

Module de programmation avancée

Quelques mécanismes du langage java

Master 1 MIAGE – Année 2020-2021

Philippe Lahire

Sommaire

- Rappel sur la gestion des flots de données
- Classes Internes
- Réflexivité et Introspection
- Chargement dynamique
- Annotations
- Persistance

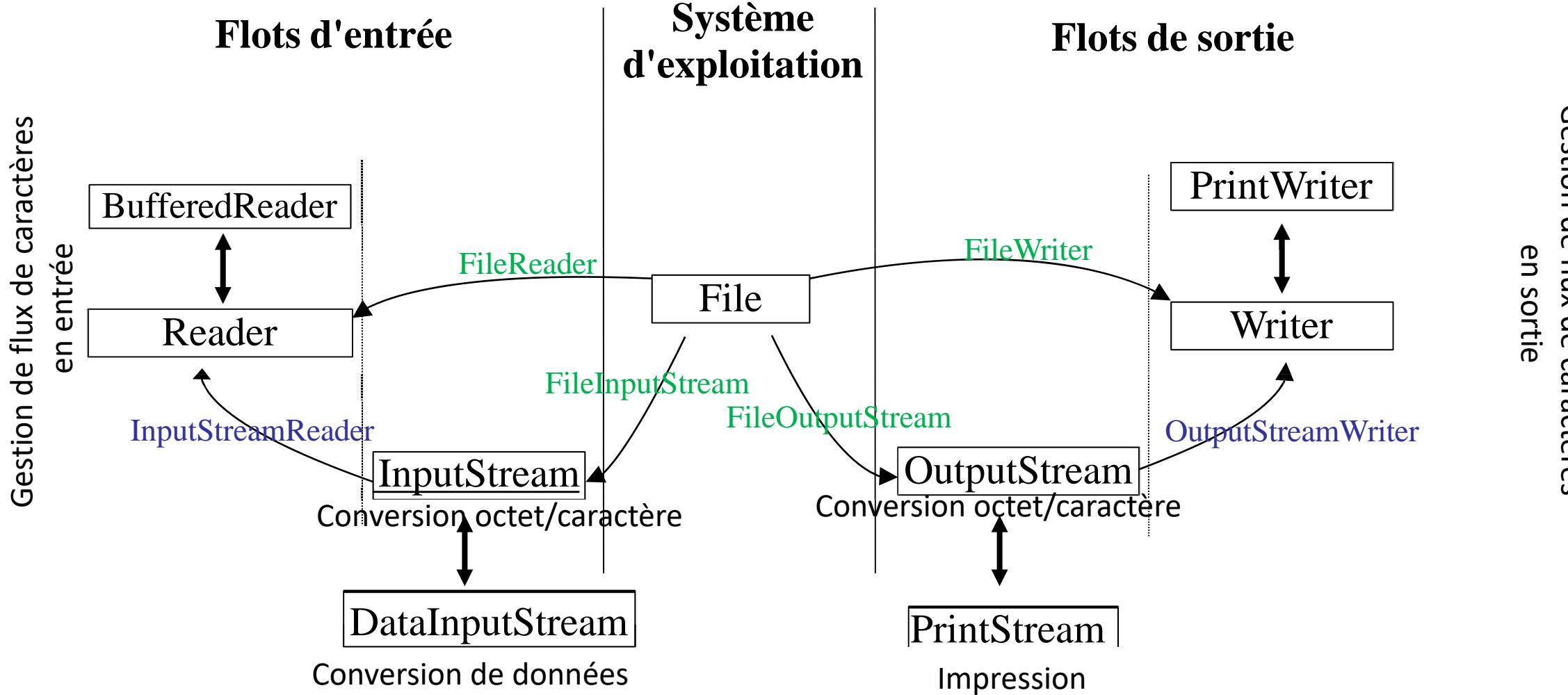
Agenda

- 4 séances de cours 2 heures
- 5 séances de TP/TD de 2 heures
- Une note de TP/TD
 - Réponses aux questions
 - Qualité des exercices
- Une note d'examen

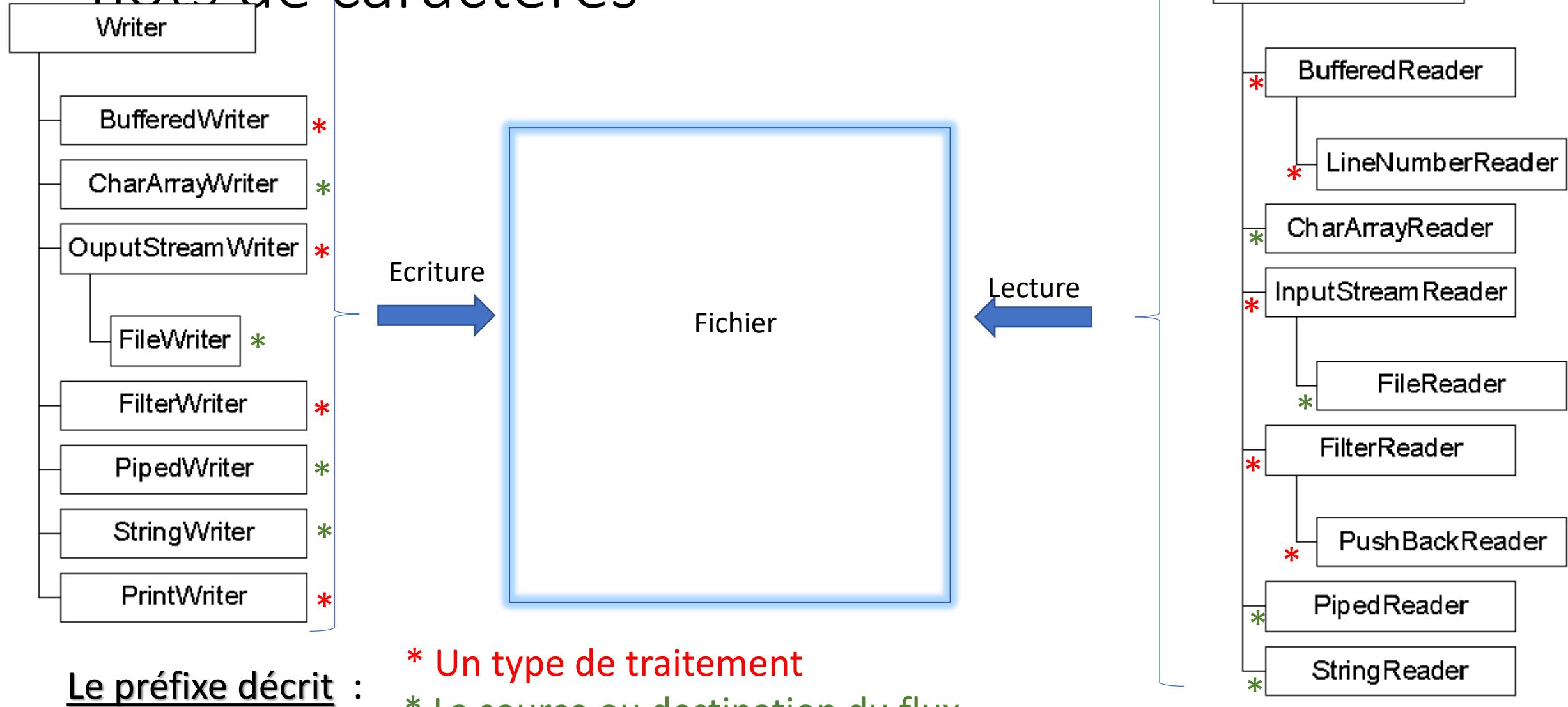
Flots et entrées-sorties

Quelques rappels

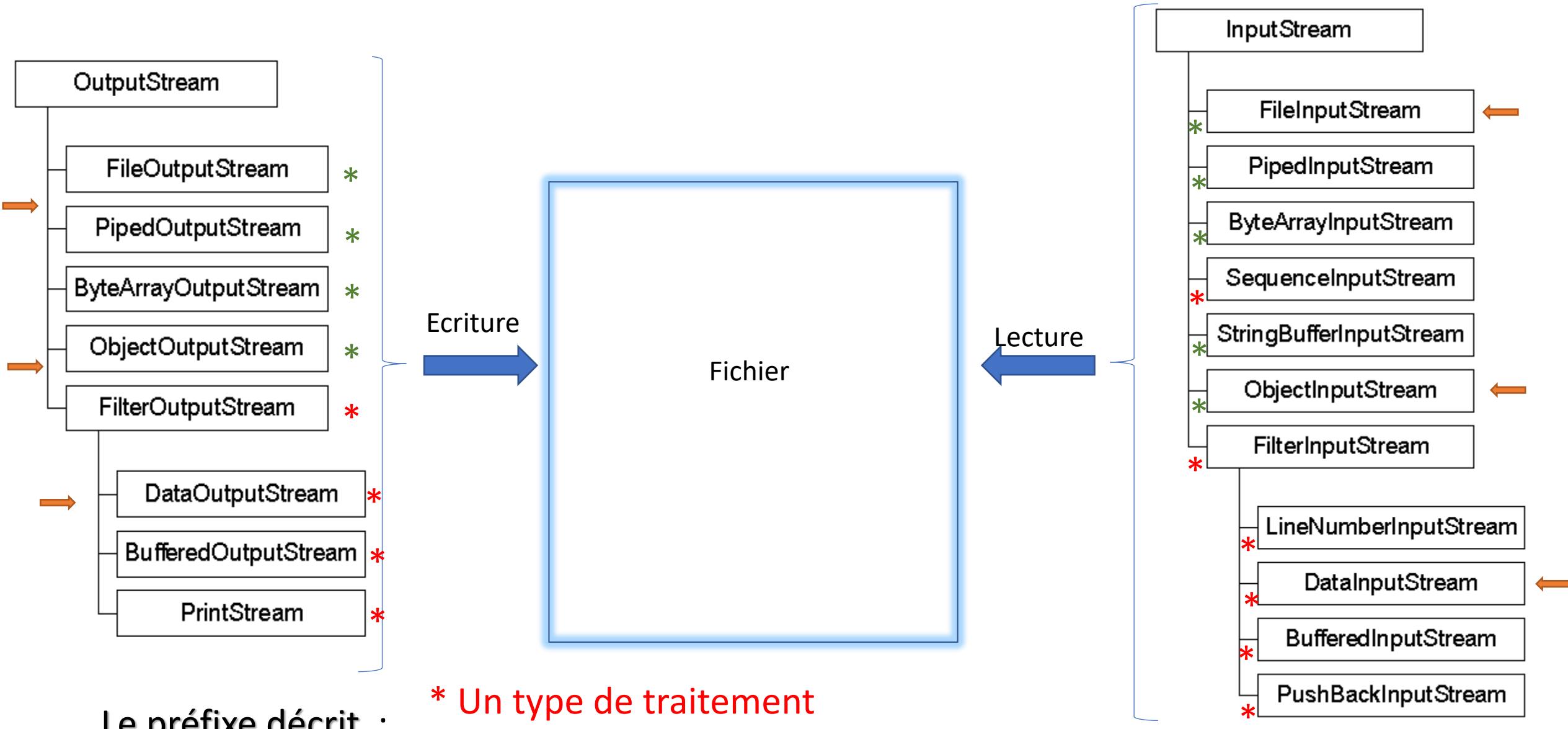
Survol du paquetage java.io (1)



