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
}
```

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() {       }
}
```

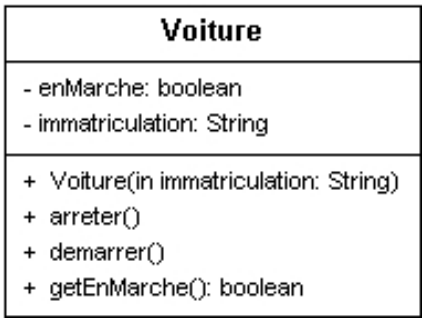


FIGURE – La classe `Voiture`

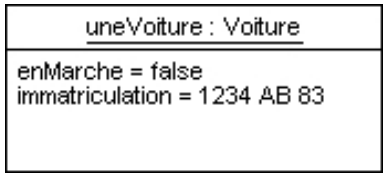


FIGURE – L'instance `uneVoiture`

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)

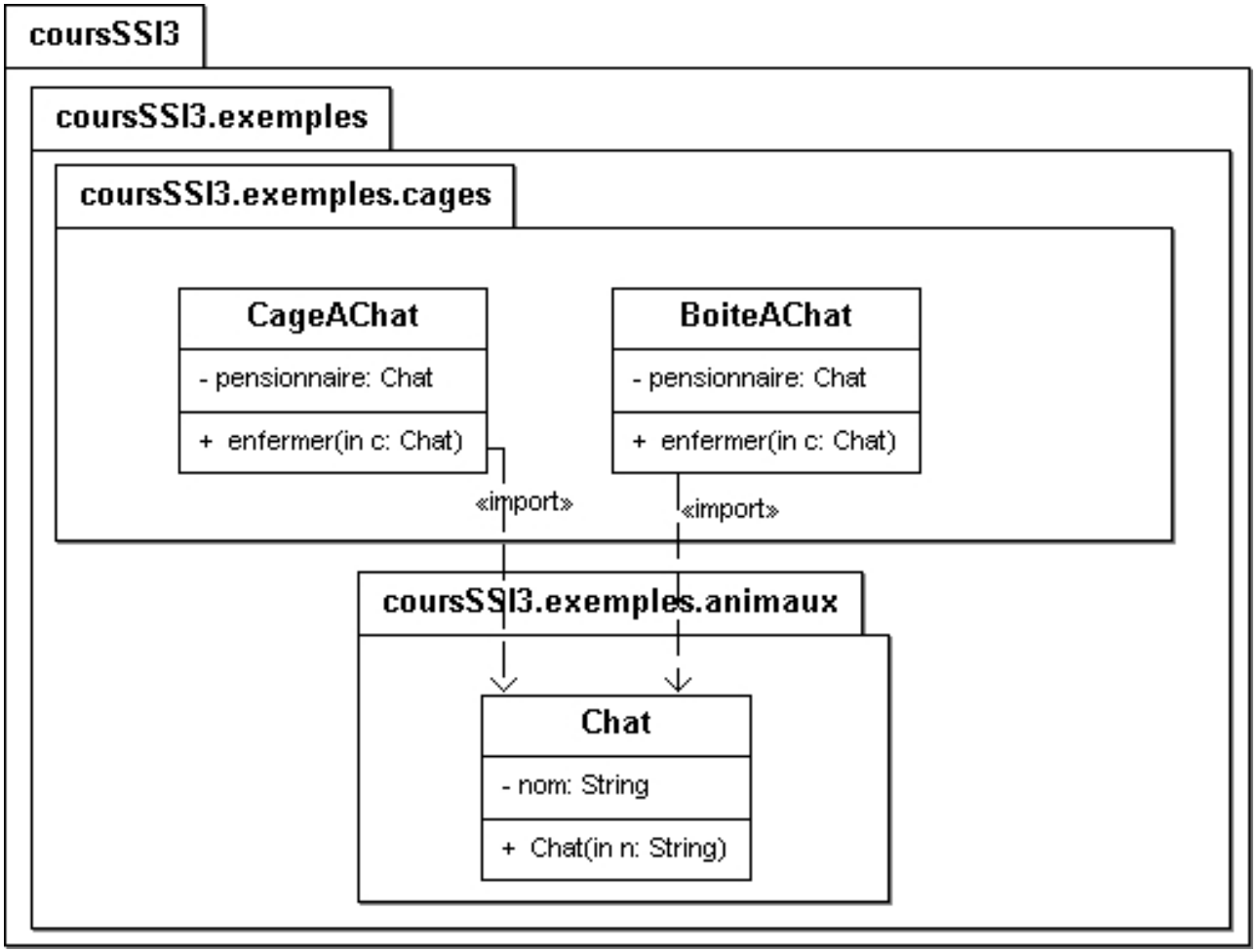


FIGURE – Les paquetages en UML

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)

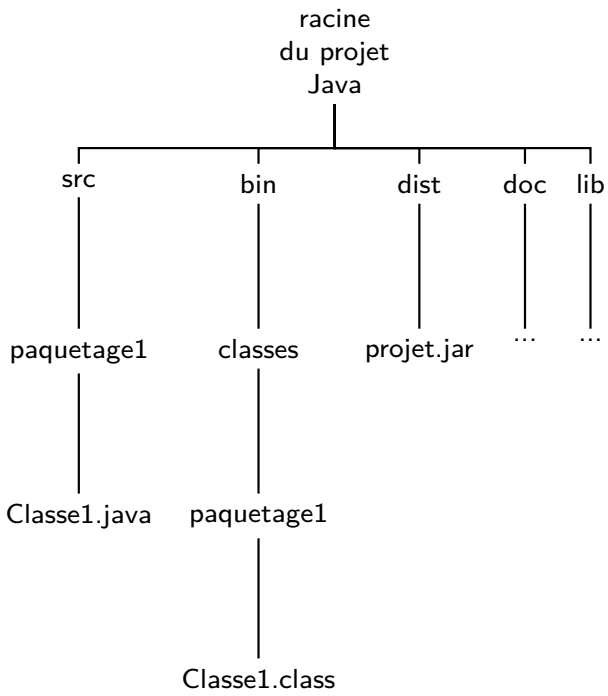
Les conventions - Documentation (2/3)

- 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

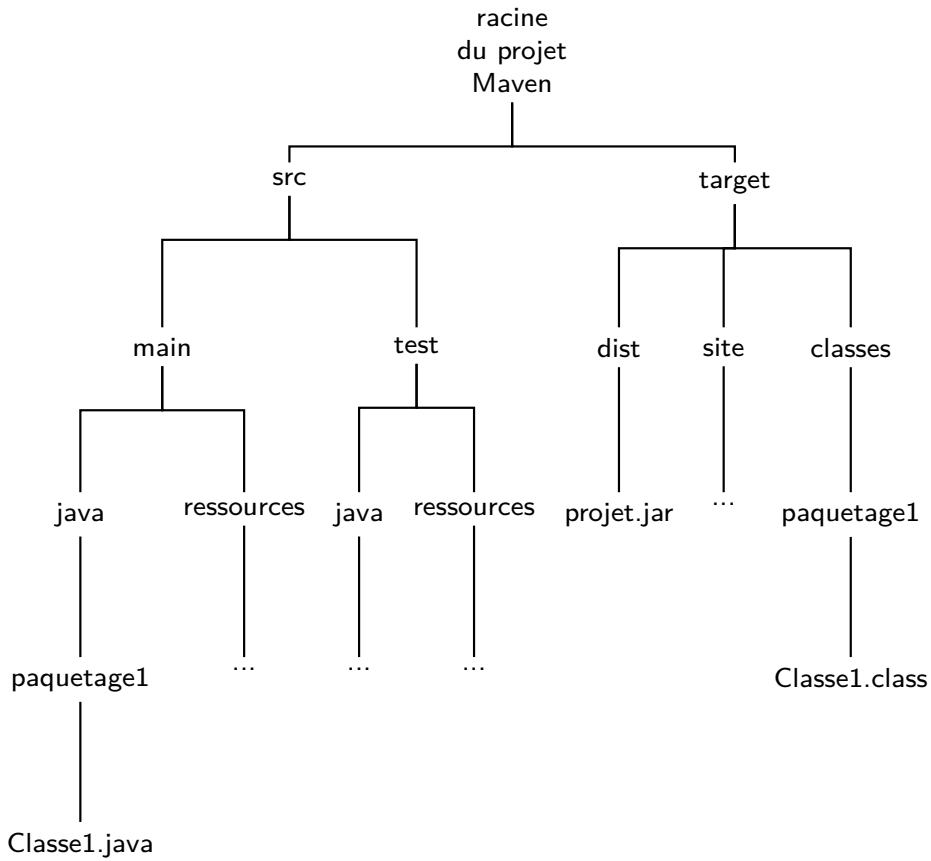
```
/** Cette classe est un exemple de documentation simple
 *  @see Chien
 *  @version 2.0
 *  @since 1.0
 *  @author Emmanuel Bruno
 */
public class TrucDocumente {
        /**
         * Cette methode est un exemple
          * @param i Un entier quelconque
          * @return le double du paramètre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

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();
        }
}
```

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

L'interet des collections

- Lors d'un développement, l'un des points importants est le choix de structures de données adaptées.
- Il est donc souvent nécessaire de développer les mêmes solutions pour
  - Stocker des objets, pour les parcourir et les retrouver (avec ou sans clé)
  - Sous la forme de tableaux, listes, arbres, tables de hachages, ...

