



COURS JAVA AVANCÉ

Mme Nouria Sana

STRUCTURE DU COURS

- Cours : 1H30 par semaine par Groupe
 - DS et Examen
- TP : - 1H30 de TP java et j2EE
 - Note de TP , Examen de TP et Mini projet
- Pre - requis : Programmation OO niveau 1



JAVA EST PARTOUT

- Voilà plus de 20 ans que le langage de programmation Java existe dans l'univers du développement informatique, et il n'est pas prêt à prendre sa retraite....
 - Les travaux sur Java débutent en 1991, sous le nom du projet Green
 - Les éléments de base de Oak ont été repris pour créer un nouveau langage appelé Java, qui fut présenté pour la première fois le 23 mai 1995, lors de la conférence SunWorld à San Francisco
- Java es le langage de programmation le plus populaire au monde
- Java SE 9 derniere version
 - 12 millions de développeur exécutant Java
 - 21 milliards de mchines virtuelles connectées au cloud
 - .



EVOLUTION DE JAVA

- Le langage Java est un langage de programmation généraliste et orienté objet, développé par *Sun Microsystems*, devenu maintenant un produit libre (au sens de la GPL).
- Java a été développé pour réaliser le développement de systèmes embarqués. Dès le début, des versions gratuites du compilateur et les spécifications du langage étaient disponibles sur Internet : D'où un développement rapide et consensuel du langage.
- Aujourd'hui les évolutions de Java sont gérées par JavaSoft, dépendant de SunSoft, avec le partenariat de nombreuses grandes entreprises (Borland / Inprise, IBM, ...).



Suite au rachat de Sun, Java est passé sous l'égide d'Oracle.

- De java À java 9
- Java 8 apporte la plus importante évolution du modèle programmation de Java depuis son lancement.
 - Comme nouveautés phares, on note les expressions lambdas, les méthodes par défaut, les interfaces fonctionnelles, Profiles ou encore Streams.
- Les développeurs **d'applications d'entreprise** peuvent choisir librement dans un écosystème de 30 implémentations compatibles des standards Java EE 6 et Java EE 7
- Java 9 (sept2017) inclut plus de 80 nouveauté décrite dans un ensemble de JEP (java Enhancement Proposal)



