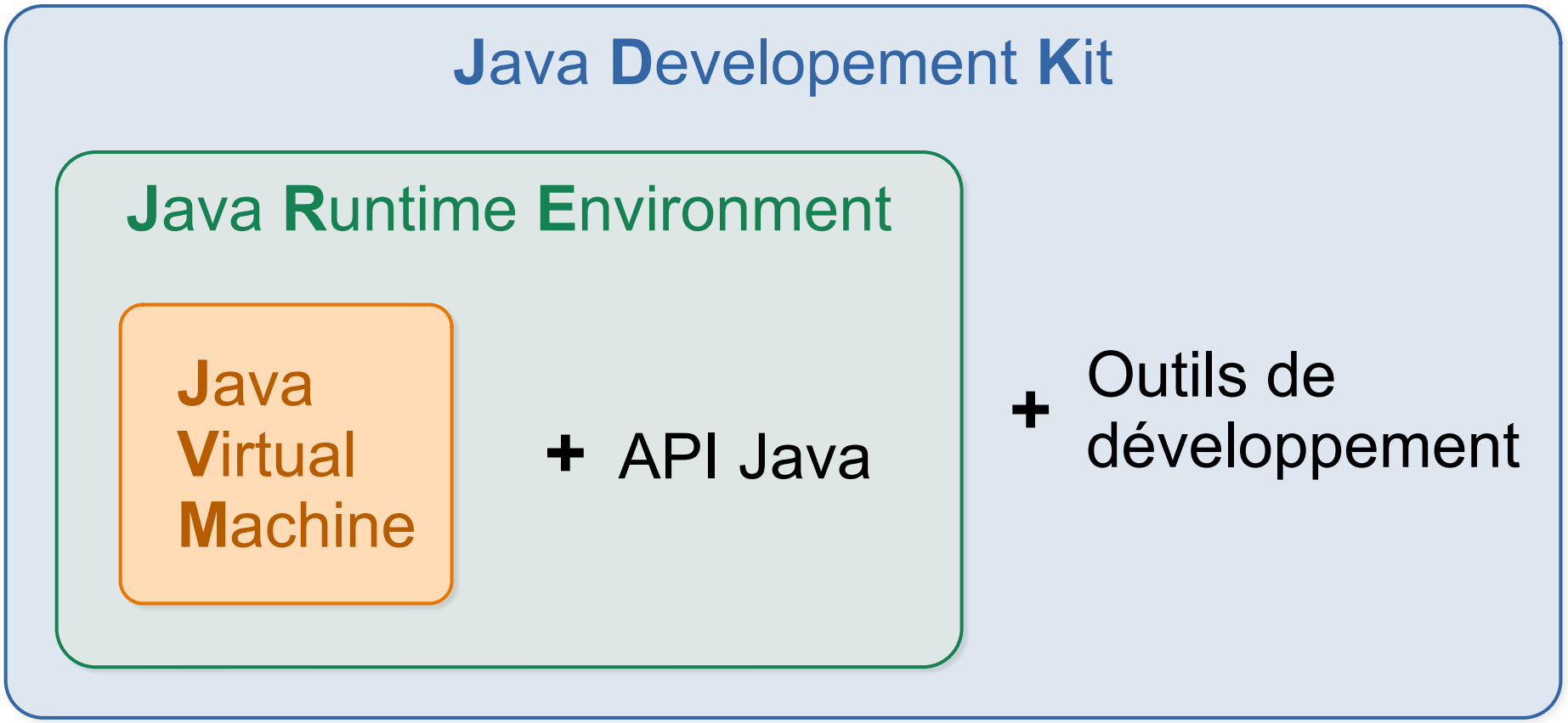
java.lang	Classes fondamentales
java.util	Collections framework
java.io	Entrées/sorties, gestion des flux, sérialization…
java.awt	Abstract Windowing Toolkit (IHM)
javax.swing	Composants Graphiques
java.net	Applications Réseau
java.sql	Accès aux données stockées dans des Bdd
java.time	Dates, heures, instants et durées
java.security	Framework de Sécurité

Java Virtual Machine (JVM)





JDK, JRE, JVM



Première application Java

- Installation du JDK
- Paramétrages des variables d'environnement
  ↳ dans variables système, on ajoute :

```
JAVA_HOME = C:\Program Files\Java\jdk1.8.0_351
PATH = %JAVA_HOME %\bin
```

- Écriture et exécution d'un premier programme en Java

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

Environnements de développement

En plus des outils de base de développement du JDK, il existe des environnements de développement intégrés (IDE)

- **Eclipse** (eclipse.org)

- **IntelliJ IDEA** (jetbrains.com/idea/)

  existe en 2 versions :
  - community edition (open-souce et gratuit)
  - utimate (payante)

- **Netbeans** (netbeans.org)

Configuration d'Eclipse

- À partir de **eclipse 4.18**, le JRE qui va exécuter eclipse est intégrée sous forme de plugin (openjdk 17)

  Il n'est plus nécessaire de le configurer dans le fichier **eclipse.ini** avec l'option -vm

```
-vm
C:\Program Files\Java\jdk1.8.0_351
```

- Dans windows → préférence
  - filtre sur **jre**

    Installed JREs → Add

    ↳ choisir : Standard VM

    JRE home : C:\Program Files\Java\jdk1.8.0_351\jre

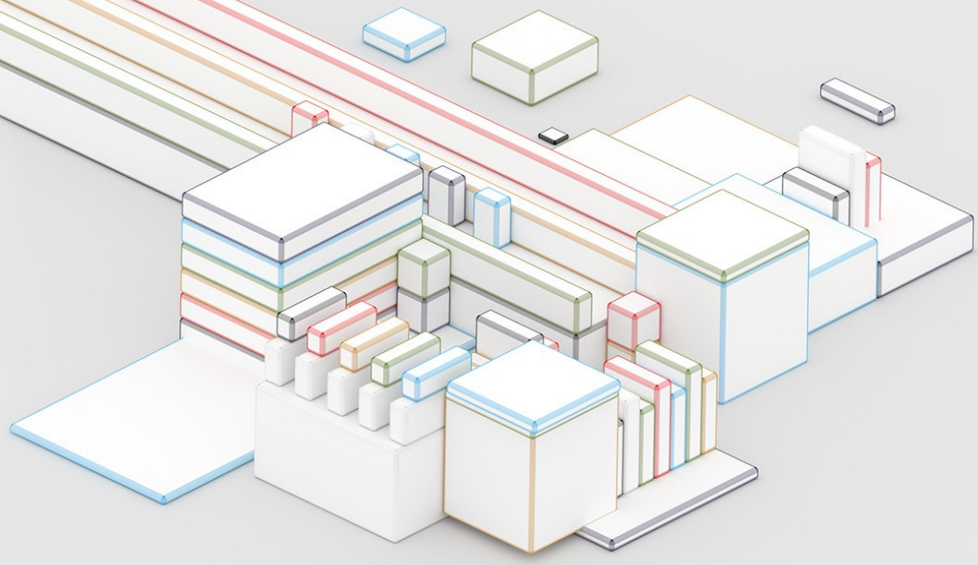
    JRE Name : jdk1.8.0_351

Configuration d'Eclipse

- filtre sur **compiler**
  JDK Compilance → Compiler compliance level → 1.8
- filtre sur **text editors**
  cocher : Insert spaces for tabs
  cocher : remove multiple spaces and backpace/delete
- filtre sur **spelling**
  décocher : enable spelling
- filtre sur **formatter**
  java → code style → Formatter
  new → profil name : Eclipse
  indentation : tab policy choisir space only

Dawan
Training is our way

Syntaxe du langage



Base du langage

- Les instructions se terminent par un **;**
- Différences entre **minuscule** et **MAJUSCULE**
- Espaces / Tabulations / CR / LF sans conséquences
- Bloc de code : suite d'instructions entre **{ }**
- Commentaire

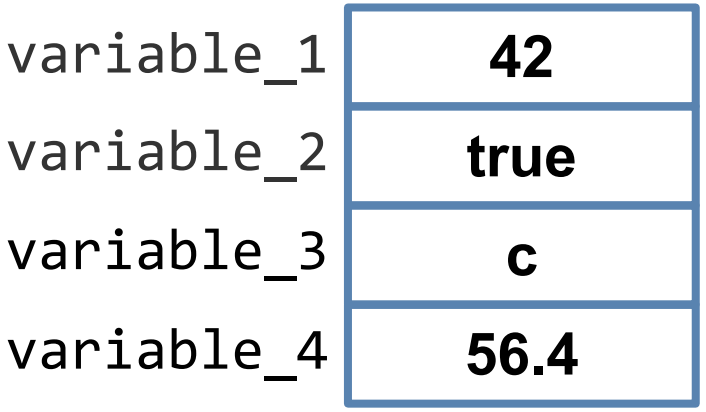
```
// Commentaire fin de ligne

/* Commentaire
   sur plusieurs lignes
*/
```

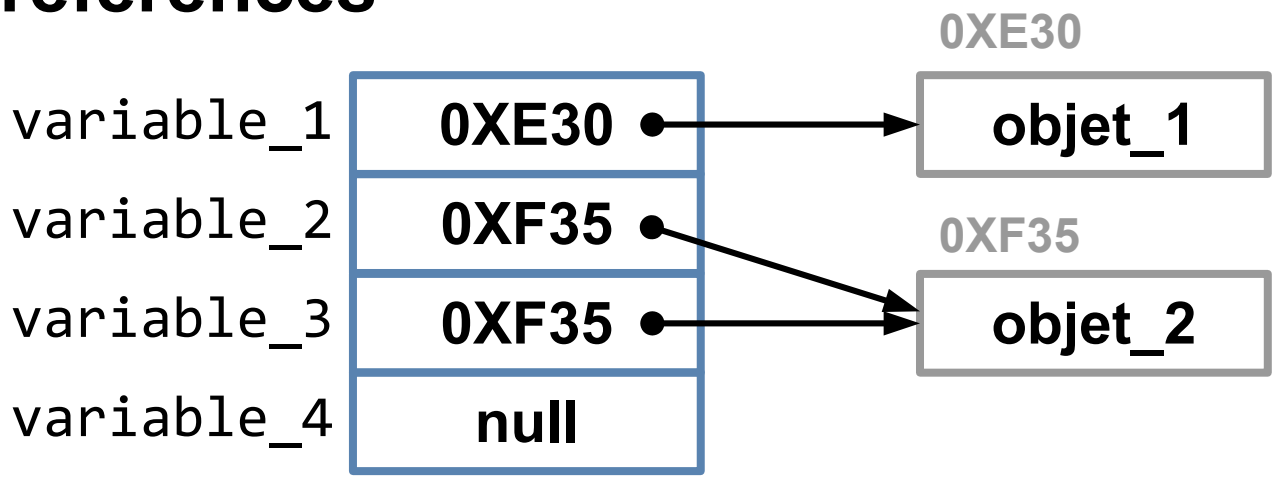
```
/**
 * Commentaire javadoc
 *
 * @author James Gosling
 * @Version 1.0
 */
```

Types de données

- **Types primitifs**



- **Types références**



Types primitifs

Type	Description	Plage de valeurs	Littéral
boolean	booléen	false ou true	true
char	caractère unicode sur **16** bits	'\u0000' à '\uFFFF'	**'a'**
byte	entier signé sur **8** bits	-128 à 127	42
short	entier signé sur **16** bits	-2^{15} à $2^{15}-1$	123
int	entier signé sur **32** bits	-2^{31} à $2^{31}-1$	420
long	entier signé sur **64** bits	-2^{63} à $2^{63}-1$	420**L**
float	réel signé sur **32** bits	$\{-3,4028234^{38}..3,4028234^{38}\}$ $\{-1,40239846^{-45}..1,40239846^{-45}\}$	4.23**f**
double	réel signé sur **64** bits	$\{1,79769134^{308}..1,79769134^{308}\}$ $\{-4,94065645^{-324}..4,94065645^{-324}\}$	1230.0 1.23e3

Variables

Zone mémoire pour stocker une information

- **Déclaration**

```
type nomVariable;
```

```
double valeur;
int i, j;
```

- **Initialisation**

```
nomVariable = valeur;
```

```
valeur = 134.8;
i = 42;
```

- **Initialisation pendant la déclaration**

```
char c = 'a';
double hauteur = 1.25, largeur = 1.26;
```

Règles de nommages

- Le nom doit commencer par : **une lettre**, _ ou **$**
- Les nombres sont autorisés **sauf en tête**
- Ne doit pas être **un mot réservé**

Correct	→	identifier	conv2Int	_test	$_data2
Faux	→	**3**dPoint	**public**	*$coffe	**while**

- Par convention les variables utilisent le **c**amel**C**ase

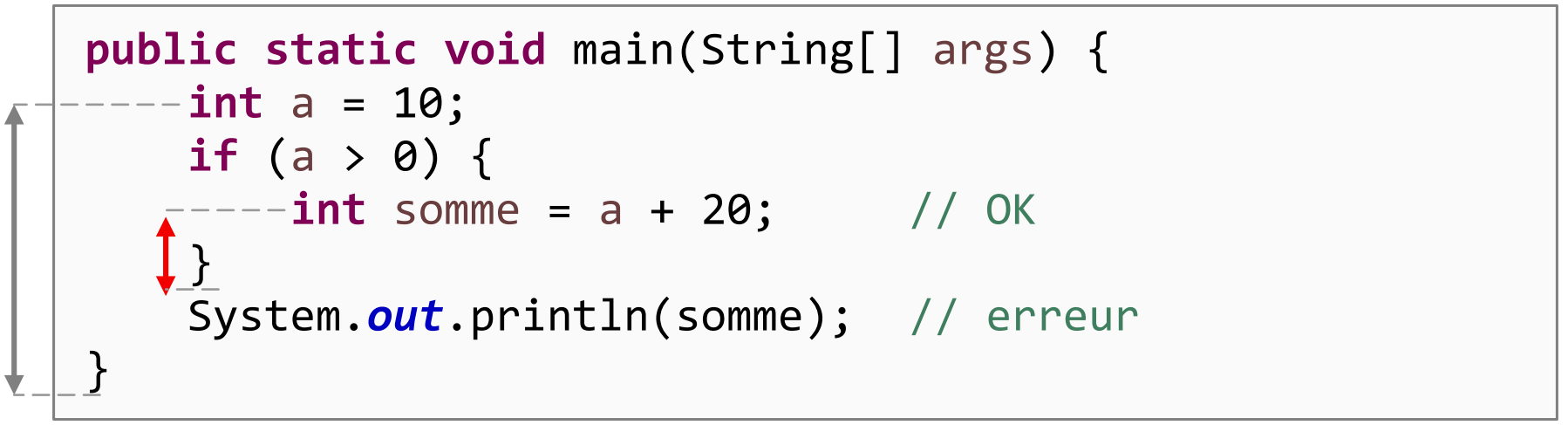
```
int nombreDeVisiteur;
int anneeNaissance;
boolean isOpen;
```

Porté d'une variable locale

- Sa portée se limite au bloc où elle est définie. Leur espace mémoire est libéré lorsque le bloc se termine (pile LIFO)

  ↳ **Chaque bloc de code a sa propre portée**

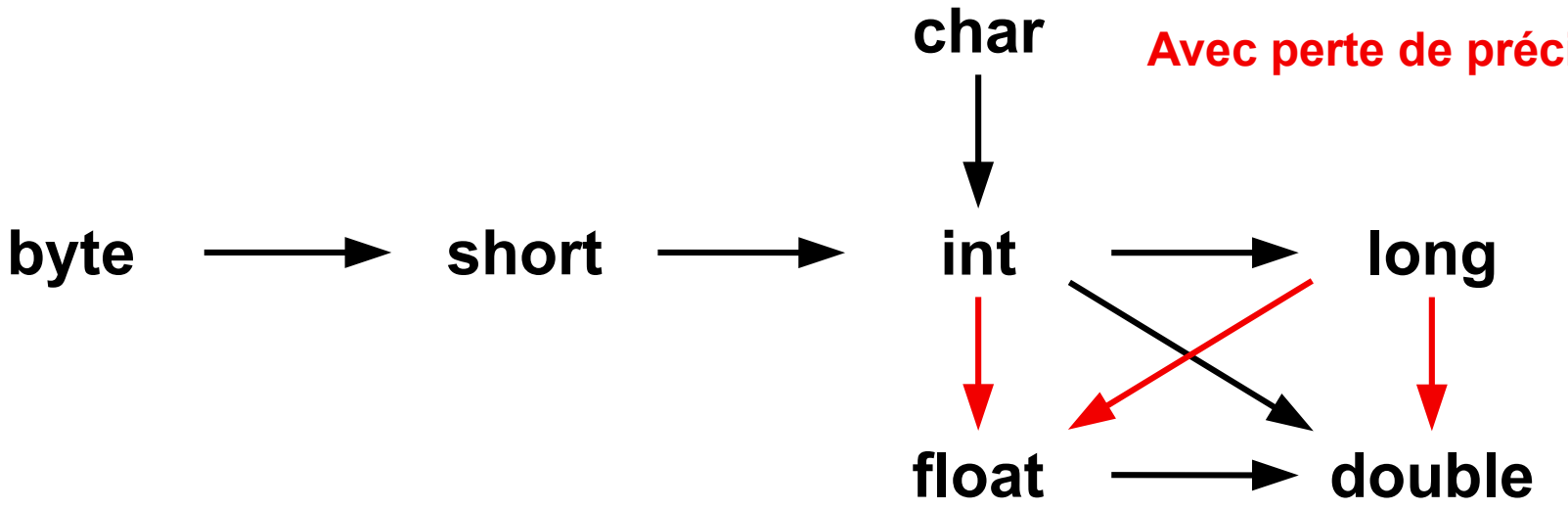
- Quand des blocs contiennent d'autre blocs
  Les blocs contenus peuvent faire référence aux variables du bloc conteneur mais pas l'inverse



```
public static void main(String[] args) {
    int a = 10;
    if (a > 0) {
        int somme = a + 20;      // OK
    }
    System.out.println(somme);   // erreur
}
```

Transtypage implicite (Automatique)

- Type inférieur vers un type supérieur
- Entier vers un réel




```
byte b = 42;
int i = b;
double d = i;
```

Transtypage explicite (cast)

- Type supérieur vers un type inférieur
- Réel vers un entier

```
type variable = (type) variableToCast;
```

```
int i = 123;
short s = (short)i;

double d = 44.95;
int j = (int)d;

// Attention au dépassement de capacité
int k = 130;
byte b = (byte)k;  // b vaut -126
```

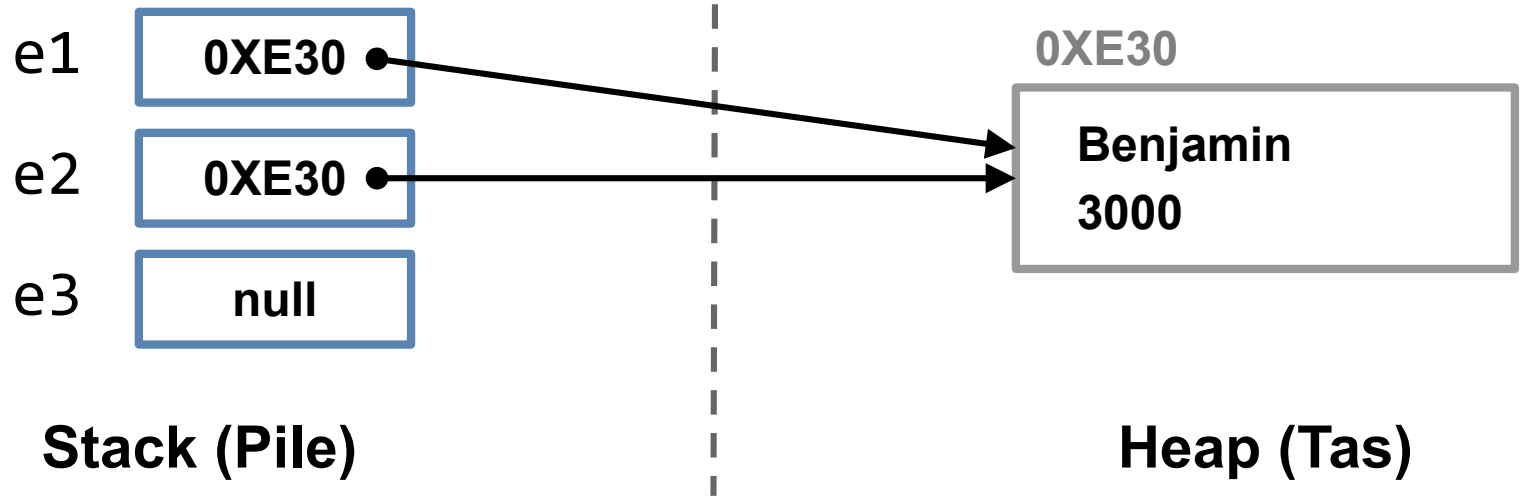
Types références → objets

Il existe des types complexes :

- String (chaîne de caractère)
- Integer (encapsulation d'un entier)
- Tableau

…

```
Employe e1 = new Employe("Benjamin", 3000);
Employe e2 = e1;
Employe e3 = null;
```



Wrappers (types enveloppes)

Les **wrappers** sont des objets identifiants des variables primitives

Type primitif	Wrapper
boolean	Boolean
byte	Byte
short	Short
int	**Integer**
long	Long
float	Float
double	Double
char	**Character**

```
char c = 'a';
Character ch =new Character(c);

int i = 3;
Integer iW = new Integer(i);
int k = Integer.parseInt("10");

Double dW = Double.valueOf("10.5");
double d = iW.doubleValue();
i = iW.intValue();

String s = Integer.toBinaryString(i);
String sh = Integer.toHexString(i);
String sd = Double.toString(d);
```

Opérateurs

- Arithmétiques : **-** **+** ***** **/** **%** (modulo)
- Incrémentation : **++**
  Décrémentation : **--**
  } pré : ++var et post : var++
- Affectations : **=** **+=** **-=** ***=** **/=** **%=** **&=**
  |= **^=** **~=** **<<=** **>>=** **>>>=**
- Comparaisons : **==** **!=** **<** **>** **<=** **>=** **instanceof**
- Logiques : **!** (non) **&&** (et) **||** (ou)
- Binaires (bit à bit) : **~** (complément) **&** (et)
  ^ (ou exclusif) **|** (ou)

(décalage) : **<<** **>>** (signe) **>>>** (0)

Promotion numérique

Elle rend compatible le type des opérandes avant qu'une opération arithmétique ne soit effectuée

1. Le type le **+ petit** est promu **vers le + grand** type des deux
2. Si une variable est **entière** et une autre en **virgule flottante**, la valeur **entière est promue en virgule flottante**
3. **byte**, **short**, **char** sont promus en **int** à chaque fois qu'ils sont utilisés avec un opérateur
4. Après une promotion le résultat aura le même type

Condition: if

```
if (condition) {
   // bloc d'instructions 1 (condition vrai)
} else {
   // bloc d'instructions 2 (condition fausse)
}
```

- **else** n'est pas obligatoire
- On peut enchaîner un **if** après un **else**

```
int i = 25;
if (i == 22) {
   // traitement 1
} else if (i == 25) {
   // traitement 2
} else {
   // traitement par défaut
}
```

Condition: switch

```
switch (variable) {
   case valeur1:
      // si variable a pour valeur valeur1
   break;
   case valeur2:
   case valeur3:
      // si variable a pour valeur valeur2 ou valeur3
   break;
   default:
      // si aucune valeur des cases ne correspond
}
```