Les classes historiques

- Les jdk 1.1 proposaient les classes : `Vector` et `HashTable`
- Ces classes ont été rendues obsolètes
- Elles existent toujours et peuvent être rencontrées dans des programmes existants

Les collections

- En java, une collection est un objet qui permet de contenir des références vers d'autres objets
- Nous avons déjà vu un type de collection : les tableaux
- Le jdk propose
  - Des Interfaces qui définissent les grands types de collections
  - Des classes abstraites qui implantent partiellement ces interfaces
  - Des classes concrètes qui précisent les implantations
- Elles sont définies dans le paquetage `java.util`
- Un mécanisme de parcours général est proposé : *les itérateurs*
- Et des outils pour trier et rechercher

Collections et indexation

- On distinguera :
  - Les collections au sens large décrites dans l'interface `Collection`
  - Et les collections indexées décrites dans l'interface `Map`
    - On associe à un objet une clé (un autre objet)
    - On peut alors indexer et rechercher par clé.
- Dans tous les cas, on pourra
  - ajouter un objet (`add()`)
  - parcourir la collection (voire obtenir un objet (`get()`))

Une hiérarchie d'Interfaces

- On distiguera parmis les collections (`Collection`)
  - Les ensembles (`Set`)
    - Les ensembles triés (`SortedSet`)
  - Les listes (`List`)
  - Les files (`Queue`)
- Et parmis les maps (`Map`) les maps triées (`SortedMap`)

Des classes abtraites

- Un ensemble de classes abstraites réalisent ces interfaces
  - Elles implantent les parties communes
    - `AbstractCollection`, `AbstractList`, `AbstractQueue`, `AbstractSequentialList`, `AbstractSet`, …
    - `AbstractMap`
- Ces classes abstraites (et donc les interfaces correspondantes) sont réalisées par des classes concrètes
  - `ArrayList`, `TreeSet`, …
  - `HashMap`, `TreeMap`, …

Algorithmique et structure de données

- Les structures de données classiques sont disponibles et utilisées pour les implantations concrètes
- Elles sont organisées dans une hiérarchie de classes et implantent des interfaces communes
- Par exemple, il existe (parmis d'autres) :
  - Des tableaux de taille variable : `ArrayList`
    - héritent de `java.util.AbstractList` et de `AbstractCollection`
    - implantent les interfaces `Collection` et `List`
  - Des listes chaînées : `LinkedList`
  - Des tables de hachages : `HashSet` et `HashMap`
  - Des arbres à balance équilibrés : `TreeSet` et `TreeMap`

Administration des collections

- L'administration des instances des collections est assurée par des méthodes statiques des classes :
  - `Collections`
    - pour trier et convertir des collections
    - pour rechercher efficacement
  - `Arrays`

Un premier exemple

- Création d'une liste de chaînes de caractères sous forme d'un `ArrayList`
- Tri et affichage

```
package coursSSI3.exemples.collections;
import java.util.*;
public class PremierExemple {
        public static void main(String[] args) {
                List l = new ArrayList();
                l.add("Medor");
                l.add("Rex");
                l.add("Brutus");
                Collections.sort(l);
                System.out.println(l);
        }
}
```

Listing 30 – coursSSI3/exemples/collections/PremierExemple.java

L'interface Collection

- La racine de la hiérarchie des collections
  - `boolean add(E o)`
  - `boolean addAll(Collection<? extends E> c)`
  - `void clear()`
  - `boolean contains(Object o)`
  - `boolean containsAll(Collection<?> c)`
  - `boolean equals(Object o)`
  - `int hashCode()`
  - `boolean isEmpty()`
  - `Iterator<E> iterator()`
  - `boolean remove(Object o)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `int size()`
  - `Object[] toArray()`
  - `<T> T[] toArray(T[] a)`

Les standards et les exceptions

- Les constructeurs sans paramètres et avec une `Collection` en paramètre sont “toujours” disponibles
  - Cela ne peut pas être garanti (pas de constructeur dans les interfaces)
  - Cela facilite les conversions entre les implantations différentes
- Toutes les méthodes de l'interface sont implantées (c'est obligatoire) mais :
  - Pour certaines spécialisations certaines méthodes n'ont pas de sens
    - Exemple : collection en lecture seule (`add()`, `put()`, ...)
  - L'implantation consiste alors à retourner une *exception* : `UnsupportedOperationException`

List et ArrayList

- L'interface `List`
  - Une collection ordonnées (une séquence)
  - L'utilisateur controle la position d'insertion des éléments
  - L'utilisateur accède à cet élément par son index (un entier)
- La classe `ArrayList`
  - Une implantation redimensionnable de l'interface `List`
  - Toutes les méthodes sont implantées
  - Il est possible de contrôler la taille du tableau utilisé en interne

Des collections “génériques” ?

- Les collections représentent des ensembles d'objets (cf. type de retour de l'interface)
- Un objet ou un ensemble d'objets extraits sont donc du type `Object`
- La conséquence est donc que les objets extraits doivent être transtypés
- De plus les fonctions prennent en paramètres des instances de `Object`, il est donc difficile de contrôler la consistance.

Un exemple de transtypage

```
package coursSSI3.exemples.collections;

import java.util.*;
import coursSSI3.exemples.animaux.*;
public class Transtypepage {
        public static void main(String[] args) {
                List l = new ArrayList();
                l.add(new Chien());
                l.add(new Chien());
                Chien c = (Chien)l.get(0);
                c.aboyer();
                ((Chien)l.get(1)).aboyer();

                // que faire ?
                List l2 = new ArrayList();
                l2.add(new Chien());
                l2.add(new Chat());
        }
}
```

Listing 31 – coursSSI3/exemples/collections/Transtypepage.java

Le parcours d'une collection (1/2)

- Les collections indexées (Listes, ...) peuvent être parcourues avec une boucle
  - Cette méthode n'est pas bonne pour l'évolutivité, elle supprime l'encapsulation des collections
- Java introduit la notion d'itérateur

Définition

Un itérateur est une instance de la classe `Iterator` qui permet d'énumérer les éléments d'une collection.

- Toutes les collections possèdent la méthode `iterator()` qui retourne un itérateur.
  - Cette itérateur peut être spécialisé en fonction des sous-classes de `Collection` (Ex : `ListIterator`).
- Un iterateur permet de modifier la collection en cours de parcours.

Le parcours d'une collection - Exemple (2/2)

```
package coursSSI3.exemples.collections;

import java.util.*;

import coursSSI3.exemples.animaux.*;

public class Parcours {
 public static void main(String[] args) {
  List listeDeChiens = new ArrayList();
  listeDeChiens.add(new Chien());listeDeChiens.add(new Chien());

  for (int i=0;i<listeDeChiens.size();i++)
   System.out.println((Chien)listeDeChiens.get(i));

  Iterator itDeChiens = listeDeChiens.iterator();
  while (itDeChiens.hasNext())
   ((Chien)itDeChiens.next()).aboyer();
  }
}
```

Listing 32 – coursSSI3/exemples/collections/Parcours.java

La conversion vers un tableau

- `toArray()` : **Attention au type de retour**
- Il est possible de passer un tableau en paramètre
  - **Il ne suffit pas de transtyper le résultat** (pourquoi ?)
  - Si celui-ci est trop petit, un tableau du même type est créé.
- Collection vide implique tableau vide (et non `null`)

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class ToArray {
 public static void main(String[] args) {
  List listeDeChiens = new ArrayList();
  listeDeChiens.add(new Chien()); listeDeChiens.add(new Chien());
  Object[] tObjets = listeDeChiens.toArray();
  Chien[] desChiens = new Chien[listeDeChiens.size()];
  listeDeChiens.toArray(desChiens);
  for(Object c:tObjets){((Chien)c).aboyer();};
  for(Object c:listeDeChiens){((Chien)c).aboyer();};
  for(Chien c:desChiens){c.aboyer();};
 }
}
```

Listing 33 – coursSSI3/exemples/collections/ToArray.java

Des collections “génériques” avec Java 5 (1/2)

- La boucle *foreach* (cf. tableaux) s’applique aux collections

Définition

Un type générique (en anglais *generic* ou aussi type paramétré) permet de définir une classe ou une interface qui aura des types différents (grace à des types passés en parmètres) lors de l’instanciation.

`http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html`

- Définir des classes spécifiques au type indiqué. On peut donc définir une “collection de `Chiens`”.
- Vérifier les paramètres des méthodes et convertir les types de retour.
- Les collections ne traitent que des objets
  - Pour les types primitifs, il faut utiliser les classes enveloppantes
  - Java 5 propose l’*autoboxing* et l’*autounboxing* qui convertit automatiquement les primitifs.

Les "génériques" de Java 5 (2/3)

```
package coursSSI3.exemples.collections;

public class Couple<T1, T2> {
          final T1 e1;
          final T2 e2;

          public Couple(T1 p1, T2 p2) {
            e1 = p1; e2 = p2;
          }
        }
```

Listing 34 – coursSSI3/exemples/collections/Couple.java

```
package coursSSI3.exemples.collections;

import coursSSI3.exemples.animaux.*;
public class Generique {


        public static void main(String[] args) {
                Chien unChien = new Chien();
                Chat unChat = new Chat();
                Couple<Chien,Chat> c =
                        new Couple<Chien,Chat>(unChien, unChat);
        }

}
```

Des collections “génériques” avec Java 5 - Exemple (2/3)

```
package coursSSI3.exemples.collections;

import java.util.*;

import coursSSI3.exemples.animaux.*;

public class Java5 {
 public static void main(String[] args) {
  List<Chien> listeDeChiens = new ArrayList<Chien>();
  listeDeChiens.add(new Chien());listeDeChiens.add(new Chien());
  // Provoque une erreur de compilation
     //listeDeChiens.add(new Chat());
  for(Chien c :listeDeChiens) c.aboyer();