Survol du paquetage java.io (2) : gestion des flots de caractères



Survol du paquetage java.io (3) : gestion des flots d'octets



Gestion de fichiers (classe Java.io.File)

- Accès de niveau système

Catégories de primitives	Méthodes
Constructeurs	File(String), etc.
Opérations de base répertoires	Open(), delete(), getName(), mkdir(), getParent()
Permissions	CanRead(), canWrite(), canExecute(), setWritable()
Fichiers d'un répertoire	ListFiles(),
Identification	getAbsolutePath(), getPath(), etc.
Type de fichier	IsDirectory(), isFile()

Ecriture sur le flot de sortie Standard (Classe System)

- Un attribut statique out (classe **System**) est le flot de sortie standard
 - *par défaut, il est dirigé vers la console*
 - *Java.io.printStream out*
- **Exemple d'utilisation**
 - `System.out.println("Bonjour");`
 - `System.out.print("Bonjour");`
- Quelques méthodes pour écrire (différents types de données)
 - *void print(String)*
 - *void println(String)*
- Surcharge de ces méthodes pour tous les types primitifs et le type **Object**
 - *void print(int), void print(double), void print(Object), ...*
 - avec un **Object**, la méthode **toString** est invoquée pour le transformer en **String**.

Ecriture sur le flot Standard d'erreur (Classe System)

- Un attribut statique err (classe **System**) est le flot standard d'erreur *par défaut, il est dirigé vers la console*
- Se comporte comme out
- Utilisation :
 - Il doit être utilisé pour afficher des messages d'erreurs
 - Souvent les deux flots sont dirigés vers l'écran
- Exemple :

```
public class MonExemple {  
    static public void main(String[] args) {  
        System.out.println("Ceci est un message normal");  
        System.err.println("Ceci est un message d'erreur");  
    }  
}
```

Ecriture dans un fichier (Classe java.io.FileWriter)

- Initialisation

```
FileWriter fw = new FileWriter ("monfichier.txt");
```

- Utilisation (basique):

- Permet d'écrire un caractère (octet?) après l'autre
- Préférer **Printwriter**

- Exemple :

```
public class MonExemple {  
    static public void main(String[] args) {  
        FileWriter fw = new FileWriter ("monfichier.txt");  
        fw.write("petite chaine");  
        fw.close();  
    }  
}
```

Ne pas oublier d'attraper les exceptions au cas où il y aurait une erreur (par exemple à l'ouverture du fichier)

Ecriture dans un fichier (Classe java.io.PrintWriter)

- Initialisation

```
FileWriter fw = new FileWriter ("monfichier.txt");
```

```
PrintWriter pw = new PrintWriter(fw);
```

- Utilisation :

- Possibilité de formatage (cf. printf en « C ») et d'écriture retardée
- Plusieurs méthodes « print » (cf. classe java.io.printStream)

- Exemple :

```
public class MonExemple {  
    static public void main(String[] args) {  
        PrintWriter pw = new PrintWriter( new FileWriter ("monfichier.txt"));  
        pw.println("petite chaine");  
        pw.close();  
    }  
}
```

Ne pas oublier d'attraper les exceptions au cas où il y aurait une erreur (par exemple à l'ouverture du fichier)

Ecriture dans un fichier ou sur le standard de sortie (synthèse)

```
import java.io.FileWriter;
import java.io.PrintWriter;
public class Ecriture {
    static public void main(String[] args) {
        try {
            FileWriter fw = new FileWriter("fic.txt");
            PrintWriter pw = new PrintWriter(fw);
            for (int i=0;i<args.length; i++) {
                pw.println(i+" "+args[i]);
                System.out.println(args[i]);
            }
            pw.close();
        }
        catch(Exception ex) { // obligatoire !
            System.err.println("Erreur sur le fichier");
        }
    }
}
```

Lecture sur le flot Standard d'entrée (Classe System)

- Un attribut statique in (classe **System**) est le flot d'entrée standard
par défaut, il est dirigé vers le clavier
- **Java.io.InputStream in**
- Exemple d'utilisation (basique)
`Int i = System.out.read(byte[] b);`
La méthode **read** remplit le tableau *b* avec les octets lus et renvoie le nombre d'octets lu.
- Existence de classes offrant des fonctionnalités plus évoluées :
 - **Java.io.BufferedReader**
 - **Java.util.Scanner**

Programmation Avancée

Classes internes ou classes imbriquées

Philippe Lahiré

Premiers éléments

Plusieurs catégories :

- Classe interne non statique
- Classe interne statique
- Classe anonyme

Plusieurs localisations :

- Dans une interface
- Directement dans une classe
- A l'intérieur d'une méthode
- A l'intérieur d'un bloc (boucle, schéma conditionnel)

Quelles utilisations ?

- Permet d'exprimer un couplage fort mais de séparer les fonctionnalités
 - Accès aux variables de la classe englobante
 - Simplifie le partage en limitant le nombre de fichiers
- Permet de ne pas exposer des classes qui n'existent que pour des raisons d'implémentation d'une classe:
Utilisation des mots clés *private* ou *public* selon le cas
- Cas des classes locales et anonymes :
 - Elles permettent d'implémenter facilement une version spécialisée
 - Redéfinition d'une méthode

Classe interne: principaux éléments

- Son accessibilité respecte les règles de visibilité du langage (private, public...)
- Pas membres statique autorisé
- Accès à l'instance de la classe englobante *nom-classe-englobante.this*
- Accès à son instance courante: *this* ou *nom-classe-interne.this*
- Accès à tous les membres de la classe englobante (résoudre ambiguïtés : *this*)
- Création depuis la classe englobante ou depuis une autre classe (syntaxe spécifique)

Une classe interne non statique

```
package diversExemples;

public class exemple1 {
    private int attribut1<----->
    protected static int attribut2;
    exemple1(){
        new exempleClasseInterne();
        System.out.println(attribut1);
        System.out.println(attribut2);
    }
    class exempleClasseInterne {
        boolean b;
        exempleClasseInterne (){
            attribut1++; attribut2++;
            b = true;
        }
    }
}
```

- Accès aux membres de la classe englobante
Y compris pour *private* ou *protected* (Membres statiques ou non)
- En cas d'homonymes :
 - `this.attribut` ou `exempleClasseInterne.this.attribut` : attribut de la classe interne (**à partir de la classe interne**)
 - `Exemple1.this.attribut` : attribut de la classe englobante

```
package diversExemples;

public class testExemple1 {
    public static void main(String[] args) {
        exemple1 e1 = new exemple1();
        exemple1.attribut2++;
        System.out.println(exemple1.attribut2);
        { //e1.attribut1++;
        //System.out.println(e1.attribut1);
        }
    }
}
```

`exemple1$exempleClasseInterne.class`
 `exemple1.class`

Classe interne ou classe membre non statique

- On peut dans une autre classe, instancier un objet de classe interne, à travers une instance de la classe englobante

```
Class ExempleUtilisation
    public void method1 () {
        exemple1 e1
        ....
        exempleClasseInterne eci = new exemple1.exempleClasseInterne();
        exempleClasseInterne eci = e1.new exempleClasseInterne();
        ...
    }
```

- Une classe interne peut aussi être créée depuis la classe englobante
 - Soit sans création de champ: pas d'accès aux membres de la classe interne
 - Soit avec création de champ: utilisation classique (voir ci-dessus eci)

Classe interne ou classe membre non statique

La différence notable entre ces les classes membres statiques ou pas se situe dans l'accès à l'instance de la classe englobante correspondante

Un objet de classe interne non statique comporte une référence à l'objet de classe externe qui l'a créé. Ce n'est pas le cas pour une classe interne statique. L'accès a lieu à travers le mot clef **this** qualifié par le nom de la classe englobante : « exemple1.**this** » .