- La variable peut être de type: **byte**, Byte, **short**, Short, **char**, Character, **int**, Integer, String et **enum**
- La valeur de **case** doit avoir une valeur constante
- **default** n'est pas obligatoire

Condition: switch

```
int jours = 7;
switch (jours) {
    case 1:
        System.out.println("Lundi");
        break;
    case 6:
    case 7:
        System.out.println("week end !");
        break;
    default:
        System.out.println("autre jour");
}
```

Condition: opérateur ternaire


```
condition ? condition_vraie : condition_fausse ;
```

Utilisation : affectation conditionnelle

```
int i = 50;
String resultat = (i < 25) ? "< à 25" : "≥ à 25";
```

équivaut à

```
int i = 25;
String resultat;
if (i < 25) {
    resultat = "< à 25";
} else {
    resultat = "≥ à 25";
}
```

Boucle: while

```
while (condition) {
   // instructions à exécuter
}
```

Tant que la condition est vérifiée, le bloc d'instructions est exécuté

```
int i = 0;
int somme = 0;
while (i <= 10) {
    somme = somme + i;
    i++;
}
System.out.println("Somme= " + somme);
```

Boucle: do while

```
do {
   // instructions à exécuter
} while (condition);
```

Identique à **while**, sauf que le test est réalisé après l'exécution du bloc

```
int i = 0;
int somme = 0;
do {
    somme = somme + i;
    i++;
} while (i <= 10);
System.out.println("Somme= " + somme);
```

Boucle: for

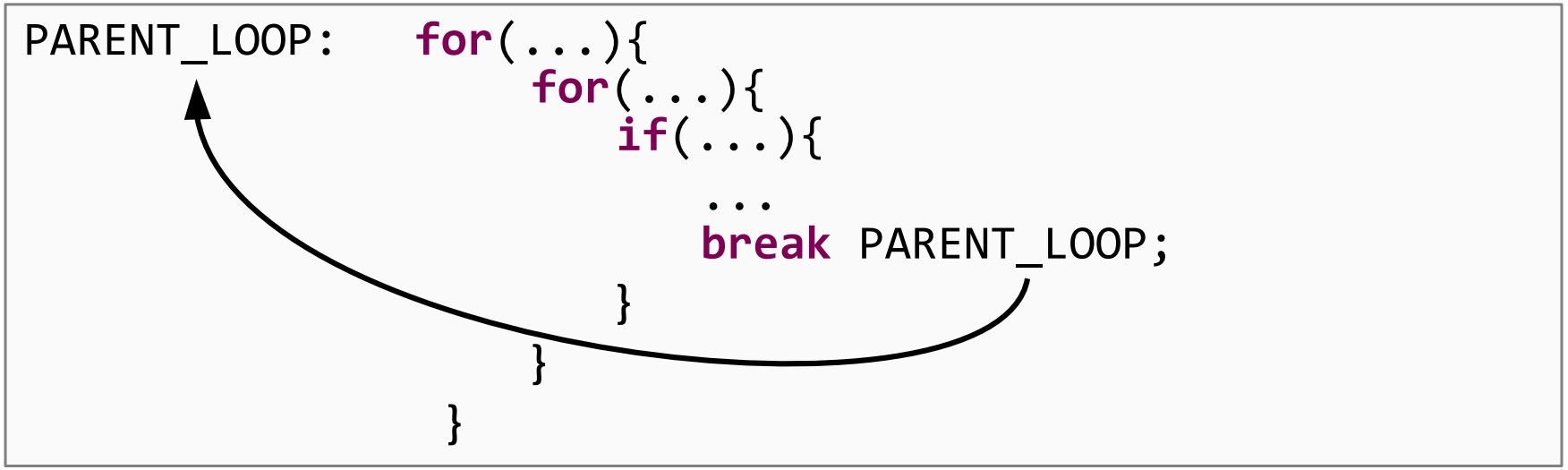
```
for(initialisation ; condition ; opération){
   // instructions à exécuter
}
```

1. **initialisation** est exécutée
2. si la **condition** est fausse ⟶ on sort de la boucle
3. le bloc d'instruction est exécuté
4. **opération** est exécutée

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

Instructions de saut

- **break**
  termine le traitement de la **boucle** ou du **switch** courant
- **continue**
  passe à l'itération suivante dans un traitement de **boucle**
- **LABEL** :
  `break` et **`continue`** peuvent être suivis d'un nom d'étiquette

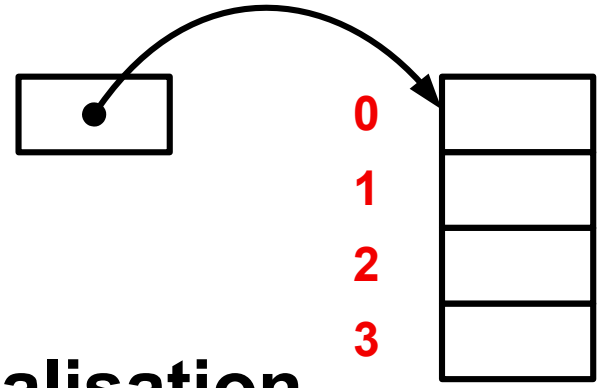


Tableaux

- **Déclaration d'un tableau**

```
type[] nom_tableau = new type[taille_tableau];
```

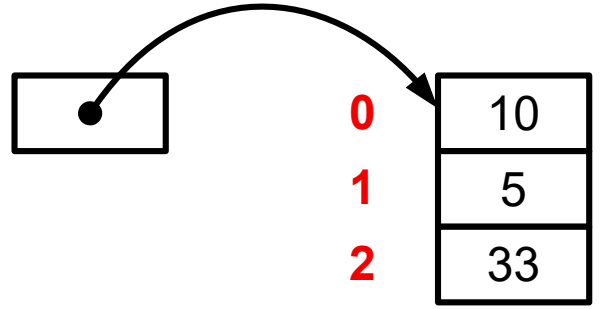
```
int[] tab = new int[4];
```



- **Déclaration d'un tableau avec initialisation**

```
type [] nom_tableau= { valeur1, valeur2, … };
```

```
int[] tab2 = { 10, 5, 33 };
```



Tableaux

- **Accès à un élément d'un tableau**

  `nom_tableau[indice]`

  L'indice d'un tableau commence à 0

```
int[] tab = { 10, 30, 40 };
System.out.println(tab[0]); // affiche 10
```

- **Taille d'un tableau**

  `nom_tableau.length`

```
int [] tab= new int[20];
int n = tab.length;    // n a pour valeur 20
```

Tableaux: itération complète (foreach)


```
for (type variable : tableau) {
   // instructions à exécuter
}
```

```
int[] tab = { 2, 4, 6 };      // → affiche : 2
for (int val : tab) {         // 4
    System.out.println(val);  // 6
}
```

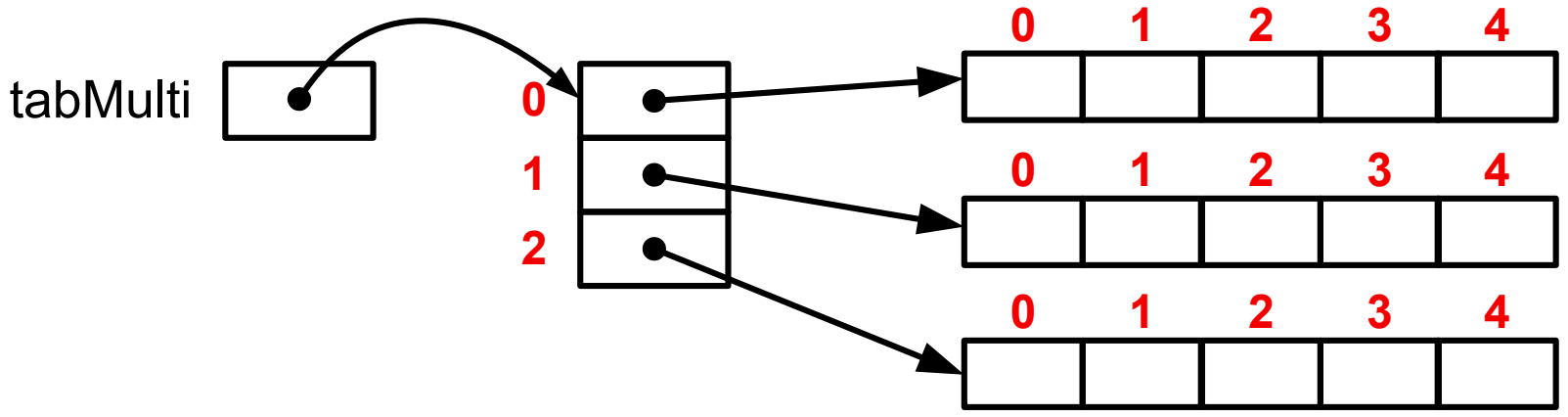
équivaut à

```
int[] tab = { 2, 4, 6 };                    // → affiche : 2
for (int i = 0; i < tab.length; i++) { // 4
    System.out.println(tab[i]);             // 6
}
```

Tableaux multidimensionnel

```
type[][] nom_tableau = new type[nb_ligne][nb_colonne];
type[][] nom_tableau = { { valeur01, valeur02, …},
                         { valeur11, valeur12, …}, … } ;
```

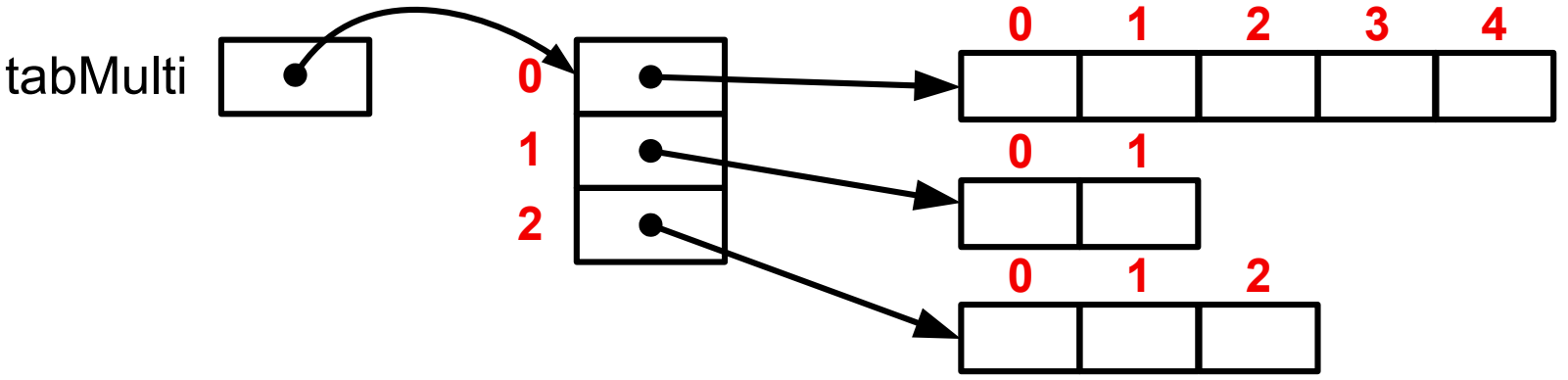
```
int[][] tabMulti = new int[3][5];
int[][] tabMulti2 = {  { 10, 3, 4, 1, 0 },
                       { 12, -9, 6, 3, 9 },
                       { 1, -1, 12, 8 } };
System.out.println(tabMulti2[0][2]);      // affiche 4
```



Tableaux en escalier

```
type[][] nom_tableau = new type[nb_ligne][];
nom_tableau[0] = new type[nb_colonne1];
nom_tableau[nb_ligne-1]  = new type[nb_colonneN];
```

```
int[][] tabMulti = new int[3][];
tabMulti[0] = new int[5];
tabMulti[1] = new int[2];
tabMulti[2] = new int[3];
int[][] tabMulti2 = {  {10, 3, 4, 1, 5 }, { 12, -9 },
                       { 1, -1, 12 } };
```



Méthodes

- **Une Méthode permet de**
  - Diviser le code en morceaux (réutilisabilité, clarté)
  - Factoriser le code
- **Déclaration d'une méthode**

```
typeDeRetour nomDeLaMethode(type arguments) {
   // instructions à exécuter
}
```

```
int somme(int a, int b) {
    return a + b;
}
```

- **Appel d'une méthode**

```
nomMethode(parametres);
```

```
int num = 20;
int res = somme(num,22);
```

Méthodes

- **Nom de la méthode**

  C'est les même règles de nommage que pour les variables

- **Type de retour**

  Il est obligatoire. S'il n'y en a pas → **void**

- **Corps de la méthode**
  - au minimum **{ }**
  - doit contenir au moins une instruction **return**
    pour void: **return;** ou il **peut être omis**

- **Arguments**
  - Ils sont séparés par **,**
  - Ils sont **passés par valeur**

Nombre d'arguments variable

```
void methode(type... argument){  }
```

- Un seul paramètre variable par méthode
- Il doit être en dernière position dans la liste d'argument

```
void walk1(int... nums) {    }
void walk2(int n, int... nums) {    }
```

- Dans le corps de la méthode, un paramètre variable est considéré comme un tableau

```
void printAll(String... strs) {
    for (String s : strs) {
        System.out.println(s);
    }
}
printAll();
printAll("foo", "bar");
printAll("foo", "bar", "baz", "toto");
```

Surcharge de méthode (overloading)

- Plusieurs méthodes peuvent avoir le même nom et des arguments différents en nombre ou/et de type
- Le type de retour n'est pas pris en compte

```
void maMethode(int param1) {
}
void maMethode(int param1, String param2) {
}
void maMethode(int param1, int param2) {
}
void maMethode(int otherParam) {
    // Faux : Le type du paramètre est le même que
              la première méthode
}
```

Surcharge de méthode (overloading)

- Si le compilateur ne trouve pas de correspondance exacte, il effectue des conversions automatiques pour trouver la méthode à appeler

```
public static void main(String[] args) {
    maMethode(1.2,3);   // correspondance exacte
    maMethode(7,5);     // conversion du 7 en double
}
public static void maMethode(double arg1, int arg2) { }
```

- Ordre d'appel des méthodes surchargées

```
meth(1,2);
```

Correspondance exacte des types	`int meth(int i,int j) {}`
Type primitif plus grand	`int meth(long i,long j) {}`
Type autoboxed	`int meth(Integer i,Integer j) {}`
Arguments variables	`int meth(int... nums) {}`

+ ↓ -

Récursivité

- **Capacité d'une méthode à s'appeler elle-même**

```
int factorial(int n) { // factoriel= 1* 2* … n
    if (n <= 1) { // condition de sortie
        return 1;
    } else {
        return factorial(n - 1) * n;
    }
}
int f = factorial(3);     // f vaut 6
```

Appels successifs

```
factorial(3)= factorial(2) * 3
    factorial(2)= factorial(1) * 2
        factorial(1)
```

Remontée des résultats

```
factorial(3) = (2) * 3 = 6
    factorial(2) = (1) * 2 = 2
        factorial(1) = 1
```

Méthode main

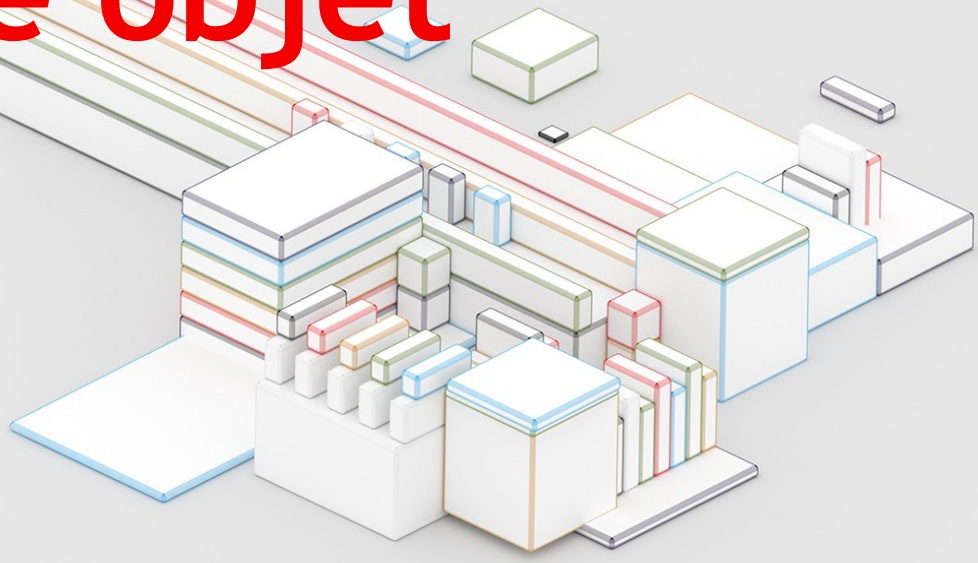
```
public static void main(String args[]) {
   // instructions à exécuter
}
```

- point d'entrée du programme
- doit être statique
- peut recevoir des paramètres depuis une ligne de commande

```
C:\Formations\java> java Application I am 35 "Hello World"
   args[0] → I
   args[1] → am
   args[2] → 35
   args[3] → Hello World
```

Dawan
Training is our way

Programmation orientée objet



Définition

L'orienté-objet = approche de résolution algorithmique de problèmes permettant de produire des programmes modulaires de qualité

Objectifs :

- Développer une partie d'un programme sans qu'il soit nécessaire de connaître les détails internes aux autres parties
- Apporter des modifications locales à un module, sans que cela affecte le reste du programme
- Réutiliser des fragments de code développés dans un cadre différent

Qu'est ce qu'un objet ?

Objet = élément identifiable du monde réel

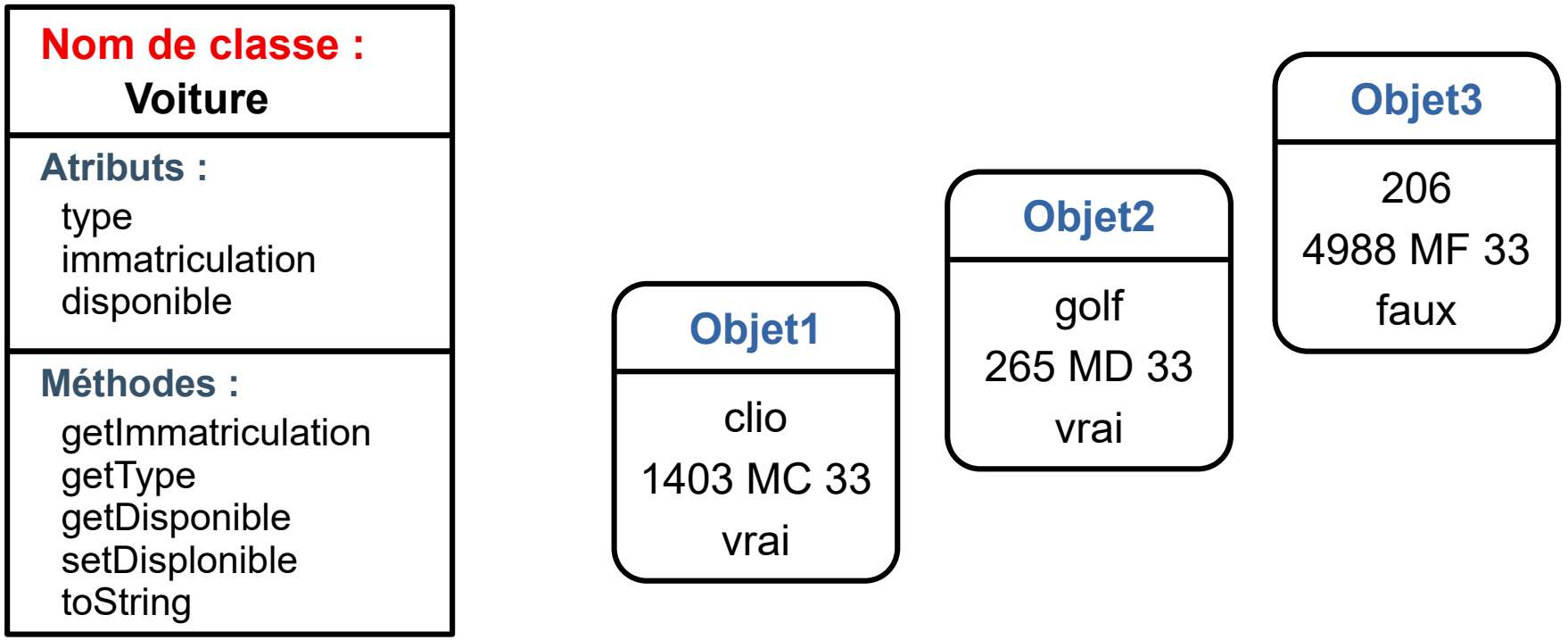
- concret (voiture, stylo,…)
- abstrait (entreprise, temps,…)

Un objet est caractérisé par :

- son **identité**
- son **état** → les données de l'objet
- son **comportement** → ce qu'il sait faire

Qu'est-ce qu'une Classe ?

- Une classe est un type de structure ayant :
  - des attributs
  - des méthodes
- On peut construire plusieurs **instances** d'une classe



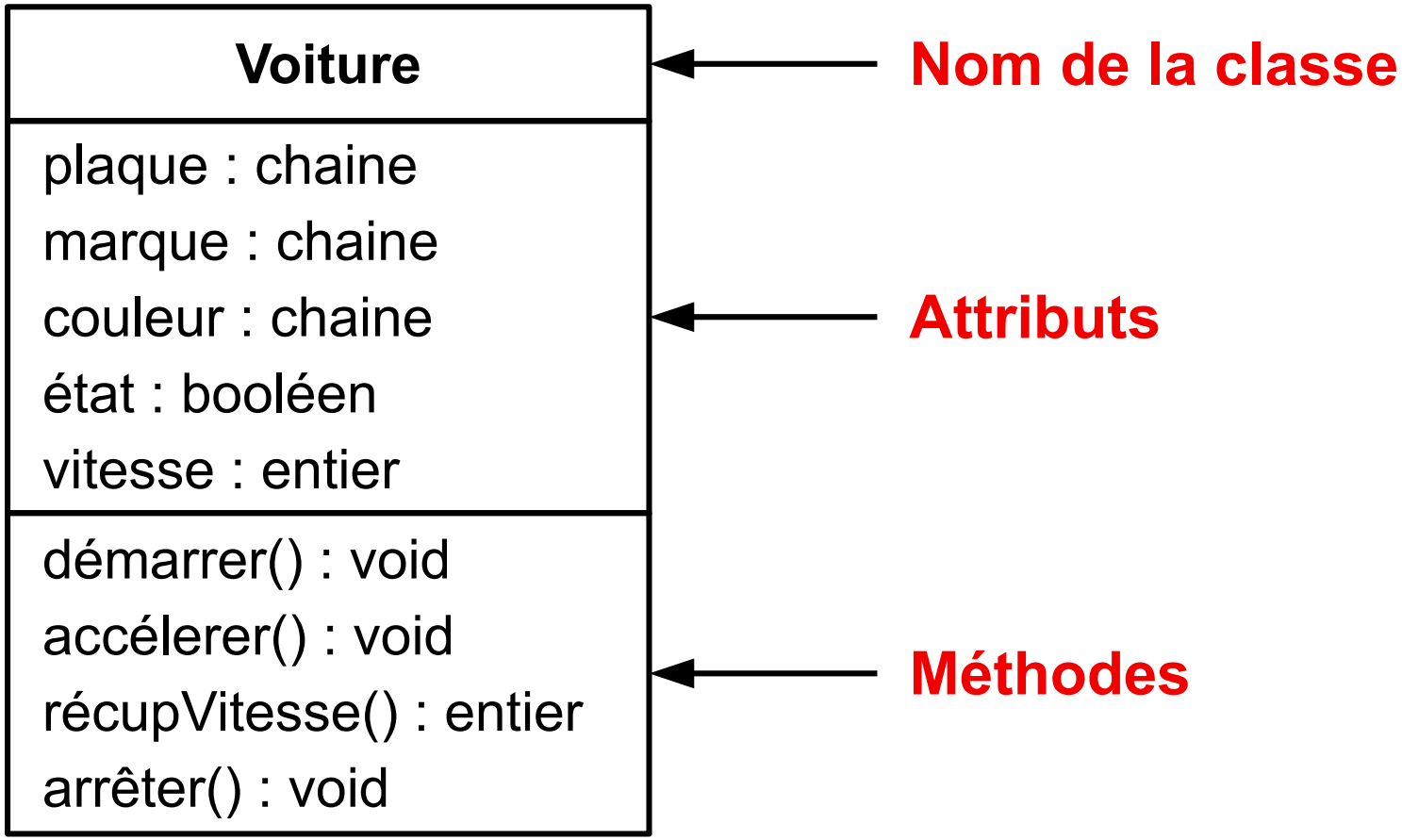
Unified Modeling Language

UML = Language pour la modélisation des classes, des objets, des intéractions etc...