  List<Integer> l = new ArrayList<Integer>();
  l.add(new Integer(3));
  int i = l.get(0);

  l.add(4);
  i = l.get(1);
  }
}
```

Listing 36 – coursSSI3/exemples/collections/Java5.java

L'utilisation des `Map`

- Une `Map` est utilisée pour associer une clé à une valeur
- Une clé est un objet qui est associé à une et une seule valeur
- Une clé doit être unique, plus précisément la valeur retournée par la méthode `equals()` de deux clés doit être différente
- En Java, il s'agit d'une interface implantée en particulier par :
  - `HashMap` : Une table de hachage (accès en $\theta(1)$)
  - `TreeMap` : Un arbre à balance équilibré qui ordonne les valeurs en fonction des clés :
    - L'accès est en $\theta(n)$ (n le nombre d'éléments)
    - Les objets doivent implanter l'interface `Comparable`
    - On peux aussi fournir un comparateur (Instance de `Comparator`) externe
- Utilisation classique :
  - ajouter et enlever des entrées
  - retrouver un objet par sa clé
  - retrouver la ou les clés associées à une valeur

L'interface Map<K,V>

- `void clear()`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `Set<Map.Entry<K,V>> entrySet()`
- `boolean equals(Object o)`
- `V get(Object key)`
- `int hashCode()`
- `boolean isEmpty()`
- `Set<K> keySet()`
- `V put(K key, V value)`
- `void putAll(Map<? extends K,? extends V> t)`
- `V remove(Object key)`
- `int size()`
- `Collection<V> values()`

Fonctionnement de `Map`

- Par défaut une `Map` stocke des instances de `Object`
- Les génériques sont utilisables (et conseillés)
- Les couples clé/valeur sont manipulés *via* l'interface `Map.Entry`
  - Trois méthodes : `getKey()`, `getValue()`, `setValue()`
  - La méthode `entrySet()` de `Map` retourne un ensemble (`Set`) de `Map.Entry`
- **Attention**, pour garantir le bon fonctionnement une clé ne peut être remplacée que par une clé égale (cf. `equals()`)
  - Un conseil : faire un ajout et une supression.

Exemple Map

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class MapSimple {
 public static void main(String[] args) {
  Map<String, Chien> m = new HashMap<String, Chien>();
  //Map<String, Chien> m = new TreeMap<String, Chien>();
  //SortedMap<String, Chien> m = new TreeMap<String, Chien>();

  m.put("Ch3",new Chien("Medor"));
  m.put("Ch1",new Chien("Rex"));
  m.put("Ch2",new Chien("Medor"));
  m.put("Ch2",new Chien("Brutus"));
  System.out.println("Le chien Ch2 est "+m.get("Ch2").nom);
 }

}
```

Listing 37 – coursSSI3/exemples/collections/MapSimple.java

Parcourir une Map

- Pour parcourir une Map
  - On récupère l'ensemble des clés (`keySet()`)
  - On parcours cet ensemble (*Itérateur* ou *for each*)
  - Pour chaque entrée on récupère éventuellement la clé (`getKey()`) et la référence vers la valeur (`getValue()`)
  - `values()` retourne la collection correspondante

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class ParcoursMap {
 public static void main(String[] args) {
  Map<String, Chien> m = new HashMap<String, Chien>();
  m.put("Ch3",new Chien("Medor"));m.put("Ch1",new Chien("Rex"));
  m.put("Ch2",new Chien("Medor"));
  Set<Map.Entry<String, Chien>> setChiens = m.entrySet();
  for(Map.Entry<String, Chien> uneEntree : setChiens)
   { System.out.print(uneEntree.getKey()+ "␣:" );
          uneEntree.getValue().aboyer(); }
 }
}
```

Listing 38 – coursSSI3/exemples/collections/ParcoursMap.java

Les critères de choix de classe

- Un des objectifs des programmes Java est la réutilisabilité
  - Il faut choisir la classe (ou l'interface) la plus adaptée
  - Mais le programme doit être utilisable dans un nombre de cas le plus large possible
- On choisira donc plutot :
  1. De manipuler les objets via des interfaces les plus générales possibles pour les paramètres
  2. Et il faudrait choisir des types de retour des objets instances des classes les plus spécifiques possibles (offrant le plus de fonctions)
  3. Cependant, ce dernier choix provoquerait des problèmes en cas de changement d'implantation ; on choisira donc aussi des interfaces plus générales.

Utilitaires

- Les classes `Collections` et `Arrays` propose des méthodes de classe pour traitre des collections et des tableaux.
- Elles permettent en particulier de trier et de rechercher dans des listes, de faire des copies, ...
- Nous avons déjà vu que l'on pouvait trier une collection de `String`, en réalité pour qu'une collection puisse être triée, ses élements doivent implanter l'interface `Comparable`.
  - `int compareTo(T o)`
    Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Attention à la consistance entre `Comparable()` et `equals()`.

Trier une collection (1/3)

```
package coursSSI3.exemples.animaux;

public class Animal implements Comparable<Animal> {
        public final String nom;
        public final int age;
        public final int poids;

        public Animal() {this("",-1,-1);}
        public Animal(String nom) {this(nom, -1, -1);}
        public Animal(String nom, int age, int poids)
        {this.nom=nom;this.age = age;this.poids=poids;}

        public int compareTo(Animal o) {
                        return (age - ((Animal) o).age);
        }
        public boolean equals(Animal o) {
                return age==o.age;
        }
        public String toString() {
                return nom+" "+age+" an(s)"+" "+poids+" kg.";}
}
```

Listing 39 – coursSSI3/exemples/animaux/Animal.java

Trier une collection (2/3)

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class Utilitaire {
 public static void main(String[] args) {
  List<Animal> l = new ArrayList<Animal>();
  //Nom, age, poids
  l.add(new Chien("Medor",2,5));
  l.add(new Chat("Figaro",3,2));
  l.add(new Chien("Brutus",1,15));

  System.out.println(l);
  Collections.sort(l); //Trier par age
  System.out.println(l);

  //Trier par poids ?
  Collections.sort(l,new ComparateurPoidsAnimal());
  System.out.println(l);
 }
}
```

Listing 40 – coursSSI3/exemples/collections/Utilitaire.java

Trier une collection (3/3)

```
package coursSSI3.exemples.collections;

import java.util.*;
import coursSSI3.exemples.animaux.Animal;
public class ComparateurPoidsAnimal
  implements Comparator<Animal> {
        public int compare(Animal arg0, Animal arg1) {
                return arg0.poids-arg1.poids;
        }

}
```

Listing 41 – coursSSI3/exemples/collections/ComparateurPoidsAnimal.java

Recherche dans une collection

```
package coursSSI3.exemples.collections;

import java.util.*;
import coursSSI3.exemples.animaux.*;

public class Recherche {
 public static void main(String[] args) {
  List<Animal> l = new ArrayList<Animal>();
  //Nom, age, poids
  l.add(new Chien("Medor",2,5));
  Chat figaro;
  l.add(figaro=new Chat("Figaro",3,2));
  l.add(new Chien("Brutus",1,15));

  Collections.sort(l);
  int positionFigaroAge = Collections.binarySearch(l, figaro);

  Collections.sort(l, new ComparateurPoidsAnimal());
  int positionFigaroPoids = Collections.binarySearch(l, figaro,
    new ComparateurPoidsAnimal());
  System.out.println(positionFigaroAge+ " "+positionFigaroPoids);
 }
}
```

Listing 42 – coursSSI3/exemples/collections/Recherche.java

Arguments variables

- Avec Java 5, il n'est plus nécessaire d'utiliser une collection pour passer un ensemble d'arguments du même type à ne méthode

```
package coursSSI3.exemples.collections;

public class VarArgs {
        public static int somme(int ... entiers) {
                int total = 0;
                for (int e:entiers) total+=e;
                return total;
        }

        public static void main(String[] args) {
                System.out.println("1+2="+somme(1,2)+
                                "␣et␣1+2+3+4="+somme(1,2,3,4));
        }
}
```

Listing 43 – coursSSI3/exemples/collections/VarArgs.java

EnumSet

Special-purpose Set and Map implementations are provided for use with enums :

- `EnumSet` - a high-performance Set implementation backed by a bit-vector. All elements of each EnumSet instance must be elements of a single enum type.
- `EnumMap` - a high-performance Map implementation backed by an array. All keys in each EnumMap instance must be elements of a single enum type.

Le contexte

- Dans notre contexte sécuriser un programme c'est :
  - Prévoir les erreurs
  - les détecter
  - et réagir
- L'un des objectifs du le langage Java est la sécurité
  - Il propose un mécanisme dédié : les exceptions

En pratique

- Pour les langages qui ne proposent pas de traitement spécifique des erreurs :
  - Les cas d'erreur doivent être prévus (analyse du code, de l'utilisation, ...)
  - Des procédures de test sont mises en place (manuelles et automatiques)
  - Des réactions aux erreurs sont prévues
    - **Traiter** : Avertissement, correction dynamique, arrêt contrôlé, ...
    - Mais aussi **remonter l'erreur** vers le code appelant.
- Le code de test est confondu avec celui de l'application
  - Difficile à lire
  - Difficile à maintenir
  - Qui traite l'erreur l'appelant ou l'appelé ?

