Le langage Java

Apprentissage en lien avec le langage UML

Le langage Java

Apprentissage en lien avec le langage UML

Organisation du cours

- Rappels sur le modèle objet, liens avec langage UML et introduction au langage Java
- Fondements du langage Java
- Programmation avancée
  - Traitement des *exceptions*
  - Utilisation des interfaces
- Paquetages importants
  - Collections
  - Entrées/sorties
  - Interfaces Homme-Machine
  - Connexions avec des bases de données relationnelles

Références

- Les documents provenant de *Sun microsystems*
  - Le site de base : `http://www.oracle.com/technetwork/java`
  - La définition du langage : `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`
  - La description de l'API : `http://docs.oracle.com/javase/specs/jls/se8/html/index.html`
  - Les tutoriaux : `http://docs.oracle.com/javase/tutorial/index.html`
- Les livres
  - " Thinking in Java ", *Bruce Eckel* `http://www.mindview.net/Books/TIJ/`
  - " Effective Java ", de *Joshua Bloch*
- Les cours sur Internet
  - Le cours de R. Grin `http://deptinfo.unice.fr/~grin/messupports/` (dont ce cours s'inspire)

La modélisation objet

- Issue des problèmes de simulation (1960)
- Utilisée en modélisation (UML), en programmation (Smalltalk, C++, Java, ...) et en bases de données (SGBDOO : presques disparus et SGBDRO)
- Concept majeur : l'encapsulation
  - Un objet ne communique avec un autre que via un ensemble de messages prédéfinis (sans connaitre leur implantation).
- Les objets sont regroupés en *classe*
- On ajoute d'autres concepts : composition, délégation, héritage, ...
- Un objet peut considéré selon deux points de vue :
  - **Structurel** : Il s'agit de l'instance d'un type (une classe), qui masque une structure derrière des opérations.
  - **Conceptuel** : Il correspond à une entité du monde réel et peut être généralisé ou spécialisé.

Les objets

Définition

Un objet permet de représenter une entité concrète ou abstraite. Il a une identité (un **identifiant**) et des propriétés (un **comportement** et un **état interne**).

- Le **comportement** détermine la façon dont l'objet réagit à des messages qu'il reçoit de son environnement. Cette réaction peut dépendre de son état interne et éventuellement le modifier.
- L'**état interne** est décrit par un ensemble de propriétés : les *attributs*. Chaque attribut possède un nom et une valeur.
  - Généralement, l'état interne n'est pas accessible directement de l'extérieur de l'objet (encapsulation).
    - Amélioration de l'évolutivité et de la sécurité.
    - Problème de performance ?

Les messages

- Un objet possède un ensemble de méthodes *publiques* ou *privées*.
- L'ensemble des méthodes publiques forme son *interface* : l'ensemble des messages qu'il peut recevoir.
- Exemples :

```
unePersonne.saluer();
unePersonne.donnerPrénom("Pierre");
unePersonne.afficherPrénom();
unePersonne.retournerPrénom();
```

Les classes - Définition (1/2)

- On remarque que certains objets possèdent des *propriétés* très proches voire identiques
  - Par exemple, identité des propriétés des voitures représentées dans un programme, ou plus généralement existence de propriétés communes entre les différents véhicules.
- On introduit la notion de *classe* :

Définition

Une classe est un ensemble d'objets dont on reconnait des similitudes sur la façon de les identifier, sur le comportement ou la description de l'état dans le cadre d'une application.

- On appelle *instance* d'une *classe* un *objet* appartenant à celle-ci.

Les classes - Utilisation (2/2)

- Une *classe* permet :
  - de décrire les *propriétés* de ses *instances*. Elle indique leurs variables d'états et leur méthodes.
  - d'instancier de nouveaux objets *via* les *constructeurs*.
  - de partager des informations entre objets ou pour la gestion des instances.
    - pour cela des propriétés peuvent être associées aux classes (au lieu des objets)

Le langage UML

- Pourquoi une méthodologie
  - Des logiciels de plus en plus complexes
  - Développés par des équipes importantes
  - Communiquer à un haut niveau d'abstraction
  - Se mettre d'accord avec les utilisateurs
  - Éviter les erreurs de conception
  - Couvrir tout le cycle de développement
  - Favoriser la réutilisabilité
- Les points forts de UML
  - UML n'est pas une méthode, UML est un langage standardisé par l'OMG
  - Notations simplifiées et non ambiguës
  - Formalisation précise des concepts
  - Couverture importante des besoins en modélisation
- Une démarche itérative et incrémentale guidée par les besoins des utilisateurs

Le langage UML - les diagrammes

Diagrammes	**Utilisation**
Cas d'utilisation	Expression des besoins
Classes et paquetages	Modèles conceptuelsde de données, organisation
Instance ou objet	Représentation du monde réel
Séquence	Exemple de fonctionnement
Collaboration	Coopération entre objets
Etat-Transition	Dynamique et fonctionnement
Composant	Modèle physique, organisation
Déploiement	Architecture, distribution

Les origines du langage Java

- En 1993, il succède au projet *oak* (echec dans la télévision)
- Sun propose un langage portable, sûr associé à une bibliothèque riche inventé par James Gosling et Patrick Naughton
- En 2009 Oracle rachète Sun
- Géré et licencié par Oracle avec l'aide de la communauté (`http://www.jcp.org`)

Les caractéristiques de Java

- Orienté objet
- Portable grâce à une machine virtuelle : " *Compile Once, Run EveryWhere* "
- Sûr : Typage fort, vérification au chargement des classes et au fur et à mesure de l'exécution.
- Sécurisé : Mécanisme de protection d'accès aux ressources (réseau, fichiers, ...)
- Multi-Tâche : Programmation *multi-thread*.
- Dynamique : Chargement des classes lors de l'exécution
- Complet : Fourni avec une grande quantité de classes (API pour les IHM, le réseau, les bases de données, ...)
- Interprété mais ...