UML 2.5 comporte ainsi 14 types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information :

- Diagramme fonctionnel
- Diagrammes structurels (statiques)
- Diagrammes comportementaux (dynamiques)

Représentation UML d'une classe



Les attributs ou les méthodes peuvent être précédés par un opérateur (**+** → public, **#** → protected, **-** → private) pour indiquer le niveau de visibilité

Déclaration d'une classe

```
public class Voiture
{
   // Attributs

   // Méthodes

}
```

Voiture.java

- Le nom d'une classe commence toujours par une majuscule
- Les règles de nommage sont les mêmes que pour les variables
- Une seule classe public par fichier, le nom du fichier est celui de cette classe

Variables d'instances

- Les variables d'instance définissent **l'état de l'objet**
- Elles sont également appelées **attributs**
- La valeur d'un attribut est **propre à chaque** instance

```
public class Voiture {
    String marque;
    String plaque;
    String couleur;
    // …
}
```

- **Accès à un attribut**

`instance.attribut`

```
clio.couleur
```

Méthodes d'instances

- Méthodes qui définissent **un comportement** d'une instance
- Elles sont déclarées dans la classe
- Elles peuvent être surchargées

```
public class MaClasse {
    public void maMethode() {
        // ...
    }
}
```

- **Appel d'une méthode d'instance**

```
instance.methode();
```

```
clio.deplacer();
```

Variables locales

- Variables temporaires qui existent seulement pendant l'exécution de la méthode

```
public class MaClasse {
    // …
    void maMethode() {
        int monNombre = 10;
    }

    void maMethode2() {
        System.out.println(monNombre);
        // Erreur de compilation
    }
}
```

Initialisation des variables

- **Variables d'instance**

  Si une variable d'instance n'est pas initialisée, elle prend une valeur par défaut

boolean	false
byte, short, int, long	0
float, double	0.0
char	\u0000
référence	null

```
public class Voiture {
    String marque;    // null
    int nbKilometre = 1000;
    // ...
}
```

- **Variables locales**

  Les variables locales n'ont pas de valeur par défaut et **doivent obligatoirement être initialisées**

Constructeur

- Le constructeur est une méthode spéciale dans la classe appelée à la création d'instances
- Un constructeur:
  - porte le nom de la classe
  - n'a pas de type de retour

```
public class Voiture {
    public Voiture() { // pas de type de retour
        // Corps du constructeur
    }
}
```

- On peut surcharger le constructeur

Constructeur par défaut

- Lorsqu'une classe ne comporte pas de constructeur, java ajoute automatiquement un constructeur par défaut

```
public class MaClasse {
    // public MaClasse() { } → implicitement généré par java
}
```

- Si un constructeur est ajouté à la classe, java n'ajoute plus automatiquement le constructeur par défaut.
Il devra être ajouté explicitement

```
public class MaClasse {
    public MaClasse() {   }   // doit être ajouté
    public MaClasse(String str) {   }
}
```

Le mot clé this

- Le mot clé **this** fait référence à l'objet en cours
- On peut l'utiliser pour :
  - manipuler l'objet en cours

```
maMethode(this);
```

  - faire référence à une variable d'instance

```
public class MaClasse {
    private int nombre;

    public MaClasse(int nombre) {
        this.nombre = nombre;
    }
}
```

Le mot clé this

- Faire appel au constructeur propre de la classe
  this doit être la première instruction du constructeur

```
public class MaClasse {
    private int a;
    private int b;

    public MaClasse(int a) {
        this(a, 16);
    }

    public MaClasse(int a, int b) {
        this.a = a;
        this.b = b;
    }
}
```

Cycle de vie d'un objet

- Un objet est instancié avec **new**

```
String str=new String("hello world");
```

- Un objet peut être collecté par le garbage collector lorsqu'il n'y a plus de référence qui pointe sur lui

- Garbage collector
  - Travail en arrière plan
  - Intervient lorsque le système a besoin de mémoire ou de temps en temps (priorité faible)

Variables de classes

- Variables partagées par toutes les instances de classe
- Elles sont déclarées avec le mot clé **static**
- **Pas besoin d'instancier** la classe pour les utiliser
- Chaque objet détient la **même** valeur de cette variable

```
public class Voiture {
    String type;
    static int nbVoitures;
    // …
}
```

- **L'appel de ses variables :**

`Classe.variableDeClasse;`

```
Voiture.nbVoiture;
```

Méthodes de classes

- Méthodes définissant un comportement global ou un service particulier
- Déclarées avec le mot clé **static**
- Peuvent être surchargées (même nom, ≠ paramètres)
- N'utilisent pas de variables d'instance parce qu'elles doivent être appelées depuis la classe

```
public class MaClasse {

    public static void maMethode() {
        // ...
    }
}
```

- **Appel des méthodes de classes**

```
MaClasse.maMethode();
```

Méthode principale (main)

- La méthode main représente le point d'entrée d'une application en exécution
- Elle peut être
  - intégrée dans une classe existante
  - écrite dans une classe séparée

```
public class Launch {

    // Méthode principale
    public static void main(String[] args) {
        // Création d'une instance de la classe Voiture
        Voiture Clio = new Voiture();
    }
}
```

Portée des variables

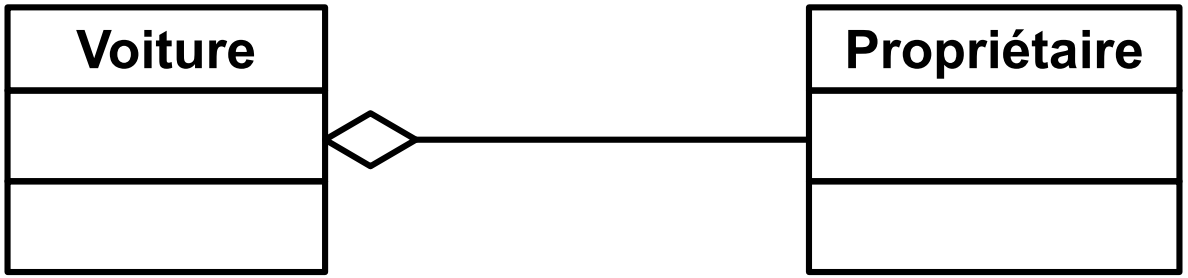
- Chaque bloc de code à sa propre portée
- Quand des blocs contiennent d'autre bloc. Les blocs contenus peuvent faire référence aux variables du bloc conteneur mais pas l'inverse

```
int a=10;
if(a>0){
    int somme= a+20;        // OK
}
System.out.println(somme);   // erreur
```

- **variable locale :** de sa déclaration → à la fin du bloc
- **variable d'instance** : de sa déclaration → jusqu'à la destruction de l'objet par le garbage collector
- **variable de classe** : de sa déclaration → à la fin du programme

Agrégation

- **Agrégation = associer un objet avec un autre**
  ex : Objet Propriétaire à l'intérieur de la classe Voiture




```
public class Proprietaire {
    // …
}
public class Voiture() {
    Proprietaire owner ;
    //  …
}
```

Accessibilité

- **Accessibilité = utilisation de facteurs de visibilité**

public	accessible par toutes les classes
protected	accessible par toutes les sous-classes et les classes du même package
« rien »	accessible seulement par les classes du même package **(default)**
private	accessible seulement dans la classe elle-même

```
public class Point {
    private int x, y;
    public Point(int x, int y) {
        // …
    }
}
```

Encapsulation

- **Encapsulation =**
  - Regroupement de code et de données
  - Masquage d'information par l'utilisation d'accesseurs (**getters et les setters**) afin d'ajouter du contrôle
- L'encapsulation permet de restreindre les accès aux membres d'un objet, obligeant ainsi l'utilisation des membres exposés

Qu'est-ce qu'un JavaBean ?

- **Une simple classe Java qui doit avoir :**
  - un constructeur public sans arguments (et éventuellement d'autres constructeurs)
  - des attributs privés
  - des getters et setters pour chaque attribut
  - un moyen de sérialisation (généralement, implémente java.io.Serializable)
- **POJO = Plain Old Java Object**
  - Un POJO est un objet Java lié à aucune autre restriction que celles forcées par la spécification du langage

Packages

Package = groupement de classes qui traitent un même problème pour former des "bibliothèques de classes"

- La hiérarchie des packages correspond à l'arborescence dans le système de fichier : un package correspond à un répertoire
- Une classe appartient à un package si la 1er instruction du fichier source est :

```
package nompackage;
```

- Les règles de nommage sont les même que pour les variables
- Par convention les packages:
  - sont toujours en minuscule
  - commencent par le nom de domaine inversé de l'entreprise

```
package fr.dawan.monpackage;
```

Packages

- Pour utiliser une classe, on peut au choix :
  - Être dans le même package
  - Préfixer par le package (à chaque utilisation)

    `monpackage.MaClasse variable ;`

    ```
    java.util.Date date;
    ```
  - Au début du fichier importer la classe ou le package entier

    ```
    import nompackage.MaClasse;
    import nompackage.*;
    ```
- Le package **java.lang** est importé automatiquement
- Un package par défaut est attribué aux classes qui sont définies sans déclarer une appartenance à un package
  ↳ correspond au répertoire de travail

Import static

- **import static** permet d'importer **uniquement** les variables et les méthodes de classe d'une classe
- On a plus besoin du nom de la classe pour appeler une méthode ou pour accéder à une variable

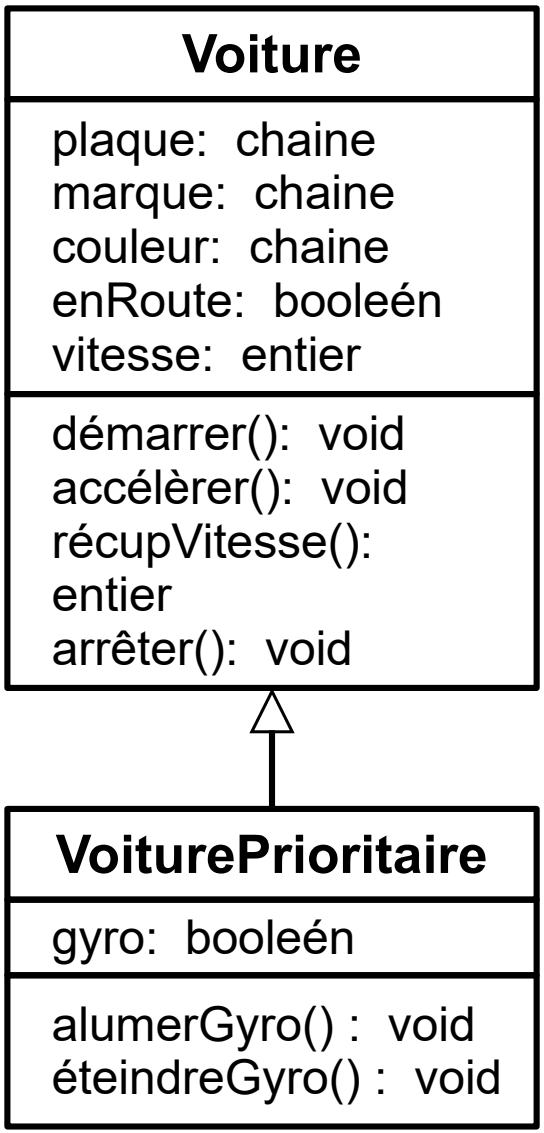
```
import static java.util.Arrays.sort;
public class Test {
    public static void main(String[] args) {
        int tab[] = { 5, -2, 6, 8 };
        sort(tab);
        // ...
    }
}
```

- Si dans la classe, on définit une méthode ou une variable qui porte le même nom elle a la priorité sur celle importée

Héritage

- L'héritage permet de créer la structure d'une classe à partir des membres d'une autre classe

- La sous-classe hérite de tous les attributs et méthodes de sa classe mère



Héritage en java

- **En Java, L'héritage est simple**
  Une classe ne peut hériter que d'une seule classe mère
- Utilisation du mot clé **extends**

```
public class VoiturePrioritaire extends Voiture{
    private boolean gyrode;
    // les actions
}
```

- Utilisation du mot clé **super** (référence à la classe mère)

```
private void demarrer() {
    gyrode = true;
    super.demarrer();
}
```

Redéfinition (overriding)

- **La redéfinition consiste à réimplémenter une version spécialisée d'une méthode héritée d'une classe mère**
- Les signatures des méthodes dans la classe mère et la classe fille doivent être identiques
- Le type de retour de la méthode redéfinie doit être du même type ou un sous-type de celui de la méthode de la classe mère
- L'accessibilité d'une méthode redéfinie peut être moins restrictive que la méthode de la superclasse
  protected → public
- On ne peut pas redéfinir des méthodes privée et/ou de classe
- Une méthode annotée avec **@override** doit obligatoirement être redéfinie (vérification à la compilation)

Utilisation de final

Le mot clé **final** permet

- de déclarer une constante

```
public static final int X = 3;
```

- d'interdire la redéfinition d'une méthode

```
public final void methode1() {
    // ...
}
```

- d'interdire l'héritage à partir de la classe

```
public final class Classe1() {
    // ...
}
```

Polymorphisme

Le polymorphisme est la propriété d'une entité de pouvoir se présenter sous diverses formes

Ce mécanisme permet de faire collaborer des objets entre eux sans que ces derniers aient déclarés leur type exact

Exemples :

- On peut avoir une voiture prioritaire avec le type Voiture
- On peut créer un tableau de Voitures et placer à l'intérieur des objets de type Voiture et d'autres de type VoiturePrioritaire

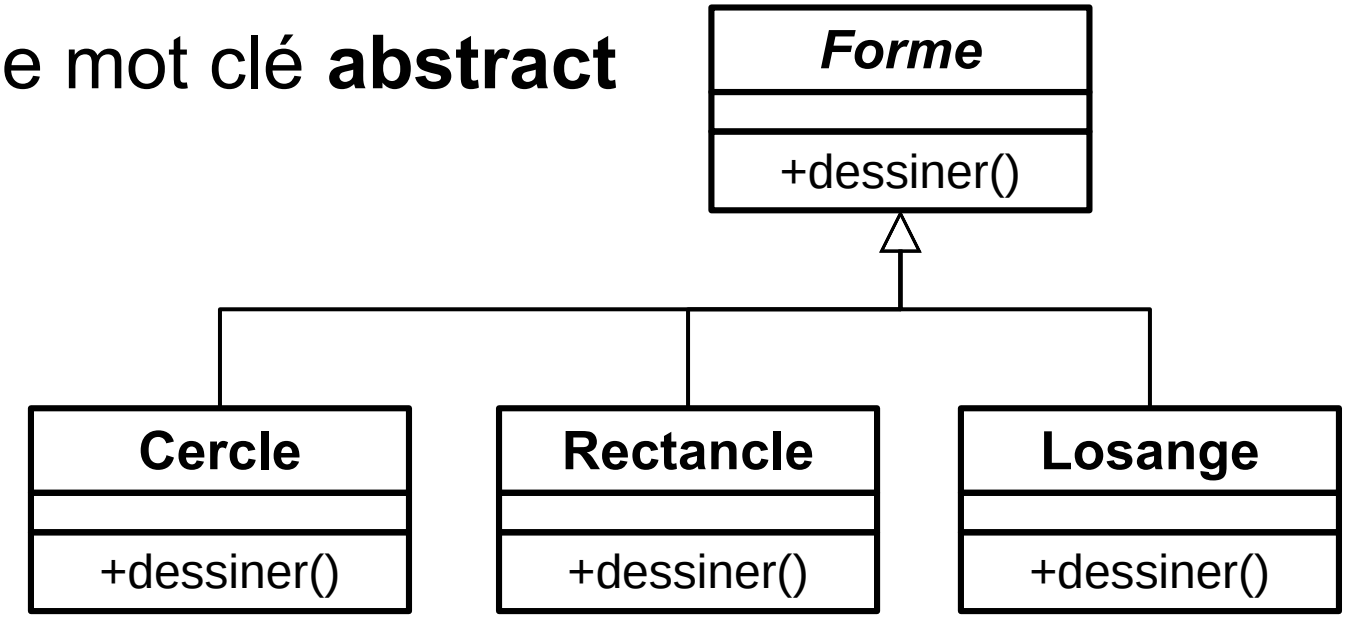
Polymorphisme

- Un objet Java est accessible à l'aide d'une référence :
  - du même type que l'objet
  - qui est une superclasse de l'objet
  - qui définit une interface que l'objet implémente (directement ou via une superclasse)
- Le type de l'objet
  ↳ détermine quelles propriétés existent dans l'objet en mémoire
- Le type de la référence à l'objet
  ↳ détermine quelles méthodes et variables sont accessibles au programme Java
- La conversion d'un objet :
  - d'une sous-classe en une superclasse est implicite
  - d'une superclasse en une sous-classe nécessite une conversion explicite

Classe Abstraite

Une classe **qui ne peut être instanciée**

- Définit un type de squelette pour les sous-classes
  - Si elle contient des méthodes abstraites,
    les sous-classes doivent implémenter le corps des méthodes abstraites
- Déclarée avec le mot clé **abstract**



Classe Abstraite

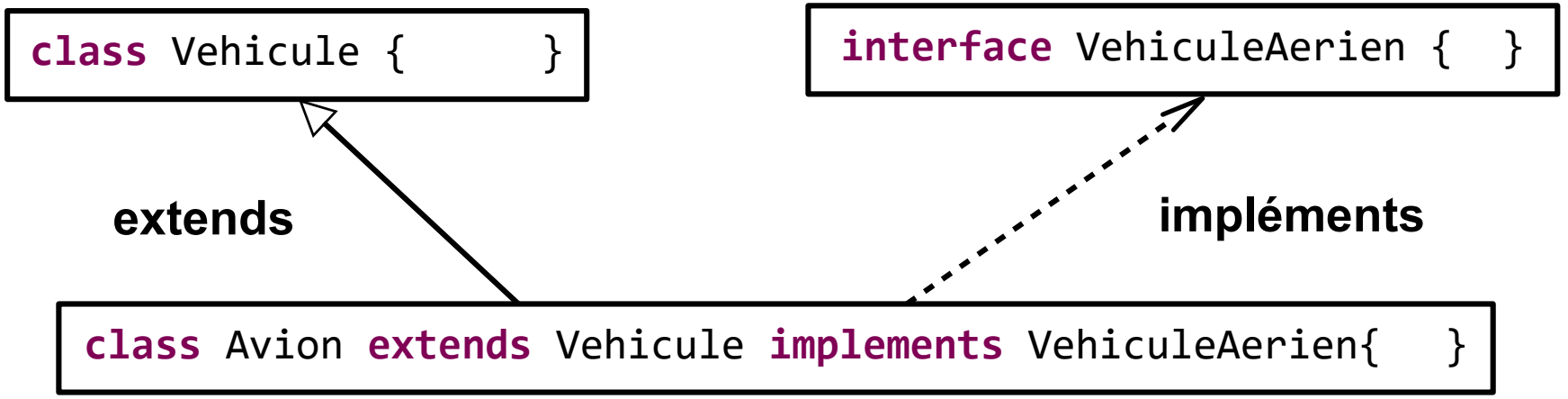
- **Exemple**

```
public abstract class Animal {
    private int age;
    public void eat() {
        System.out.println(" animal is eating ");
    }
    public abstract String getName();
}

public class Swan extends Animal {
    public String getName() {
        return "Swan";
    }
}
```

Interfaces

- Une pseudo classe abstraite marquée par le mot clé **interface** contenant juste des signatures de méthodes (et des constantes) afin de montrer les capacités



- **Implémentation d'une interface**

  Une classe peut implémenter plusieurs interfaces, chacune séparée par ,

```
public class Elephant implements WalksOnFourLegs, Herbivore { }
```

Classe Object

Toutes les classes héritent implicitement de la classe Object

- **toString()** Représenter un objet sous forme d'une chaîne de caractère
  par défaut retourne : nomDeLaClasse@hashcode
- **equals(Object obj)** Permet de comparer deux objets
  par défaut, si this a la même référence que obj
  ↳ retourne vrai
- **hashcode()** Calcul un code numérique pour l'objet
  si equals est redéfinit, hashcode doit l'être aussi
- **clone()** Pour dupliquer un objet
  interdit par défaut, pour l'utiliser il faut :
  - redéfinir la méthode par une méthode public
  - la classe implémente l'interface clonable

Énumération

- Une énumération est un type de données, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs qui sont des constantes prédéfinies

```
public enum Direction {
   NORD, EST, SUD, OUEST
}
Direction dir=Direction.NORD;
```

- **Les énumérations possèdent des méthodes**

name() et toString()	renvoie une chaîne de caractères contenant le nom de la constante
valueOf()	renvoie la valeur énumérée à partir de sa chaîne de caractères
ordinal()	retourne l'index de la valeur selon l'ordre de déclaration (commence à partir de 0)
values()	retourne un tableau de toutes les valeurs énumérées disponibles

Les conventions

- Indentation significative, 80 caractères par ligne
- Accolades : ouvrantes en fin de ligne, fermantes isolées

```
public Voiture() {
}
```