Une classe, des tests et des réactions

```
package coursSSI3.exemples.exceptions;
import coursSSI3.exemples.animaux.Chien;

public class Traineau {
        public final int capacite;
        protected int occupation = 0;
        protected Chien[] contenu;
        public Traineau(int capacite) {
                this.capacite = capacite;
                contenu = new Chien[capacite];
        }
        public boolean estComplet() {
                return occupation == capacite;}
        public boolean estVide() {return occupation == 0;}

        public void ajouter(Chien c) {
                if (!estComplet()) contenu[occupation++] = c;
                else System.out.println("Traineau complet !");
        }
        public void liberer() {
                if (!estVide()) contenu[--occupation] = null;
                else System.out.println("Traineau vide !");
        }
}
```

Listing 44 – coursSSI3/exemples/exceptions/Traineau.java

Un programme principal incomplet

```
package coursSSI3.exemples.exceptions;
import coursSSI3.exemples.animaux.*;

public class Test {
        public static void main(String[] args) {
                Traineau t;
                int nbChiens;
                // Creation d'un traineau de taille donnee
                t = new Traineau(Integer.parseInt(args[0]));
                // Nb de chiens
                nbChiens = Integer.parseInt(args[1]);

                int i = nbChiens;
                while (i--> 0 && !t.estComplet())
                        t.ajouter(new Chien());
                i = nbChiens;

                System.out.println("occupation : "
                                +100*nbChiens/t.capacite);

                while (i--> 0 && !t.estVide())
                        t.liberer();
        }
}
```

Listing 45 – coursSSI3/exemples/exceptions/Test.java

Avec Java

- Dans le programme précédent
  - Le programme teste et réagit à la même erreur plusieurs endroits
  - que se passe-t-il si :
    - Il n'y a pas assez de paramètres
    - Les paramètres (des chaînes) ne représentent pas nombres
    - Le premier paramètre vaut 0
- Pour traiter cela, java propose :
  - un mécanisme le bloc `try…catch…finally`
    - On exécute et on réagit éventuellement : mécanisme implicite
    - Séparation du code et du traitement des erreurs
    - Erreurs classiques : Division par zéro, dépassement de tableaux, ...
    - Extensibilité des erreurs
  - une hiérarchie de classe pour les erreurs

Avec Java

- Toutes les erreurs potentielles doivent être traitées par l'appelant
- Sauf celles de type RuntimeException :
  [arrows=<-, levelsep=25pt, nodesep=5pt]Object Throwable Error Exception RuntimeException ArithmeticException IllegalArgumentException NumberFormatException IndexOutOfBoundsException ArrayIndexOutOfBoundsException

Un programme principal amélioré

```
package coursSSI3.exemples.exceptions;

import coursSSI3.exemples.animaux.*;

public class TestErreur {
 public static void main(String[] args) {
  Traineau t = null;
  int nbChiens = 0;
  try {
    t = new Traineau(Integer.parseInt(args[0]));
    nbChiens = Integer.parseInt(args[1]);

    int i = nbChiens;
    while (i--> 0 && !t.estComplet()) t.ajouter(new Chien());
    i = nbChiens;
    while (i--> 0 && !t.estVide()) t.liberer();
  } catch (ArrayIndexOutOfBoundsException e) {
          System.out.println(
        "Test exception <TailleTraineau> <nbCheins>");
  } catch (NumberFormatException e) {
          System.out.println("Parametres non numeriques !"); }
 }
}
```

Listing 46 – coursSSI3/exemples/exceptions/TestErreur.java

La remontée d'erreur

- La première réaction possible a une erreur c'est de la transmettre au programme appelant
- En Java, on utilise l'instruction **`throw`** *`exception`*
  - dans le corps d'un programme (par exemple dans un `catch`)
- C'est le comportement par défaut pour les `RuntimeException`
- Toute erreur est interceptée au plus tard par le programme principal encadré par défaut par :

```
try
...
catch(Throwable t)
System.err.println(t.printStackTrace());
```

Une classe qui remonte des exceptions

```
package coursSSI3.exemples.exceptions;

import coursSSI3.exemples.animaux.Chien;

public class TraineauThrows extends Traineau {

 public TraineauThrows(int capacite) {super(capacite);}

 public int getRatioOccupation() {
  try {
        return 100*occupation/capacite;
  } catch (ArithmeticException e) {
        return -1;}
 }

 public int getRatioOccupationSur() {
  try {
          return 100*occupation/capacite;
  } catch (ArithmeticException e) {
        System.out.println("Ratio non calculable");
        throw e; }
 }
}
```

Listing 47 – coursSSI3/exemples/exceptions/TraineauThrows.java

La création d'une nouvelle exception

- Pour traiter les cas d'erreur particulier à une application :
  - On étend la classe `Exception`
  - Ces erreurs devront être traitées par l'utilisateur
- Quand une méthode peut émettre une erreur, on l'indique dans la déclaration avec la directive `throws`
- Pour l'émettre, on crée une instance qui est émise pas `throw`

Des exceptions personnelles

```
package coursSSI3.exemples.exceptions;

public class TraineauVideException extends Exception {

}
```

Listing 48 – coursSSI3/exemples/exceptions/TraineauVideException.java

```
package coursSSI3.exemples.exceptions;

public class TraineauPleinException extends Exception {

}
```

Listing 49 – coursSSI3/exemples/exceptions/TraineauPleinException.java

Une classe qui émet ses exceptions

```
package coursSSI3.exemples.exceptions;

import coursSSI3.exemples.animaux.Chien;

public class TraineauErreur extends TraineauThrows {

        public TraineauErreur(int capacite) {
                super(capacite);
        }

        public void ajouterSur(Chien c)
          throws TraineauPleinException {
                if (!estComplet()) contenu[occupation++] = c;
                else throw new TraineauPleinException();
        }

        public void libererSur()
          throws TraineauVideException {
                if (!estVide()) contenu[--occupation] = null;
                else throw new TraineauVideException();
        }

}
```

Listing 50 – coursSSI3/exemples/exceptions/TraineauErreur.java

Le programme principal fini

```
package coursSSI3.exemples.exceptions;
import coursSSI3.exemples.animaux.*;
public class TestErreur2 {
  public static void main(String[] args) {
    TraineauErreur t = null;  int nbChiens = 0;
    try {t = new TraineauErreur(Integer.parseInt(args[0]));
      nbChiens = Integer.parseInt(args[1]); // Nb de chiens
      int i = nbChiens;try {while (i--> 0)
          t.ajouterSur(new Chien());
      } catch (TraineauPleinException e) {
        System.out.println("Le traineau est trop petit !");}

      System.out.println("occup.: "+t.getRatioOccupationSur());

      i = nbChiens;try {while (i--> 0 && !t.estVide())
          t.libererSur();
      } catch (TraineauVideException e) {
        System.out.println("Il n'y a plus de chiens !");}

    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("TestExcep <TTraineau> <nbChiens>");
    } catch (NumberFormatException e) {
      System.out.println("Parametres non numeriques !");
    }
  }
}
```

Listing 51 – coursSSI3/exemples/exceptions/TestErreur2.java

La programmation assertionnelle

- Dans certains, cas le traitement de erreur peut être utilisé pour vérifier que le programme fonctionne dans un état prévu :
  - On vérifie que les variables restent dans des domaines précis
  - C'est lourd à mettre en place et à maintenir
- Java 5 propose les assertions
  - Cela permet de vérifier dynamiquement que des conditions sont toujours vraies
  - Quand ce n'est pas le cas, une exception est levée

L'utilisation des assertions

```
package coursSSI3.exemples.exceptions;

public class VehiculeAvecAssertion {
        int vitesse=0; /* la vitesse en km/s */

        public static void main(String args[]) {
                VehiculeAvecAssertion v =
                        new VehiculeAvecAssertion();

                v.vitesse = -3;
                v.vitesse = 400000;

                assert v.vitesse >= 0 && v.vitesse <= 300000;
        }

}
```

Listing 52 – coursSSI3/exemples/exceptions/VehiculeAvecAssertion.java

Utilisation des Flots (streams)

- En Java, la communication entre le programme et les échanges de données entre un programme et l'extérieur (autre programme, fichier ou application réseau, ...) sont réalisées à travers un " flot " de données
- Un flot permet de transporter **séquentiellement** des données. Les données sont transportées une par une (ou bloc) et dans l'ordre
- Le cycle d'utilisation d'un flot de données est le suivant :
  1. Ouverture du flot
  2. Tant qu'il y a des données à lire (ou à écrire), on lit (ou on écrit) la donnée suivante dans le flot
  3. Fermeture du flot

Sources ou destinations de flots

- Fichier
- Socket pour échanger des données sur un réseau
- Données de grandes tailles dans une base de données (blob) (images, par exemple)
- Pipe entre 2 processus
- Tableau d'octets (en mémoire)
- Chaîne de caractères
- URL (adresse Web)
- ...

Le paquetage `java.io`

- La plupart des classes liées au E/S sont définies dans le paquetage `java.io`
- Il prend en compte un grand nombre de flots :
  - On distingue deux grands types de flots :
    - Les octets
    - Les caractères
  - différentes sources et destinations
  - Il est possible de *décorer* les flots :
    - le modifier au fur et à mesure
- Le nombre de classes de ce paquetage est important, il faut regarder la javadoc.

Les grands types de flots

- Les *flots d'octets* servent à lire ou écrire des octets " bruts ", qui représentent des données codées, manipulées par un programme
- Les *flots de caractères* servent à lire ou écrire des données qui représentent des caractères lisibles par un homme (codés en Unicode)

Les types de classes

- Pour les deux grands type de flots (octets et caractères), on distingue :
  - Les classes associées à une source ou une destination “ concrète ”
    - `FileReader` pour lire un fichier
  - Les classes qui permettent de “ décorer ” un autre flot
    - `BufferedReader` qui ajoute un tampon (buffer) pour lire un flot de caractères

La décoration des flots

- Les fonctionnalités de base d'un flot sont la lecture ou l'écriture (méthodes `read` ou `write`)
- Selon les besoins, on peut lui ajouter d'autres fonctionnalités (appelées décorations) :
  - utilisation d'un buffer pour réduire les lectures ou écritures réelles
  - Codage ou décodage des données manipulées
  - Compression ou décompression de ces données
  - etc...