Notre classe `exempleClasseInterne` possède une méthode

```
public void method1 (int attribut1, int t) {
        exemple1.this.attribut1 = attribut1;
        attribut2 = t;
}
```

Une classe interne peut accéder à tous les membres (champs et méthodes) de l'objet englobant (variables de classe ou d'instances): ici, suppose que « attribut2 » n'est pas statique



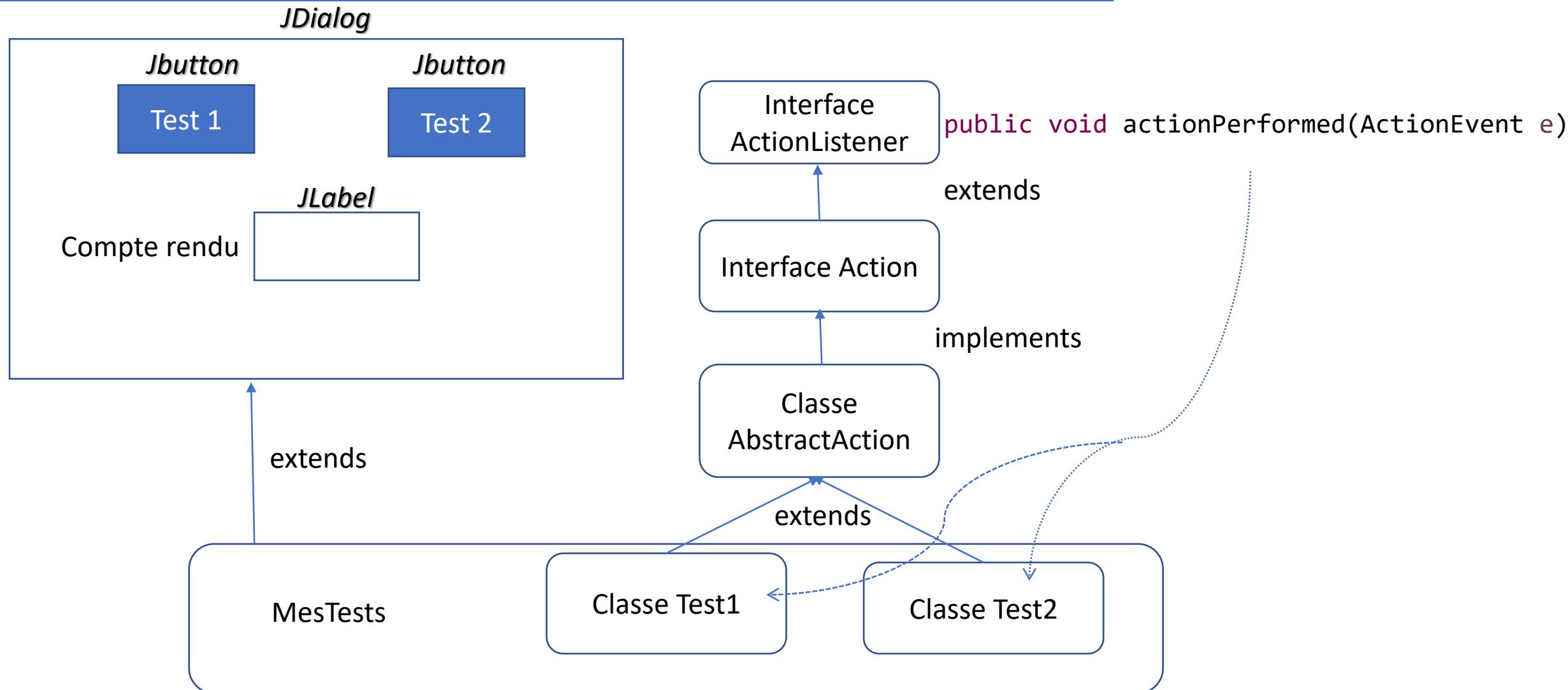
Permet de gérer les ambiguïtés de nommage en fonction du « scope »

Exemple d'utilisation : java.io.Filenamefilter

```
class Exemple {  
    String extension;  
  
    class Filtre implements  
        FilenameFilter {  
        public boolean accept(File dir, String name) {  
            return name.endsWith("." + extension);  
        }  
    }  
  
    void liste(String dir) { File f =  
        new File(dir);  
        String[] ts ;  
        ts = f.list(this.new Filtre());  
    }  
}
```

Création d'une instance de la classe interne

Exemple d'utilisation : Interface graphique et API SWING(1)



Exemple d'utilisation : Interface graphique et API SWING (2)

```
package miage.cours;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MesTests extends JDialog {

    private class Test1 extends AbstractAction {
        public Test1() {super("Test1");}

        public void actionPerformed(ActionEvent e) {
            //exemple stockage d'une trace à l'écran
            compteRendu.setText("test1 effectué");
        }
    }

    private class Test2 extends AbstractAction {
        public Test2() {super("Test2");}

        public void actionPerformed(ActionEvent e) {
            //exemple stockage d'une trace à l'écran
            compteRendu.setText("test2 effectué");
        }
    }
}
```

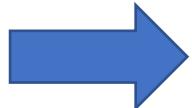
```
private class Test2 extends AbstractAction {
    public Test2() {super("Test2");}
    public void actionPerformed(ActionEvent e) {
        //exemple stockage d'une trace à l'écran
        compteRendu.setText("test2 effectué");
    }
}
private void initTest() {
    ...
    this.add(new JButton(new Test1()));
    this.add(new JButton(new Test2()));
    compteRendu = new JLabel("En attente");
    this.add(compteRendu);
    ...
}
```

Voir le « main » transparent suivant

Création d'une instance des classes internes

Exemple d'utilisation : la classe Java.util.Arrays (3)

```
public static void main(String[] args) {  
    MesTests bt1 = new MesTests();  
    bt1.initTest();  
    MesTests bt2 = new MesTests();  
    bt2.initTest();  
}  
}
```

- 
- Le champ *compteRendu* est le même pour *bt1* et *bt2*:
Test1 et Test2 peuvent être « static »
 - *bt1* et *bt2* ont chacun leur champ *compteRendu* propre :
Test1 et Test2 sont non static

Classe interne non statique et interface

- Une interface ne peut pas être déclarée en interface interne « non statique » comme une classe membre ordinaire car on ne peut pas instancier d'objet à partir d'une interface.
- Si le mot clef **static** est « oublié », le compilateur java le "rajoutera" automatiquement et l'interface sera considérée comme statique !

```
class Exemple {  
    static interface InterfaceInterne extends AutreInterface {  
        ...  
    }  
    public void method();  
}
```

Classes locales (Premiers éléments)

- Déclarée au sein d'un **bloc de code** Java.
généralement dans le corps d'une méthode d'une autre classe qui la contient (nommée classe englobante). Elle adopte un schéma de visibilité semblable à celui d'une variable locale. Une classe locale est instanciable comme une classe membre.
On ne peut utiliser les mots-clés *protected*, *public* ou *private*
- Associée à un objet instancié de la classe englobante.
- Visible et utilisable qu'au sein du bloc de déclaration
- Une classe locale peut utiliser des variables (locales, d'instances ou paramètres) visibles dans le bloc où elle est déclarée (condition: déclaration en mode **final**).
Variable ou paramètre « final » ou n'être modifié ni par la méthode ni par la classe interne.
- Peut utiliser des variables d'instances ou de classes

Une classe locale (non statique)

```
package diversExemples;

public class exemple3 {
    private int attribut1; <-- final
    public int attribut2;
    public void method3 (char attribut1){
        final int attribut2 = 25; <-- final
        class exemple3ClasseLocale1 {...}
        for (int i=0; i<5;i++) {
            final int a = i; <-- final
            class exemple3ClasseLocale2 {...}
            exemple3ClasseLocale2 obj;
            if (i==3) {
                obj = new exemple3ClasseLocale2();
                class exemple3ClasseLocale3 {...}
            }
        }
    }
}
```

int j = attribut2; // local à method3()
int c = attribut1; // local à exemple3

int j = a + attribut 1 + exemple3.this.attribut2
// a est local à la boucle « for »

→ Une classe locale **ne peut pas** être qualifiée en public, private, protected ou static.

Exemple d'utilisation : java.io.Filenamefilter

```
class Exemple {  
  
    String extension ="";  
  
    void liste(String dir) {  
  
        File f = new File(dir);  
        String[] ts ;  
  
        class Filtre implements FilenameFilter {  
            public boolean accept(File dir, String name) {  
                return name.endsWith("." +extension);  
            }  
        }  
  
        ts = f.list(this.new Filtre());  
    }  
}
```

La classe n'est visible que dans la méthode « *liste* » (pas membre de *Exemple*)



Création d'une instance de la classe interne

Classe interne non statique : autre exemple

```
Java.awt.Point getP (int x, int y) {  
    class MonPoint extends java.awt.Point {  
        MonPoint (int x, int y) {  
            this.x = x;  
            this.y = y;  
        }  
        Return new MonPoint (x, y);  
    }  
}
```

→ Permet de personnaliser des objets sans créer de nouveau fichier
(masquer l'implémentation)

Une classe interne statique (ou classe membre statique)

- Une classe membre statique est une classe java définie dans la partie déclaration des membres d'une autre classe qui la contient (nommée classe englobante), puis qualifiée par le modificateur **static**. Une classe membre statique est instanciable.
- Une classe membre statique ne peut pas être associée à un objet instancié de la classe englobante .
- **Syntaxe :**

```
public class Enveloppe {  
    <membres {public, protected, private} attributs >  
    <membres {public, protected, private} méthodes >  
    <membres {public, protected, private} classes membres statiques >  
}
```

Une classe membre statique accède à **tous les membres statiques** de sa classe englobante qu'ils soient **publics** ou **privés**, sans nécessiter d'utiliser le nom de la classe englobante pour accéder aux membres (raccourcis d'écriture) :

Une classe interne statique (ou classe membre statique)

```
package diversExemples;

public class Exemple2 {
    private static String titre = "test0";
    private int attribut;
    * static class Exemple2Interne {
        int attribut2;
        public exemple2Interne(){
            titre = "test";
            //attribut = 2;
            System.out.println(titre);
        }
    }

    public static void main(String[] args) {
        System.out.println(exemple2.titre);
        new Exemple2.exemple2Interne();
        Exemple2 e2 = new Exemple2();
        e2.attribut = 1;
        System.out.println(e2.titre);
    }
}
(*) private, protected, public
```

- Accès aux membres de la classe englobante
 - Y compris pour *private* ou *protected*
 - Membres statiques seulement
- Création d'une instance d'Exemple2
 - Accès aux membres statiques et non statiques
 - Pas d'accès aux membres de la classe interne
- Sans création d'instance d'Exemple2
 - Accès aux membres statiques
 - Pas d'accès aux membres de la classe interne

Exécution

```
test0
test
test
```

- `exemple2$exemple2Interne.class`
- `exemple2.class`

Une classe interne statique (ou classe membre statique)

Hypothèse: notre classe (statique) Exemple2Interne possède :

public void method1 () {

meth1ClEnglobante (titre); // méthode « static » dans la classe englobante Exemple2

Exemple2. meth1ClEnglobante (titre); // titre est « static »

meth1ClEnglobante (Exemple2.titre);

Exemple2. meth1ClEnglobante(Exemple2.titre);

meth1ClEnglobante (attribut) // Non valide (si attribut ou meth1ClEnglobante non « static »)

}

- Utilisation des raccourcis d'écriture
- Un objet de classe interne statique ne comporte pas de référence à l'objet de classe externe qui l'a créé



Exemple d'utilisation : la classe Java.util.Arrays (1)