- Ordre de déclaration

statique → d'instance

attribut → constructeur → méthode

private → protected → public

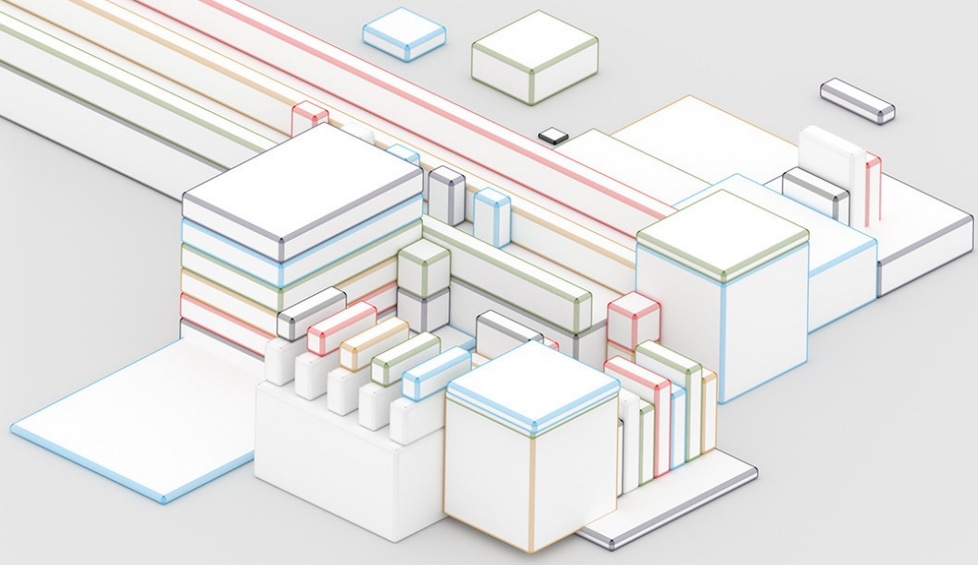
- Nommage

un.package UneClasse UneInterface uneMethode

uneVariable unAttribut UNE_CONSTANTE

Dawan
Training is our way

Classes essentielles



String

- Chaîne immuable de caractères unicodes
  - ↳ Une fois créé, elle n'est plus modifiable
- **Longueur** : `length`
- **Comparaison** : `compareTo compareToIgnoreCase …`
- **Concaténation** : `concat join`
- **Découpage** : `split substring`
- **Recherche** : `startsWith endsWith replace charAt indexOf lastindexOf contains …`
- **Mise en forme** : `toLowerCase toUpperCase format ,trim`
- On peut chaîner les méthodes

```
String str = "abcd".concat("ef").toUpperCase(); // ABCDEF
```

StringBuilder

- **La chaîne peut être modifiée** → pas de création de chaîne intermédiaire
  Elle retourne une référence à elle même
- **charAt**(), **indexOf**(), **length**() et **substring**()
  ↳ idem String
- **append**(String str) — ajouter une chaîne
- **insert**(int offset, String str) — insérer une chaîne (offset commence à 0)
- **delete**(int start, int end) — supprimer les caractères entre start et end (non inclut)
- **toString**() — convertie un StringBuilder en String

StringTokenizer

- Permet de décomposer une chaîne de caractères en une suite de mots séparés par des délimiteurs
- **Constructeurs**
  - **`StringTokenizer`**`(String str)`

    str: chaîne à analyser, les délimiteurs sont Espaces / Tab / CR / LF
  - **`StringTokenizer`**`(String str, String delim)`

    str: chaîne à analyser ,***delim*** : chaîne du délimiteur
  - **StringTokenizer**(String str, String delim, boolean returnValue)

    idem , si ***returnValue*** est vrai les délimiteurs font partie de la chaîne renvoyer
- **Méthodes**
  - **hasMoreTokens**() indique s'il reste des éléments à extraire
  - **nextToken**() renvoie l'élément suivant
  - **countTokens**() renvoie le nombre d'éléments

Date

- En java 8, il existe 3 API de gestion du temps :
  - java.util.Date (depuis le jdk 1.0)
  - java.util.Calendar (depuis le jdk 1.1)
  - java.time (depuis le jdk 8)

LocalDate	Contiens uniquement la Date
LocalTime	Contiens uniquement l'heure
LocalDateTime	Contiens la date et l'heure

- La méthode statique **now**() donne la date et l'heure courante
- La méthode statique **of**() permet de créer une date ou une heure spécifique

Date

- **LocalDate**

  Le mois peut être passé soit en entier soit sous forme d'énumération **Month**

  - **of**(int year, int mount,int dayOfMonth)
  - **of**(int year, Mounth mount,int dayOfMonth)
- **LocalTime**
  - **of**(int hour, int minute)
  - **of**(int hour, int minute, int second)
  - **of**(int hour, int minute, int second, int nanoSecond)
- **LocalDateTime**
  - **of**(LocalDate date, LocalTime time)
  - des méthodes combinant des paramètres de **LocalTime** et **LocalDate**

Manipuler les dates et le temps

- La date et le temps sont immuables
- On manipule une date et un temps avec les méthodes plus et minus : **plusYears**(), **plusMonths**(), **plusWeeks**(), **plusDays**(), **plusHours**(), **plusMinutes**(), **plusSeconds**(), **plusNanos**()
- **Periods** → contiens une période

```
Period m = Period.ofMonth(1); // période d'un mois
```

ofYears(), **ofMonths**(), **ofWeeks**(),**ofDays**(), **of**(year,month,day)

On ne peut pas chaîner les Periods (sinon, dernier pris en compte)

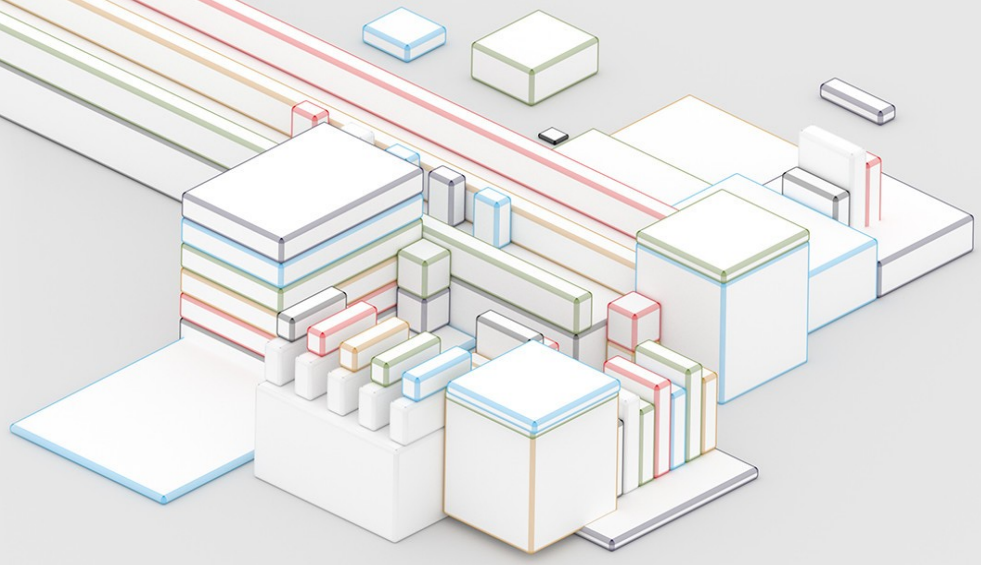
```
LocalDate date = LocalDate.of(2015, 1, 20);
Period period = Period.ofMonths(1); // période d'un mois
System.out.println(date.plus(period)); // affiche 20/02/2015
```

Maths

- Toutes les méthodes sont statiques
- Classe contenant des implémentations standard (abs(), cos(), floor(), min(), round(), pow(), sqrt()…)
- Constantes E et PI

```
int i = (int) Math.floor(5.99 / 2);    // 2
long l = Math.round(0.51);             // 1
double d = Math.sin(Math.PI * 0.75);   // 0.707106781
double d2 = Math.pow(2.0, 3.0);        // 8.0
double d3 = Math.sqrt(9.0);            // 3.0
double d4 = Math.abs(-4.0);            // 4.0
int min = Math.min(3, 67);             // 3
double r = Math.random(); // nombre aléatoire de 0.0 à 1.0
```

Exceptions



Définition

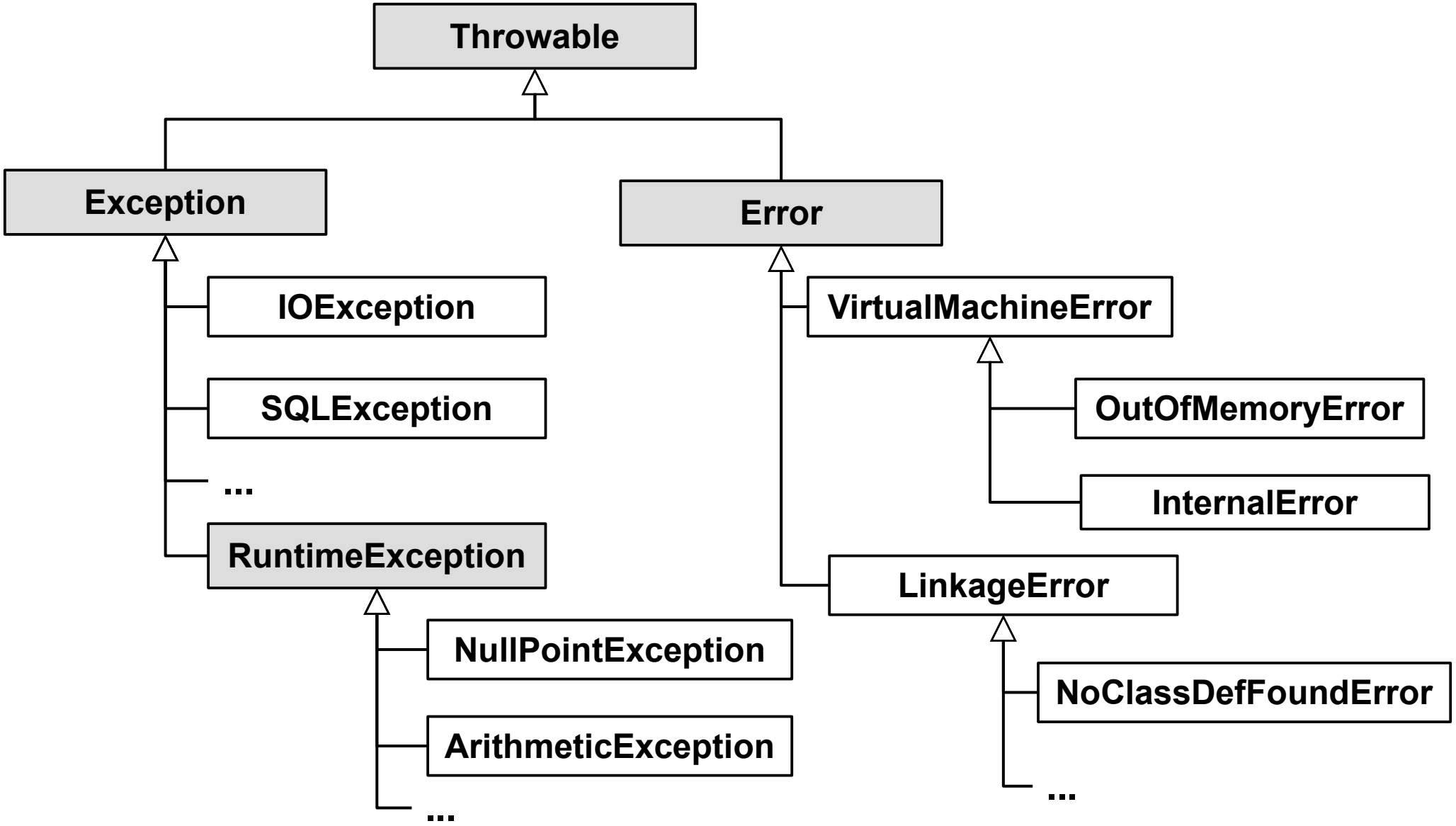
Situations inattendues ou exceptionnelles

qui surviennent pendant l'exécution d'un programme,

interrompant le flux normal d'exécution

Java Documentation

Classe Throwable



Types d'exceptions

- **Checked exceptions**

  Le développeur doit obligatoirement les anticiper et coder des lignes pour les traiter

  Exemple : Ouvrir un fichier qui n'existe pas

- **Errors**

  On ne doit pas les identifier et le programme s'arrête en les rencontrant

  Exemple : La JVM charge une classe inexistant

- **Runtime exceptions**

  Ne peuvent être prévues (dans certains cas)

  Exemple : Essayer de lire une valeur en dehors d'un tableau

Bloc try et catch

- Utilisé pour encadrer un bloc susceptible de déclencher une exception

```
try {
    // des lignes de code susceptibles
    // de lever une exception
} catch (IOException e) {
    // capture et traitement de l'exception de type
       IOException
} finally {
    // toujours exécuté
    // même sans exception ou une exception imprévue
}
```

Bloc try et catch

```
try {
    FileReader lecteur = new FileReader(nomDeFichier);
} catch (FileNotFoundException e) {
    // capture FileNotFoundException
    System.err.println("FileNotFoundException catched :");
    e.printStackTrace();
} catch (IOException e) {
    // capture IOException
    System.err.println("IOException catched :");
    e.printStackTrace();
}
```

Lever ou propager une exception

- **Lever une exception**

  Le mot clé **throw** est utilisé pour déclencher une exception à n'importe quel moment

```
if (age < 0) {
    throw (new Exception("Impossible: age négatif"));
}
```

- **Propager une exception**

  Le mot clé **throws** est utilise pour dire a la méthode de ne pas récupérer l'exception localement mais plutôt l'envoyer dans la méthode appelante

```
public void readFile(String path) throws FileNotFoundException
{
    FileReader reader = new FileReader(path);
}
```

Créer ses propres exceptions

- Il faut seulement hériter de la classe **Throwable** ou une sous-classe (généralement la classe **Exception**)

```
public class MonException extends Exception {
    // Une exception valide !
}

public class NegativeNumberException extends NumberException
{
    public NegativeNumberException(int num) {
        super("Le nombre " + num + "est négatif");
    }
    // C'est une exception valide également !
}
```

Relancer une exception

- Une exception peut être partiellement traitée, puis relancée

```
public void readFile(String path) throws IOException {
    try{
        // ...
    }
    catch(IOException e){
        // Traitement de l'exception
        throw e;  // Relance de l'exception
    }
}
```

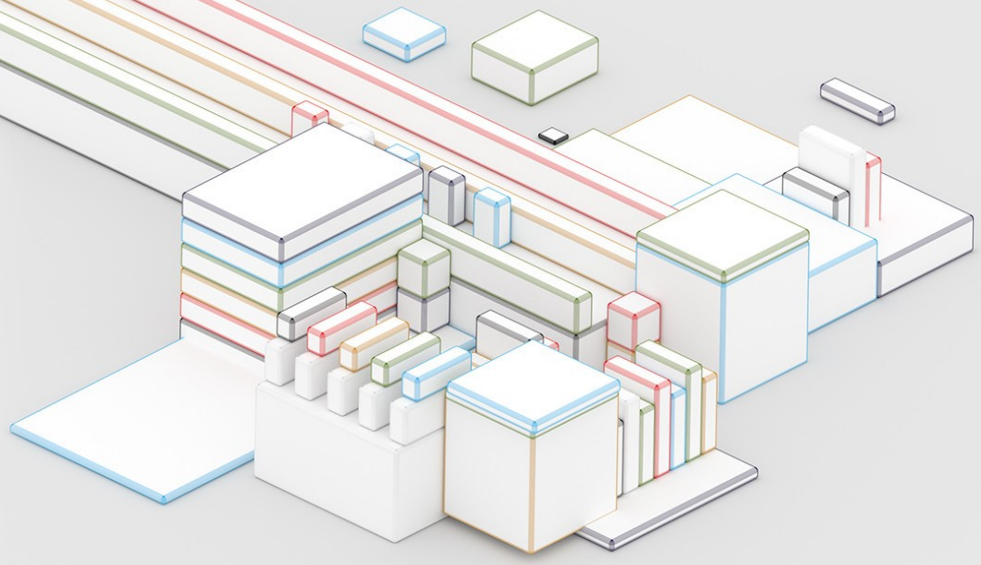
Relancer une exception d'un autre type

```
public void readFile2(String path) throws MonException {
    try{
        // ...
    }
    catch(IOException e){
        // Traitement de l'exception
        throw new MonException("Lecteur impossible", e );
        // ↪ Relance une exception d'un autre type
    }
}
```

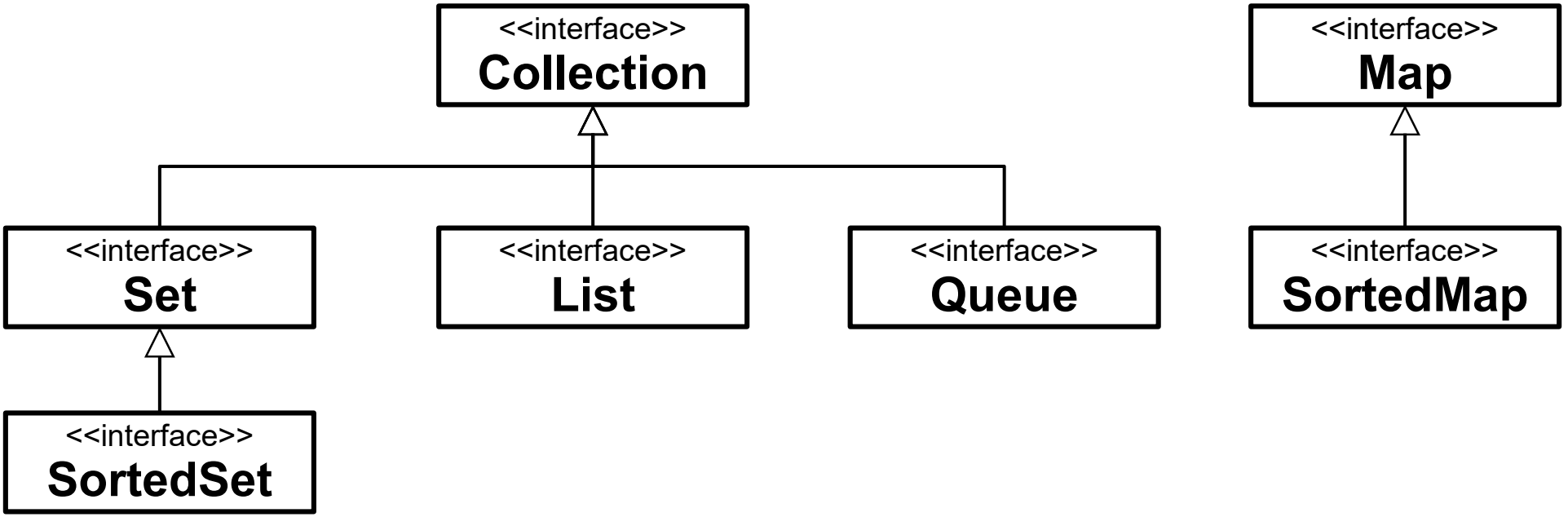
En paramètre, on passe l'exception originale

Dawan
Training is our way

Collections



Les collections



- Une collection est un objet qui contient d'autres objets
- Les interfaces et les classes se trouvent dans le paquetage **Java.util**

Interface Collection

add(T t)	ajouter un objet à la collection
remove(T t)	retirer un objet à la collection
contains(T t)	renvoie true, si l'objet passé en paramètre est contenu dans la collection
size()	renvoie le nombre d'éléments de la collection
isEmpty()	renvoie true, si la collection est vide
clear()	efface la collection
toArray(T[] a)	convertit la collection en tableau
retainAll(collection)	retire tous les éléments de la collection qui ne se trouvent pas dans la collection passée en paramètre
containsAll(collection)	retourne true, si tous les éléments de la collection passée en paramètre se trouvent dans la collection courante

Parcourir une collection

- **Les itérateurs**

  La méthode **iterator()** de l'interface collection retourne une instance d' Iterator, qui contient 3 méthodes :

  - **hasNext()** renvoie true, si la collection possède encore des éléments à itérer
  - **next()** renvoie l'élément suivant
  - **remove()** retire de la collection l'élément courant (pas supportée par toutes les implémentations)

- **boucles "for each"**

```
for(type element : collection) {
   //…
}
```

Interface List

L'interface List correspond à un groupe d'objet indexé

add(int index, T t) **addAll(int index, collection)**	Insère un ou plusieurs éléments à la position de l'index
remove(int index)	Retire l'élément à la position de l'index. L'élément est retourné par la méthode
set(int index, T t)	Remplace l'élément placé à la position index par celui passé en paramètre. L'élément retiré est retourné par la méthode
get(int index)	Revoie l'élément placé à l'index
indexOf(Object o) **lastIndexOf(Object o)**	Revoient le premier et le dernier index de l'objet passé en paramètre
subList(int debut, int fin)	Renvoie la liste composé des éléments compris entre debut, et fin – 1. Retourne une sur la liste pas une copie

Les génériques

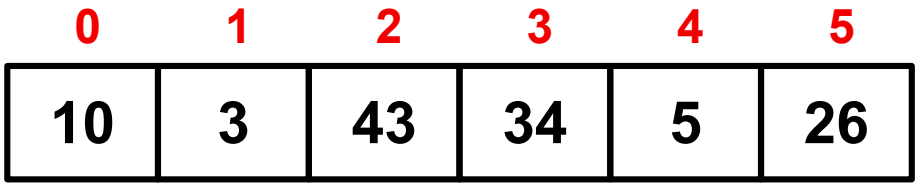
- Utilisés pour typer une variable
- Depuis Java 5.0, ils sont utilisés dans les collections

  ArrayList est une collection générique, que l'on spécialise par les symboles < >

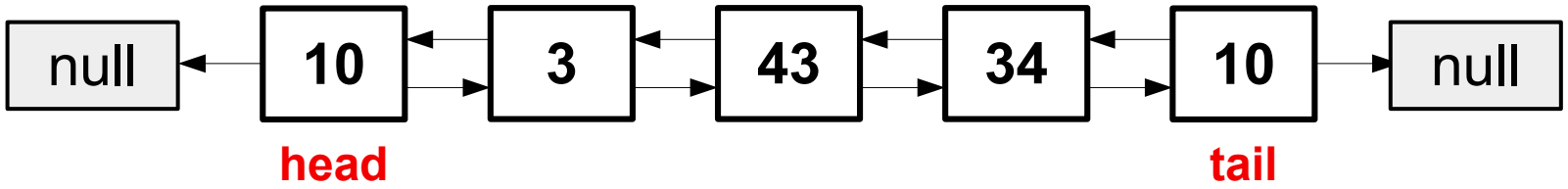
```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
list.add("world");
System.out.println(list.size() + " mots");
for (String item : list) {
    System.out.print(item);
}
```

Implémentation de List

- **ArrayList** : un tableau à taille variable
  ↳ accès rapide a un élément donné



- **Vector :** idem ArrayList mais synchronisé
- **LinkedList :** Une liste chaînée
  ↳ insertion en début et en fin de liste rapide



Méthode supplémentaire : **addFirst**(Object o), **addLast**(Object o), **getFirst**(), **getLast**(), **removeFirst**(), **removeLast**()

Interface Set

L'interface Set correspond à un ensemble d'objets qui n'accepte pas de doublons (2 objets égaux avec equals)

- L'ajout d'un élément dans un Set peut échouer, si cet élément s'y trouve déjà, dans ce cas, add(T t) renvoie false
- **Les principales implémentations :**
  - **HashSet** : Offre des performances constantes pour les opérations add, remove, contains et size
    Il faut éviter l'itération sur de grand ensemble
  - **LinkedHashSet** :La classe est une extension de HashSet qui améliore les performances de l'itération
  - **TreeSet** : garantit que les éléments sont rangés dans leur ordre naturel

Interface SortedSet

Avec l'interface SortedSet, tous les objets sont automatiquement triés dans un ordre que l'on peut définir

comparator()	Renvoie l'objet instance de Comparator
first() **last()**	Renvoient le plus petit et le plus grand objet de l'ensemble
headSet(T t)	Renvoie une instance de SortedSet contenant tous les éléments strictement plus petit que l'élément passé en paramètre
tailSet(T t)	Renvoie une instance de SortedSet contenant tous les éléments plus grands ou égaux que l'élément passé en paramètre
subSet(T inf, T sup)	Renvoie une instance de SortedSet contenant tous les éléments plus grands ou égaux que inf, et strictement plus petits que sup

Comparable et Comparator

Avec SortSet, on peut comparer deux objets en :

- Implémentant l'interface **Comparable** :
  Il n'a qu'une seule méthode **compareTo**(T t) qui renvoie :
  - **0** si égal
  - **1** si plus grand que l'argument
  - **-1** plus petit quel'argument
- Fournissant au constructeur de **SortedSet**, une instance l'interface de **Comparator**
  Il n'a qu'une seule méthode **compare**(T t1,T t2) qui renvoie :
  - **0** si t1 égal t2
  - **1** si t1 est plus grand que t2
  - **-1** si t1 est plus petit que t2

Interface Queue

L'interface Queue modélise une file d'attente simple

add(T t)
offer(T t)

Ajoutent un élément à la liste

Si la capacité maximale de la liste est atteinte :

- **add()** lance une exception **IllegalStateException**
- **offer()** retourne **false**

remove()
poll()

Retirent un élément de la file d'attente

Si aucun élément n'est disponible :

- **remove()** lance une exception **NoSuchElementException**
- **poll()** retourne **null**

element()
peek()

Examinent toutes les deux l'élément disponible, sans le retirer de la file d'attente

Si aucun élément n'est disponible :

- **element()** lance une exception **NoSuchElementException**
- **peek()** retourne **null**

Interface Map

L'interface Map correspond à un groupe de clé/valeur
Une clé repère une et une seule valeur

put(K key, V value)	Associe une clé à une valeur
get(K key)	Récupérer une valeur à partir d'une clé
remove(K key)	Supprime la clé passée en paramètre de la table, et la valeur associée
keySet()	Renvoi un Set contenant toutes les clés de la table de hachage
values()	retourne l'ensemble de toutes les valeurs stockées dans la table
clear()	Efface tout le contenu de la table
size()	Renvoie le cardinal de la table
isEmpty()	Indique si la table est vide

Interface Map

putAll(Map map)	Ajouter toutes les clés de la table passée en paramètre
containsKey(K key)	Tester si la clé passée en paramètre sont présentes dans cette table
containsValue(V value)	Tester si la valeur passée en paramètre sont présentes dans cette table
entrySet()	Retourne un Set, dont les éléments sont des Map.Entry

- **Map.Entry** permet de modéliser les couples (clé, valeur) d'une table de hachage
  - **getKey()** et **getValue()** retournent la clé et la valeur de ce couple
  - **setValues()** modifie la valeur associé à une clé durant l'itération
- Implémentation : **HashMap**, **TreeMap** ...