Les classes de base du paquetage `java.io`

- `InputStream` (Lecture d'octets)
- `OutputStream` (Ecriture d'octets)
- `Reader` (Lecture de caractères Unicode)
- `Writer` (Ecriture de caractères Unicode)
- `File` (Manipulation de noms de fichiers et de répertoires)
- `StreamTokenizer` (Segmentation lexicale d'un flot d'entrée)

Les sources et destinations concrètes

- Fichiers :
  - `File{In|Out}putStream`
  - `File{Reader|Writer}`
- Tableaux
  - `ByteArray{In|Out}putStream`
  - `CharArray{Reader|Writer}`
- Chaînes de caractères
  - `String{Reader|Writer}`

Les décorateurs

- Pour ajouter un tampon sur une entrée/sortie :
  - `Buffered{In|Out}putStream`
  - Buffered{Reader|Writer}
- Pour lire et écrire des types primitifs sous une forme binaire :
  - `Data{In|Out}putStream`
- Pour compter les lignes lues :
  - `LineNumberReader`
- Pour écrire sous forme de chaînes de caractères :
  - `PrintStream`
  - `PrintWriter`
- Pour permettre de remettre un caractère lu dans le flot :
  - `PushbackInputStream`
  - `PushbackReader`

La lecture et l'écriture de flots d'octets

Classe d'entrée	**Classe de sortie**	**Utilisation**
`InputStream`	`OutputStream`	Classes abstraites de base pour les lecture et écriture d'un flot de données
`FilterInputStream`	`FilterOutputStream`	Classe mère des classes qui ajoutent des fonctionnalités à Input/OutputStream
`BufferedInputStream`	`BufferedOutputStream`	Lecture et écriture avec buffer
`DataInputStream`	`DataOutputStream`	Lecture et écriture des types primitifs
`FileInputStream`	`FileOutputStream`	Lecture et écriture d'un fichier
	`PrintStream`	Possède les méthodes `print()`, `println()` utilisées par `System.out`

La classe `InputStream`

- Classe abstraite
- C'est la racine des classes qui concernent la lecture d'octets depuis un flot de données
- Elle fixe les méthodes de base
- Elle possède un constructeur sans paramètre
- La lecture d'un flot d'octets avec `InputStream` :
  - `FilterInputStream` (Décorateur - Doit être sousclassée)
    - `BufferedInputStream` (Entrées bufférisées)
    - `DataInputStream` (Lecture des types primitifs)
  - `FileInputStream` (Lecture des octets d'un fichier)
  - `ObjectInputStream` (Lecture d'un objet sérialisé)

Méthodes de la classe InputStream

- Interface publique de cette classe :
  - `abstract int read() throws IOException`
  - `int read(byte[] b) throws IOException`
  - `int read(byte[] b, int début, int nb) throws IOException`
  - `long skip(long n) throws IOException`
  - `int available() throws IOException`
  - `void close() throws IOException`
  - `synchronized void mark(int nbOctetsLimite)`
  - `synchronized void reset() throws IOException`
  - `public boolean markSupported()`
  - La lecture est bloquante

Les sous-classes de InputStream

- InputStream
  - FileInputStream
  - PipedInputStream
  - FilterInputStream
    - LineNumberInputStream
    - DataInputStream
    - BufferedInputStream
    - PushbackInputStream
  - ByteArrayInputStream
  - SequenceInputStream
  - StringBufferInputStream
  - ObjectInputStream

Un exemple : lire des octets dans un fichier

- Cet exemple utilise
  - `FileInputStream` : fichier source
  - `BufferedInputStream` : décorateur d'ajout d'un buffer
  - `DataInputStream` : décorateur de lecture des primitifs
- En général, on n'utilise que le flot décoré (pas besoin des variables intermédiaires)

```
package coursSSI3.exemples.es;
import java.io.*;
public class Decorateur {
public static void main(String[] args) throws IOException {
  FileInputStream fis = new FileInputStream("fichier");
  BufferedInputStream bis = new BufferedInputStream(fis);
  DataInputStream dis = new DataInputStream(bis);
  double d = dis.readDouble(); String s = dis.readUTF(); //UTF
  int i = dis.readInt();
  dis.close();
  DataInputStream disBis = new DataInputStream(
    new BufferedInputStream(new FileInputStream("fichier2")));
  i = dis.readInt();
  dis.close();  } }
```

Listing 53 – coursSSI3/exemples/es/Decorateur.java

La lecture des octets d'un fichier

- Pour lire un fichier qui contient des octets qu'on ne peut lire sous forme de types Java particuliers (images, vidéo, etc, ...)
- La chaîne de caractères passé au constructeur de File peut être un chemin relatif ou absolu
- Les chemins relatifs sont relatifs au répertoire dans lequel on lance la commande java, et pas par rapport au répertoire qui contient la classe

```
package coursSSI3.exemples.es;
import java.io.*;
public class LectureOctets {
        public static void main(String[] args)
          throws IOException {
                File f = new File("fichier");
                int tailleFichier = (int)f.length();
                byte[] donnees = new byte[tailleFichier];
                DataInputStream dis =
                  new DataInputStream(
                   new FileInputStream(f));
                dis.readFully(donnees);
                dis.close();
        }
}
```

Listing 54 – coursSSI3/exemples/es/LectureOctets.java

L'écriture d'un flot d'octets

- `OutputStream` (Classe abstraite de base)
  - `FilterOutputStream` (Décorateur)
    - `BufferedOutputStream` (Sorties bufférisées)
    - `DataOutputStream` (Ecriture de types primitifs)
    - `PrintStream` (Utilisé par System.out - Ne pas utiliser autrement)
  - `FileOutputStream` (Ecriture des octets d'un fichier)
  - `ObjectOutputStream` (Ecriture d'un objet sérialisé)

La classe `OutputStream`

- Interface publique de cette classe (ajouter `throws IOException` à toutes les méthodes) :
  abstract void write(int b)
  void write(byte[] b)
  void write(byte[] b,
  int debut, int nb)
  void flush()
  void close()
- Remarque : avec la méthode `write(int b)`, seul l'octet de poids faible de b est écrit dans le flot

Les particularités de `PrintStream`

- Cette classe possède les deux méthodes `print()` et `println()` qui écrivent tous les types de données sous forme de chaînes de caractères
- Aucune des méthodes de `PrintStream` ne lève d'exception ; on peut savoir s il y a eu une erreur en appelant la méthode `checkError()`
- Attention, `println()` n'effectue un `flush()` (vidage des tampons) que si le `PrintStream` a été créé avec le paramètre " autoflush "

Un exemple de ByteArrayOutputStream

- Ecrire un table d'octet en mémoire sans connaire la taille à l'avance

```
package coursSSI3.exemples.es;
import java.io.ByteArrayOutputStream;
public class EcritureOctet {
        private static int p=4;
        private static int[] T={4,5,-2,5};

        public static int getValeur() {return T[p++];}

        public static void main(String[] args) {
                ByteArrayOutputStream out =
                 new ByteArrayOutputStream();
                int b;
                while ((b = getValeur()) > 0) {
                 out.write(b); }
                byte[] octets = out.toByteArray();
        }
}
```

Listing 55 – coursSSI3/exemples/es/EcritureOctet.java

L'écriture de types primitifs dans un fichier

```
package coursSSI3.exemples.es;

import java.io.*;

public class EcriturePrimitifs {
        public static void main(String[] args)
          throws IOException {
                DataOutputStream dos =
                        new DataOutputStream(
                          new BufferedOutputStream(
                          new FileOutputStream("fichier")));
                dos.writeDouble(27.7);
                dos.writeUTF("Pierre");
                dos.writeInt(56);
                dos.close();
        }

}
```

Listing 56 – coursSSI3/exemples/es/EcriturePrimitifs.java

- Le constructeur `FileOutputStream(String nom, boolean append)` permet d'ajouter à la fin du fichier ; sinon, le contenu du fichier est effacé à la création

Les exceptions liées aux entrées-sorties

- `Exceptions`
  - `IOException` (Exception durant une entrée-sortie)
    - `EOFException` (Lecture d'une fin de fichier)
    - `FileNotFoundException` (Fichier n'existe pas)
    - `ObjectStreamException` (Problème lié à la sérialisation)

Le traitement des exceptions

```
package coursSSI3.exemples.es;

import java.io.*;

public class ErreurES {
   public static void main(String[] args) {
                try {
                        DataInputStream dis =
                         new DataInputStream(
                           new FileInputStream("fichier"));
                        try { while (true) {
                          double d= dis.readDouble();
                          }
                        }catch(EOFException e) { }
                         catch(IOException e) { }
                         finally {
                           try {dis.close();}
                           catch (IOException e) {}
                           }
                } catch(FileNotFoundException e) { }
        }
}
```

Listing 57 – coursSSI3/exemples/es/ErreurES.java

Classes de lecture d'un flot de caractères

- `Reader` (Classe abstraite de base)
  - `FilterReader` (Décorateur)
  - `InputStreamReader` (Lecture de caractères Unicode à partir d'un flot d'octets)
    - `FileReader` (Lecture de caractères Unicode à partir des octets d'un fichier)
  - `BufferedReader` (Entrées bufférisées)

Classes d'écriture d'un flot de caractères

- `Writer` (Classe abstraite de base)
  - `FilterWriter` (Décorateur)
  - `OutputStreamWriter` (Ecriture de caractères Unicode sous forme d'octets)
    - `FileWriter` (Ecriture de caractères Unicode dans un fichier, sous forme d'octets)
  - `BufferedWriter` (Ecriture bufférisée)
  - `PrintWriter` (Fournit des méthodes print() et println())

Lecture et écriture de flots de caractères

Classe pour entrée	**Classe pour sortie**	**Fonctions fournies**
`Reader`	`Writer`	Classes abstraites de base
`InputStreamReader`	`OutputStreamReader`	Ponts entre les flots d'octets et les flots de caractères
`FileReader`	`FileWriter`	Lecture et écriture de caractères à partir des octets d'un fichier (codage par défaut)
`BufferedReader`	`BufferedWriter`	Lecture et écriture avec buffer
	`PrintWriter`	Possède les méthodes `print()` et `println()`

print et println de PrintWriter

- Ces méthodes sont surchargées pour tous les types primitifs, les tableaux de caractères et les classes `String` et `Object`
- Elles ne lancent jamais d'exceptions (comme toutes les autres méthodes de `PrintWriter`) ; `boolean checkError()` permet de savoir s'il y a eu une erreur avec le flot sous-jacent
- Attention, `println()` n'effectue un `flush()` (vidage des buffers) que si le `PrintWriter` a été créé avec le paramètre `autoflush`

Séparateur de lignes

- La façon de séparer les lignes dépend du système d'exploitation
- Pour être portable utiliser
  - `println` de `PrintWriter` (le plus simple)
  - `writeLine` ou `newLine` de `BufferedWriter`
  - ou la propriété système `line.separator` (`System.getProperty("line.separator")`)
- Ne pas utiliser le caractère \n qui ne convient pas, par exemple, pour Windows