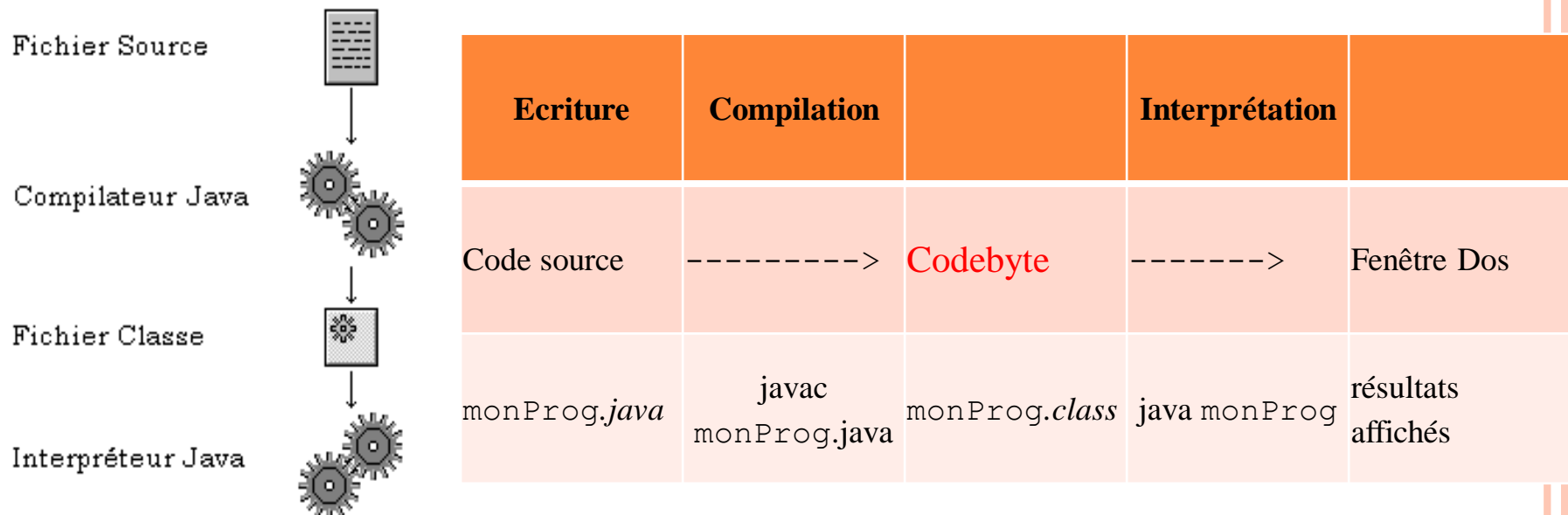
PRINCIPALES CARACTÉRISTIQUES, EN BREF

...

- Java est un langage complet, qui permet également d'écrire des programmes classiques (et pas seulement des applets dans des pages HTML), comme n'importe quel langage informatique, du type Pascal ou C++.
- Son atout primordial est son universalité, c'est-à-dire son indépendance des plates-formes matérielles. Cette indépendance est située à 2 niveaux :
 - le source est portable : il n'y a pas besoin de le recompiler quand on change de système.
 - le fichier binaire résultat de la compilation est lui aussi indépendant ! En effet cette compilation ne génère pas de code machine mais un pseudo-code universel, le **Java byte-code**, qui ressemble à du langage machine d'un processeur virtuel, en ce sens que ce code n'est intégré dans aucun processeur existant. Ce code compilé est inclus dans les fichiers **.class**.
 - Ceux-ci peuvent être distribués sur le réseau Internet, sous forme de petits programmes, les applets.
 - l'exécution seule nécessite un interpréteur Java, appelé *MVJ*="machine virtuelle Java". Seule cette MVJ dépend étroitement de la plate-forme, on pense même pour accélérer l'exécution d'intégrer la MVJ directement dans le système d'exploitation.
- Java est un langage-objet, apte à permettre le développement de grandes applications, intégrant des facilités pour réutiliser des modules déjà mis au point (appelées *paquetage* ou *packages*).
- Les applets Java, destinées à transiter sur le réseau Internet ont été conçues avec un maximum de sécurité (pas d'implémentation de manipulation de fichiers notamment)
- L'explosion de la bulle d'Internet et le fait que Java ne dépende pas d'une plateforme particulière ont contribué au succès du langage de programmation.



- Le **JDK** (Java Development Kit) est l'outil de base pour tout développement Java. Il est gratuit ! (il peut être chargé sur le site JavaSoft).
- Ce kit contient tout le nécessaire pour développer des applications ou des applets Java :
 - Le compilateur (en ligne de commandes),
 - une machine virtuelle,
 - un ensemble complet de classes de bases regroupées en packages.



POURQUOI UTILISER L'APPROCHE OBJET ?

La motivation essentielle de cette approche est d'augmenter les possibilités de réutilisation de ce que nous développons : Le monde de la programmation est encore proche du stade artisanal où chaque programmeur passe son temps à redévelopper ce que d'autres (souvent lui-même) ont déjà réalisé.

Les moyens de réutilisation offerts par l'approche objet sont multiples :

L'encapsulation des données et du code dans une même entité permet de garantir la cohérence des objets (qui se suffisent à eux-mêmes, contrairement aux procédures qui dépendent la plupart du temps de données externes à elles-mêmes). Cette cohérence est indispensable pour envisager de réutiliser un objet dans un autre contexte (pas suffisant, mais nécessaire)

La notion d'encapsulation par une interface permet de normaliser le fonctionnement des objets : il est possible de changer le fonctionnement interne d'un objet particulier, sans modifier la manière de l'utiliser (c'est à dire le reste du programme)

La notion d'héritage permet de réutiliser ce qui a déjà été défini lors de la conception d'un objet pour en créer de nouveaux.

La notion de polymorphisme, permet de manipuler de manière identique des objets ayant des comportements totalement différents (des comportements qui ne sont pas obligatoirement connus au moment où l'on définit ces manipulations).

De manière plus générale, l'approche objet permet de construire des programmes mieux conçus, car plus évolutifs et souvent plus sûrs.