Classe Collections

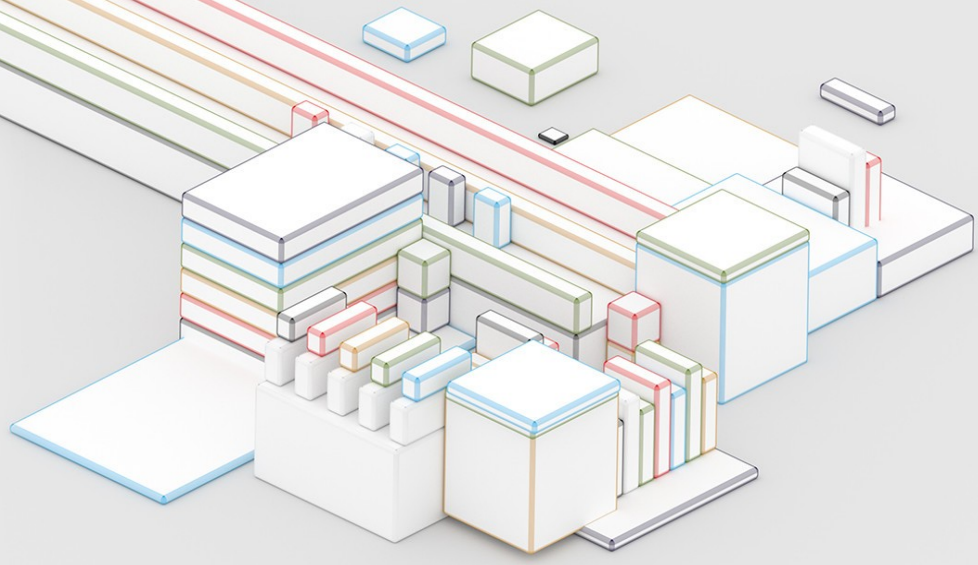
- Collections est une classe utilitaire pour travailler avec des collections
  - trie (liste)
  - recherche (liste)
  - copies
  - minimum et maximum
  - ...

Classe Arrays

- **Arrays** est une classe utilitaire qui traite des tableaux

fill(int[] tab, int val)	initialisation d'un tableau
equals(int[] tab1, int[] tab2)	comparaison de deux tableaux
toString(int[] tab)	méthode toString() pour les tableaux
sort(int[] tab)	tri d'un tableau
binarySearch(int[] tab, int key)	recherche d'un élément dans un tableau **trié**, • retourne l'index de l'élément recherché • Si le tableau n'est pas trié → le résultat est imprévisible • Si la valeur n'est pas trouvée → retourne une valeur négative, qui a pour valeur l'index où il pourrait être inséré tout en préservant l'ordre du tableau multiplié par -1 en ajoutant -1

Entrées/Sorties



File

La classe **File** est une représentation d'un chemin d'accès (absolu ou relatif) au fichier et au dossier

```
File(String pathname)
File(String parent,String child)
File(File parent, String child)
```

pathname → chemin du fichier / dossier
child → nom du fichier / dossier
parent → chemin du dossier parent

`boolean mkdir()`	créer un dossier
`boolean createNewFile()`	créer un fichier vide
`boolean delete()`	effacer un fichier ou un dossier vide
`boolean exists()`	tester l'existence d'un fichier / dossier
`String getName()`	nom du fichier ou du dossier
`String getPath()`	chemin du répertoire parent
`File[] listFiles()`	tous les fichiers et dossiers du répertoire
`boolean isDirectory()` `isFile()/isHidden()`	retourne true si le chemin est un dossier / un fichier / caché
`separator, separatorChar`	caractère de séparation indépendant de l'OS (/ → unix , \\ → windows)

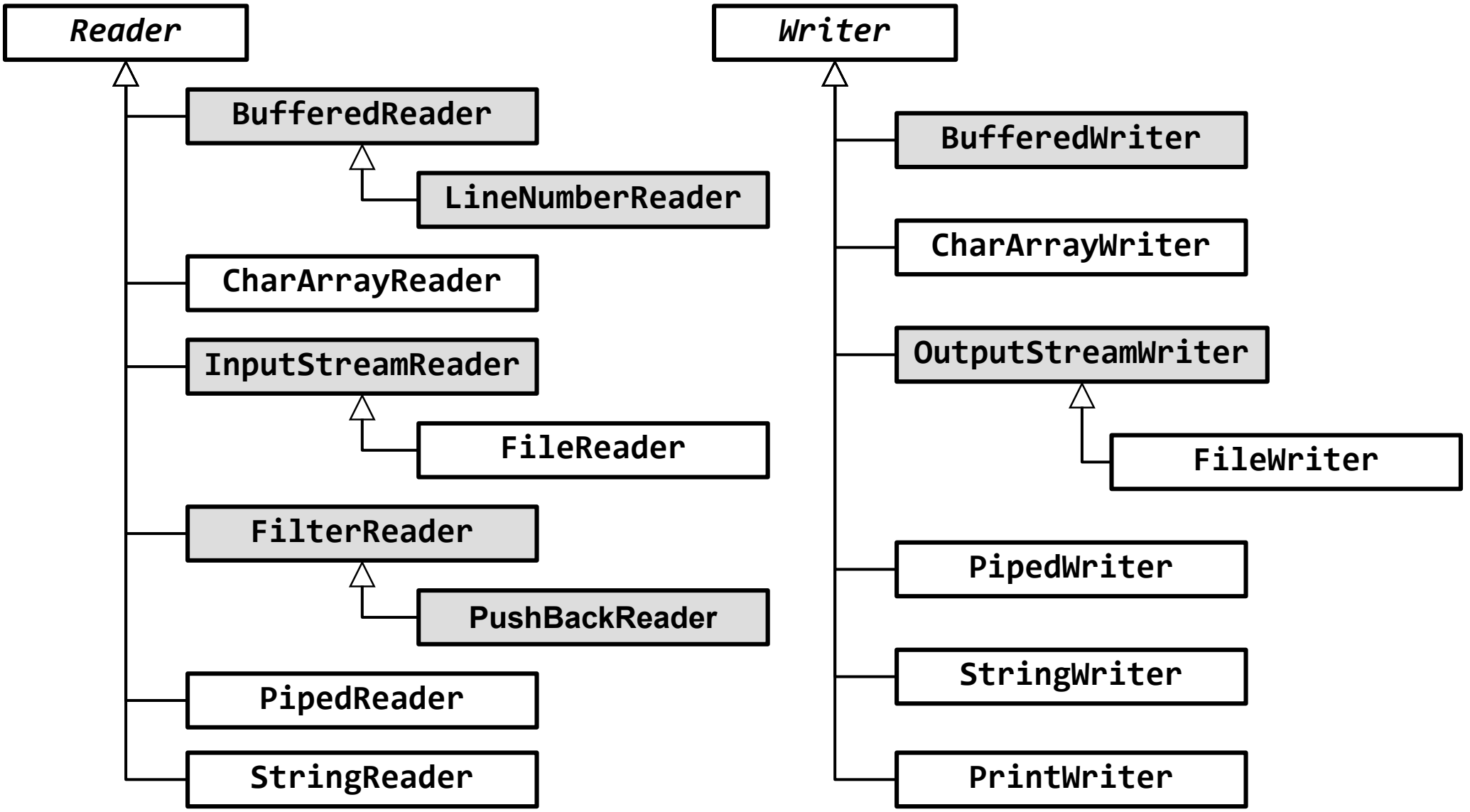
Définition

Stream = flux de données (sériel, temporaire, en entré ou en sortie)

- Le package **java.io** englobe les classes permettant la gestion des flux
- Principe d'utilisation d'un flux:
  - ouverture du flux
  - lecture/écriture de l'information
  - fermeture du flux (close)
- 4 classes abstraites

	Lecture	Écriture
Texte	`Reader`	`Writer`
Binaire	`InputStream`	`OuputStream`

Les flux de caractères



Reader

Classe abstraite qui est la classe mère de toutes les classes qui gèrent des **flux de caractères en lecture**

`int read()`	lit un caractère	retourne le nombre de caractères lue ou **-1**, si la fin du flux est atteinte
`int read(char[])`	lit plusieurs caractères et les places dans un tableau	
`int read(char[], int off, int max)`	lit au maximum ***max*** caractères et les places dans un tableau à partir de l'indice ***off***	
`long skip(long n)`	saute ***n*** caractères dans le flux et renvoie le nombre de caractères sautés	
`boolean ready()`	indique si le flux est prêt à être lu	
`mark(int num)`	place un marqueur sur la position courante dans le flux reste valide tant que l'on n'a pas lue **num** caractères	
`reset()`	revenir à la position marquée	
`boolean markSupported()`	indique si le flux supporte le marquage	

Writer

Classe abstraite qui est la classe mère de toutes les classes qui gèrent des **flux de caractères en écriture**

`write(int c)`	écrit le caractère dans le flux
`write(char[])`	écrit un tableau de caractères dans le flux
`write(char[],int of,int len)`	écrit une portion du tableau de caractère qui commence à l'indice ***of*** et de taille ***len***
`write(String)`	écrire la chaîne de caractères en paramètre dans le flux
`write(String,int of,int len)`	écrit une portion d'une chaîne de caractères qui commence à l'indice **`of`** et de taille **`len`**
`Writer append(CharSequence)`	ajoute des caractères à la fin du flux renvoie une référence au flux appelant
`Writer append(CharSequence ,int begin, int end)`	idem pour les caractères de ***`begin`*** à ***`end-1`***

Les flux de caractères avec un fichier

- **`FileReader`** et **`FileWriter`** permettent de gérer des flux de caractères avec des fichiers

`FileReader``(String)` **`FileReader`**`(File)`	créer un flux en lecture
`FileWriter``(String)` **`FileWriter`**`(File)`	créer un flux en écriture, si le fichier existe les données seront écrasées
`FileWriter``(String,boolean)`	**`true`** → les données sont ajoutées **`false`** → les données sont écrasées

- **`BufferedReader`** et **`BufferedWriter`** permettent de gérer des flux de caractères tamponnés avec des fichiers

`BufferedReader``(Reader)`	**`Reader`** permet de préciser le flux à lire
`BufferedReader``(Reader, int)`	***`int`*** permet de préciser la taille du buffer
`String` **`readLine`**`()`	lire une ligne de caractères dans le flux

Les flux de caractères avec un fichier

BufferedWriter(Writer writer) — ***writer*** → le flux utilisé pour l'écriture

BufferedWriter(Writer, int s) — ***s*** → la taille du buffer

flush() — vide le tampon en écrivant les données dans le flux

newLine() — écrire un séparateur de ligne dans le flux

- **PrintWriter** permet d'écrire dans un flux des données formatées

PrintWriter(Writer) — le flux peut être **Writer** ou **OutputStream**

PrintWriter(OutputStream) — le tampon est automatiquement vidé

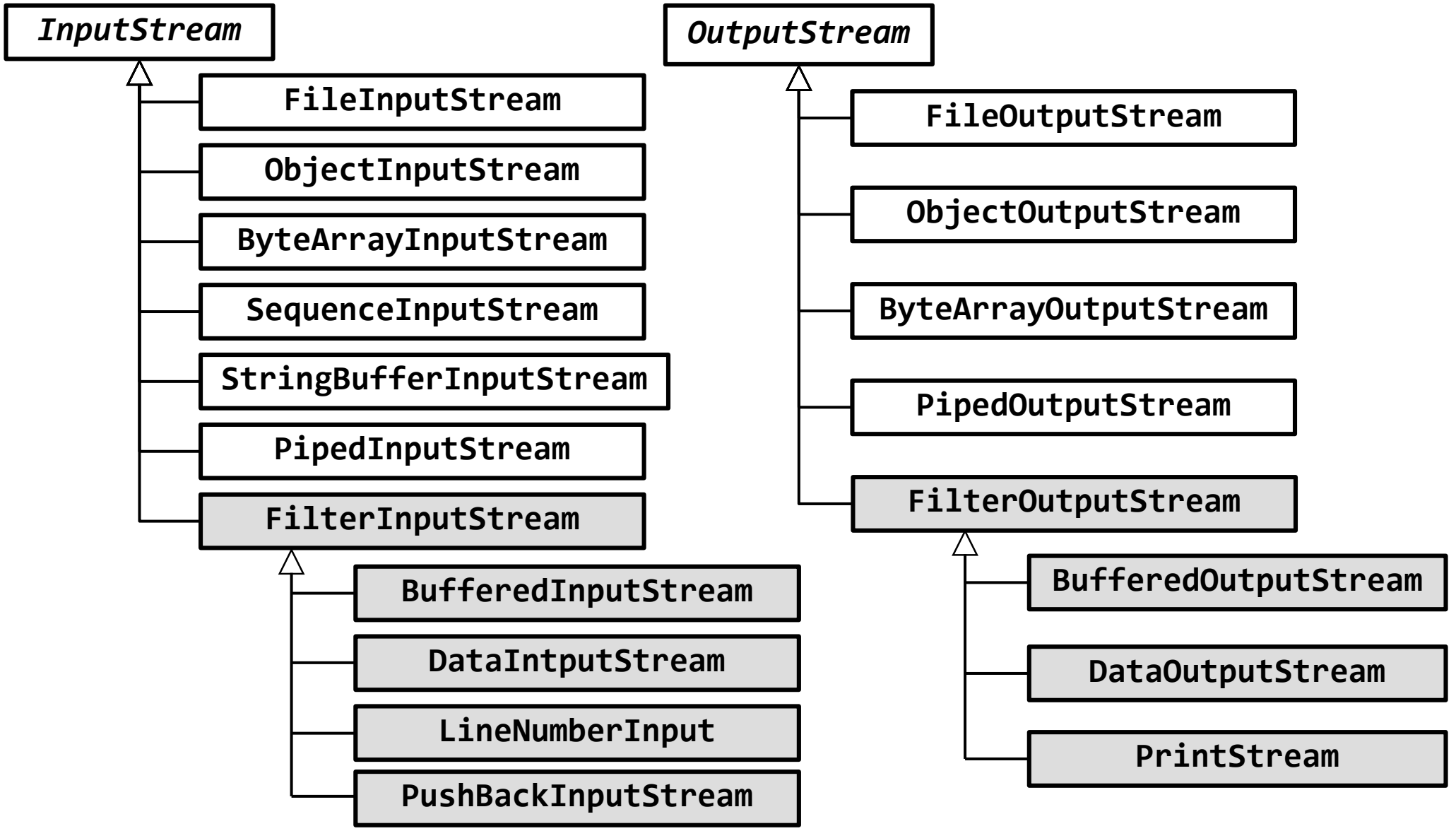
PrintWriter(Writer, boolean f) — ***f*** → indique, si le tampon doit être automatiquement vidé

PrintWriter(OutputStream,boolean f)

print( ), **println**( ) — acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux

flush() — vide le tampon en écrivant les données dans le flux

Les flux d'octets



InputStream

- Classe abstraite qui est la classe mère de toutes les classes qui gèrent des **flux d'octets en lecture**

int **read**()	retourne la valeur de l'octet lu
read(byte[] b)	lit plusieurs octets et les places dans un tableau retourne le nombre d'octet lu
int **read**(byte[], int off, int max)	lit au maximum max octets les places dans un tableau à partir de l'indice off retourne le nombre d'octet lu
long **skip**(long)	saute n octets dans le flux et renvoie le nombre de octets sautés
int **available**()	retourne une estimation du nombre d'octets qu'il est encore possible de lire dans le flux

retourne -1, si la fin du flux est atteinte

OutputStream

- Classe abstraite qui est la classe mère de toutes les classes qui gèrent des **flux d'octets en écriture**

write(int b)	écrit un octet dans le flux
write(byte[]b)	écrit un tableau d'octet dans le flux
write(char[] b, int off, int len)	écrit une portion du tableau d'octet caractère qui commence à l'indice **off** et de taille **len**
flush()	Vide le flux de sortie et force l'écriture de tous les octets de sortie mis en mémoire tampon

try with ressource

- Depuis Java 7, try with ressource permet de définir une ou plusieurs ressources qui seront automatiquement fermées à la fin de l'exécution du bloc de code
- Les ressources :
  - sont séparées par un point-virgule
  - doivent implémenter l'interface **AutoCloseable**

```
try (BufferedReader br = new BufferedReader(new FileReader(path)))
{
    System.out.println(br.readLine());
}

try(FileOutputStream output = new FileOutputStream("source.txt");
    FileInputStream input = new FileInputStream("cible.txt")){
    //…
}
```

Fichier properties

Les fichiers de properties sont utilisés pour stocker des paramètres de configuration (extension : .properties)

- **Format de fichier**
  Chaque ligne du fichier contient une propriétés
  Elle est composé d'une clé et d'une valeur associée

  `key=valeur`

  Les lignes commençant par **#** et par **!** sont des commentaires

```
#Commentaire sur le contenu du fichier
#Fri May 08 14:29:34 CEST 2020
version=1.0
user=JohnDoe
```

Classe Properties

- **Chargement depuis un fichier de .properties**

  `load(InputStream in)`
  `load(Reader read)`

- **Chargement depuis un fichier XML**

  `loadFromXML(InputStream in)`

```
Properties appProps = new Properties();
appProps.load(new FileInputStream(appConfigPath));
```

- **Lire / obtenir une valeur correspondante à une clé**

  `getProperty(String key)` → si la clé n'existe, retourne **null**

  `getProperty(String key, String defaultValue)`
  ↳ si la clé n'existe pas, retourne ***`defaultValue`***

```
String version = appProps.getProperty("version");
```

Classe Properties

- **Fixer une propriété**

  `String` **`setProperty`**`()` → si la clé existe on met à jour la valeur sinon on ajoute une nouvelle propriété

```
appProps.setProperty("name", "NewAppName");
```

- **Supprimer une propriété**

  `remove``()`

```
appProps.remove("version");
```

- **Enregistrer dans un fichier .properties**

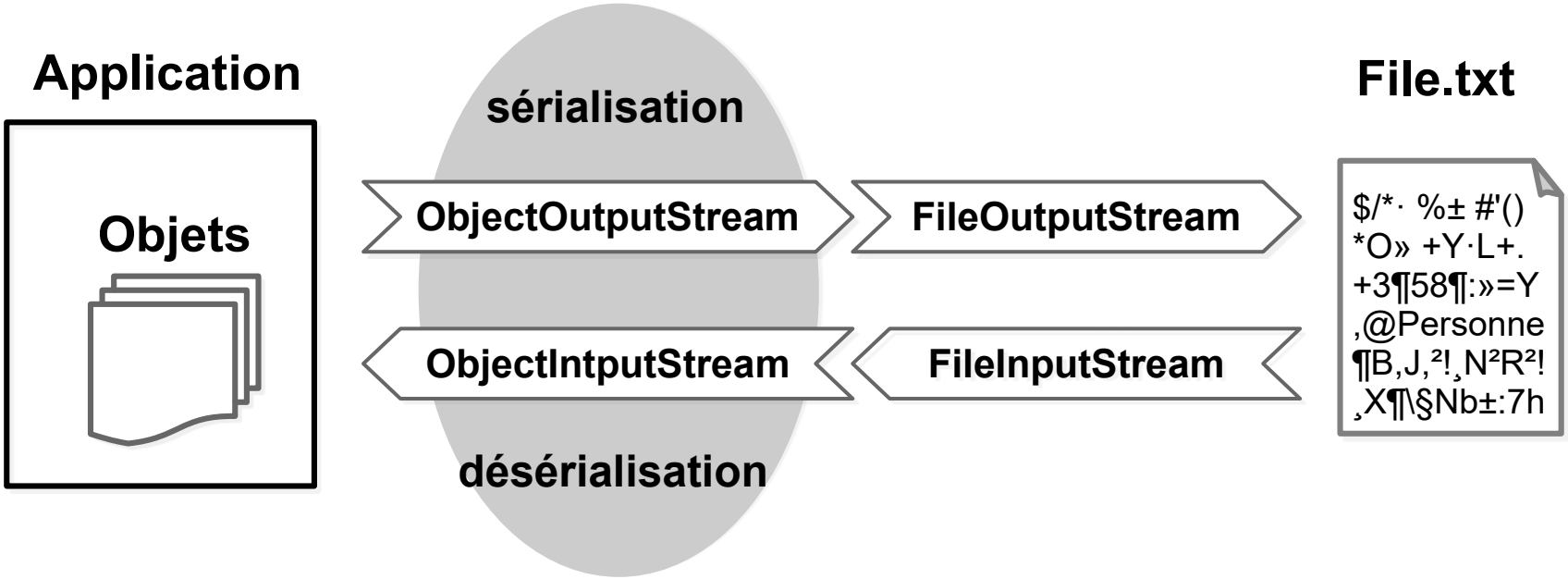
  `store``(OutputStream / Writer, String comments)`
  ↳ dans un fichier .properties