Codage

- En Java les caractères sont codés en Unicode
- Ce n'est souvent pas le cas sur les périphériques source ou destination des flots (le plus souvent ASCII étendu ISO 8859-1 pour les français)
- Des classes spéciales permettent de faire les traductions entre le codage Unicode et un autre codage
- Un codage par défaut est automatiquement installé par le JDK, conformément à la locale

Flots de caractères et les flots d'octets

- `InputStreamReader` et `OutputStreamWriter` sont des classes filles de Reader et Writer
- Leur constructeur prend en paramètre un flot d'octets ; par exemple, `public OutputStreamWriter(OutputStream out)`
- On peut préciser un codage particulier en paramètre de leur constructeur si on ne veut pas le codage par défaut
- Ecriture-lecture dans un fichier de texte
  - `File{Reader|Writer}` sont des classes filles de `InputStreamReader` et `OutputStreamWriter`
  - Elles permettent de lire et d'écrire des caractères Unicode dans un fichier, suivant le codage par défaut (utiliser leur classe mère si on veut un autre codage)

Fichier composé de lignes de texte

- En lecture, on utilise la classe `BufferedReader` qui comprend la méthode `readLine()`
- En écriture, on utilise la classe `PrintWriter` qui comprend les méthodes `print()` et `println()`

Lire les lignes de texte d'un fichier

```
package coursSSI3.exemples.es;

import java.io.*;

public class LireLigne {
  public static void main(String[] args) {
    try {
       String ligne;
       FileReader fr = new FileReader("fichier");
       BufferedReader br = new BufferedReader(fr);
       try {
        while ((ligne = br.readLine()) != null) {
            }
        } catch (IOException e) {  }
        finally {
          try {br.close();}
          catch(IOException e) { }
          }
       } catch (FileNotFoundException e) { }
  }
}
```

Listing 58 – coursSSI3/exemples/es/LireLigne.java

Écrire des lignes de texte dans un fichier

```
package coursSSI3.exemples.es;

import java.io.*;

public class EcrireLigne {
  public static void main(String[] args) {
    try {
       PrintWriter pw = new PrintWriter(
         new BufferedWriter(new FileWriter("fichier")),
          true);
       String ligne="le texte";
       pw.println(ligne);
       pw.close(); // vide les buffers
      } catch(IOException e) {}
  }

}
```

Listing 59 – coursSSI3/exemples/es/EcrireLigne.java

Les séparateurs des données

- Pour les flots d'octets, il suffit de relire les données dans l'ordre dans lequel elles ont été écrites, avec le décodage approprié
- Pour les flots de caractères, on doit explicitement mettre des séparateurs entre les données ; par exemple, pour distinguer un nom d'un prénom
- `StringTokenizer` et `StreamTokenizer` facilitent la relecture de données écrites avec des séparateurs

Entrées-sorties sur clavier-écran

- Lecture de caractères tapés au clavier :

```
package coursSSI3.exemples.es;

import java.io.IOException;

public class LireClavier {
        public static void main(String[] args) {
                int n;char car;
                String s = "";
                try {
                 while (true) {
                  n = System.in.read();
                  if (n == -1) break;
                  car = (char)n;  s += car;
                  }
                 }
                 catch(IOException e) {
                   System.err.println("Erreur I/O" + e);
                 }
        }
}
```

Listing 60 – coursSSI3/exemples/es/LireClavier.java

Nouveau paquetage java.nio

- Reprend toute l'architecture des classes pour les entrées/sorties
- notion de canal (channel) et de buffer
- Utilise les possibilités avancées du système d'exploitation hôte pour optimiser les entrées-sorties et offrir plus de fonctionnalités

La Classe `File`

- Cette classe représente la notion de fichier, indépendamment du système d'exploitation
- Un fichier est repéré par un chemin abstrait composé d'un préfixe optionnel (nom d'un disque par exemple) et de noms (noms des répertoires parents et du fichier lui-même)
- Attention, `File fichier = new File("/bidule/truc");` ne lève aucune exception si /bidule/truc n'existe pas dans le système de fichier
- Constructeurs
  - Les chemins passés en premier paramètre peuvent être des noms relatifs ou absolus
  - File (String chemin)
  - File (String cheminParent, String chemin)
  - File (File parent, String chemin)
- La classe File offre des facilités pour la portabilité des noms de fichiers

Les fonctionnalités de la classe `File`

- Elle permet d'effectuer des manipulations sur les fichiers et répertoires considérés comme un tout (mais pas de lire ou d'écrire le contenu) :
  - lister un répertoire,
  - supprimer, renommer un fichier
  - créer un répertoire
  - créer un fichier temporaire
  - connaître les droits que l'on a sur un fichier (lecture, écriture)
  - etc.

Méthodes de `File`

- informatives
  - `boolean exists()`
  - `boolean isDirectory()`
  - `boolean isFile()`
  - `boolean canRead()`
  - `boolean canWrite()`
  - `long lastModified()`
  - `long length()`
- actions
  - `boolean delete()` (true si suppression réussie)
  - `boolean mkdir()`
  - `boolean mkdirs()` (peut créer des répertoires intermédiaires)
  - `boolean renameTo(File nouveau)`
  - `boolean setLastModified(long temps)`
  - `boolean setReadOnly()`

Méthodes pour les noms

- `String getName()` (nom terminal)
- `String getPath()` (chemin absolu ou relatif)
- `File getAbsoluteFile()`
- `String getAbsolutePath()`
- `String getParent()`
- `File getParentFile()`
- `URL toURL()` (de la forme file:url)

Classe URL

- `protocole://machine[:port]/cheminPage`
- Cette classe du paquetage java.net fournit de nombreux constructeurs
- Elle permet d'extraire des éléments à partir d'un URL (`getPort`, `getHost`, ...)
- Les données associées à l'URL peuvent être obtenues par les méthodes `openConnection` et `openStream` ou par la méthode `getContent`

Lire le code HTML d'une page Web

- La classe `URL` fournit la méthode `InputStream openStream() throws IOException`

```
package coursSSI3.exemples.es;
import java.io.*;
import java.net.*;

public class LireURL {
  public static void main(String[] args) throws IOException {
    String ligne;
    URL url = new URL("http://www.univ-tln.fr/index.html");
    InputStream is = url.openStream();
    BufferedReader br =
      new BufferedReader( new InputStreamReader(is));
    while ((ligne = br.readLine()) != null) {
       System.out.println(ligne); }
 }
}
```

Listing 61 – coursSSI3/exemples/es/LireURL.java

Définition de la Sérialisation

- Sérialiser un objet c'est transformer l'état (les valeurs des variables d'instances) d'un objet en une suite d'octets
- On peut ainsi conserver l'état d'un objet pour le retrouver ensuite et reconstruire un autre objet avec le même état
- On utilise ObjectOutputStream et ObjectInputStream

Qu'est-ce qui est sérialisé ?

- Les valeurs des variables d'instance (pas des variables de classe) des instances sérialisées
- Des informations sur les classes des objets sérialisés ; en particulier :
  - nom de la classe
  - noms, types, modificateurs des variables à sauvegarder
  - des informations qui permettent de savoir si une classe a été modifiée entre la sérialisation et la désérialisation

Utilisation de la sérialisation

- Conserver un objet dans un fichier ou une base de données pour le récupérer plus tard
- Conserver la configuration d'un composant, pour pouvoir le réutiliser plus tard dans une application (JavaBean)
- Transmettre les paramètres (de types non primitifs) d'une méthode appelée sur un objet distant (RMI) : le paramètre est sérialisé, les octets sont transmis sur le réseau et le paramètre est reconstruit sur la machine distante

Interface Serializable

- Pour pouvoir être sérialisé, un objet doit être une instance d'une classe qui implémente l'interface Serializable
- Cette interface ne comporte aucune méthode ; elle sert seulement à marquer les classes sérialisables
- La plupart des classes du JDK sont sérialisables
- Certaines classes ne peuvent pas être sérialisées (InputStream par exemple) ; d'autres ne doivent pas l'être (par sécurité)
- Si on ne veut pas qu'une variable d'instance soit sérialisée, on la déclare transient : `private transient int val;`
  - Quand l'objet sera désérialisé, la valeur de cette variable devra être recalculée s'il en est besoin sinon elle recevra la valeur par défaut de son type

Compression/décompression

- Le paquetage `java.util.zip` fournit des classes filtres pour compresser des flots :
  - `GZIPInputStream` (et `GZIPOutputStream`) pour travailler avec des donné au format GZIP
  - `ZipInputStream` (et `ZipOutputStream`) pour lire ou écrire des entrées (le plus souvent des fichiers) compressées au format ZIP
  - Pour lire les fichiers zip, la classe `java.util.zip.ZipFile` permet des traitements plus performants

Exemple de compression

- compresser un objet sérialisé

```
package coursSSI3.exemples.es;

import java.io.*;
import java.util.zip.GZIPOutputStream;

import coursSSI3.exemples.animaux.Chien;
public class Compression {
        public static void main(String[] args)
          throws IOException {
                FileOutputStream fos =
                   new FileOutputStream("fichier");
                GZIPOutputStream gz =
                   new GZIPOutputStream(fos);
                ObjectOutputStream oos =
                   new ObjectOutputStream(gz);
                oos.writeObject(new Chien("Medor"));
        }
}
```

Listing 62 – coursSSI3/exemples/es/Compression.java

Exemple de compression

- récupérer l'objet sérialisé

```
package coursSSI3.exemples.es;

import java.io.*;
import java.util.zip.GZIPInputStream;

import coursSSI3.exemples.animaux.Chien;

public class Decompresser {
        public static void main(String[] args)
          throws IOException, ClassNotFoundException {
                Chien c;
                FileInputStream fis =
                  new FileInputStream("fichier");
                GZIPInputStream gz =
                  new GZIPInputStream(fis);
                ObjectInputStream ois =
                  new ObjectInputStream(gz);
                c= (Chien)ois.readObject();
        }

}
```