UN PEU DE SYNTAXE EN JAVA

- **Les classes** : le mot clé `class`
- **TOUT est CLASSE**

[modificateurs] **class** *nomClasse* **extends**
nomSurClasse **implements** *interface*
 { [déclaration des attributs]
 [déclaration de méthodes]
 }

Et le test :

```
public class Test {  
    public static void main(String[] args) {  
        Cls cls = new Cls();  
        cls.foo();  
        cls.bar();  
        cls.baz();  
    }  
}
```



EXEMPLE

```
class Interrupteur {    // Interrupteur est le nom de la classe
boolean open ;        // Ceci est un attribut (une données
                       // membre de la classe)

void setState (boolean newstate) {    // Ceci est une méthode
    open = newstate ;
}

void printState() {                // Ceci est une deuxième méthode
    if (open) System.out.println( " L'interrupteur est ouvert " ) ;
    else System.out.println( " L'interrupteur est fermé " ) ;
}
}
```



LES MOTS CLÉS

- **Modificateur de classe : public** (visible en dehors du package) **ou private** (non visible en dehors du package)
 - Une seule classe publique par fichier .java
- **Modificateur de classe : final** :
 - *Cette classe ne peut être dérivée*
- **Le mot clé : final**
 - Le mot clé **final**, utilisé pour un attribut, permet de spécifier que cet attribut est une constante (sa valeur ne pourra jamais être modifiée). Un attribut **final** doit obligatoirement être initialisée lors de sa déclaration (puisque'on ne peut pas le modifier après cela) !
- **Le mot clé : static**
 - Le mot clé **static**, **utilisé pour un attribut**, permet d'indiquer que cet attribut est commun à tous les objets de la classe concernée : il s'agit d'un attribut de la classe elle-même, et si on modifie cet attribut pour un objet donné, il sera modifié pour tous les objets de la classe (puisque c'est le même).
 - Il est possible de définir **une méthode de type static**. De même que pour les attributs, cela signifie que les actions de cette méthode concernent la classe entière (pas un objet particulier). Pour cette raison, **une méthode static, ne peut accéder que les attributs static** de la classe : lorsque l'on utilise une telle méthode vouloir accéder à un attribut appartenant à un objet particulier n'a pas de sens.

CRÉATION D'UN OBJET

- *Déclare le nom d'un objet de la classe sans définir l'objet lui-même*

nomClasse nomObjet;

- *construit l'objet*

nomObjet = new constructeurClasse([liste de paramètres]);

- *permet de fusionner les 2 étapes*

nomClasse nomObjet = new constructeurClasse ([.....]);




```
class Personne {  
    private String nom ;  
    ...  
    public Personne(String nom) { this.nom = nom ; }  
    public String getNom() { return nom ; }  
}
```

```
public class Test  
{  
    public static void main(String [] a)  
{ Personne moi = new Personne("Toto") ;  
    System.out.println( moi.getNom() ) ;}  
}
```



LE MOT CLÉ : THIS

```
class Date {  
    int jour =1 ;  
    int mois =1 ;  
    int an =1990 ;  
  
    Date( ) { an = 2000 ; }           /* peut aussi s'écrire : this.an = 2000 */  
    Date( int an ) {  
        this.an = an ;  /* Le paramètre an cache l'attribut an */  
    }  
  
    Date( int jour, int mois, int an ) {  
        this.jour = jour ;  
        this.mois = mois ;  
        this(an) ;           /* appel du deuxième constructeur */  
    } }
```



APERÇU SUR LES ENTREE/SORTIE EN JAVA

○ Sortie écran

`System.out.println(« texte »+objet) ;`

○ Entrée clavier

- Définir une méthode qui permet de lire un caractère au clavier

- `System.in.read()` : permet uniquement de lire des variables de type bytes

```
import java.io.* ;  
  
public static String lireChaine () throws IOException {  
    byte[] lu = new byte[15]; System.in.read(lu,0,15); // prend en parametre  
    une variable de type byte String texte = new String (lu,0,lu.length); return  
    texte; }
```

- Utiliser la classe Scanner du package java.util.Scanner

- Cette classe permet de lire au clavier des variables de type différents (int, short, byte, , line).