Un extrait des fonctionnalités

```
public class Arrays {  
    /* It contains various methods for manipulating arrays (such assorting and searching). */  
    ...  
    public static <T> void sort(T[] a, Comparator<? super T> c)  
        /* Sorts the specified array of objects according to the order induced by the  
        specified comparator. All elements in the array must be mutually comparable by the specified  
        comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements  
        e1 and e2 in the array). */  
  
    public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)  
        /* Searches the specified array for the specified object using the binary search algorithm.  
        The array must be sorted into ascending order according to the specified comparator */  
    ...  
}
```

Exemple d'utilisation : la classe Java.util.Arrays (2)

On veut pouvoir utiliser les méthodes de manière à ce qu'un tableau d'étudiants soit trié de telle manière à ce que les étudiants soit classés suivant l'ordre de leur note au module « programmation avancée ». Cela doit se faire de la manière la plus compacte possible

```
package miage.cours;  
import java.util.Comparator;  
  
public class Etudiant extends Personne {  
    private final int noteJava  
    // l'attribut 'nom' est protected  
    // et hérité de la classe Personne  
  
    public Etudiant (String nom, Integer n) {  
        this.nom = nom; noteJava = n;  
    }  
}
```

```
public static class ComparateurEtudiant  
    implements Comparator<Etudiant> {  
  
    public int compare(Etudiant e1,  
                      Etudiant e2) {  
        if (e1 == null) {return -1;}  
        if (e2 == null) {return 1;}  
        Integer i = e1.noteJava;  
        int res = i.compareTo(e2.noteJava);  
        if (res == 0) {  
            res = e1.nom.compareTo(e2.nom);}  
        return res;  
    }  
}
```

Evite de créer des liens « extends » ou « implements » dans Etudiant

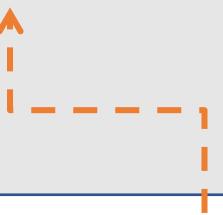
Exemple d'utilisation : la classe Java.util.Arrays (3)

```
package miage.cours;
import java.util.Arrays;
public class Exemple1 {

    Etudiant[] etudiants ;

    exemple1(){
        etudiants = new Etudiant[] { new Etudiant ("etudiantA", 10),
                                    new Etudiant ("etudiantC", 11), new Etudiant ("etudiantB", 11)};
    }

    public static void main(String[] args) {
        Exemple1 e ;
        e = new exemple1();
        Arrays.sort(e.etudiants, new Etudiant.ComparateurEtudiant());
        ...
    }
}
```



Intérêt que la classe *ComparateurEtudiant* soit déclarée « static »

Classe anonyme

- Une classe **anonyme** est une **classe locale qui ne porte pas de nom**.
- Une classe anonyme possède toutes les propriétés d'une classe locale.
- Comme une classe locale n'a pas de nom on ne peut pas définir un constructeur.
- Une classe anonyme est instanciée immédiatement dans sa déclaration selon une syntaxe spécifique :

```
new <identificateur de classe> ( <liste de paramètres de constructions>) {  
    <corps de la classe>  
}
```

Quand utiliser une classe anonyme ?

Principaux cas d'utilisation :

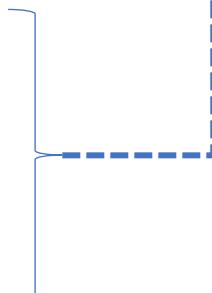
- Une classe anonyme étend concrètement une classe déjà existante abstraite ou non, ou bien implémente concrètement une interface.
- Une classe anonyme sert lorsque l'on a besoin d'une classe pour un seul usage unique, elle est définie et instanciée là où elle doit être utilisée.
- L'exemple le plus marquant d'utilisation de classe anonyme est l'instanciation d'un « écouteur d'évènements »

Quand utiliser une classe anonyme ?

```
Class A {  
    public String titre;  
    A (String s){ titre = s; }  
    public void meth1 (String x){titre = x;}  
}
```

```
Class A1 extends A {  
    public void meth1(String x) {  
        super(x);  
        titre = titre.concat(" fin");}  
}
```

```
public void method () {  
    A x = new A() {  
        public void meth1(String s){  
            super(x);  
            nom = nom.concat(" fin");  
        }  
    };  
}
```



Une classe interne anonyme

```
package diversExemples;

public class exemple3 {
    java.awt.Point p;

    exemple3 (int x, int y) {
        p = new java.awt.Point (x, y);
    }

    void display () {
        System.out.println (p.toString());
    }

    void display2 () {
        java.awt.Point p2 = new java.awt.Point (p.x,p.y) {
            public String toString() {
                return "mon toString: " + p.x + " / " + p.y + " ...";
            }
        };
        System.out.println (p2.toString());
    }
}
```

- Dans une méthode ou un bloc
- Permet de (re)définir localement une méthode
- S'applique à une classe ou une interface
- Très intéressant pour l'implantation des « callback »

....

```
public static void main(String[] args) {
    exemple3 e1 = new exemple3(3,4);
    e1.display2(); -----> mon toString: 3 / 4 ...
    e1.display(); -----> java.awt.Point[x=3,y=4]
}
```

Classe anonyme: exemple

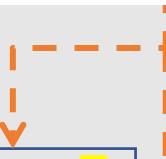
```
Java.awt.Point getP (int x, int y) {  
    return new java.awt.Point(int x, int y) {  
Redéfinition    public String toString() {  
        return "Mon point"+ "x="+ x+ ", y=" +y  
    }  
}  
}
```

→ Permet de personnaliser des objets sans créer de nouveau fichier (masquer l'implémentation)

Exemple d'utilisation : java.util.EventListener.ActionListener

```
Package sun.jvm.hotspot;
...
public class HSDB implements ObjectHistogramPanel.Listener, SAListener {
    ...
    JMenu menu = new JMenu("File");
    JMenuItem item;
    item = createMenuItem("Attach to HotSpot process...",
        new ActionListener() {public void actionPerformed(ActionEvent e) {...}});
    menu.add(item);
    item = createMenuItem("Open HotSpot core file...",
        new ActionListener() {public void actionPerformed(ActionEvent e) {...}});
    menu.add(item);
    ...
}
```

Création d'instances de classes anonymes



Lambda: premiers éléments

- Les expressions lambda. nommées *closures* ou fonctions anonymes
- Leur objectif: permettre de passer en paramètre un ensemble de traitements.

```
monBouton.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        System.out.println("clic");  
    }  
});
```

ActionListener a une seule routine abstraite : *ActionPerformed* et son paramètre de type *ActionEvent*

```
monBouton.addActionListener(event -> System.out.println("clic"));
```

→ le compilateur va déterminer que le paramètre de *addActionListener* est l'interface *ActionListener* et créer une instance de la classe « **anonyme** » générée

Lambda : quelle syntaxe ?

- L'expression lambda passée en paramètre permet de définir une implémentation d'une **interface fonctionnelle*** sous la forme d'une expression de la forme :
(arguments) -> corps

(paramètres) -> expression;
(paramètres) -> { traitements; }

zéro, un ou plusieurs paramètres dont le type peut être déclaré explicitement ou inféré

- L'opérateur **->** sépare le ou les paramètres du bloc de code qui va les utiliser.
- Le type du paramètre n'est pas obligatoire

le compilateur va tenter de réaliser une inférence du type pour le déterminer selon le contexte

- Une expression lambda est typée de manière statique. Ce type doit être une interface fonctionnelle.

(* utilisation facultative : le compilateur « sait » si les contraintes sont vérifiées

Interfaces fonctionnelles (à partir de java 8)

```
@FunctionalInterface  
public interface E1 {  
    methodeAbstraite(Type param);  
    default void methodE1() { // implémentation }  
    static methodE2 () { // même règle que dans une classe }  
}
```

Interface fonctionnelle :

- interface contenant une unique méthode *abstraite*
- *toutes les méthodes doivent être public*
- autant de méthodes statiques ou de méthodes par défaut que l'on veut
- Marquage par annotation: `@FunctionalInterface`

Exemple d'interfaces qui vérifient les contraintes :

Comparator<T> qui définit la méthode int compare(T o1, T o2)

ActionListener qui définit la méthode void actionPerformed(ActionEvent)

Exemple d'interface fonctionnelle

```
@FunctionalInterface  
public interface Consumer<T> {  
  
    /**  
     * Performs this operation on the given argument.  
     *  
     * @param t the input argument  
     */  
    void accept(T t);
```

Exemple d'utilisation : la classe Java.util.Arrays (1)

On veut pouvoir utiliser les méthodes de manière à ce qu'un tableau d'étudiants soit trié de telle manière à ce que les étudiants soit classés suivant l'ordre de leur note au module « programmation avancée ». Cela doit se faire de la manière la plus compacte possible

```
package miage.cours;  
import java.util.Comparator;  
  
public class Etudiant extends Personne {  
    private final int noteJava  
    // l'attribut 'nom' est privé  
    // et hérité de la classe Personne  
  
    public Etudiant (String nom, Integer n) {  
        this.nom = nom; noteJava = n;  
    } }
```

Pas besoin d'ajouter de classe interne si dans la classe utilisatrice il y a une lambda

Comparator<T> doit alors être une interface fonctionnelle

Exemple d'utilisation : la classe Java.util.Arrays (2)

```
package miage.cours;
import java.util.Arrays;
import java.util.Comparator;
public class TesterLesLambda {
    Etudiant[] etudiants ;
    TesterLesLambda(){
        etudiants = new Etudiant[] { new Etudiant ("etudiantA", 10),
            new Etudiant ("etudiantC", 11), new Etudiant ("etudiantB", 11)}; }

    public static void main(String[] args) {
        TesterLesLambda e = new TesterLesLambda();
        Comparator<Etudiant> triParNom = (Etudiant e1, Etudiant e2) -> {
            if (e1 == null) {return -1;}
            if (e2 == null) {return 1;}
            Integer i = e1.noteJava;
            int res = i.compareTo(e2.noteJava);
            if (res == 0) {res = e1.nom.compareTo(e2.nom);}
            return res; }
        Arrays.sort(e.etudiants, triParNom);
    ...
}
```