Du code source à l'exécution

- La compilation
  - A partir du code source (fichiers `.java`)
  - et à l'aide d'un compilateur (commande `javac`)
  - créaton de *bytecode* (pseudo-code) Java (fichiers `.class`)
- Le *bytecode* (ou pseudo-code)
  - Code assembleur Java
  - Suite d'instructions pour la machine "virtuelle" Java
- Les machines virtuelles
  - *JVM* : *Java Virtual Machine*
  - Implantation de la machine virtuelle sur système donné (Windows, Unix, MacOS, ...)
  - Interprétation et optimisation du code (`http://docs.oracle.com/javase/specs/jvms/se8/html/index.html`)
  - Oracle fournit une implantation de référence
  - Autres implantations disponibles dont une opensource (openjdk)

Code source et bytecode

```
/**
 * Cette application Java affiche simplement "Bonjour a tous !"
 * sur la sortie standard.
 */
class BonjourATous {
    public static void main(String[] args) {
        //Affichage du texte
        System.out.println("Bonjour␣a␣tous␣!");
    }
}
```

Listing 1 – BonjourATous.java

javap -c BonjourATous

```
Compiled from "BonjourATous.java"
class BonjourATous {
  BonjourATous();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Bonjour a tous !
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Listing 2 – BonjourATous.javap

Les différentes plateformes et version

- Implantation de référence fournie par Oracle
  `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
  - Java Development Kit (JDK) : Développement et exécution
  - Java Runtime Environment (JRE) : Exécution uniquement
- Différentes versions, Java 1.0 (1996), Java 1.1 (1997), Java 1.2 ou Java 2 (1998) , Java 1.3 (2000), Java 1.4 (2002), Java 1.5 ou Java 5 (2004) : plus maintenues, ni supportées
- Java 6 (fin en 12/2018), Java 7 (fin en 12/2022), Java 8 et bientôt (09/2018) Java 9.
  - Apparition de nouvelles fonctionnalités
  - Disparition de certaines (d'abord marquées *deprecated*)
- Différentes éditions
  - Java Standard Edition
  - Java Enterprise Edition : Ajout d'APIs utiles pour les serveurs (servlets, ...)
  - Java Micro Edition : Version allégées pour les programmes embarqués (Javacard, ...)

Développer en Java

- Au minimum, un JDK et un éditeur de texte
- Un outils de fabrication de projet : *Maven* (`https://maven.apache.org/`, Gradle, Ant, ...)
- Un IDE (Integrated Development Environment)
  - Assistance au développement (navigation dans les sources, saisie assistée, déboggage, ...)
  - Intégration d'outils UML (création de diagramme, *reverse* et *round-trip engineering*)
  - De nombreuses solutions : Netbeans, Jbuilder, ...
  - Un projet *opensource*, très complet et supporté par IBM : Eclipse (`http://www.eclipse.org`)
  - Un très bon éditeur avec une version communauté et une commercial : Intellij Idea de Jetbrains (`https://www.jetbrains.com/idea`)

Java en résumé

- Les avantages
  - **Portabilité** grâce au *bytecode* executable sur n'importe quelle JVM
  - **Sécurité** grâce à la conception du langage et aux vérifications lors de l'execution faites par la JVM
  - **Richesse** grande quantité de bibliothèques disponibles en standard
- L'inconvénient principal : la lenteur
  - L'interprétation du *bytecode* et les vérifications faites par la JVM ralentissent l'exécution
  - Diverses techniques d'optimisation à la volée (*hotspot*) sont intégrées dans les JVM
  - Dans les cas où la vitesse est cruciale : préférer des langages comme C ou C++
    - Il est possible d'intégrer du code natif en java au prix de la perte de portabilité

Les types

- Le langage Java est **fortement typé**
  - Toute expression a un type connu au moment de la compilation.
- On distingue deux familles de types :
  - Les **types primitifs**
    - Booléen : `boolean` (1 bit)
    - Entier : `byte` (1 octet), `short` (2 octets), `int` (4 octets), `long` (8 octets)
    - Flottant : `float` (4 octets), `double` (8 octets)
    - Caractère : char (2 octets, codage unicode)
  - Les **références**
    - Elles indiquent des objets
    - Elles comprennent les classes, les interfaces et les tableaux.

Les identificateurs en Java

- Ils constituent les noms de classes, d'instances, de méthodes, ...
- Ils sont constitués d'une lettre unicode (de préférence un caractère ASCII) suivi d'une suite de longueur quelconque de lettres, de chiffres et de « _ ».
- Ils doivent être différents des mots réservés et des constantes `true`, `false` et `null`.

Les constantes

- Nombre
  - Entier : de type **long** si elle est suffixé par `L` sinon de type **int**
  - Flottant : de type **float** si elle est suffixé par `F` sinon de type **double**
- Caractère : Un caractère unicode entre deux « ' »
- Booléen : `false` et `true`
- Référence : `null` (référence qui n'indique rien)

Les objets et les références

- Un objet est l'instance d'une classe ou un tableau.
- La valeur d'une référence (on dira simplement référence) indique
  - un objet
  - ou rien si sa valeur est `null`
- Les opérateurs que l'on peut appliquer à une référence sont :
  - L'accès à un champ
  - L'appel de méthode
  - Le transtypage (cast)
  - La concaténation de chaînes (qui converti l'objet par appel de la méthode `toString()`)
  - L'opérateur `instanceOf()`
  - La comparaison de références ($==$ et $!=$)
  - L'opérateur conditionnel ?

Les classes

- La classe `Object` généralise toutes les autres. Une référence de type `object` peut indiquer n'importe quel objet (cf. héritage).
- Les principales méthodes sont :
  - `public final Class<?> getClass()` . . .
    - retourne un objet qui représente la classe de l'objet (cf. introspection).
  - `public String toString()` . . .
    - retourne une `String` qui représente l'objet.
  - `public boolean equals(Object obj)` . . . et `public int hashCode()` . . .
    - définissent la notion d'égalité de deux objets. (cf. collections)
  - `protected Object clone() throws CloneNotSupportedException` . . .
    - crée un double de l'objet.
  - `protected void finalize() throws Throwable` . . .
    - est exécutée juste avant que l'objet ne soit détruit (cf. ramasse miette).

Les variables (1/2)

- On distingue sept types de variables
  - Les variables d'*instance*
    - Déclarées à l'extérieur de toute méthode
    - Visibles depuis toutes les méthodes de l'instance
  - Les variable de *classe*
    - Visibles depuis toutes les instances de la classe
    - Créées et initialisées au chargement de la classe
  - Les variables *locales*
  - Les composants de tableaux (variables anomymes).
  - Les paramètres des méthodes
  - Les paramètres des constructeurs
  - Le paramètre de traitement des erreurs (*exception*)

Les variables (2/2)

- Une variable déclarée `final` ne pourra être affectée qu'une seule fois.
  - Si la valeur est une référence, celle-ci indiquera toujours le même objet mais l'état de celui peut changer.
  - Une variable d'instance `final` est une constante pour chaque instance
- Toutes les variables doivent être inialisées avant d'être utilisées
- Ils existe des valeurs par défaut :
  - Pour les types numériques (même `char`) 0 (en fonction du format)
  - Pour les `boolean` : `false`
  - Pour les références : `null`

Les méthodes - Familles (1/2)

- On distingue plusieurs « familles » de méthodes
  - Celles dédiées à l'accès ou à la modification des variables d'instance : les *accesseurs* et les *modificateurs*.
  - Celles qui sont privées et qui ne sont appelées que par d'autres méthodes de la classe
  - Celles qui sont accessibles de l'extérieur (cf. protection des classes), et qui sont accessibles par d'autres instances de la même classe ou d'autres. Elles forment l'interface de la classe.

Les méthodes - Surcharge (2/2)

Définition

On appelle **surcharge** d'une méthode, la définition d'une autre méthode de même nom mais ayant une *signature* différente

Définition

La **signature** d'un méthode est composée du nom de la méthode et du type de ses paramètres.

- Attention, le type de retour ne fait pas partie de la signature
  - Deux méthodes ne peuvent pas différé uniquement par leur type de retour.

Les classes - Exemple