  `storeToXML``(OutputStream / Writer, String comments)`
  ↳ dans un fichier .xml

```
appProps.store(new FileWriter("app.properties"),"commentaires");
```

Sérialisation d'objets

- La sérialisation permet de sauvegarder l'état d'un objet dans un support persistant




```
class Test implements java.io.Serializable{…}
```

Sérialisation d'objets

- Pour définir un attribut non sérialisable, on utilise le mot clef **transient** :

```
public transient String password = "";
```

- Les attributs **static** ne sont pas sérialisés
- ObjectOutputStream pour la sérialisation :

```
void writeObject(Object o);
```

- ObjectInputStream pour la désérialisation :

```
Object readObject();
```

Sérialisation XML

- La sérialisation XML a été conçue pour être utilisée avec les JavaBeans

  Elle utilise l'introspection pour sérialiser/désérialiser les objets

- **XMLEncoder** → pour sérialiser un objet en XML

```
void writeObject(Object o);
```

- **XMLDecoder** → pour désérialiser un objet sérialisé avec XMLEncoder

```
Object readObject();
```

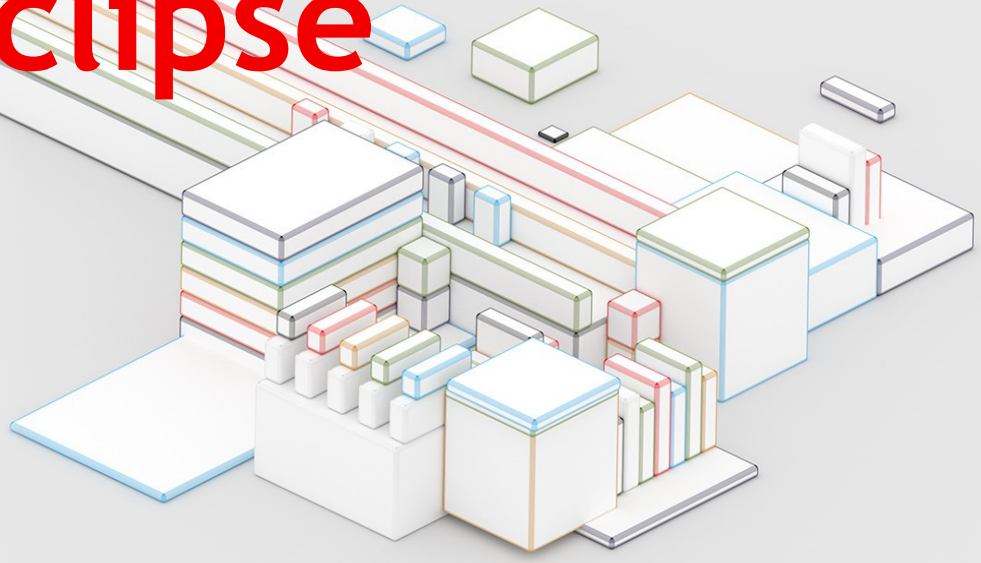
- **Empêcher la sérialisation d'un attribut**
  - ne pas coder d'accesseur/modifieur pour l'attribut

Sérialisation XML

- Utiliser la réflexion avec la classe **PropertyDescriptor** qui décrit une propriété d'un javabean

```
// Obtenir les PropertyDescriptors d'un javabean
BeanInfo info = Introspector.getBeanInfo(JavaBeans.class);
PropertyDescriptor[] propertyDescriptors =
info.getPropertyDescriptors();
for (PropertyDescriptor descriptor : propertyDescriptors) {
    if (descriptor.getName().equals("attribut")) {
        descriptor.setValue("transient", Boolean.TRUE);
    }
}
```

Les fonctions de base de Git dans Eclipse



La gestion de version

Version **C**ontrol **S**ystem (système de contrôle de version)

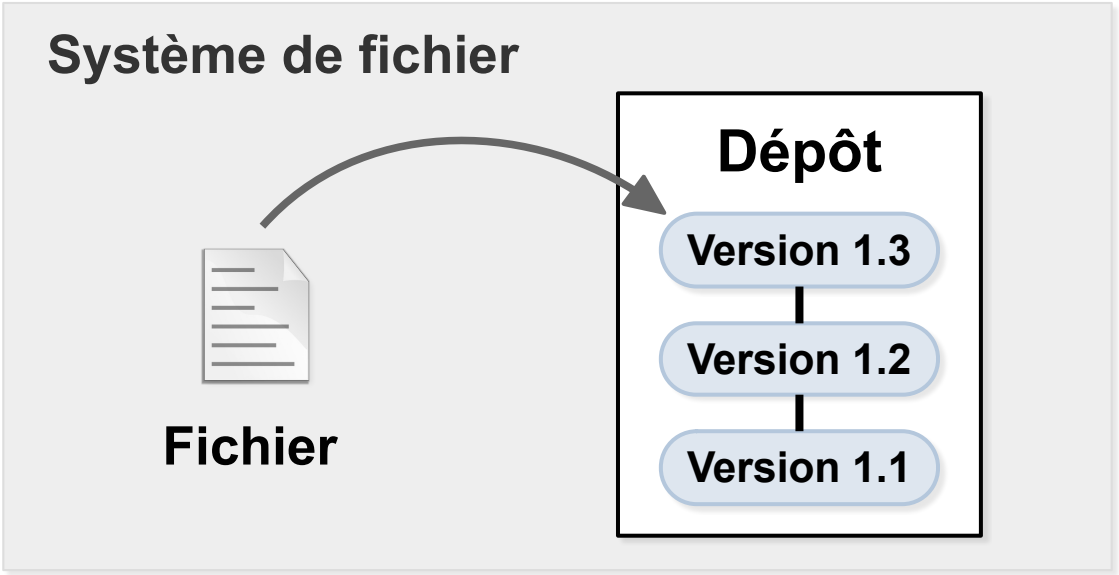
Source **C**ontent **M**anagement (gestion du contrôle de source)

- **Définition**

  Un système qui enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure d'un fichier à tout moment

- **Utilités**
  - Machine à remonter le temps
  - Documentation des versions détaillées et datées
  - Facilite la collaboration

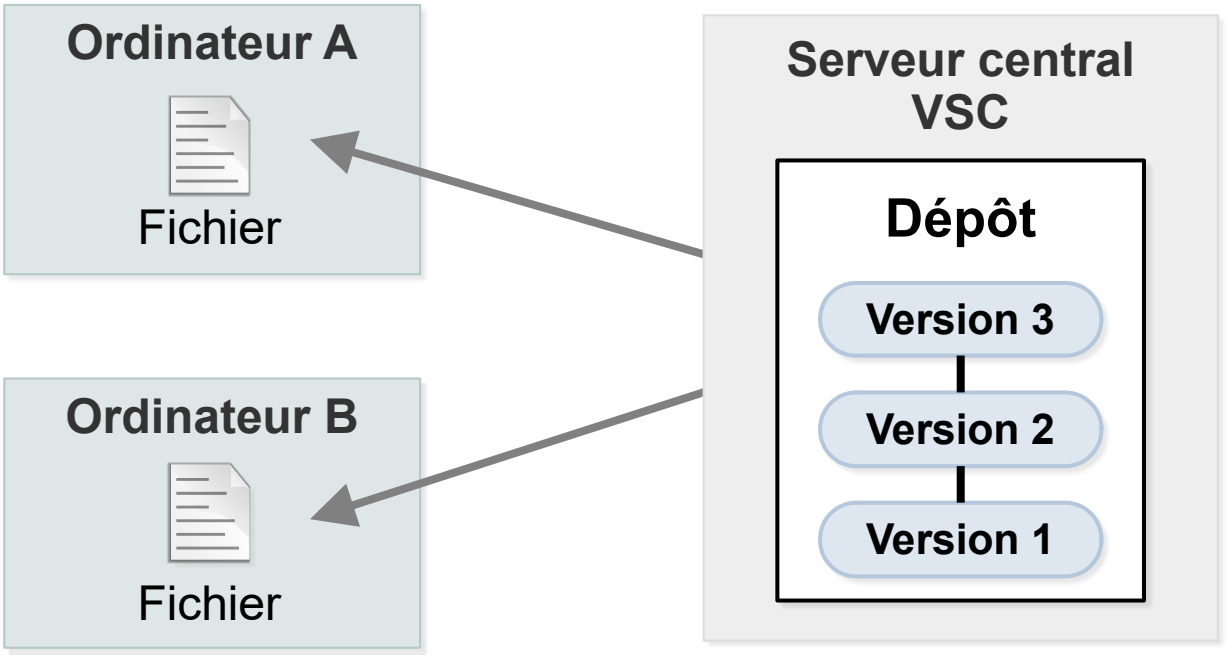
Contrôle de version local



- **Avantage**
  - gestion et utilisation simples
- **Inconvénients**
  - limiter au même système de fichier
  - est très sensible aux pannes

Contrôle de version centralisé

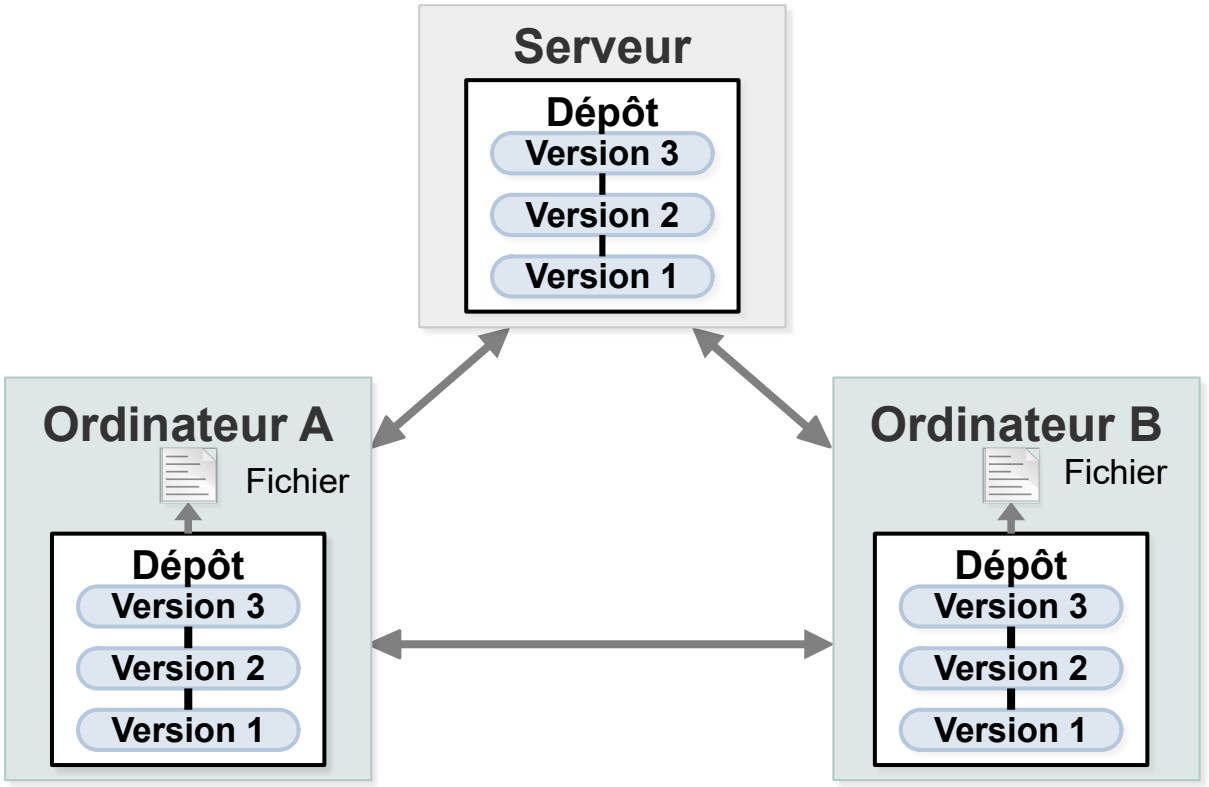
Les clients peuvent extraire les fichiers du dépôt central



Un serveur central contient tous les fichiers sous gestion de version

- **Avantages**
  - permet le travail collaboratif
  - chacun peut savoir qui fait quoi
  - un administrateur peut gérer des droits
- **Inconvénients**
  - s'il y a une coupure du serveur, plus personne ne peut collaborer

Contrôle de version distribué



Les clients n'extraient plus seulement la dernière version d'un projet,

mais ils dupliquent complètement le dépôt

- **Avantages**
  - plus de dépendance vis à vis du serveur
  - sécurité grâce à la redondance des dépôts

Git

1991 Noyau Linux gérer via des patchs et des archive

2002 Utilisation de **Bitkeeper**

2005 Bitkeeper devient payant
Linus Torvalds crée un successeur de Bitkeeper

Objectifs

- décentralisé
- libre
- conception simple
- développements non-linéaires (branches)
- vitesse
- compacité des données

} capacité à gérer des projets d'envergure tels que le noyau Linux

Fonctionnement

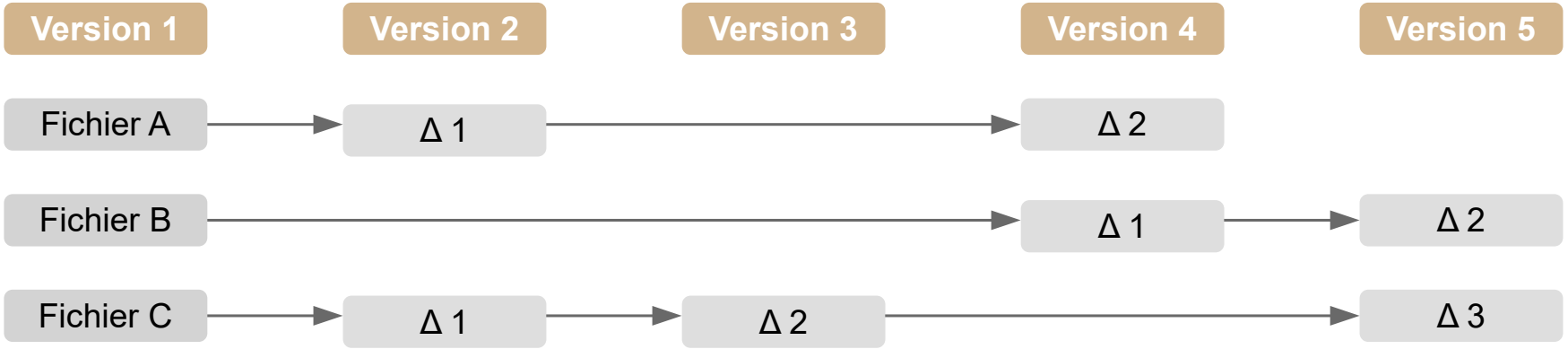
- **Travailler en local**
  - La plupart des opérations se déroulent localement
  - Pas de ralentissement dû à la latence du réseau

  exemple:
    - parcourir l'historique d'un fichier
    - travailler en mode «hors connexion»
      (dans le train, en dehors du réseau d'entreprise …)

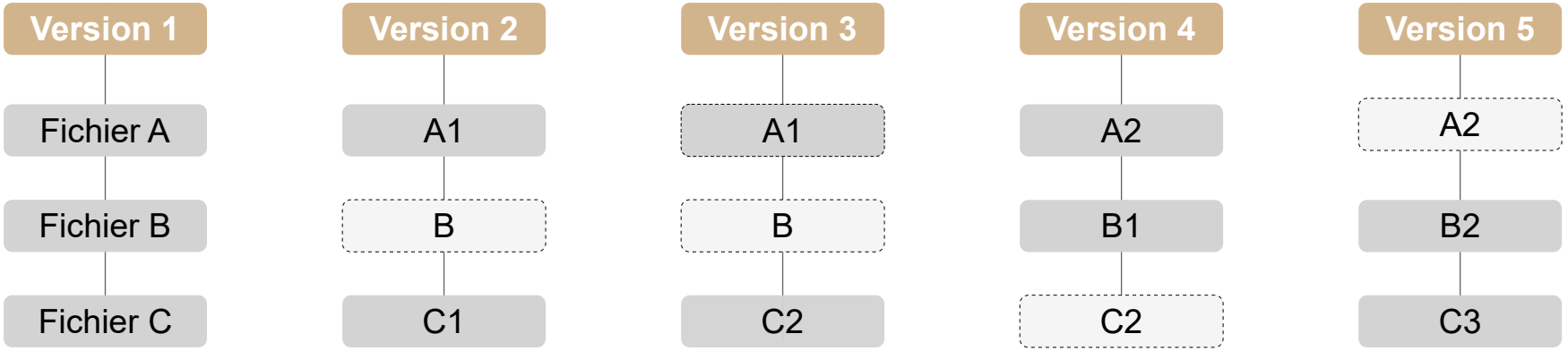
- **Mode de stockage**
  - Git gère les versions sous formes d'instantanés (snapshot) et non des différences
  - Un système de référence permet d'éviter de stocker plusieurs fois un fichier non-modifié

Fonctionnement

- **Stockage par différences**



- **Stockage par flux d'instantanés**



Fonctionnement

- **Signature SHA-1**

  Une chaîne de caractères composée de 40 caractères hexadécimaux

  Calculée en fonction du contenu du fichier ou de la structure du répertoire considéré

- **Exemple**

  Un fichier **git.txt** qui contient la chaîne de caractère :

  - **git** a pour signature :

    ↪ 46f1a0bd5592a2f9244ca321b129902a06b53e03

  - **GIT** a pour signature **:**

    ↪ 406750c04c6d320640c2a7c8d2c4b52d5b9965bf

Fonctionnement

- Tous les éléments que git manipulent (fichiers, dossiers, commits ...) sont rangés dans un dictionnaire **clé/valeur** dont la clé est la signature SHA-1

- Git ne va pas identifier un fichier en fonction de son nom, mais à partir de l'empreinte générée par son contenu

- Cela permet de détecter facilement la moindre modification dans un fichier

- Pas de risque de modification non-gérée d'un fichier, ni de corruption lors d'un transfert

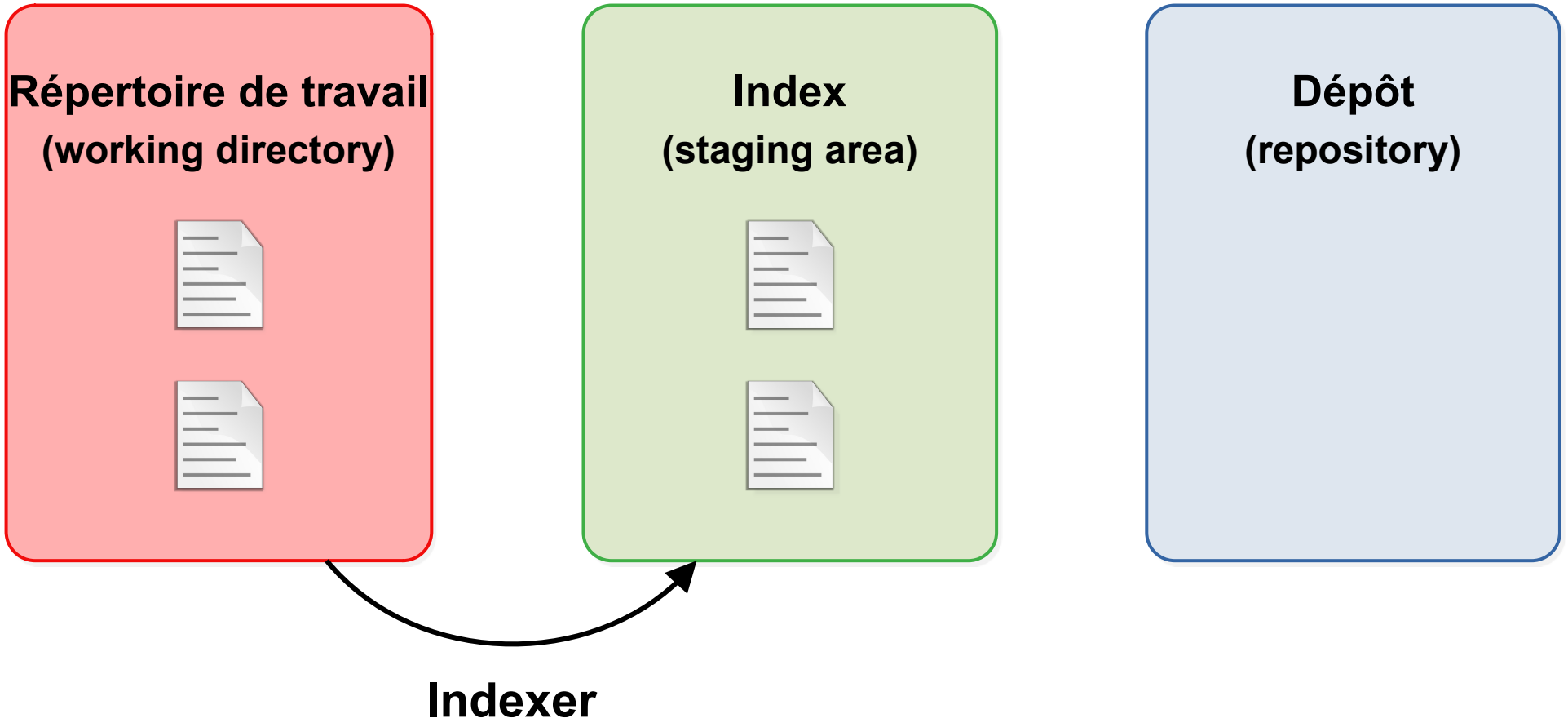
Répertoire de travail, index et dépôt

Répertoire de travail
(working directory)