Ici le type des paramètres est facultatif

Exemple d'utilisation : Interface graphique et API SWING

```
package miage.cours;  
  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class MesTests extends JDialog {  
  
    private void initTest() {  
        ...  
        JButton b1 = new JButton("test1");  
        JButton b2 = new JButton("test2");  
        b1.addActionListener(event->  
            compteRendu.setText("test1 effectué"));  
    }  
}
```

b2. addActionListener(event->
 compteRendu.setText("test2 effectué"));

compteRendu = new JLabel("En attente");
this.add(compteRendu);
this.add(b1);
this.add(b2);
...
}
le « main » est inchangé
}

Programmation Avancée

Réflexivité et Introspection

Philippe Lahire

Introduction et définitions

- Réflexivité : Capacité à se décrire soit même
- Introspection
 - Mécanisme qui permet l'accès dynamique à la structure d'un programme
 - connaître.inspecter les classes, les objets, les méthodes à l'exécution
 - Le faire dans le même langage = introspection réflexive
- Meta-programmation
 - capacité de modifier les mécanismes du langage à l'aide d'un programme META
 - Meta + réflex = inspecter et modifier le comportement
 - Java = introspection réflexive, pas de protocole de métaprogrammation

Utilisation :

Environnements de développement

Echange de classe dans les systèmes distribués

Programmation d'outils spécifiques

Paquetage *java.lang.reflect* (et *java.lang*)
Des éléments syntaxiques

Types statiques et types dynamiques

Types d'une référence ou d'un attribut/champ

- Type statique de « p » : Personne
- Type dynamique de « p » :
 - Liaison dynamique : Etudiant

De la compilation à l'exécution

- Type statique :
 - Vérification à la compilation:
setAge existe bien dans Personne
- Type dynamique de « p » :
 - Liaison dynamique réalisée à l'exécution :
setAge: Version de la classe Etudiant

```
Class Personne
Personne()
setAge(int i)

extends

Class Etudiant
Etudiant()
setAge(int i)
```

```
Class Exemple {
    Personne p;

    exemple () {
        p = new Etudiant();
        p.setAge(21);
    }
}
```

Run-Time Type Identification (RTTI)

- Java maintient ce qu'on appelle l'Identification de Type à l'exécution sur tous les objets
- Permet de connaître le **type dynamique** d'une référence
- La classe `Class` et le RTTI

Permet de faire du contrôle de type

```
Personne p1 = new Personne("Philippe L.");
Personne p2 = new Etudiant("Henri L.")
Class<? extends Personne> c1 = p1.getClass();
System.out.println(c1.getName() +" "+ p1.nom);
Class<? extends Personne> c2 = p2.getClass();
System.out.println(c2.getName() +" "+ p2.nom);
```

Permet l'accès à « `getName()` »

- Affiche à l'exécution

```
miage.m1.cm.Personne:Philippe L.
miage.m1.cm.Etudiant:Henri L.
```

→ *getClass retourne la classe obtenue par liaison dynamique*

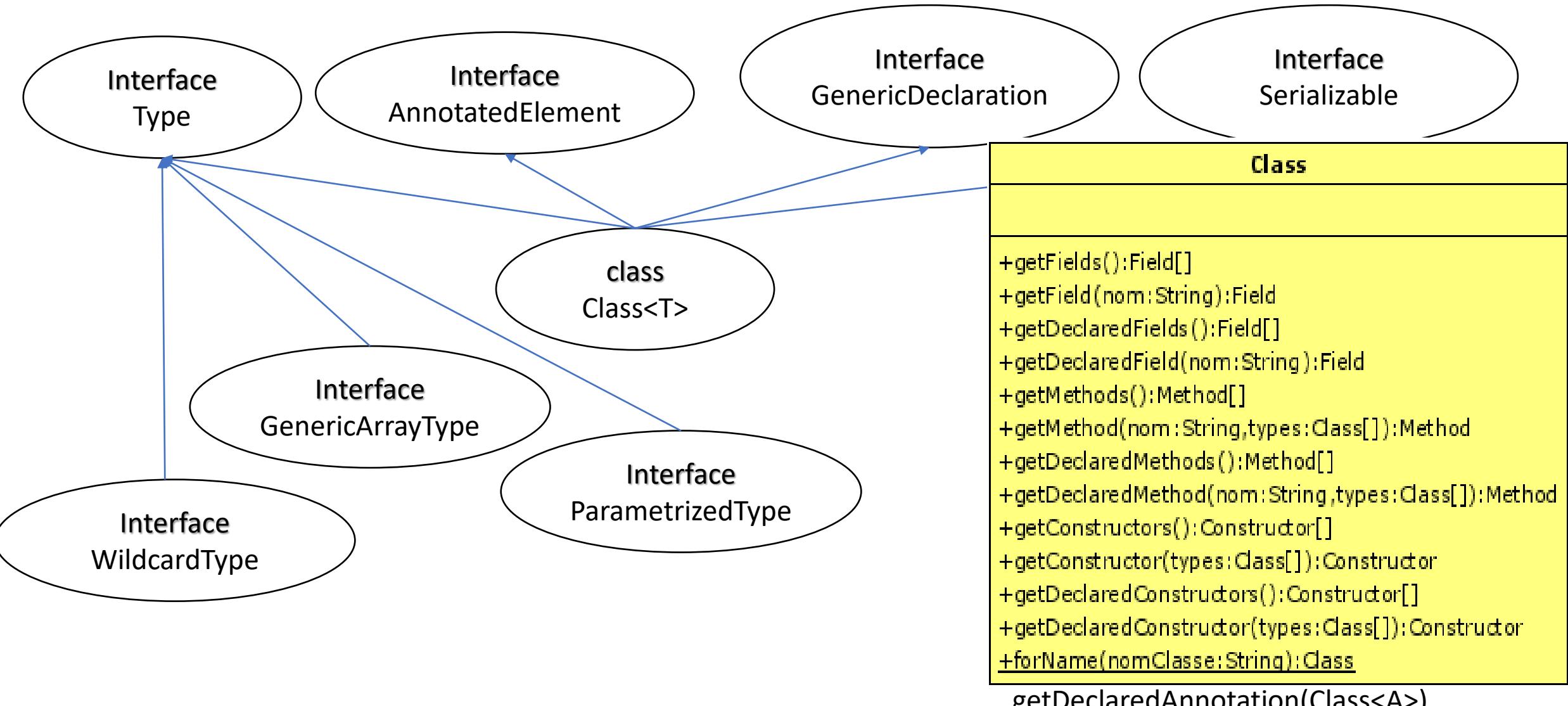
Champ statique « class »

- C'est une construction du langage
- Tout se passe comme si toute classe ou type primitif ou tableau avait un champ statique « class »
- Un objet de type class: un type, pas forcément une classe
- Exemple de compatibilités de type :

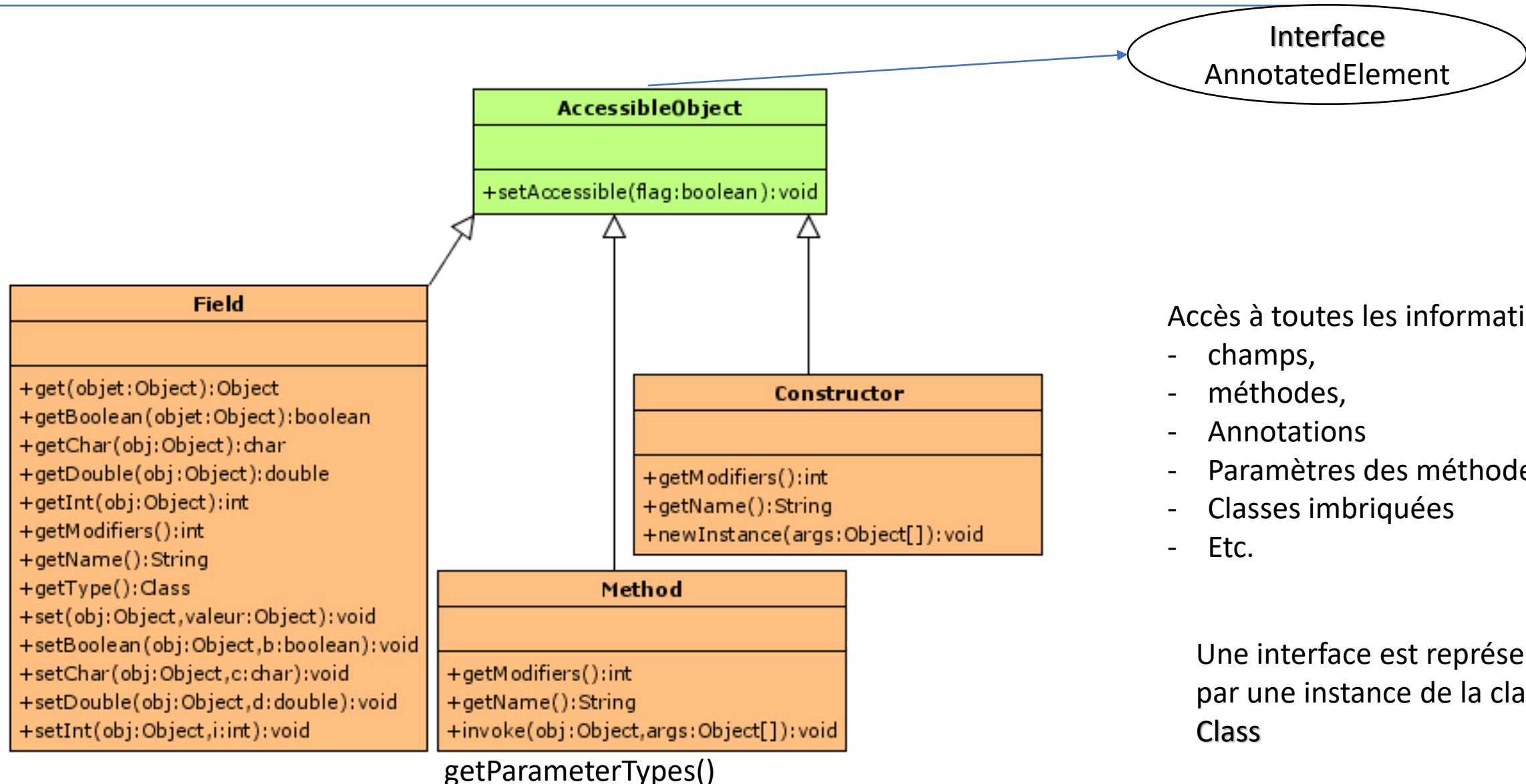
```
Class<Etudiant> cl1 = Etudiant.class;
Class<Integer> cl2 = int.class;
Class<Double> cl3 = double.class;
Class<Void> cl4 = void.class
Class<Etudiant[]> = Etudiant[].class
```

nomClasse.class
nomClasse[].class
typePrimitif.class
Void.class

La classe Class (hiérarchie et quelques fonctionnalités)



Description des éléments d'une classe



Interface
AnnotatedElement

Accès à toutes les informations :

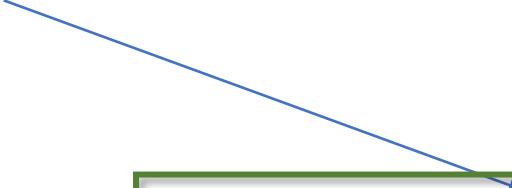
- champs,
- méthodes,
- Annotations
- Paramètres des méthodes
- Classes imbriquées
- Etc.