```
public class Voiture {
// Variables d'instances
 private String immatriculation = "";
 private String marque = "";
 private boolean enMarche;
// Constructeur
 public Voiture(String immatriculation,
            String marque) {
  this.immatriculation = immatriculation;
  this.marque = marque;
 }
 public Voiture(String immatriculation) {
  this(immatriculation, "");
}
//Modificateur
 public void setMarque(String marque){
  this.marque = marque;
 }
// Accesseur
 public String getMarque() {
  return marque;
}
//Methodes
 public void demarrer() { }
 public void arreter() {        }
}
```

Listing 3 – Voiture.java

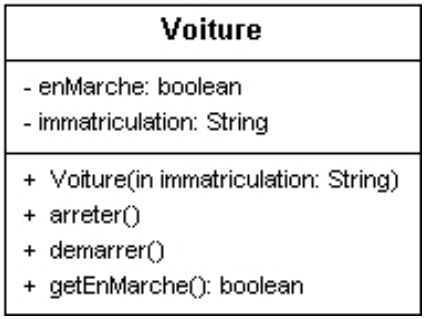


FIGURE – La classe `Voiture`

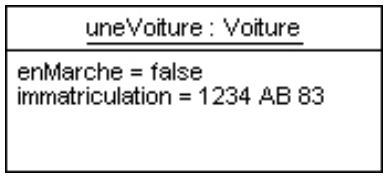


FIGURE – L'instance `uneVoiture`

Les paramètres

- En Java les paramètres sont passés par valeur
- En particulier pour les références :
  - Si valeur de la référence est changée dans une méthode *m()*, pas de changement hors de *m*
  - Si l'objet référencé est modifié dans une méthode *m()* alors c'est l'objet d'origine qui est modifié.

```
class Parametres {
 /* ATTENTION CETTE METHODE EST FAUSSE */
 private static void abandonner(Voiture v) { v = null; }
 private static void changeMarque(Voiture v) {
  v.setMarque("Fiat");
 }
 public static void main(String[] args) {
  Voiture uneVoiture = new Voiture("1234␣AB␣83");
  uneVoiture.setMarque("Rolls-Royce");
  changeMarque(uneVoiture);
  abandonner(uneVoiture);
  System.out.println(uneVoiture);
 }
}
```

Listing 4 – Parametres.java

Les méthodes et variables de classe

- Pour déclarer une variable de classe on utilise le mot clé `static`
  - Une variable de classe (`static`) `final` est une constante pour le programme
- L'initialisation est faite au chargement de la classe
- L'invocation d'une méthode de classe se fait en la préfixant du nom de la classe
  - Facultatif dans la classe courante
  - Attention, il est possible de préfixer par une instance (fortement déconseillé pour la lisibilité)
  - Des méthodes de classe et d'instance ne peuvent avoir la même signature
- Dans le cas, d'une iniatilisation complexe, on peut utiliser un bloc `static` (executé uniquement au chargement de la classe)

Les méthodes de classes - Exemple

```
public class Chien {
        private String tatouage;
        private String nom;

        private static int nbChiens = 0;
        private static final int NB_REFUGES = 3;
        private static int[] occupationRefuge =
                new int[NB_REFUGES];
        // Constructeur
        public Chien(String t, String n) {
                tatouage = t;nom = n;nbChiens++;
        }
        // Methode statique de gestion des instances
        public static int getNbChiens() {
                return nbChiens;
        }

        // Bloc statique d'initialisation
        static {
                for (int i = 0; i < NB_REFUGES; i++) {
                        occupationRefuge[i] = -1;
                }
        }
}
```

Listing 5 – Chien.java

Protection des classes, méthodes et variables

- En java la protection se fait en fonction des classes et non des objets
- Un *membre* (méthode ou attribut) peut être
  - `public` : accessible depuis toutes les instances
  - `protected` : accessible depuis les classes du même paquetage et les sous-classes
  - (par défaut) : accessible depuis les classes du même paquetage
  - `private` : accessible depuis les instances de la même classe

	Classe	Paquetage	SousClasse	Tout
private	Oui	Non	Non	Non
rien	Oui	Oui	Non	Non
protected	Oui	Oui	Oui	Non
public	Oui	Oui	Oui	Oui

Un programme Java

- Un programme Java est un ensemble de classes dont au moins une est executable
  - On ajoute la méthode de classe `main`
- Pour exécuter ce programme :
  - On compile : `javac BonjourATous.java`, cela produit le fichier `BonjourATous.class`
  - On execute (la classe, pas le fichier `.class`) : `java BonjourATous`

```
/**
 * Cette application Java affiche simplement "Bonjour a tous !"
 * sur la sortie standard.
 */
class BonjourATous {
    public static void main(String[] args) {
        //Affichage du texte
        System.out.println("Bonjour a tous !");
    }
}
```

Listing 6 – BonjourATous.java

L'instanciation - Les constructeurs (1/3)

- Pour créer une nouvelle instance d'une classe on utilise un de ses *constructeurs*
- Ils servent aussi à initialiser l'état interne du nouvel objet
- Un constructeur :
  - possède le même nom que la classe
  - n'a pas de type de retour
  - peut être *surchargé*
  - est invoqué avec l'opérateur `new`

L'instanciation - Le constructeur par défaut (2/3)

- Si **aucun** constructeur n'est donné :
  - un constructeur par défaut sans paramètres et ayant la même accessibilité que la classe est créé sinon le constructeur sans paramètre n'est pas créé implicitement
- Une autre constructeur peut être invoqué avec la méthode `this` et les paramètres correspondants

L'instanciation - Exemple (3/3)

- Voir Listing 3 pour la déclaration de la classe `Voiture`.
- Voir Listing 5 pour la déclaration de la classe `Chien`.
- **Attention, la simple déclaration n'instancie pas les objets**