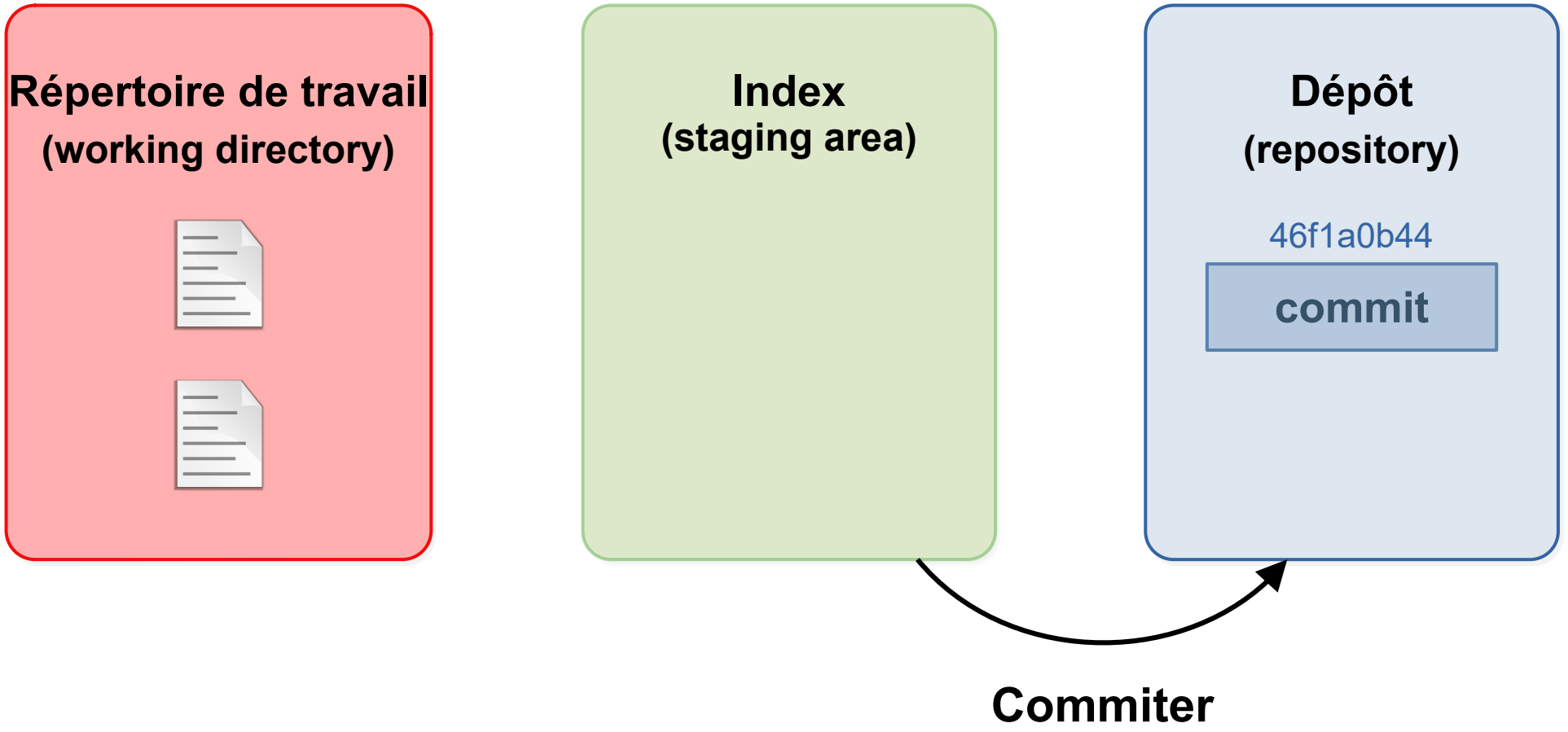
Index
(staging area)

Dépôt
(repository)

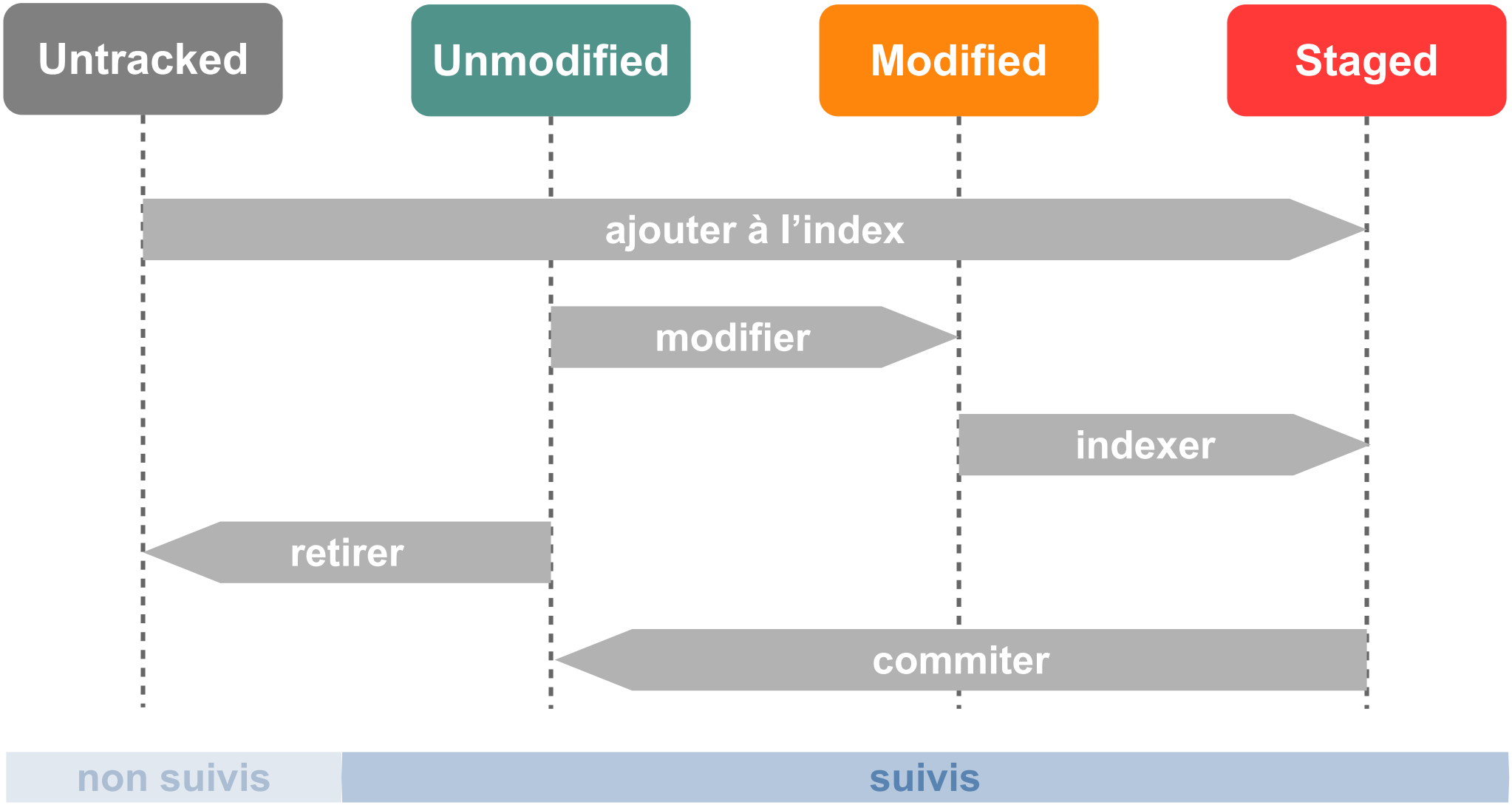
Répertoire de travail, index et dépôt



Répertoire de travail, index et dépôt



États des fichiers



Ignorer des fichiers

- On peut ignorer certains fichiers (fichiers de log, fichiers générés…) qui ne seront pas ajouter à l'index
- À la racine du projet, on ajoute le fichier **.gitignore**, qui contient la liste des règles des fichiers à ignorer
- **Syntaxe**
  - Une ligne par règle
  - Les lignes vides ne sont pas prises en compte
  - Les lignes commençant par **#** sont des commentaires
  - Une ligne contenant un chemin complet suivi du nom de fichier exclut uniquement le fichier ciblé
  - Un dossier finie par **/**
  - Un fichier à la racine du dépôt commence par **/**

Ignorer des fichiers

- * correspond à un ou plusieurs caractères inconnues

```
*.log  # ignore tous les fichiers de log
```

- ** indique une série de répertoires inclus

```
log/**/daily # ignore par exemple log/daily
                log/weeks/daily  log/april/weeks/daily
```

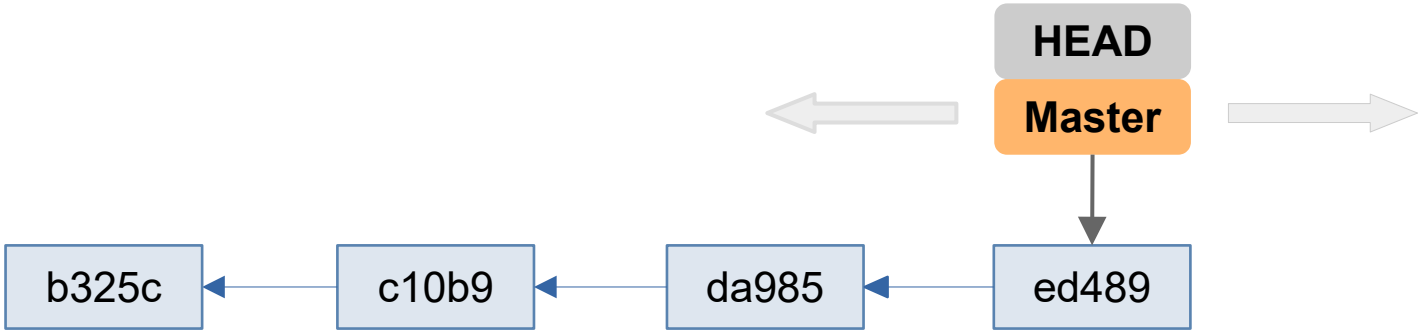
- ? correspond à un caractère inconnue
- **[abc]** correspond aux caractères a,b ou c
- **[0-9]** correspond à un caractère de l'intervalle 0 à 9
- **!** signifie que les fichiers ciblés ne sont pas ignorés

```
*.log          # ignore tous les fichiers de log
!errors.log    # le fichier errors.log n'est plus ignoré
```

- **Le fichier .gitignore doit être versionné**

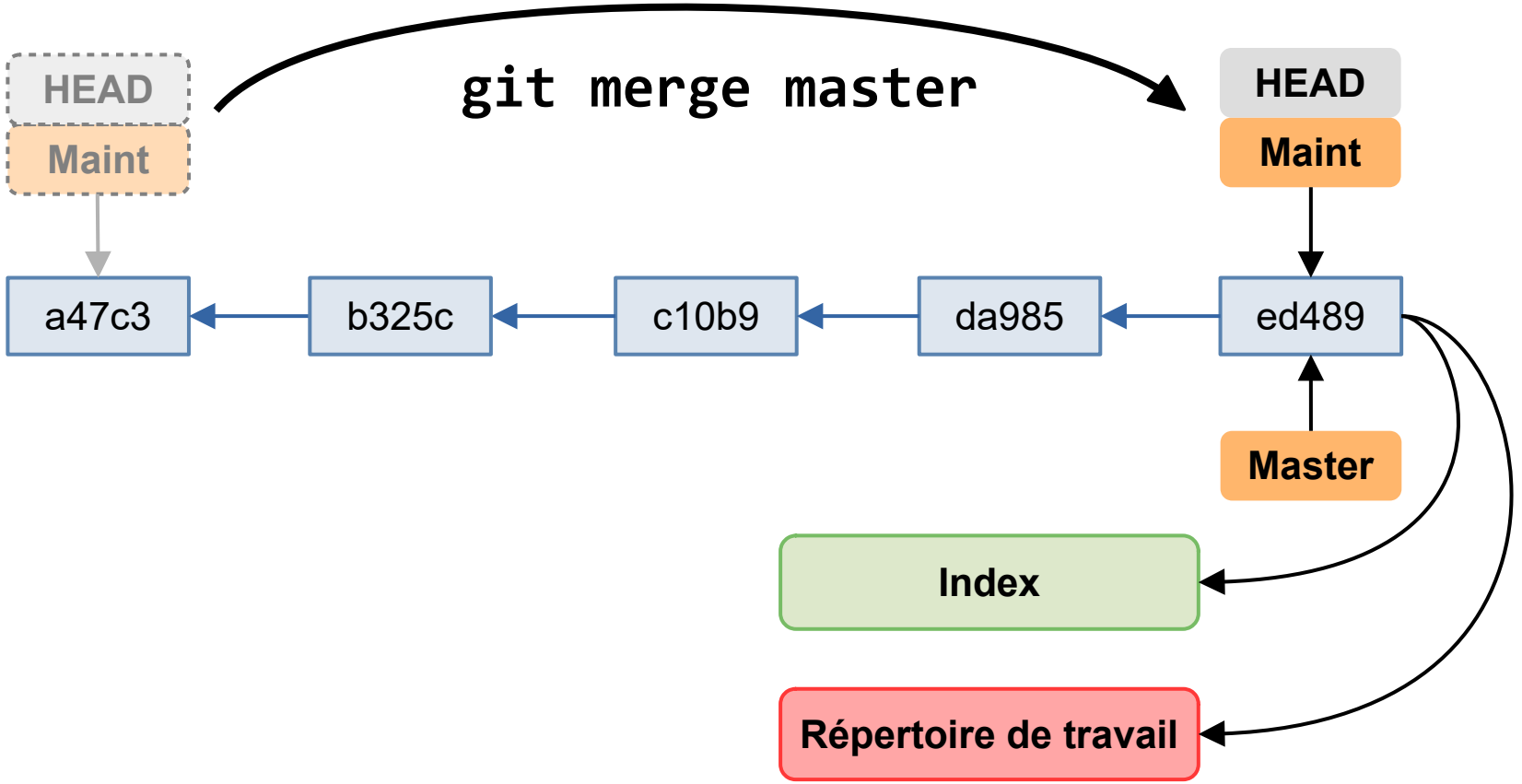
Les branches

- Une branche correspond à une version parallèle de celle en cours de développement
- Elles permettent de:
  - développer de nouvelle fonctionnalités
  - corriger des bugs
- Une branche est une référence sur un commit
  À chaque commit, la branche se déplace pour rester en permanence sur le dernier commit



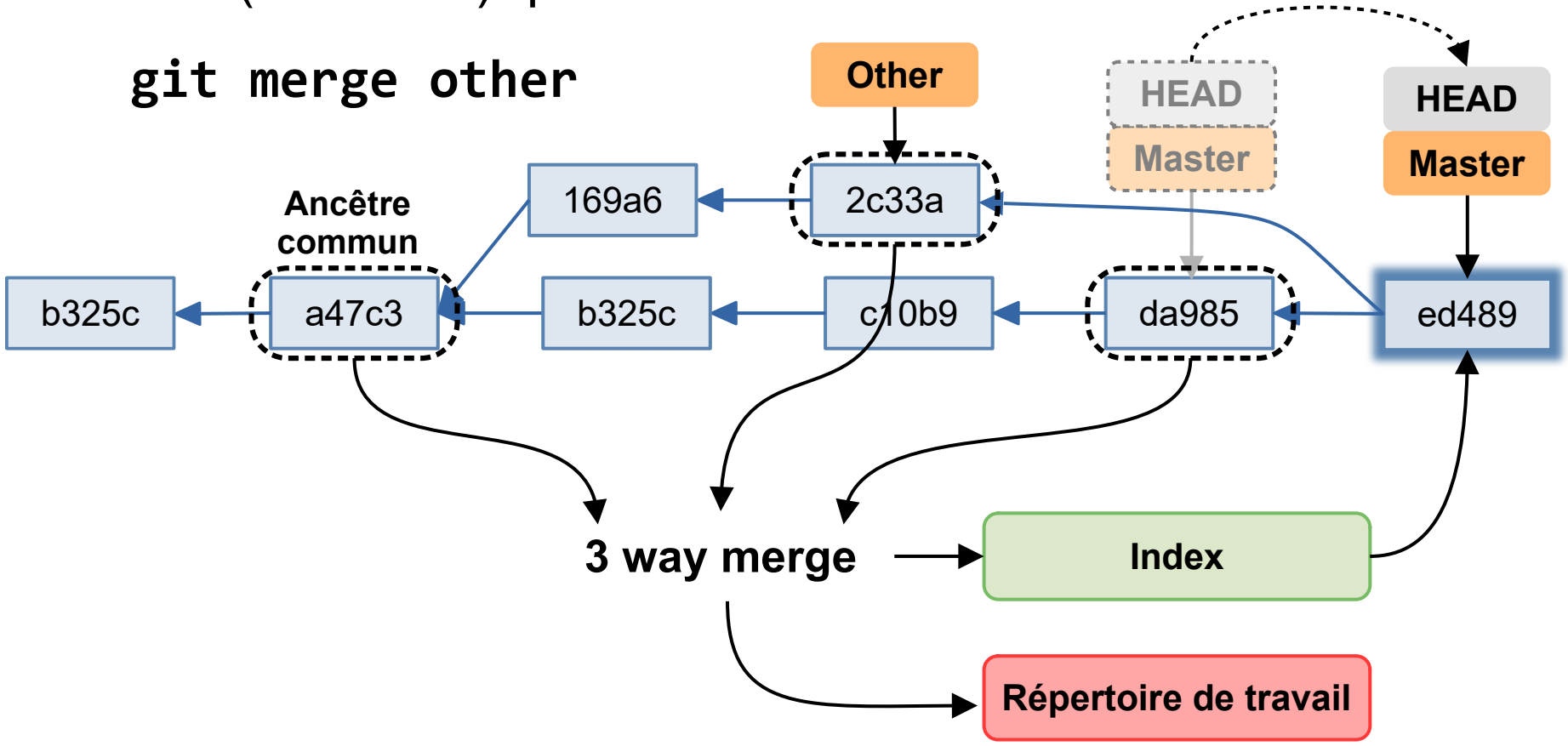
Fusion simple (Fast Forward Merge)

- La branche que l'on veut fusionner n'a pas divergé (pas de nouveau commit) pendant qu'on travaillait sur l'autre branche → Déplacement de la branche



Fusion à trois sources (Three way merging)

- Si les branches ont divergé (des nouveaux commits ont été fait sur chacune d'elles), Git va fusionner les deux dernier commit des branches et leur ancêtre commun et créer un nouveau commit (de fusion) qui va réunir les différentes modifications



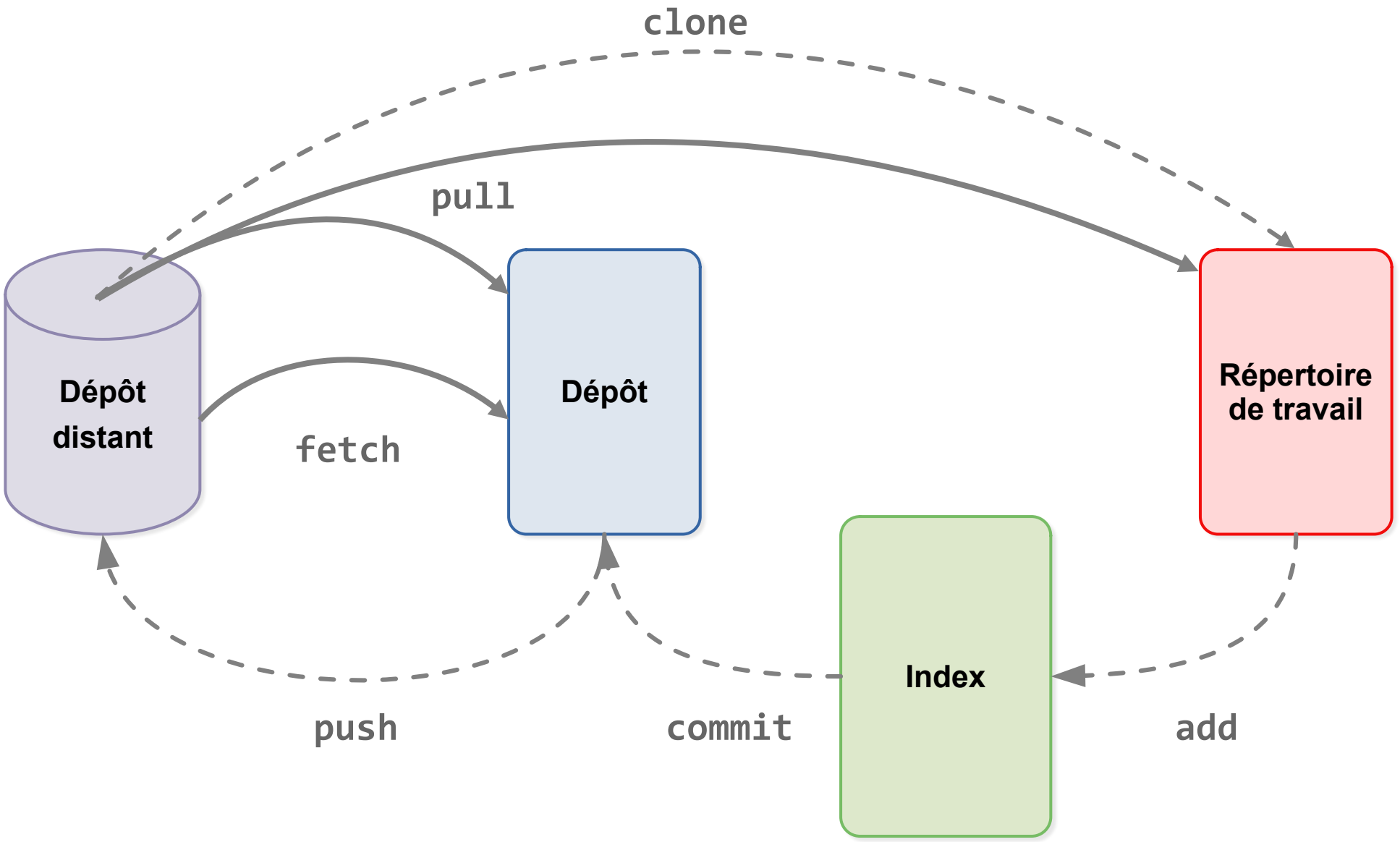
Conflit de fusion

- Si une même partie d'un même fichier a été modifié différemment dans les deux branches, Git ne peut effectuer la fusion automatiquement
- Le fichier à fusionner dans le répertoire de travail est annoté

```
<<<<<<< HEAD: index.html
contenu de la branche master
=======
contenu de la branche iss53
>>>>>>> iss53 : index.html
```

- Pour marquer le conflit comme résolu, on ajoute le fichier à l'index
- Quand tous les conflits sont résolus, on valide la fusion avec un commit

Dépôt distant



Configuration de l'utilisateur

Avec git, il faut au minimum configurer le nom et l'email de l'utilisateur

- Dans eclipse : window→préférence
  - filtre sur **git**
    Configuration → onglet : user setting