Une interface est représentée par une instance de la classe
Class

Charger une classe à partir de son nom

`public static Class<?> forName(String className)`

nomClasse : peut être un nom d'interface ou de classe



```
String nomClasse = "java.lang.reflect.Array";
try {
    Class<?> cl = Class.forName(nomClasse);
    Method[] md = cl.getDeclaredMethods();
    System.out.println(md[0].getName());
} catch (ClassNotFoundException x) {}
```



Utile pour charger des classes dont on ne connaît pas le nom à l'avance

Créer des instances sans utiliser « new »

```
public T newInstance(Object ... initargs)
    throws InstantiationException, IllegalAccessException,
        IllegalArgumentException, InvocationTargetException
```

nomClasse : peut être un nom d'interface ou de classe

`Object o = cl.newInstance();`
(constructeur par défaut)

```
String nomClasse = "miage.m1.cm.Personne";
try {
    Class<?> cl = Class.forName(nomClasse);
    Constructor<?>[] cc = cl.getConstructors();
    Object = cc[0].newInstance(...); ←
} catch (ClassNotFoundException | ... x) {}
```

Dépend du constructeur si plusieurs dans la classe

→ Utile pour créer des instances de classes dynamiquement

Voir le contenu d'un champ « public »

```
String nomClasse = "miage.m1.td1.Personne";
try {
    Class<?> cl = Class.forName(nomClasse);
    Constructor<?>[] cc = cl.getConstructors();
    Object o = cc[0].newInstance();
    Field f = o.getClass().getField("age");
    System.out.println(f.getInt(o));
} catch (ClassNotFoundException | ... y) {}
```

-
- Un objet de type Field ne contient pas la valeur de l'attribut mais la « structure » de l'attribut
 - f.getType pour récupérer le type statique
 - F.get(o).getClass.getTypeName() pour le type dynamique

Autres méthodes de la classe « Class <T> »

- `String getName(): nom de la classe`
- `String toString();`
- `public native Class<? super T> getSuperclass()`
- `public Class<?>[] getInterfaces()`
- `public native boolean isInterface();`
- `public native boolean isInstance(Object obj); (= instanceof)`
- `public native boolean isAssignableFrom(Class<?> cls)`
- `public <U> Class<? extends U> asSubclass(Class<U> clazz)`
- `public Class<?> getComponentType()`

si le type de la classe est un tableau

Analyser une classe (1): la structure d'une classe

Deux classes dans `Java.Lang`:

- La classe `Class`
- La classe `Object`

Dans `java.lang.reflect`:

- Trois classes : `Field`, `Method`, `Constructor`
- Mais aussi d'autres: `Modifier`, etc.
- Les quatre classes (`Class...Constructor`) possèdent `getName()`
- `Field` possède `getType()` qui renvoie un objet de type `Class`
- `Method` et `Constructor` ont des méthodes pour obtenir le type de retour et le type des paramètres et surtout des méthodes pour les exécuter

 `Object` et `Class` sont le point de départ qui permet d'accéder à « tout »

Analyser une classe (2) : reconnaître les « modifiers »

- **Field, Method, Constructor** possèdent `getModifiers()` qui renvoie un `int`,
Interprétation de l'entier bit par bit (0 ou à 1): signifient static, public, private, etc...
- On utilise les méthodes statiques de `java.lang.reflect.Modifier` pour interpréter cette valeur :
 - `String toString(int)`
 - `boolean isFinal(int)`
 - `boolean isPublic(int)`
 - `boolean isPrivate(int)`
 - ...
- **Field, Method, Constructor** (d'autres méthodes):
 - `Class getDeclaringClass()`
 - `Class[] getExceptionTypes()` et `Class[] getParameterTypes()`
Constructor et Method uniquement

Analyser une classe (3): Les champs

- `Field[] getFields()` :
Ne renvoie que les champs publics, locaux et hérités
- `Field[] getDeclaredFields()` :
Renvoie tous les attributs locaux uniquement
- Ces deux méthodes renvoient un tableau de longueur nulle si
 - Pas de champs
 - La classe est un type prédéfini (`int`, `double`...)

Quelques informations complémentaires :

- Le lieu de la déclaration (classe): `getDeclaringClass()`
- Le type « statique » du champ (Classe, primitif ou générique): `getType()` et `getGenericType()`
- Annotations (*voir chapitre du cours concerné*)

Analyser une classe (4): les méthodes / constructeurs

- **Method[] getMethods()**
Ne renvoie que les méthodes publiques, locales et héritées
- **Method[] getDeclaredMethods()**
Renvoie toutes les méthodes locales uniquement
- **Constructor[] getConstructors()**
Ne renvoie que les constructeurs publics
- **Constructors[] getDeclaredConstructors()**
Renvoie tous les constructeurs
- Accéder aux paramètres : classe Parameter
 - Assez proche des fonctionnalités offertes par la classe FIELD: *getType*, *getParameterizedType*, annotations, etc.

Analyser un objet inconnu

Nous *avons vu* comment déterminer les noms et les types des champs d'un objet

- Obtenir un objet de type `Class`
- Appeler `getDeclaredFields()` sur l'objet obtenu

Nous *allons voir* comment obtenir la valeur des champs d'un objet

- on ne connaît pas l'objet à examiner à l'avance
- On ne connaît pas sa classe

Accéder à la valeur d'un champ

- Utiliser la méthode `get(Object obj)` de la classe **Field**
 - Un type primitif la valeur est encapsulée dans un objet
 - Un champ statique: le paramètre est ignoré
 - Le champ doit être accessible (dépend des « modifiés »)
 - Gestion d'exceptions: accès illégal, type non conforme

Si **f** est un objet de type **Field** Alors

Si **obj** est un objet de la classe dont **f** est le champ Alors

f.get(obj) renvoie la valeur de l'attribut **f** de l'objet **obj**

Sur un exemple (1)

```
public class Personne {  
    private String nom;  
    private String prenom;  
    private int age;  
    public Personne(String nom, String prenom, int age) {  
        this.nom = nom;  
        this.prenom = prenom;  
        this.age = age;  
    }  
}
```

Sur un exemple (2)

Supposons que l'on connaisse une instance et le nom des champs :

```
Personne p = new Personne("R.", "Philippe", 33);
...
Class cl = p.getClass();
Field f = cl.getField("prenom");
Object v = f.get(p);
System.out.println(v);
```

On ne peut pas accéder à tout : par défaut, le mécanisme de réflexion respecte le contrôle des accès.

- Exemple : Le champ "prenom" est **private** !
- Génération d'une exception: *IllegalAccessException*

Sur un exemple (3)

On peut voir les champs d'un objet mais pas toujours consulter leur valeur

Solutions de contournement :

- mettre l'attribut "prenom" public
- mettre le code dans la classe Personne
- si un programme Java n'est pas contrôlé par un gestionnaire de sécurité (*policy manager*) qui le lui interdit, il peut outrepasser son droit d'accès.

Invoquer la méthode setAccessible() d'un objet Field, Method ou Constructor

- Autre méthode (méthode statique) :

`AccessibleObject.setAccessible(AccessibleObject[] array, boolean flag)`

Sur un exemple (4)

Code mis à jour :

```
Personne p = new Personne("R.", "Philippe", 33);  
...  
Class cl = p.getClass();  
Field f = cl.getField("prenom");  
f.setAccessible(true);  
Object v = f.get(p);  
System.out.println(v);
```

Cas d'une classe avec des types prédéfinis

- Types prédéfinis : char, boolean, float, long, int, short, byte
Des valeurs, pas des objets
- Des classes associées: Float, Double, Integer, Short, Long, Character, Byte
Encapsulation de la valeur (boxing) et opération inverse (unboxing)
- Accès aux valeurs
 - `f.get(obj)` renvoie un **Object** !
 - La valeur lue est encapsulée
 - Exemple :
 - `Field f = cl.getField("age"); // age est un int`
 - `Object v = f.get(p); // v est un Integer`
- La classe Field contient des primitives dédiées :

`getBoolean (Object): boolean`

`getInt (Object): int`

`getChar (Object) : char`

`getFloat (Object) : float`

Cas d'une classe paramétrée (générique)

Exemple :

```
Exemple e;  
e = new exemple();  
Field f = e.getClass().getDeclaredField(" tab_p");  
Type t = f.getGenericType();  
System.out.print("type:");  
System.out.println(t.toString());
```

```
Class Exemple {  
    public ArrayList<Personne> tab_p;  
  
    exemple () {  
        tb_p = new ArrayList();  
    }  
}
```

Affichage :

type:java.util.ArrayList<miage.m1.td1.Personne>

Utilité des mécanismes réflexifs (1)

Exemple d'utilisation : Augmenter la taille d'un tableau

```
static Object[] AgrandirTailleTableau(Object[] tab, int newLength) {  
    Object[] newTab = new Object[newLength];  
    System.arraycopy(tab, 0, newTab, 0,  
                     Math.min(tab.length, newLength));  
    return newTab;  
}
```

Application :

...
String[] oldTab = {"aaa", "bbb", "ccc", "ddd"};

Object[] newTab = new Object[5];

newTab = AgrandirTailleTableau(oldTab, 5);