Listing 63 – coursSSI3/exemples/es/Decompresser.java

Interface avec l'utilisateur

- Les programmes peuvent interagir avec un utilisateur
  - Demande d'information
  - Saisie de données
  - Affichage des résultats
- Cet échange se fait via un interface utilisateur (UI en anglais)
  - Elle peut en mode texte ou graphique

Interface graphique

- Une interface graphique (GUI) est composée d'une ou plusieurs fenêtres
- Une fenetre contient un ensemble de composants graphiques (*widgets* en anglais) :
  - boutons
  - listes déroulantes
  - menus
  - champ texte
  - ...
- Difficulté majeure :
  - Le programme est guidé par les actions de l'utilisateur
  - Celui peut utiliser plusieurs objets graphiques à un même moment
    - saisie d'un texte, clic sur un bouton, ...
  - L'ordre de ces actions ne peut pas être complètement prévu

Programmation événementielle

- Pour simplifier la programme dans ce type de cas on utilise un nouveau mode de programmation
  - La programmation événementielle
- La structure d'un programme de ce type est :
  - les actions de l'utilisateur produisent des événements qui sont empilés
  - le programme traite les évènements avec des actions préprogrammées

Les écouteurs

- Dans les IHM Java, on distinguera deux types d'objets es *observateurs* et les *observés*
- les widgets (boutons, ...) sont les observés
- on associe aux composants graphiques des observateurs (*listeners* en Anglais, ou aussi écouteur)
  - un écouteur traite des classes d'évènements particuliers (clic souris, frappe au clavier, ...).
- Le fonctionnement est le suivant :
  - L'écouteur est prévenu par le composant graphique qu'un évènement les concerne
  - L'écouteur va exécuter le code prévu pour réagir à cet évènement

Les API en Java

- On distingue deux bibliothèques :
  - AWT (Abstract Window Toolkit), à partir du JDK 1.1
  - Swing, à partir de Java 2
- Elles composent avec Java2D l'ensemble de bibliothèques (*framework*) graphiques proposé par Java appelé
  - Java Foundation Classes : JFC
  - Swing s'appuie sur AWT
    - Le fonctionnement est le même
    - les classes de Swing héritent des classes de AWT
- Comment choisir en AWT et SWING
  - Swing permet tout ce qu'AWT permet et même plus
  - Les composants swing sont plus jolis mais plus lourd (et donc plus lent)

Les paquetages de base

- AWT : `java.awt` et `java.awt.event`
- Swing : `javax.swing`, `javax.swing.event`
  - plus ceux pour des composants de haut niveau dans `javax.swing` :
    - `table`, `tree`, `text`, `filechooser`, `colorchooser`

  et ceux liés à l'apparence aussi dans `javax.swing` :
  - pluggable look and feel (plaf)
  - `plaf`, `plaf.basic`, `plaf.metal`, `plaf.windows`, `plaf.motif`

Un premier exemple : ouvrir un fenêtre

```
package coursSSI3.exemples.ihm;

import java.awt.Frame;

import javax.swing.JFrame;

public class Fenetre extends JFrame {
        public Fenetre() {
                super("Une fenetre");
                setSize(300, 200);
                pack(); //Ne pas mettre si la fenetre est vide
                setVisible(true);
        }

        public static void main(String Args[]) {
                Fenetre f = new Fenetre();
        }

}
```

Listing 64 – coursSSI3/exemples/ihm/Fenetre.java

Taille d'une fenêtre

- Il est possible de spécifier préciser les caractéristiques de la fenêtre
  - Taille : `setSize(int l, int h)` (ou instance de `Dimension`)
  - Position : `setLocation(int x, int y)` (ou instance de `Point`), fixe le sommet haut/gauche
  - Les deux : `setBounds(int x, int y, int l, int h)` (ou instance `Rectangle`)
- Il est aussi possible d'adapter la fenêtre à son contenu :
  - La méthode `pack()` fixe la taille nécessaire à la fenêtre la taille en fonction des tailles préférées de chacun de ses composants

Classe `java.awt.Toolkit`

- Les sous-classes de la classe abstraite `Toolkit` permettent de communiquer avec le système d'exploitation de la machine réelle.
- La méthode `getDefaultToolkit()` retourne un instance d'une classe qui implante `Toolkit` (cf. propriété `awt.toolkit`)
- Les méthodes classiques sont : `getScreenSize()`, `getScreenResolution()`, `beep()`, `getImage()`, `createImage()`, `getSystemEventQueue()`

Positionnement d'une fenêtre et icône

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import javax.swing.JFrame;

public class FenetrePlacee extends JFrame {
        public FenetrePlacee() {
                Toolkit tk = Toolkit.getDefaultToolkit();
                Dimension d = tk.getScreenSize();
                int hauteurEcran = d.height;
                int largeurEcran = d.width;
                setSize(largeurEcran / 2, hauteurEcran / 2);
                setLocation(largeurEcran / 4, hauteurEcran / 4);
                Image img = tk.getImage("icone.gif");
                setIconImage(img);
                //pack();
                setVisible(true);
        }

        public static void main(String[] args) {
                FenetrePlacee f = new FenetrePlacee();
        }
}
```

Listing 65 – coursSSI3/exemples/ihm/FenetrePlacee.java

Composants lourds

- Pour afficher des fenêtres (instances de JFrame), Java s'appuie sur les fenêtres fournies par le système d'exploitation hôte dans lequel tourne la JVM
- Les composants Java qui, comme les JFrame, s'appuient sur des composants du système hôte sont dit « lourds »
- L'utilisation de composants lourds améliore la rapidité d'exécution mais nuit à la portabilité et impose les fonctionnalités des composants

Composants légers

- Au contraire de AWT qui utilise les widgets du système d'exploitation pour tous ses composants graphiques (fenêtres, boutons, listes, menus, etc.), Swing ne les utilise que pour les fenêtres de base « top-level »
- Les autres composants, dits légers, sont dessinés dans ces containers lourds, par du code « pur Java »
- Attention, les composants lourds s'affichent toujours au-dessus des composants légers

Containers lourds

- Il y a trois sortes de containers lourds (un autre, `JWindow`, est plus rarement utilisé) :
  - `JFrame` fenêtre pour les applications
  - `JApplet` pour les applets
  - `JDialog` pour les fenêtres de dialogue
- Pour construire une interface graphique avec Swing, il faut créer un (ou plusieurs) container lourd et placer à l'intérieur les composants légers qui forment l'interface graphique

Libérer les ressources associées à une `JFrame`

- En tant que composant lourd, une `JFrame` utilise des ressources du système sous-jacent
- Si on ne veut plus utiliser une `JFrame` (ou JDialog ou JWindow), mais continuer l'application, il faut lancer la méthode `dispose()` de la fenêtre ; les ressources seront rendues au système
- Voir aussi la constante `DISPOSE_ON_CLOSE` de l'interface `javax.swing.WindowConstants`

Classe JComponent

- La plupart des widgets de Swing sont des instances de sous-classes de la classe JComponent
- Les instances des sous-classes de `JComponent` sont de composants « légers »
- `JComponent` hérite de la classe `Container`
- Tous les composants légers des sous-classes de JComponent peuvent donc contenir d'autres composants

Les Containers

- Des composants sont destinés spécifiquement à recevoir d'autres éléments graphiques :
  - les containers « top-level » lourds JFrame, JApplet, JDialog, JWindow
  - les containers « intermédiaires » légers JPanel, JScrollPane, JSplitPane, JTabbedPane, Box (ce dernier est léger mais n'hérite pas de JComponent)

JPanel

- JPanel est la classe mère des containers intermédiaires les plus simples
- Un JPanel sert à regrouper des composants dans une zone d'écran
- Il n'a pas d'aspect visuel déterminé ; son aspect visuel est donné par les composants qu'il contient
- Il peut aussi servir de composant dans lequel on peut dessiner ce que l'on veut, ou faire afficher une image (par la méthode paintComponent)

Ajouter des composants

- Les containers « top-level » ne peuvent contenir directement d'autres composants
- Ils sont associés à un autre container, le « content pane » dans lequel on peut ajouter les composants
- On obtient ce content pane par (topLevel est un container lourd quelconque ; JFrame par exemple)
  - `Container contentPane = topLevel.getContentPane();`

Placer les composants

```
package coursSSI3.exemples.ihm;

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

public class Composants extends Fenetre {

        public Composants() {
                super();
                Container contentPane = getContentPane();
                JLabel label = new JLabel("Bonjour");
                JButton b1 = new JButton("Cliquez␣moi␣!");
                contentPane.add(label, BorderLayout.NORTH);
                contentPane.add(b1, BorderLayout.SOUTH);
                pack();
        }

        public static void main(String[] args) {
                Composants c = new Composants();
        }