    Ajout du nom utilisateur

    ↳ Bouton Add Entry…

    Key : user.name
    Value : *Prénom Nom*

    Ajout de l'email de l'utilisateur

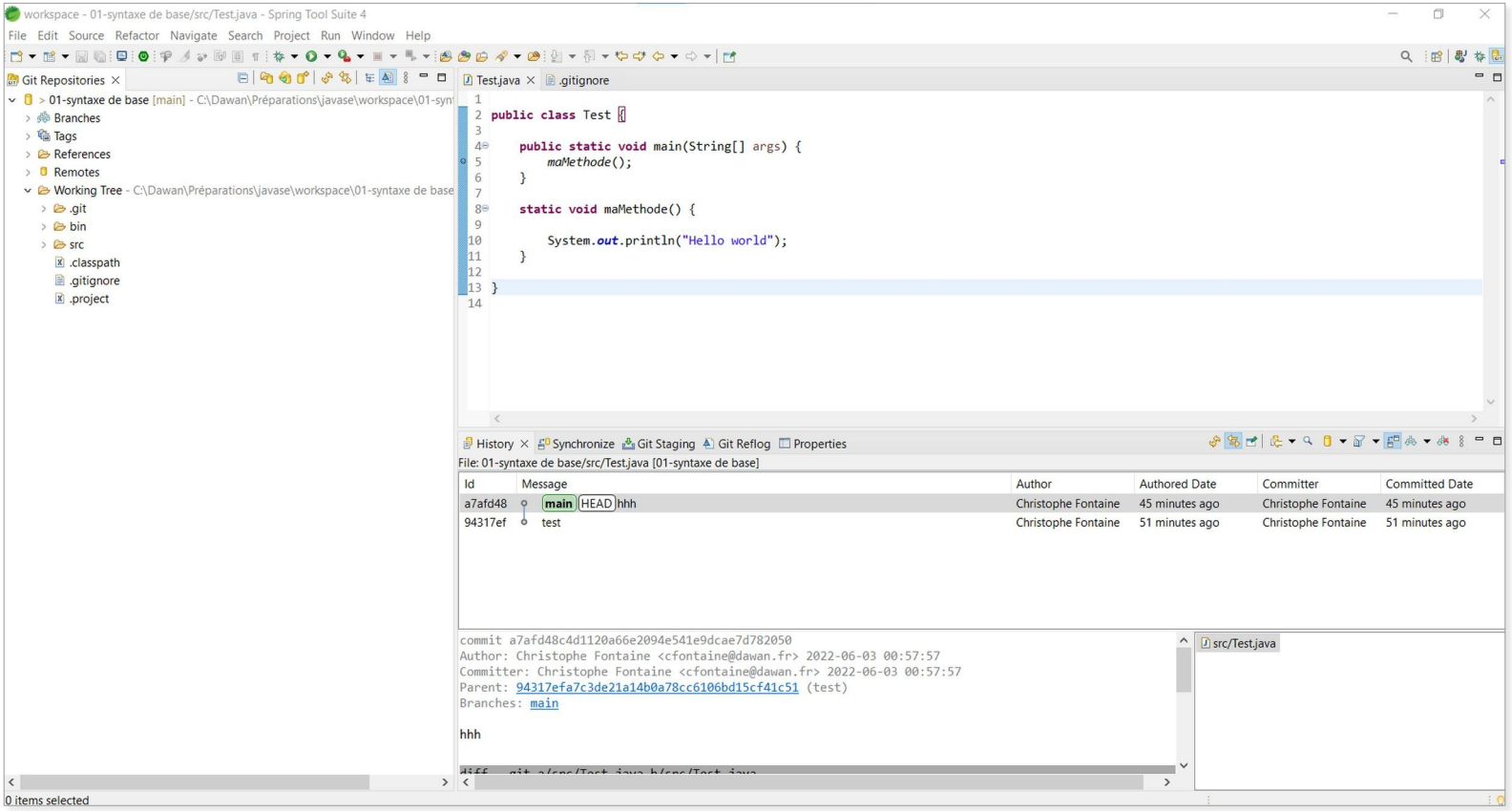
    ↳ *Bouton Add Entry…*

    Key : user.email
    ***Value** : votre email*

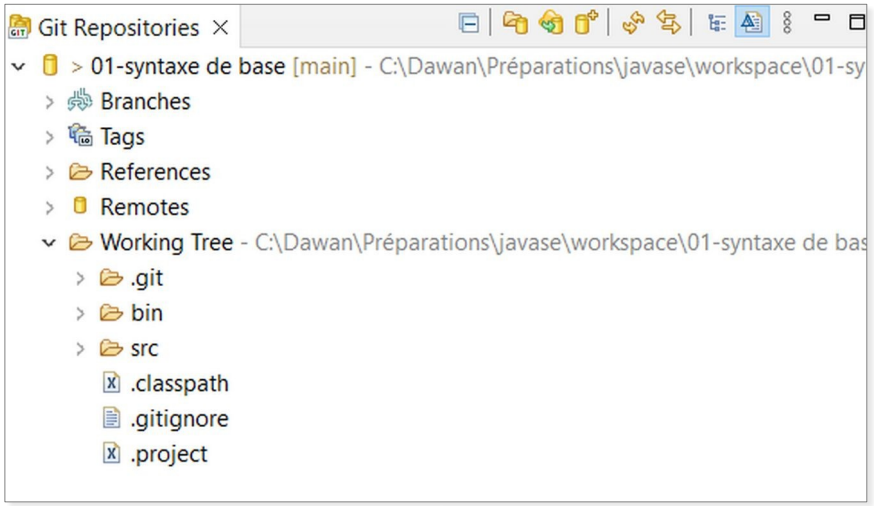
Perspective git

- Pour utiliser git dans eclipse, il faut ouvrir la perspective git

  window→perspective→open perspective→git



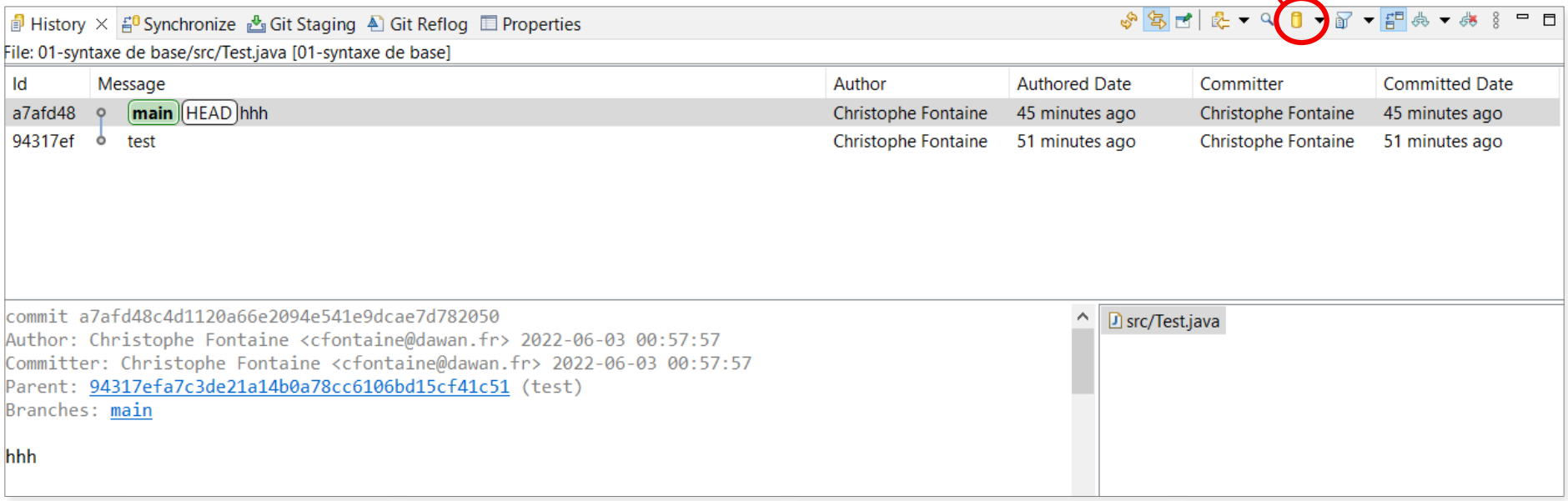
Git Repositories



- La fenêtre **Git Repositories** permet de :
  - Créer ou cloner un dépôt git
  - Gérer les branches > Branches
  - Gérer les étiquettes > Tags
  - Gérer les dépôts distants > Remotes
  - Visualiser le répertoire de travail Working Tree

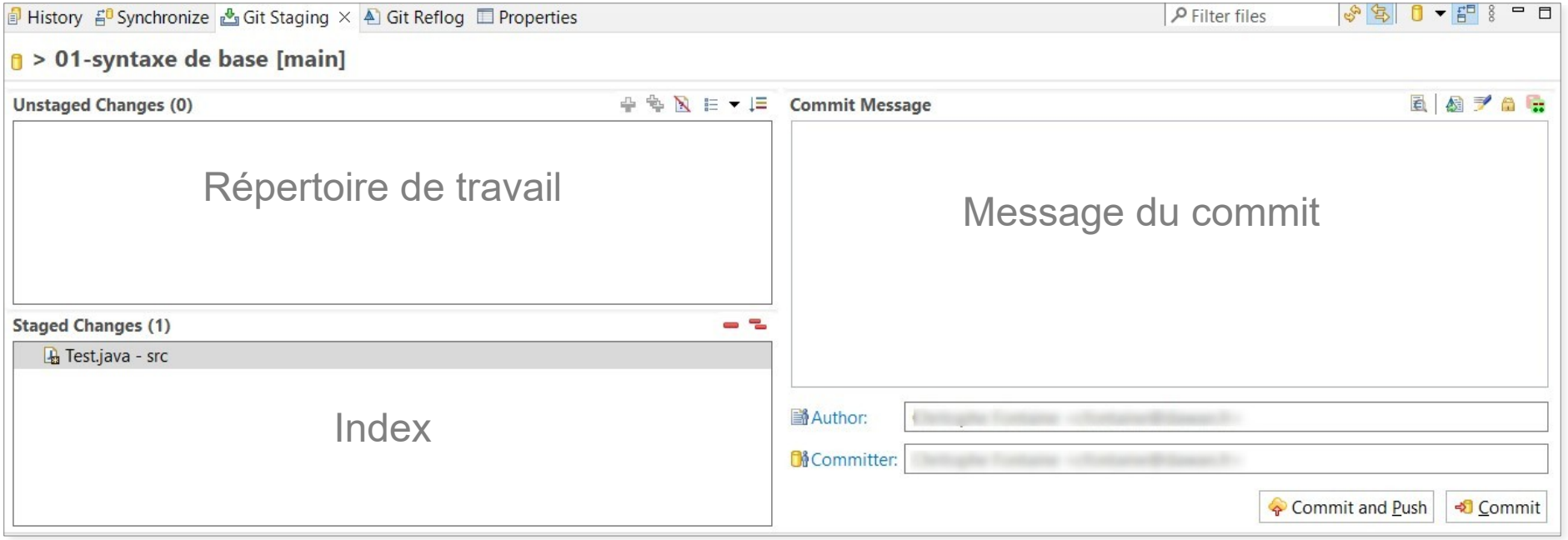
History

Pour sélectionner le repository



- La fenêtre **History** permet de visualiser l'historique des commits et la position des branches et des étiquettes sur les commits …

Git Staging

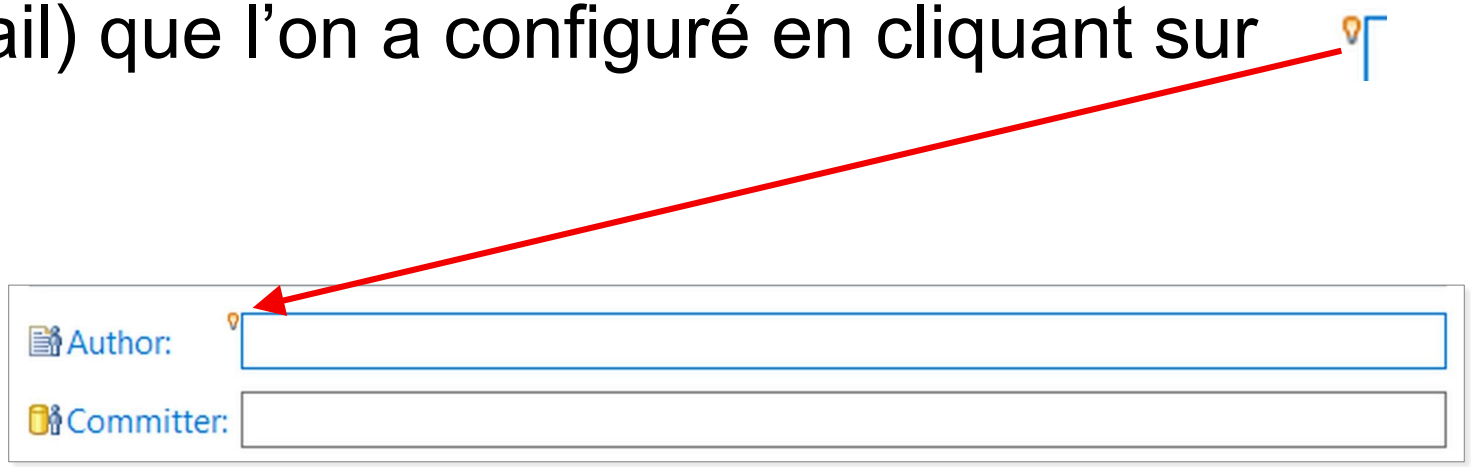


- La fenêtre **Git Staging** permet :
  - D'indexer les fichiers 
  - De faire un commit 

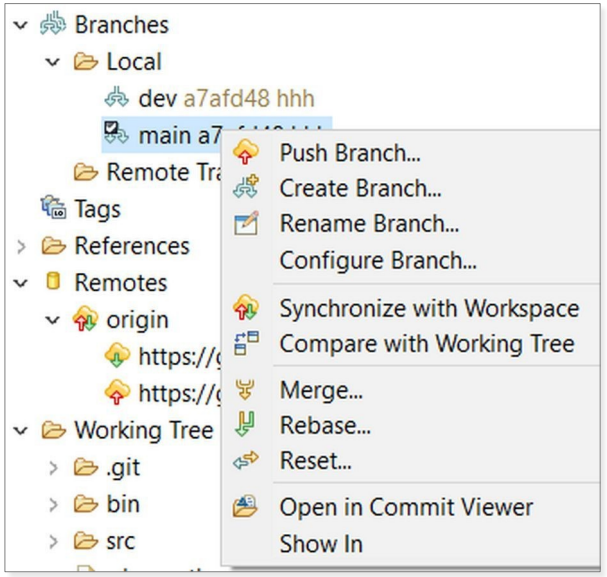
  - De faire un commit et d'envoyer le contenu du dépôt local vers le dépôt distant 

Git Staging

- On peut définir le contenu des champs **author** et **commiter** avec les valeurs de l'utilisateur git (nom + email) que l'on a configuré en cliquant sur



Gestion des branches

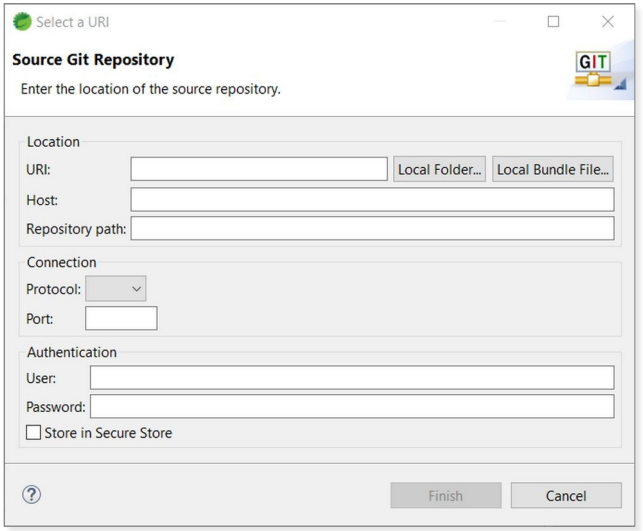


- dans fenêtre **Git Repositories** → **Branches** on peut :
  - Créer / renommer des branches
  - Fusionner les branches (Merge…)
  - Rebaser les branches (Rebase...)
  - Pousser une brache (Push Branch...)vers un dépôt distant

Gestion des dépôts distants

- dans fenêtre **Git Repositories** → **Remotes** on peut définir le dépôt distant
- Si on clone un dépôt, il sera configuré par contre, si on créer un nouveau dépôt git, on devra l'ajouter
- Remotes→Create Remote…
  - On définie le nom du dépôt distant (remote name) par défaut : origin
  - On choisit **configure push** ou configure fetch
  - URI → bouton Change...

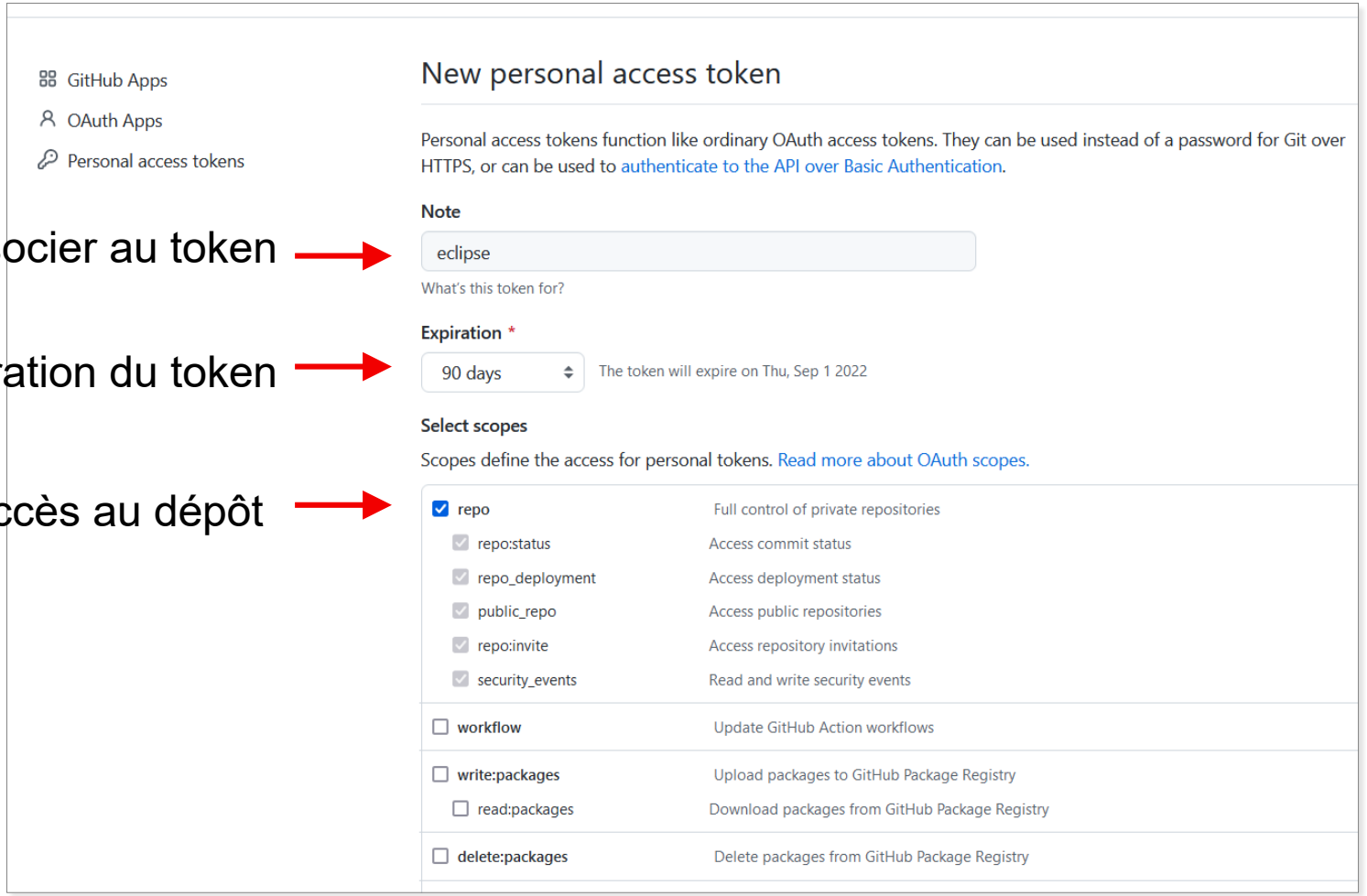
Gestion des dépôts distants



- Dans le champs URI, on place l'adresse du repository github, gitlab …
- Dans authentification, on définie le user et le mot de passe du compte gitlab, github ne gére plus les mots de passe, il faudra générer un token et le placer dans password
- Ensuite finish et save

Génération d'un token pour s'identifier avec github

- Settings→Developper settings→Personal access tokens→Generate new tokens

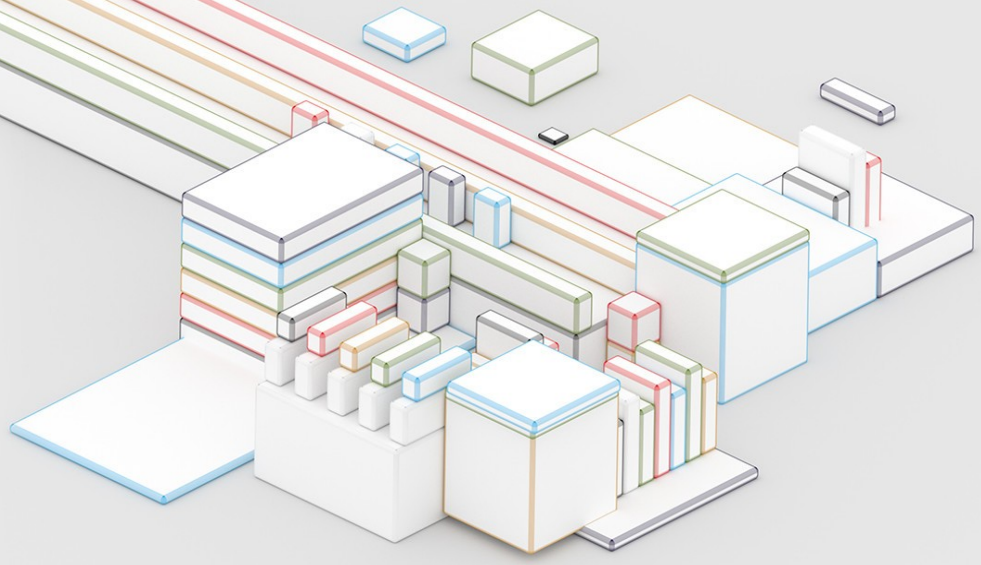


Nom associer au token →

Date d'expiration du token →

Droit d'accès au dépôt →

Annexe



Raccourcis clavier Eclipse

Raccourcis clavier	Description
Ctrl + Shift + L	Afficher la liste des raccourcis clavier
Ctrl + Shift + O	Organisation des imports
Ctrl + Shift + :	Commenter / Dé-commenter la sélection
Ctrl + Shift + F	Formater la sélection ou tout le fichier
Ctrl + Espace	Autocomplétion
Ctrl + Shift + X ou Y	Passer la sélection en majuscule ou minuscule
Ctrl + D	Effacer le ligne courante
Alt + ↑ ou ↓	Déplacer la ligne courante vers le haut ou vers le bas
Ctrl + Alt + ↑ ou ↓	Dupliquer la ligne courante vers le haut ou vers le bas
Alt + Shift + Z	Envelopper le code sélectionné dans un bloc
Shift + Alt + S	Ouvrir le menu source
Shift + Alt + T	Ouvrir le menu refactor
Shift + Alt + R	Renommer l'identifiant

Raccourcis clavier Eclipse

Raccourcis clavier	Description
`Ctrl + 1`	QuickFix (dépend de la position de la souris)
`Ctrl + S`	Sauvegarder le fichier en cours d'édition
`Ctrl + Shift + S`	Sauvegarder tous les fichiers
`Ctrl + Z`	Annuler
`Ctrl + Shift + N`	Créer une nouvelle ressource (projet, class …)
`Ctrl + Shift + B`	Ajouter un point d'arrêt à la ligne courante
`Ctrl + F11` 	Redémarrer le dernier programme exécuté
`F11` 	Démarrer en mode debug
En mode debug	Dupliquer la ligne courante vers le haut ou vers le bas
`F5` 	Exécution pas à pas : entrée dans la méthode
`F6` 	Exécution pas à pas : évaluer la méthode
`F8` 	Reprendre l'exécution
`Ctrl + F2` 	Arrêter l'exécution et le débogage

Hiérarchie des opérateurs

+

Opérateurs	Description	Associativité
() [] .		→
++ --	post-incrémentation	→
+ - ! ~ ++ --	pré-incrémentation, unaire	←
(cast) new		←
* / %		→
+ -	binaire	→
<< >> >>>		→
< <= > >= instanceof		→
== !=		→
&		→
^		→
|		→
&&		→
||		←
? :	ternaire	←
= += -= *= /= %= &= ^= |= <<= >>= >>>=	affectation	←

-