System.out.println(newTab.getClass().getTypeName());

java.lang.Object[]

String[] ts = (String[]) newTab;



Impossible : Object n'hérite pas de String

Utilité des mécanismes réflexifs (2)

Exemple d'utilisation : Augmenter la taille d'un tableau

- Utilisation de `java.lang.reflect.array` et de `java.lang.Class`
- `Array` permet de créer des instances et de lire ou modifier les éléments

```
static Object AgrandirTailleTableau2(Object[] tab, int newLength) {  
    Class type = tab.getClass().getComponentType();  
    Object newTab = Array.newInstance(type, newLength);  
    int oldLength = Array.getLength(tab);  
    for (int i=0;i<Math.min(oldLength, newLength);i=i+1) {  
        Array.set(newTab, i, tab[i]);  
    }  
    return newTab;  
}
```

```
Object newTab = AgrandirTailleTableau2(oldTab, 5);  
System.out.println(newTab.getClass().getTypeName());  
String[] newTab2 = (String[]) newTab;
```

`java.lang.String[]`

Invoquer dynamiquement une méthode (1)

Utiliser de `java.lang.reflect.Method`

La classe `Method` possède

`Object invoke(Object o, Object[] args);`

- `o` est l'objet dont on veut appeler la méthode
 - Si on veut invoquer une méthode statique, `o` vaut `null`
- `args` est la liste des paramètres

Une solution pour simuler : `p.getNom()` où ‘`p`’ une instance de `Personne` et ‘`m`’ représente la méthode `getNom()`

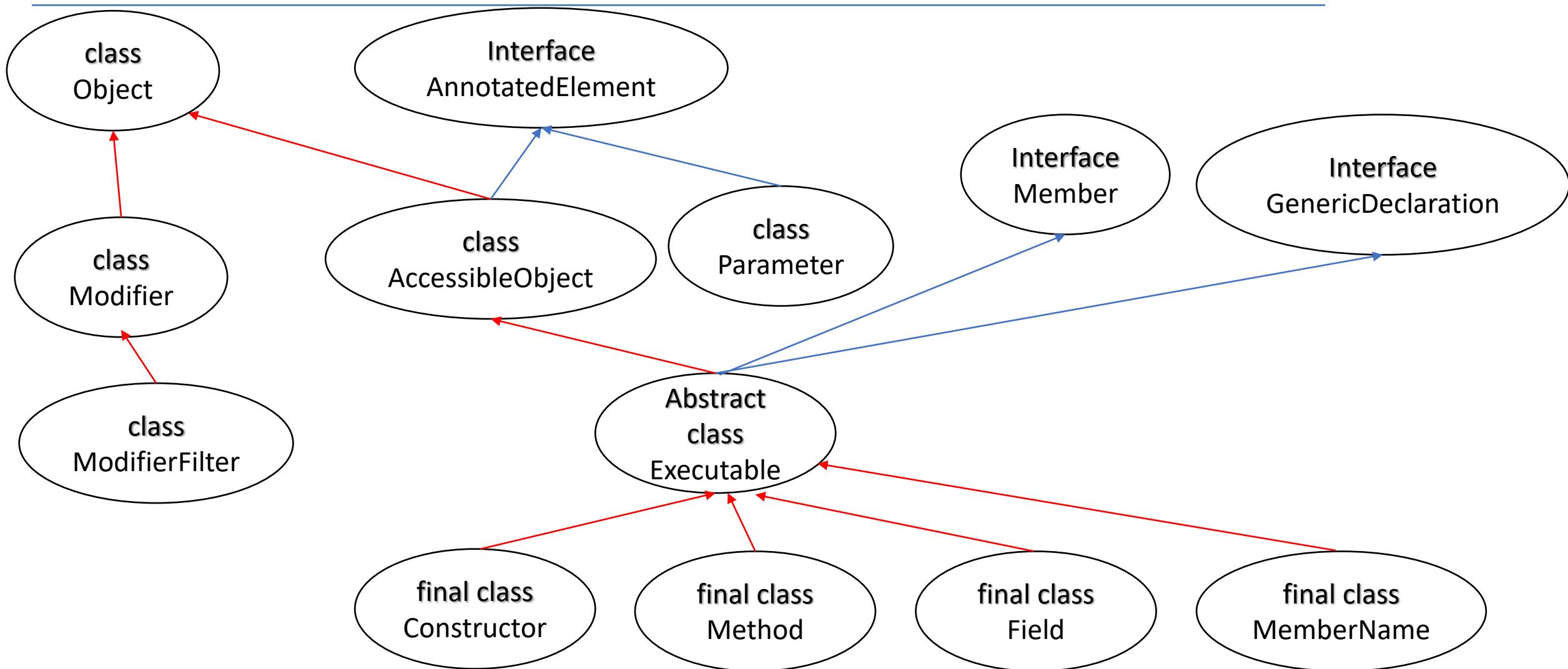
```
Method m = Personne.class.getMethod("getNom", null);
String n = (String) m.invoke(p, null);
```

Invoquer dynamiquement une méthode (2)

- Dans le cas (paramètre et résultat) de types primitifs utiliser les classes enveloppantes **Integer**, **Float**, **Double**, ...
- Si **m** représente **setAge(int a)** de la classe **Personne**

```
Method m = Personne.class.getMethod("setAge", Integer.class);
Object[] args = {new Integer(33)};
m.invoke(p, args);
```

Description des éléments d'une classe (2)



Programmation Avancée

Chargement dynamique

Philippe Lahiré

JVM, JDK et JRE

- la machine virtuelle Java (JVM): permet l'exécution des programmes
- l'environnement d'exécution Java (JRE): crée la machine virtuelle Java ou JVM et s'assure que les dépendances sont disponibles pour les programmes Java

Un JRE inclut une JVM

- le kit de développement Java (JDK): créer des programmes Java qui peuvent être exécutés par la JVM et le JRE

Un JDK inclut un JRE

- La version actuelle de java: 15



La JVM charge les classes (byte code) dynamiquement pendant l'exécution

JVM, JDK et JRE (Rappels)

Version actuelle : JRE 15.0.2

- Il existe plusieurs implémentations de la JVM, et il est important pour le développeur de savoir comment charger et exécuter les fichiers de classe en utilisant la JVM de son choix.
- Le JDK (ou Java SE (Standard Edition) Development Kit) est l'ensemble des outils dont le développeur a besoin pour développer des logiciels basés sur Java. Il existe plusieurs JDKs
- Le JRE (Java SE Runtime Environment) est l'environnement d'exécution Java. C'est donc un logiciel conçu pour exécuter du code Java.

Plusieurs types de class Loader (hiérarchie)

- **Bootstrap ClassLoader**

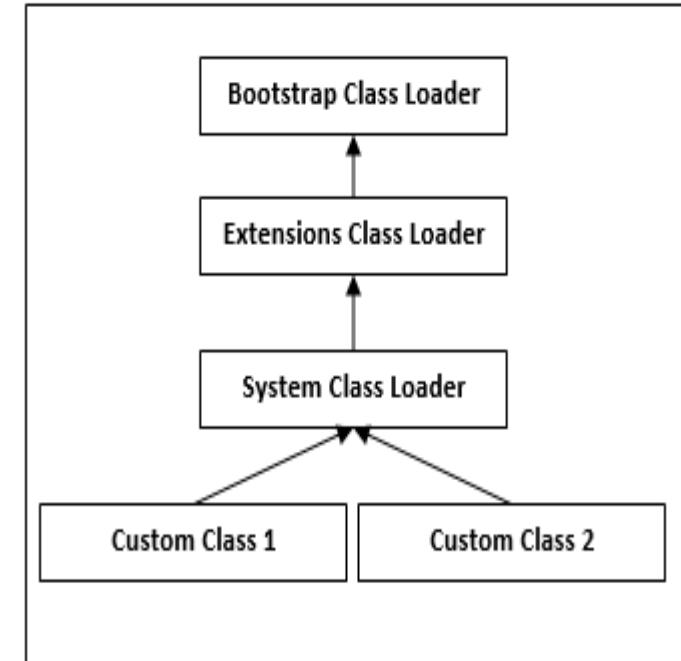
Il charge les fichiers de classe JDK standard à partir de rt.jar et d'autres classes de base. C'est un parent de tous les chargeurs de classe. Il n'a pas de parent.

- **Extension ClassLoader**

Il délègue la requête de chargement de classe à son parent. Si le chargement d'une classe échoue, il charge les classes du répertoire jre/lib/ext ou de tout autre répertoire comme java.ext.dirs.

- **System/Application classLoader**

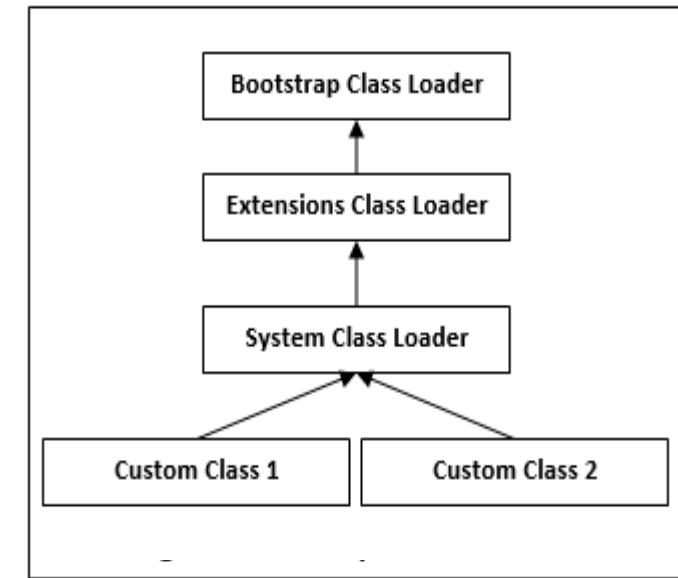
Il charge des classes spécifiques à l'application à partir de la variable d'environnement CLASSPATH. Il peut être défini lors de l'invocation du programme en utilisant « -cp ».