```
/**
 * Un exemple de programme qui
 * instancie une Voiture et deux Chiens
 * @author Emmanuel Bruno
 * @version 0.1
 * @see Voiture, Chien
 */
public class Instanciation {
 public static void main(String[] args) {
  Voiture uneVoiture = new Voiture("1234 AB 83");
  uneVoiture.setMarque("Rolls-Royce");
  uneVoiture.demarrer();
  System.out.println("uneVoiture est une "
                  +uneVoiture.getMarque());
  Chien c1 = new Chien("C1","Rex");
  Chien c2; c2 = new Chien("C2","Medor");
  System.out.println("Il y a "+Chien.getNbChiens()+" chiens");
  }
}
```

Listing 7 – Instanciation.java

Les références

- Une variable dont le *type* est une *classe* (ou une *interface*) a pour valeur une *référence* vers un *objet* d'un type compatible.
- Lors de l'instanciation de cet objet de la mémoire lui a été allouée dans le tas.
- En Java, un objet n'est pas détruit explicitement par appel d'un destructeur, mais lorsqu'il n'est plus référencé.
- Ce travail est effectué automatiquement par un *ramasse-miette* ou *garbage-collector*
- Que se passe-t-il lors de l'appel de la méthode `test()` et après qu'elle soit terminée ?

```
class RamasseMiette {
        private static void  test() {
                Chien unChien = new Chien("X1","Rex");
                Chien monChien = unChien;
        }

    public static void main(String[] args) {
        test();
    }
}
```

Listing 8 – RamasseMiette.java

Le ramasse miette

- L'objectif du ramasse-miette est de déterminer les objets qui ne peuvent plus être utilisés (abscence de référence) et de libérer la mémoire occupée
- Ce travail est réalisé au fur et à mesure de l'éxécution
- **Amélioration de la sécurité** : la gestion de la mémoire n'est plus confiée à l'utilisateur
- **Perte de performance** (variable) : Coût de la « surveillance »
- En Java, le ramasse-miette est une tâche qui s'execute en parallèle et qui agit à des moments aléatoires ou quand le système a besoin de mémoire
- Si l'objet à détruire possède la méthode `finalize()`, elle est invoquée au moment de la destruction.
  - Un objet peut donc être ressucité (en affectant `this` à une variable statique)
  - La méthode `finalize()` n'est appelée qu'une seule fois.
  - Le passage du rammasse-miette peut être demandé : `Runtime.getRuntime.gc()`

L'édition de liens dynamique

- Les classes Java n'indiquent pas explicitement où se trouvent le code correspondant aux classes utilisées
- Le compilateur recherche les classes dans le *classpath*.

Définition

Le classpath est une liste de chemins contenant des classes et des paquetages. Il peut aussi indiquer des fichiers archive (`.jar`).

- Le `classpath` peut être indiqué par une variable d'environnement (CLASSPATH) ou par une option de compilation (préférable).
- Le compilateur compile automatiquement et si nécessaire les classes utilisées.
- Au moment de l'execution, la machine virtuelle doit aussi trouver les classes nécessaires (grâce à un *classpath* éventuellement différent).

Les classes de base - Les tableaux (1/3)

- Un tableau est objet
  - La taille n'est pas fixée à la déclaration
    - `int mesEntiers[];`
  - Mais à la création avec l'opérateur `new`
    - `mesEntiers = new int[3];`
- Ils sont indicés à partir de 0
  - `mesEntiers[0] = 3;`
- Et peuvent être initialisés directement avec des valeurs primitives ou des objets
  - `int [] autresEntiers = {4,8,12+2}`
- On accède à leur taille *via* la variable d'instance `length`.
- Attention instancier un tableau n'instancie pas les objets
  - `Voiture mesVoitures = new Voiture[3];`

Classes de base - Les chaînes de caractères (2/3)

- En Java les chaînes de caractères sont des objets instances de la classe `String`.
- Un littéral s'écrit entre guillemets et deux **littéraux** ayant la même valeur sont le même objet.
  - `String message = "hello";` et `String message2 = "hello";`
  - `String message = new String("hello");`
- Les opérations de base sont la concaténation avec l'opérateur '+', la comparaison avec les méthodes `equals()` et `compareTo()`.
- Une `String` Java ne peut pas être modifiée
  - `String message = "hello";message = "bye bye";`
- Pour utiliser des chaînes modifiables on utilise les classes `StringBuffer` (compatible avec les *threads*) et `StringBuilder` (plus rapide mais mono-thread)

Classes de base - Enveloppement des types primitifs (3/3)

- Pour appliquer des traitements (constantes, conversions, ...) sur des types primitifs des classes leurs sont associées :
  - Classes enveloppantes : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`.
- Cela permet d'accéder à des constantes
- De faire des conversions
- D'utiliser des méthodes qui prennent en paramètres des objets (cf. Collections)

```
class Enveloppement {
 public static void main(String[] args) {
  System.out.println("Max des integers : "+Integer.MAX_VALUE);
  int i = 345; float f = 3.12F;
  Number T[] = {new Integer(i), new Float(f)};
  String num = "345";
  System.out.println(num+" + 1 = "+
                  (Integer.parseInt(num,10)+1));
        }
}
```

Listing 9 – Enveloppement.java

Expressions, Instructions et Blocs

Définition

Une expression est une suite de variables, d'opérateurs et d'appels de méthode qui respecte la syntaxe de Java et qui possède une et une seule valeur.

Définition

Une instruction est une unité exécutable.

- L'affectation est une expression
- On peut produire une instruction en ajoutant un ';' à une expression.
- Les déclarations sont des instructions, ainsi que les structures de contrôle (boucles, ...)

Définition

Une bloc est une suite de zéro ou plusieurs instructions.

- Un bloc est une instruction (indiquée entre '{' et '}')

Les branchements - if

```
/**
 * Utilisation du if
 * @author Emmanuel Bruno */
public class If {
 public static void main(String[] args) {
  String s = "1 test";
  System.out.print("\""+s+"\""+" commence par ");

  if (Character.isLetter(s.charAt(0)))
   System.out.println("une lettre.");
  else if (Character.isDigit(s.charAt(0)))
   System.out.println("un chiffre.");
    else
     System.out.println("non(une lettre ou un chiffre).");
 }
}
```

Listing 10 – If.java

Les branchements - switch

- L'instruction `switch` exécute des sauts en fonction de la valeur d'expressions entières, de chaines de caractères ou de type énumérés (Enum).