`import java.util.Scanner ;`

`Scanner x = new Scanner(System.in);`

`x.nextInt();`

`x.nextLine(); }`



L'héritage et le Polymorphisme

- Il est possible de dériver une classe à partir d'une autre classe : la **classe fille** ainsi créée (ou **sous-classe**, ou **classe dérivée**) hérite alors des caractéristiques de sa **classe mère** (ou **super-classe**), tout en la spécialisant avec de nouvelles fonctionnalités
- le mot clé **extends**
 - il n'y a pas d'héritage multiple



EXAMPLE

```
class Felin {
    boolean afaim = true ;
    void parler() {}
    void appeler() {
        System.out.println("Le Félin est
        appelé") ;
        if (afaim) parler() ;
    }
}

final class Chat extends Felin {
    String race ;
    void parler() { System.out.println("miaou!"); }
}

final class Tigre extends Felin {
    void parler() { System.out.println("Groar!"); }
    void chasser() { ... }
}
```

```
Tigre tigre = new Tigre() ;
tigre.chasser() ;           //
OK
tigre.appeler() ;
```

// OK : méthode héritée de la classe Felin

```
Tigre tigre = new Tigre() ;
Felin felin ;
felin = tigre ;
```

// **OK** : c'est une conversion implicite, en fait
la classe Tigre héritant de Felin,
// tout Tigre est aussi un Felin

```
tigre = felin ;
```

// **Erreur** à la compilation : Tous les félins ne
sont pas des tigres
//=> une conversion explicite est obligatoire

```
tigre = (tigre)felin ;
```



```
Felin felin ;  
Tigre tigre = new Tigre() ;  
felin = tigre ;  
felin.parler() ;
```

```
// la référence felin est de type Felin, mais l'objet réellement référencé est de  
// type Tigre => c'est la méthode Tigre.parler() qui est réellement appelée
```

```
felin.chasser() ;
```

```
// Erreur à la compilation : La méthode n'existe pas dans la classe Felin  
// (bien que définie dans Tigre)
```

```
tigre = felin ;  
tigre = (tigre)felin ;  
tigre.parler();  
tigre.chasser();
```

```
// Erreur à la compilation : Conversion explicite nécessaire
```

```
// OK
```

```
// OK
```

```
// OK
```

```
Chat chat = new Chat() ;  
felin = chat ;  
lion = (lion)felin ;
```

```
// Erreur détectée lors de l'exécution : ClassException  
// (impossible à détecter par le compilateur)
```



Accès à la super-classe d'une classe

- Le mot clé : **super(...)**

```
class Mere {  
    int attribut ;  
    Mere() { attribut = 1 ; }  
    Mere(int attribut) ( this.attribut = attribut ; }  
    void print() { System.out.println("base" +  
        attribut) ; }  
}
```

```
class fille extends Mere {  
    Fille( int a ) { super(a) ;  
        }  
    void print() {  
        System.out.println("dérivée") ;  
        super.print() ;  
    }  
}
```

```
Fille f = new Fille(2) ;  
Mere m = f ;
```

```
m.print() ;
```

```
// Affiche :  
dérivée  
base 2
```



POLYMORPHISME

- Exemple Ville - Capitale



○ Le mot clé : abstract

```
abstract class Felin {  
    boolean afaim = true ;  
    abstract void parler() ;  
    void appeler() {  
        System.out.println("Le Félin est appelé") ;  
        if (afaim) parler() ;  
    }  
}  
class Tigre extends Felin {  
    void parler() { System.out.println("Groar!"); }  
}
```



SOMMAIRE

1. **LES PACKAGES ET L'ENCAPSULATION**
 2. **LES INTERFACES – GESTION DES EXCEPTIONS**
 3. **INTERFACES FONCTIONNELLES ET EXPRESSIONS LAMBDA**
 4. **LES CLASSES DE BASE ET COLLECTIONS**
 5. **LES STREAMS**
 6. **JAVAFX ET PROGRAMMATION ÉVÉNEMENTIELLE**
 7. **IMPLÉMENTATION DU MVC EN JAVA**
 8. **PROGRAMMATION CONCURRENTIELLE ET THREADS**
 8. **E/S : ACCÈS AUX BASES DE DONNÉES AVEC JAVA – LES FICHIERS (À FAIRE PAR L'ETUDIANT)**
- 