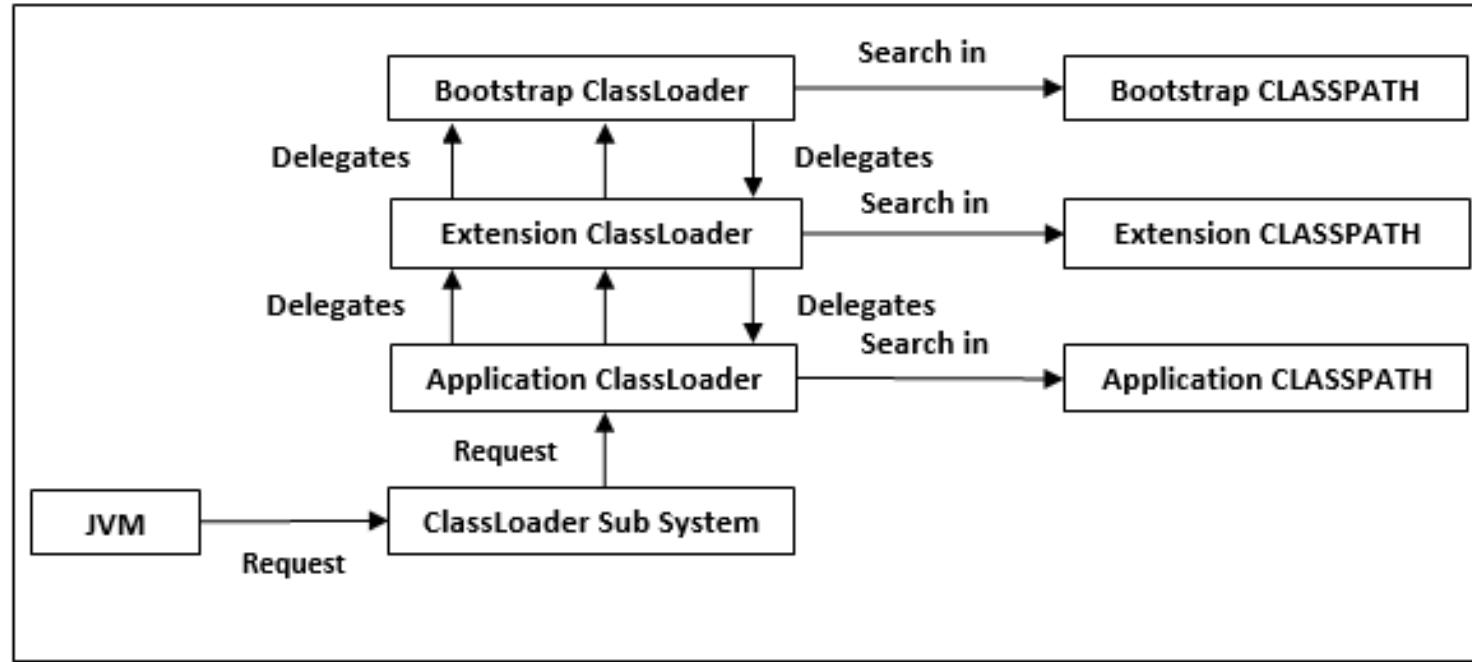
Des échanges formalisés entre les différents types de « class loader »

La classe `java.lang.ClassLoader.class`

- C'est une classe abstraite
- Chargement à partir de différentes ressources
File System, réseau, etc.
- Charger des classes à l'exécution (en fonction des besoins)
Chargement des classes dépendantes
- Existence d'une hiérarchie de *Class Loader*
- Fonctionnement suivant trois principes :
 - Délégation: Le chargement d'une classe est déléguée à son parent
Sauf s'il ne la trouve pas
 - Visibilité: Un fils voit les classes chargées par ses parents mais pas l'inverse
 - Unicité: Une classe n'est chargé qu'une fois
Un fils ne peut charger une seconde fois une classe dans la JVM



Class Loaders : échanges entre fils et père



Un dialogue organisé :

- Si la classe n'est pas trouvé (déjà chargée) à son niveau un class loader interroge son parent jusqu'à « la racine »
- Une fois la racine atteinte et si 1 trouvée, le *bootstrap class loader* cherche dans « son espace »
- S'il ne le retrouve pas il interroge son fils et ainsi de suite...
- Si le class loader « feuille » (qui a fait la demande initiale) ne le trouve pas alors il cherche dans son espace
- Si la classe est trouvée, il la charge sinon déclenchement d'une exception

Moment du chargement d'une classe

➤ Une classe est chargée à l'exécution :

- A l'exécution du byte code correspondant à “new”
- Au moment d'accéder à une référence statique d'une classe

Exemple : *System.out*

→ La détermination des liens peut être Statique (dépend de l'implémentation)

- Au moment où on invoque la méthode `Class.forName()`

→ La détermination des liens est obligatoirement dynamique

➤ Possibilité de précharger les classes en avance de leur utilisation

➤ Le chargement est suivi d'une phase d'édition de lien

L'édition de lien

- **Vérification** de la représentation binaire
 - Code opérations valides, saut vers le début d'une instruction, signatures des méthodes
 - `VerifyError`
- **Préparation** de la classe
 - Création des champs statiques et initialisation à leur valeur par défaut => pas d'exécution de code à ce stade (=> Initialisation)
- **Résolution** des références symboliques
 - Chargement des classes référencées et résolution de lien (vers attributs, méthodes, constructeurs...)
 - `IllegalAccessException` (accès à un champs non autorisé)
 - `InstantiationException` (instantiation d'une classe abstraite)
 - `NoSuchFieldError`, `NoSuchMethodError`, `UnsatisfiedLinkError`
 - (méthode native non implémentée)

Chargement de classes : principales méthodes

java.lang.ClassLoader :

- ✓ `loadClass(String name, boolean resolve)`: Chargement de la classe
- ✓ `defineClass(String name, byte[] b, int off, int len)`: Méthode “final” qui transforme un tableau d’octets en une instance de la classe Class. Si contenu invalide: génère une exception `ClassFormatError`.
- ✓ `findClass(String name)`: Recherche l’existence de la classe et retourne une instance de Class. Ne charge pas la classe.
- ✓ `findLoadedClass(String name)`: Vérifie si la classe a déjà été chargée
- ✓ `resolveClass (Class<?> c)` : édition de lien ou rien suivant l’implémentation

java.lang.Class :

- ✓ `Class.forName(String name, boolean initialize, ClassLoader loader)`: Chargement et initialisation de la classe. Permet de choisir un ClassLoader (sinon le *Bootstrap ClassLoader* est utilisé).

Ce que pourrait être la méthode loadClass

```
protected synchronized Class<?> loadClass(String name, boolean resolve)
                                         throws ClassNotFoundException
{
    Class c = findLoadedClass(name);
    try {
        if (c == null) {
            if (parent != null) { c = parent.loadClass(name, false); }
            else { c = findBootstrapClass0(name); }
        }
    catch (ClassNotFoundException e)
    { System.out.println(e); }
}
}
```

Chargée ?

Délégation

Non trouvée

Ce que pourrait être la méthode loadClass

```
protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException
{
    Class<?> c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) { c = parent.loadClass(name, false); }
            else {c = findBootstrapClassOrNull(name); }
        } catch (ClassNotFoundException e) { .... }
        if (c == null) {c = findClass(name);}
    }

    if (resolve) {resolveClass(c);}
    return c
}
```

 A priori ne redéfinir que la méthode « findClass »

Pourquoi personnaliser le chargement de classe

- Charger des classes depuis le WWW,
- Charger des classes depuis une BD,
- Charger des classes « différemment »
- ...
- Gérer des problématiques de compatibilité :

Exemple : On considère un programme qui utilise plusieurs bibliothèques (librairies) indispensables au bon fonctionnement d'une application, l'une compatible java 1.4 d'autres compatibles java 8 et +.

Solution : Il faut donc les charger dynamiquement, suivant la version du client.

Exemple de classLoader : java.net.URLClassLoader

- Hérite de la classe `java.security.SecureClassLoader`
 - Support de la vérification d'une « politique de droits »
 - Hérite de `java.lang.ClassLoader`
- Permet de rechercher les classes dans une liste ordonnée de chemins contenant des fichiers JAR ou des répertoires.
- Quelques constructeurs :
 - `public URLClassLoader(URL[] urls)`
 - `public URLClassLoader(URL[] urls, ClassLoader parent)`
- Redéfinition de :
 - `Class<?> findClass(final String name)`
 - `Class<?> defineClass(String name, Resource res)`

Personnaliser le chargement d'une classe (1)

- Démarche : Créer une classe fille de `java.lang.ClassLoader`
 - Redéfinir la méthode `findClass` pour aller chercher le code à charger
 - Ou/et Redéfinir la méthode `loadclass`

```
public class MonclassLoader extends java.lang.ClassLoader {  
  
protected Class<?> loadClass(String name, boolean resolve)  
    throws ClassNotFoundException {  
if (super.findLoadedClass(name) == null) {  
    System.out.println("Classe " + name + " non chargée");  
    Class<?> c = super.loadClass(name, resolve); return c;}  
else {  
    System.out.println("Classe " + name + " déjà chargée");  
    Class<?> c = super.loadClass(name, resolve); return c; }  
}  
  
MonclassLoader() {super();}
```

Personnaliser le chargement d'une classe (2)

```
public static void main(String[] args) {
    Class<?> c;
    TestChargementDynamique t;
    try {
        t = new TestChargementDynamique();
        c = Class.forName("java.net.URLClassLoader", true, t); // (1)
        c = Class.forName("miage.examples.classTestChargement", true, t); // (2)
        c = t.loadClass("java.net.URLClassLoader"); // (3)
        c = t.loadClass("miage.examples.classTestChargement"); // (4)
        System.out.println(c.getName()); // (5)
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

- (1) La classe `java.net.URLClassLoader` n'est pas encore chargée
- (2) La classe `miage.examples.classTestChargement` n'est pas encore chargée
- (3) La classe `java.net.URLClassLoader` est déjà chargée
- (4) La classe `miage.examples.classTestChargement` est déjà chargée
- (5) `miage.examples.classTestChargement`

Rappel: Crédation d'un fichier .jar (Eclipse)

- Crédation d'un projet
- Exportation :
 - menu « File »
 - Export wizard: Jar file
 - Sélection des fichiers à mettre dans le .jar
 - Choisir une localisation (et s'en souvenir) *
- Attention lors de la réutilisation :
 - « file:/// » + chemin

(*) en remplaçant « \ » par « / »