```
/**
 * Utilisation de Case avec enum
 * @author Emmanuel Bruno */
public class Case {
 public enum Jour { LUNDI, MARDI, MERCREDI,
                JEUDI, VENDREDI, SAMEDI, DIMANCHE }

 public static void main(String[] args) {
  Jour j = Jour.SAMEDI;
  switch (j) {
   case LUNDI:  case MARDI:  case MERCREDI:
   case JEUDI:  case VENDREDI:
    System.out.println("jour de semaine.");
        break;
   case SAMEDI: case DIMANCHE:
    System.out.println("jour de fin de semaine.");
    break;
   default:
    System.out.println("un nouveau jour ?");
    break;
  }
 }
}
```

Listing 11 – Case.java

```
/**
 * Utilisation de case avec String
 * @author Emmanuel Bruno */
public class CaseString {

 public static void main(String[] args) {
  String j = "Samedi";
  switch (j) {
   case "Lundi":  case "Mardi":  case "Mercredi":
   case "Jeudi":  case "Vendredi":
    System.out.println("jour de semaine.");
        break;
   case "Samedi": case "Dimanche":
    System.out.println("jour de fin de semaine.");
    break;
   default:
    System.out.println("un nouveau jour ?");
    break;
  }
 }
}
```

Listing 12 – CaseString.java

Les boucles while, do...while et for

```
/**
 * Creation et affichage d'un tableau avec la boucle while */
class While {
  public static void main(String[] args) {
  Chien[] mesChiens = {new Chien("C1","Rex"),
        new Chien("C2","Medor"),new Chien("C3","Pluto")};
  int chienCourant=0;
  while (chienCourant<mesChiens.length) {
          System.out.println(mesChiens[chienCourant++]);}
  }
}
```

Listing 13 – While.java

```
/**
 * Affichage des arguments d'un programme avec la boucle For
 * @author Emmanuel Bruno */
class For {
  public static void main(String[] args) {
  for (int i = 0; i < args.length; i++)
    System.out.print(i == args.length-1?args[i]+"." :args[i]+", ");
    System.out.println();
  }
}
```

Listing 14 – For.java

Le foreach

```
/**
 * Exemple de For each de java 5
 * @author Emmanuel Bruno
 */

class For5 {
  public static void main(String[] args) {
    Chien[] mesChiens = new Chien[3];

    mesChiens[0]=new Chien("C1","Rex");
    mesChiens[1]=new Chien("C2","Medor");
    mesChiens[2]=new Chien("C3","Pluto");

    for (Chien c : mesChiens) {
      System.out.print(c + "␣");
  }
 }
}
```

Listing 15 – For5.java

La structuration d'un programme les paquetages (1/3)

- Les classes Java peuvent être organisées hiérarchiquement en paquetages (packages)
- Ils s'agit d'une arborescence similaire à celles des répertoires dans laquelle sont rangées les classes
- On indique en en-tête de la classe le paquetage auquel elle appartient avec le mot-clé `package`

Définition

Le nom complet d'une classe est composé du nom de la classe préfixé du chemin à suivre dans les paquetages pour arriver jusqu'à elle.

- Le nom complet est facultatif dans les classes du même paquetage. Pour les autres, il faut utiliser l'instruction `import` *`nom_complet`* pour pouvoir omettre le chemin.
- Pour ajouter toutes les classes d'un paquetage on utilise « `*` » au lieu du nom d'une classe (par défaut `java.lang.*` est importé).

La structuration d'un programme les paquetages (2/3)

```
package coursSSI3.exemples.animaux;

public class Chat extends Animal {
        public Chat() {
        }

        public Chat(String nom) {
                super(nom);
        }

        public Chat(String nom, int age, int poids) {
                super(nom, age, poids);
        }

        public void miauler() {
                System.out.println("Miou");
        }
}
```

Listing 16 – coursSSI3/exemples/animaux/Chat.java

```
package coursSSI3.exemples.cages;

public class BoiteAChat {
 private coursSSI3.exemples.animaux.Chat pension
 public void enfermer(coursSSI3.exemples.animaux
   {pensionnaire = c;} }
```

Listing 17 – coursSSI3/exemples/cage/BoiteAChat.java

```
package coursSSI3.exemples.cages;
import coursSSI3.exemples.animaux.Chat;
public class CageAChat {
 private Chat pensionnaire = null;
 public void enfermer(Chat c)
   {pensionnaire = c;} }
```

Listing 18 – coursSSI3/exemples/cage/CageAChat.java

La structuration d'un programme les paquetages (3/3)

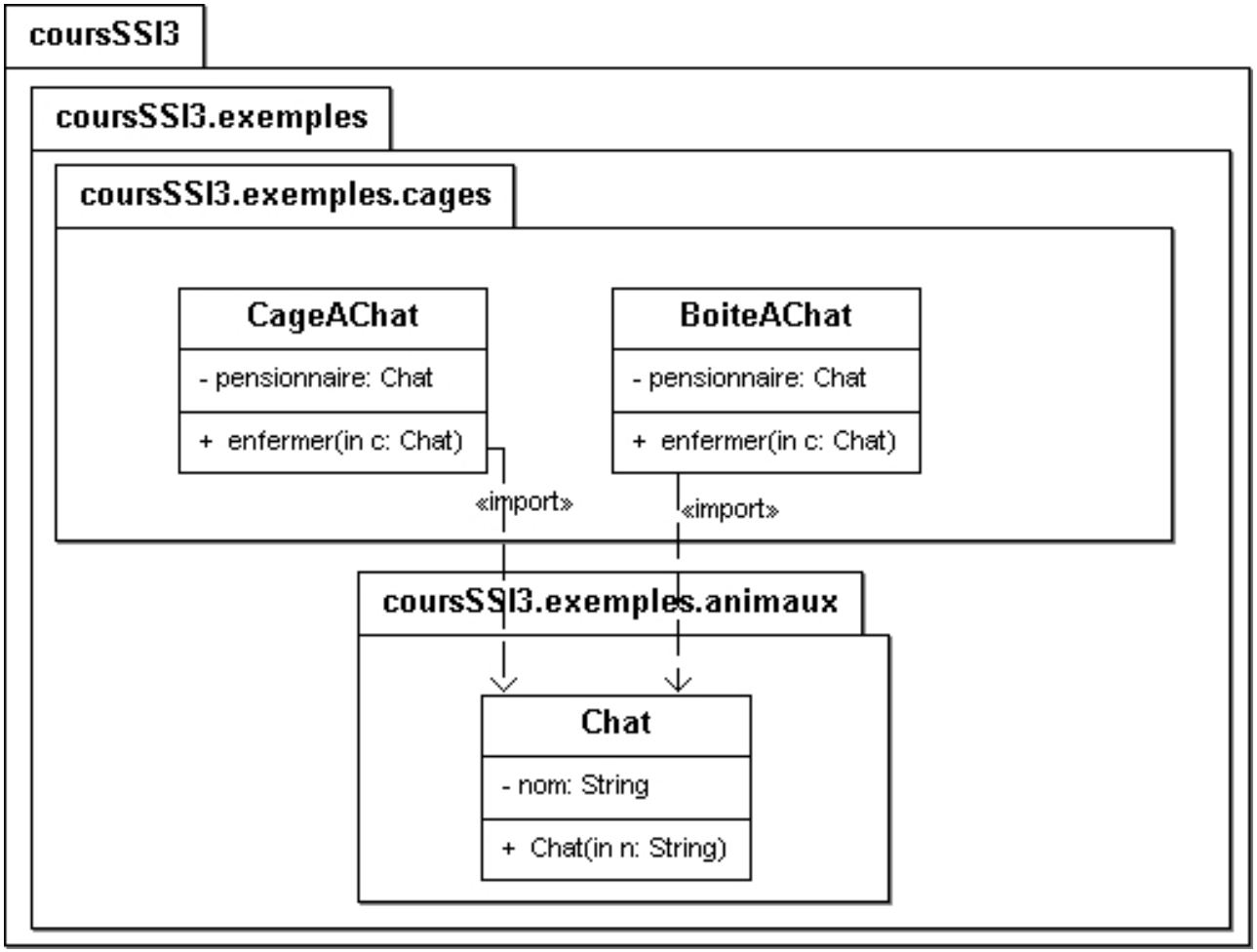


FIGURE – Les paquetages en UML

Les paquetages standards

- En général, les noms de package sont préfixés d'un nom de domaine " inversé " pour garantir l'unicité des noms et facilité la réutilisation. `fr.univtln.bruno.samples.Voiture` (Attention au '-' interdit dans les identificateurs).
- Le langage Java est fourni avec un grand nombre de paquetages standards :
  - `java.lang` : Les classes de base de Java
  - `java.util` : Des utilitaires
  - `java.io` : La gestion des entrées-sorties
  - `java.awt` et `javax.swing` : Les interfaces graphiques
  - `java.applet` : Les applets
  - `java.net` : La gestion du réseau

Les conventions - Notations (1/3)

- En Java, il existe de nombreuses conventions de notation
- Seuls les noms de classes et de constantes commencent par une majuscule
  - `Chien` et `Voiture` sont des classes
- Les noms de constantes sont entièrement en majuscules et les mots séparés par des '_'
  - `Chien.NB_REFUGES` est une constante
- Dans les identificateurs les débuts de mots sont marqués par une majuscule (Camel Notation)
  - `nbChiens` et `monChien` sont des identificateurs
  - `maVoiture.demarre()` (deux identificateurs)

Les conventions - Documentation (2/3)

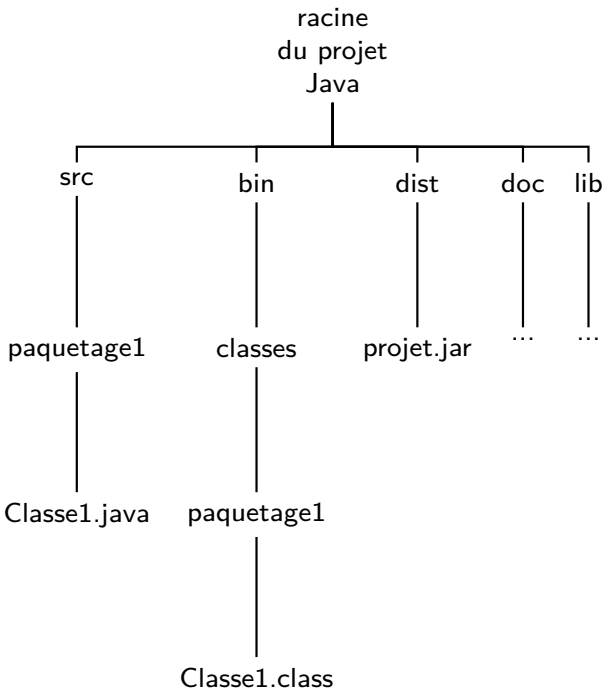
- En Java, la document fait partie du processus de développement
- L'outil `javadoc` (`http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html/`) permet de générer la documention d'un projet en HTML.
- C'est l'outil utilisé par Oracle pour documenter son JDK
- Pour cela, `javadoc` utilise le code source du programme et des commentaires standardisés ajoutés par le développeur

```
/** Cette classe est un exemple de documentation simple
 *  @see Chien
 *  @version 2.0
 *  @since 1.0
 *  @author Emmanuel Bruno
 */
public class TrucDocumente {
        /**
         * Cette methode est un exemple
          * @param i Un entier quelconque
          * @return le double du paramètre
          * @deprecated Utiliser la nouvelle methode X
          */
        public int uneMethode(int i) { return 2*i}
}
```

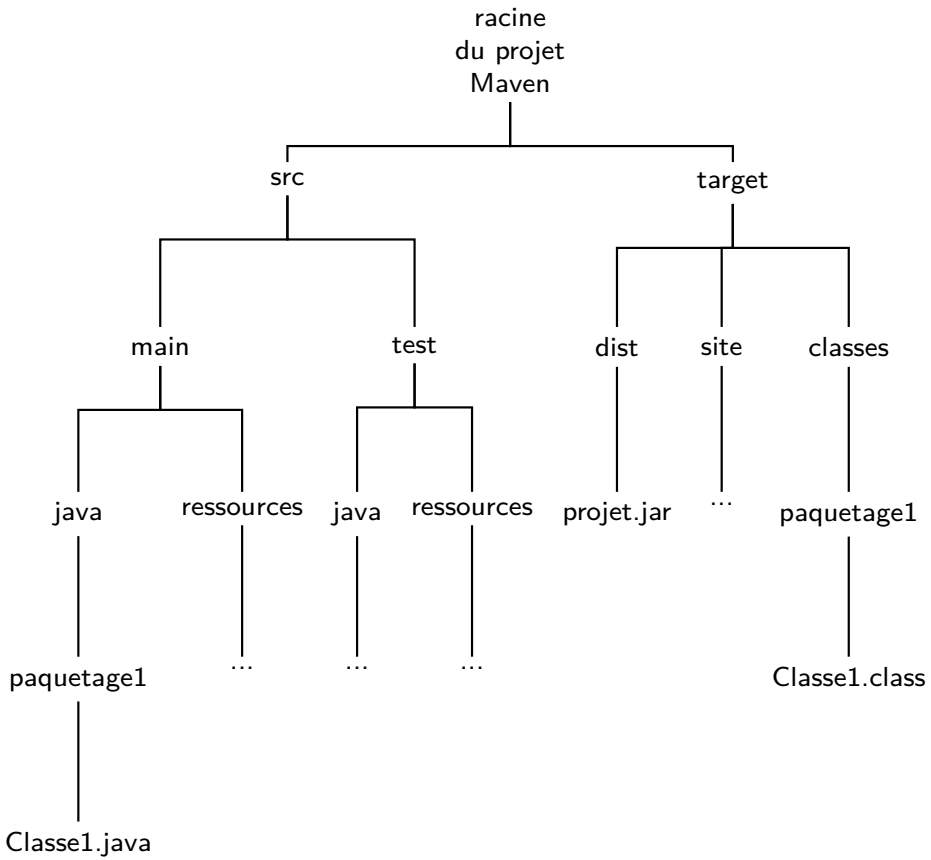
Listing 19 – TrucDocumente.java

Les conventions - Organisation (3/3)

- Un projet Java a généralement la forme suivante :



- Un projet Java avec Maven a généralement la forme suivante :



Le traitement simple des exceptions (1/2)

- Le langage Java utilise les exceptions pour traiter les erreurs
- Ce mécanisme sera traité plus tard en détail
- Quand une méthode peut provoquer une erreur
  - Le développeur qui décrit la méthode doit l'indiquer dans la définition
  - Le développeur qui invoque la méthode doit indiquer un traitement à réaliser en cas d'erreur
    - **Ne rien faire et transmettre l'erreur**. On ajoute la directive `throws` suivie du nom de l'exception à transmettre après la définition de la méthode
    - **Réagir.** On imbrique le bloc concerné dans un bloc `try...catch`

Le traitement simple des exceptions (2/2)

```
/**
 * Cette application Java illustre intuitivement
 * le traitement des erreurs
 */
import java.io.*;
class Erreur {
    public static void main(String[] args) throws IOException {
        int i=0;
        int T[] = new int[10];
        try {
                System.out.println(32/i);
        } catch (java.lang.ArithmeticException e) {
                System.out.println("Division par 0");
        }

        FileReader f = new FileReader("File.txt");
    }
}
```

Listing 20 – Erreur.java

Les fichiers archive `.jar`

- Un programme Java est donc un ensemble de classes distribuées dans des paquetages
- Pour faciliter la distribution des programmes on crée une archive avec l'outil `jar` (Java ARchive)
- Il suffit alors d'ajouter le fichier `.jar` à la variable ou au paramètre `classpath`
- Il est aussi possible d'executer directement un fichier `.jar` en indiquant quel est la classe executable par défaut (cf. TP)
- Un tutoriel se trouve ici : `https://docs.oracle.com/javase/tutorial/deployment/jar/`

Pourquoi réutiliser ?

- Le paradigme objet permet de structurer une application
- Des parties de celle-ci peut-être partagées ou réutilisées
  - Cela comprend la conception et le développement
- Ces parties peuvent aussi être étendues
- Cela accélère de développement et facilite la maintenance
- Mais demande plus de travail préliminaire

UML et Java

- Dans cette partie nous nous limiterons aux modèles UML structurels dédiés aux aspects statiques des objets
  - Les diagrammes de classe
  - Les diagrammes d'instance
- Nous examinerons en parallèle l'implantation possible en Java

La délégation

- Un première solution pour réutiliser en objet : La délégation
- Un classe ou une instance peut :
  - Appeler une méthode d'une autre classe
  - Instancier un objet d'une autre classe et "déléguer"

```
package coursSSI3.exemples.reutilisation;

public class DelegationPure {
        public void action() {
                Voiture v = new Voiture();
                v.demarrer();
        }
}
```

Listing 21 – coursSSI3/exemples/reutilisation/DelegationPure.java

- Il est difficile de formaliser cela : " C1 utilise C2 "

L'association en UML

- Au niveau de l'analyse un attribut ne peut être une classe
- On définit la notion d'**association** pour indiquer une relation entre deux classes
- Une association possède un **nom**, une cardinalité et éventuellement des rôles

L'aggrégation en Java

- La représentation des associations en Java se fait à l'aide d'attributs d'instance
- Dans le cas d'une cardinalité déterminée *n..m*, il est possible d'utiliser un tableau
- Plus généralement, nous utiliserons des *Collections* (cf. cours sur les Collections)
- Les contraintes exprimées sur le modèle UML :
  - Ordonné, Ou-exclusif, sous-ensemble, ...doivent être vérifiées par le programme

La navigabilité en Java

- La navigabilité en Java est matérialisée par :
  - la présence d'un attribut
  - des accesseurs permettant d'obtenir les valeurs
- Attention, au coût et à la complexité des associations bi-directionnelles.
  - Maintien de la cohérence

L'aggrégation et la composition en Java

- L'aggrégation par rapport à la composition est plutot sémantique
- La composition impose de vérifier la contrainte de co-existance :
  - La destruction du composé impose la destruction des composants
  - Mais aussi la destruction d'un composant être interdite sans s'assurer au préalable de la destruction du composant.
- En Java, c'est le rôle du ramasse-miette, mais attention à ne pas laisser de références erronées.

L'héritage en Java

- On indique que la classe fille étend la classe mère : `extends`

```
package coursSSI3.exemples.reutilisation;

public class Vehicule {
 private String marque;
 public void setMarque(String marque) {this.marque=marque;}
 public String getMarque() {return marque;}
 public void avance() {}
}
```

Listing 22 – coursSSI3/exemples/reutilisation/Vehicule.java

```
package coursSSI3.exemples.reutilisation;

public class VehiculeAMoteur extends Vehicule {
 private String typeMoteur; /* Augmentation de l'etat Interne */
 public VehiculeAMoteur(String marque, String typeMoteur) {
  setMarque(marque);
  this.typeMoteur = typeMoteur;
 }

 public void avance() {/* Modification du comportement*/}
 public void demarre() {/* Nouveau comportement */}
}
```

Listing 23 – coursSSI3/exemples/reutilisation/VehiculeAMoteur.java

L'héritage en Java

- Si `extends` n'est pas précisé la classe étend la classe `Object`
- L'héritage multiple est interdit
- La classe fille peut
  - ajouter des variables, des méthodes, des constructeurs.
  - *redéfinir* ou *surcharger* des méthodes

Définition

La redéfinition d'une méthode consiste à définir une méthode ayant la méthode signature qu'une méthode définie dans une classe ancêtre.

L'héritage en Java

- Le principe d'utilisation de l'héritage en Java est le même que pour la conception :
  - Si `B extends A`
  - Alors toute instance `b` de `B` est un (*is a*) `A`
- Par exemple, une Voiture *v* est un Vehicule
- **Il est interdit d'utiliser l'héritage dans un autre contexte**

Les constructeurs et l'héritage

- La classe fille hérite de tous les membres (attributs et méthodes)
- **Attention** en fonction des protections (`private`), il se peut qu'elle ne puisse y accéder
  - (cf. `protected`)
- Les constructeurs ne sont pas hérités
  - Mais il est possible d'appeler ceux de la super classe avec `super()`
  - Il est toujours possible d'appeler un autre constructeur avec `this`
- Attention, les deux instructions précédentes ne peuvent chacune être que la **première instruction** d'un constructeur.

Les constructeurs et l'héritage

- Si ni `this()` ni `super()` ne sont précisés, `super()` est ajouté par défaut.
- La première instruction de tous les constructeurs est donc un appel au constructeur de la super classe.
- Quel est donc le premier constructeur appelé et pourquoi ?

Les constructeurs et l'héritage

```
package coursSSI3.exemples.reutilisation;

public class Animal {
 private String espece="";
 public Animal(String espece)
        {this.espece=espece;}
}
```

Listing 24 – coursSSI3/exemples/reutilisation/Animal.java

```
package coursSSI3.exemples.reutilisation;
public class Chien extends Animal {
        private enum Race {
                BOXER, CANICHE, DOGUE
        };
        private Race race;

        public Chien() {
                super("canide"); /* espece= canide ? */
        }
        public Chien(Race race) {
                this(); /* Que se passe-t-il sans this ? */
                this.race = race;
        }
}
```

Listing 25 – coursSSI3/exemples/reutilisation/Chien.java

Limitations imposées par Java

- Le type de retour d'une méthode surchargée doit être le même que celui de la méthode d'origine
- La nouvelle méthode ne doit pas être moins accessible
  - Par exemple une méthode `public` ne peut pas devenir `private`
  - *Quelle est la raison ?*

Le polymorphisme

Définition

Le polymorphisme est un mécanisme qui permet d'envoyer à plusieurs objets de types différents un même message chacun réagissant de façon propre.

- L'utilisation du polymorphisme est en partie liée à l'héritage mais pas seulement (cf. Interfaces)
- On distinguera donc :

Définition

Le type réel d'un objet qui est celui de l'objet effectivement créé en mémoire.

Définition

Le type déclaré d'un objet qui est celui de la référence à travers laquelle il est manipulé.

- Le type déclaré permet de savoir quel message peuvent être envoyé et le type réel quelle action **sera réellement exécutée**.

Le polymorphisme

```
package coursSSI3.exemples.reutilisation;
public class Polymorphisme {

        public class Animal {
                void crier() {System.out.println("Je␣crie.");}
        }
        class Chien extends Animal{
                void crier() {System.out.println("ouaf␣ouaf");}
        }
        class Chat extends Animal{
                void crier() {System.out.println("miaou␣miaou");}
        }

        public void crier() {
                Animal animaux[] = {new Animal(),
                                new Chien(), new Chat()};
                for(Animal animal :animaux) animal.crier();
        }

        public static void main(String args[]) {
                new Polymorphisme().crier();
        }
}
```

Listing 26 – coursSSI3/exemples/reutilisation/Polymorphisme.java

Le transtypage

- Il possible de forcer le compilateur à considérer un objet comme étant d'un type différent :
  - de son type réél
  - de son type déclaré
- Les seuls cast possibles sont ceux entre classe filles et mères. On distingue :
  - le *upcast* vers la classe mère (Toujours possible)
  - le *downcast* vers la classe (**Attention Danger**, cf. exemple suivant)

Le transtypage

```
package coursSSI3.exemples.reutilisation;

public class Cast {

        public abstract class Animal {
                abstract void crier();
        }
        class Chien extends Animal{
                void mord() {System.out.println("grr grr");}
                void crier() {System.out.println("ouaf ouaf");}
        }
        class Chat extends Animal{
                void crier() {System.out.println("miaou miaou");}
        }

        public void attaque() {
                Animal animaux[] = {new Chat(),
                                    new Chien(), new Chat()};
                ((Chien)animaux[1]).mord();
                /* Erreur a l'execution mais pas a la compilation
                ((Chien)animaux[2]).mord(); */
        }

        public static void main(String args[]) {
                new Cast().attaque();
        }
}
```

Listing 27 – coursSSI3/exemples/reutilisation/Cast.java

Les classes abstraites

- En général la spécialisation d'une classe peut entrainer la redéfinition d'une ou plusieurs méthodes
- Dans certains cas, la définition du corps d'une méthode n'a pas de sens pour la classe mère
  - Par exemple, tous les `Animaux` peuvent `crier()`.
  - Mais on peut définir un cri général...
  - Le cri dépend de la sous-classe (`Chien`, `Chat`, ...)
- Cependant la définition de la méthode doit rester dans la classe mère :
  - Pour garantir la structuration objet
  - Pour permettre le polymorphisme (lié à l'héritage)
- On parle alors de *Classe abstraite*
  - Attention, ces classes ne peuvent être instanciées puisque les définitions de méthodes sont incomplètes.

Les classes abstraites

- En java, les méthodes dont on ne donne pas le corps ainsi que les classes concernées doit être marquées `abstract`
- Les classes qui spécialisent une classe abstraite doivent définir les méthodes abstraites ou être marquée `abstract`

```
package coursSSI3.exemples.reutilisation;

public class ClasseAbstraite {
public abstract class Animal {
        abstract void crier();
}

public abstract class Bovin extends Animal {
        abstract void ruminer();
}

public class Vache extends Bovin {
        void crier() {/* Definition */}
        void ruminer() {/* Definition */}
}
}
```

Listing 28 – coursSSI3/exemples/reutilisation/ClasseAbstraite.java

Les interfaces

- En java la définition d'une classe et l'héritage permettent de définir à la fois
  - l'état interne des instances
  - et le comportement (éventuellement par spécialisation)
- Cependant cela impose, un relation hiérarchique stricte de type "est un"
- Il est parfois utile de pourvoir imposer à des objets instances de classe sans relation d'héritage de vérifier un même comportement.
- Java comme UML propose pour cela la notion d'*Interface*

Les interfaces

Définition

Une interface est un comportement (un ensemble de signatures de méthodes) que des classes peuventt choisir de suivre (on dira d'implanter).

- En java, la déclaration d'une interface se fait avec le mot clé `interface` de la même façon que pour une classe qui ne comporterait que des méthodes abstraites et publiques.
  - L'héritage même multiples est possible entre interfaces

Les interfaces

```
package coursSSI3.exemples.reutilisation;

public class Interface {
        public interface ItrucEmpruntable {
                public void emprunte();
                public void rendu();
        }

        public class Voiture implements ItrucEmpruntable{
                boolean disponible = true;
                public void emprunte() {disponible = false;}
                public void rendu() {disponible = true;}
        }

        public class Stylo implements ItrucEmpruntable {
                boolean emprunte = false;
                public void emprunte() {emprunte = true;}
                public void rendu() {emprunte = false;}
        }

}
```

Listing 29 – coursSSI3/exemples/reutilisation/Interface.java