}
```

Listing 66 – coursSSI3/exemples/ihm/Composants.java

Les layout managers

- l'utilisateur peut changer la taille d'une fenêtre ; les composants de la fenêtre doivent alors être repositionnés
- Les fenêtres (plus généralement les containers) utilisent des gestionnaires de mise en place (layout manager) pour repositionner leurs composants
- Il existe plusieurs types de layout managers avec des algorithmes de placement différents
- Quand on ajoute un composant dans un container on ne donne pas la position exacte du composant
- On donne plutôt des indications de positionnement au gestionnaire de mise en place
  - explicites (`BorderLayout.NORTH`)
  - ou implicites (ordre d'ajout dans le container)

Algorithme de placement

- Un layout manager place les composants « au mieux » suivant
  - l'algorithme de placement qui lui est propre
  - les indications de positionnement des composants
  - la taille du container
  - les tailles préférées des composants

Dimension et Taille des composants

- La classe `java.awt.Dimension` est utilisée pour donner des dimensions de composants en pixels
  - Elle possède deux variables d'instance publiques de type `int`
    - `height`, `width`
  - Constructeur : `Dimension(int, int)`
- Tous les composants graphiques (`Component`) peuvent indiquer leurs tailles pour l'affichage
  - taille maximum, taille préférée, taille minimum
- Les méthodes : `{get|set}{Maximum|Preferred|Minimum}Size()`

Taille préférée

- La taille préférée est la plus utilisée par les layout managers ; un composant peut l'indiquer en redéfinissant la méthode `Dimension getPreferredSize()`
- On peut aussi l'imposer « de l'extérieur » avec la méthode `void setPreferredSize(Dimension)`
- Mais le plus souvent, les gestionnaires de mise en place ne tiendront pas compte des tailles imposées de l'extérieur

Layout manager

- Fixer le layout manager
  - Par défaut, les fenêtres `JFrame` ont un gestionnaire de mise en place qui est une instance de la classe `BorderLayout`
  - On peut changer le gestionnaire de mise en place d'un Container par la méthode `setLayout(LayoutManager)` de la classe `Container`
- Les types les plus courants de gestionnaire de mise en place :
  - `BorderLayout` : placer aux quatre points cardinaux
  - `FlowLayout` : placer à la suite
  - `GridLayout` : placer dans une grille
  - `BoxLayout` : placer verticalement ou horizontalement
  - `GridBagLayout` : placements complexes

BorderLayout

- Affiche au maximum 5 composants (aux 4 points cardinaux et au centre)
- Essaie de respecter la hauteur préférée du nord et du sud et la largeur préférée de l'est et de l'ouest ; le centre occupe toute la place restante
- layout manager par défaut de JFrame et JDialog
- Les composants sont centrés dans leur zone
- On peut spécifier des espacement horizontaux et verticaux minimaux entre les composants
- Si on oublie de spécifier le placement lors de l'ajout d'un composant, celui-ci est placé au centre (source de bug !)
- Règle pratique : l'est et l'ouest peuvent être étirés en hauteur mais pas en largeur ; le contraire pour le nord et le sud ; le centre peut être étiré en hauteur et en largeur

Placement dans une fenêtre complexe

- Pour disposer les composants d'une fenêtre de structure graphique complexe on peut :
  - utiliser des containers intermédiaires, ayant leur propre type de gestionnaire de placement, et pouvant éventuellement contenir d'autres containers
  - utiliser un gestionnaire de placement de type GridBagLayout (plus souple mais parfois plus lourd à mettre en oeuvre)
  - mixer ces 2 possibilités

Utiliser un JPanel

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import javax.swing.*;

public class JPanelExemple extends Fenetre {
        public JPanelExemple() {
                Container contentPane = getContentPane();
                JPanel panelBoutons = new JPanel();
                JButton b1 = new JButton("Cliquez moi !");
                JButton b2 = new JButton("Et moi aussi !");
                panelBoutons.add(b1);
                panelBoutons.add(b2);
                contentPane.add(panelBoutons, BorderLayout.NORTH);
                JTextArea textArea = new JTextArea(15, 5);
                contentPane.add(textArea, BorderLayout.CENTER);
                JButton quitter = new JButton("Quitter");
                contentPane.add(quitter, BorderLayout.SOUTH);
                pack();
        }
        public static void main(String Args[]) {
                JPanelExemple f = new JPanelExemple();
        }
}
```

Listing 67 – coursSSI3/exemples/ihm/JPanelExemple.java

FlowLayout

- Rangement de haut en bas et de gauche à droite
- Les composants sont affichés à leur taille préférée
- layout manager par défaut de JPanel et JApplet
- Attention, la taille préférée d'un container géré par un FlowLayout est calculée en considérant que tous les composants sont sur une seule ligne

Code avec FlowLayout

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import javax.swing.*;

public class FlowLayoutExplicite extends JFrame {
        public FlowLayoutExplicite() {
                JPanel panel = new JPanel();
                JTextField zoneSaisie;
                JButton bouton;
                panel.setLayout(
                        new FlowLayout(FlowLayout.LEFT, 5, 4));
                panel.add(zoneSaisie = new JTextField(20));
                panel.add(bouton = new JButton("OK"));
                getContentPane().add(panel);
                pack();
                setVisible(true);
        }
        public static void main(String[] args) {
                FlowLayoutExplicite f =
                        new FlowLayoutExplicite();
        }

}
```

Listing 68 – coursSSI3/exemples/ihm/FlowLayoutExplicite.java

GridLayout

- Les composants sont disposés en lignes et en colonnes
- Les composants ont tous la même dimension
- Ils occupent toute la place qui leur est allouée
- On remplit la grille ligne par ligne

Code avec GridLayout

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import javax.swing.*;

public class GridLayoutExemple extends JFrame {
        public GridLayoutExemple() {
                JTextField s1,s2,s3;
                JButton b1,b2,b3;
                JPanel panel = new JPanel();
                panel.setLayout(new GridLayout(3, 2));
                panel.add(s1 = new JTextField(20)); // (1,1)
                panel.add(b1 = new JButton("OK")); // (1,2)
                panel.add(s2 = new JTextField(20)); // (2,1)
                panel.add(b2 = new JButton("OK")); // (2,2)
                panel.add(s3 = new JTextField(20)); // (3,1)
                panel.add(b3 = new JButton("OK")); // (3,2)
                getContentPane().add(panel);
                pack();
                setVisible(true);
        }
        public static void main(String[] args) {
                GridLayoutExemple g = new GridLayoutExemple();
                }
}
```

Listing 69 – coursSSI3/exemples/ihm/GridLayoutExemple.java

GridBagLayout

- Comme GridLayout, mais un composant peut occuper plusieurs « cases » du quadrillage ; la disposition de chaque composant est précisée par une instance de la classe GridBagConstraints
- C'est le layout manager le plus souple mais aussi le plus complexe

Code avec GridBagLayout

```
package coursSSI3.exemples.ihm;
import java.awt.*;import javax.swing.*;
public class GribBagExemple extends JFrame {
 public GribBagExemple() {
  JPanel panel = new JPanel();JButton b1,b2,b3;
  panel.setLayout(new GridBagLayout());
  GridBagConstraints c1 = new GridBagConstraints();
  c1.gridx = 0;c1.gridy = 0;
  c1.gridheight = 1;c1.gridwidth = 2;
  c1.weightx = 0.7;c1.weighty = 0.5;
  panel.add(b1 = new JButton("Bouton␣1"), c1);
  GridBagConstraints c2 = new GridBagConstraints();
  c2.gridx = 0;c2.gridy = 1;
  c2.gridheight = 1;c2.gridwidth = 1;
  panel.add(b2 = new JButton("Bouton␣2"), c2);
  GridBagConstraints c3 = new GridBagConstraints();
  c3.gridx = 1;c3.gridy = 1;
  c3.gridheight = 1;c3.gridwidth = 1;
  panel.add(b3 = new JButton("Bouton␣3"), c3);
  getContentPane().add(panel);pack();setVisible(true);
 }
 public static void main(String[] args) {
  GribBagExemple g = new GribBagExemple(); }
}
```

Listing 70 – coursSSI3/exemples/ihm/GribBagExemple.java

Autres contraintes

- fill détermine si un composant occupe toute la place dans son espace réservé (constantes de la classe GridBagConstraint : BOTH, NONE, HORIZONTAL, VERTICAL)
- anchor dit où placer le composant quand il est plus petit que son espace réservé (CENTER, SOUTH, ...)
- insets ajoute des espaces autour des composants : contraintes.insets = new Insets(5,0,0,0) (ou contraintes.insets.left = 5)
- ipadx, iapdy ajoutent des pixels à la taille minimum des composants

BoxLayout

- Aligne les composants sur une colonne ou une ligne (on choisit à la création)
- Respecte la largeur (resp. hauteur) préférée et maximum, et l'alignement horizontal (resp. vertical)
- Layout manager par défaut de Box et de JToolBar
- Pour un alignement vertical, les composants sont affichés centrés et si possible
  - à leur largeur préférée
  - respecte leur hauteur maximum et minimum (`get{Maxi|Mini}mumSize()`)
- Pour un alignement horizontal, idem en intervertissant largeur et hauteur
- Pour changer les alignements, on peut utiliser les méthodes de la classe `Component setAlignment{X|Y}`
- Alignement (Constantes de `Component`) :
  - `{LEFT|CENTER|RIGHT}_ ALIGNMENT`
  - `{TOP|BOTTOM}_ ALIGNMENT`

Problèmes d'alignement

- Si tous les composants géré par un BoxLayout n'ont pas le même alignement, on peut avoir des résultats imprévisibles
- Par exemple, le seul composant aligné à gauche peut être le seul qui n'est pas aligné à gauche !
- Il vaut donc mieux avoir le même alignement pour tous les composants

Classe Box

- Cette classe est un container qui utilise un BoxLayout pour ranger ses composants horizontalement ou verticalement
- Elle fournit des méthodes static pour obtenir des composants invisibles pour affiner la disposition de composants dans un container quelconque : glue, étais et zones rigides

Code avec Box

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import javax.swing.*;
public class BoxSample extends Fenetre {
        public BoxSample() {
                // Penser a BoxLayout
                Box box = Box.createVerticalBox();
                box.add(new JButton("Action 1"));
                box.add(new JButton("Action 2"));
                box.add(Box.createVerticalStrut(5));
                box.add(new JButton("Action 3"));
                box.add(new JButton("Action 4"));
                box.add(Box.createRigidArea(
                        new Dimension(5, 35)));
                box.add(new JButton("OK!"));
                JPanel panel = new JPanel();
                panel.add(box);
                getContentPane().add(panel);
                pack();
        }
        public static void main(String[] args) {
                BoxSample b = new BoxSample();
        }
}
```

Listing 71 – coursSSI3/exemples/ihm/BoxSample.java

CardLayout et JTabbedPane

- CardLayout : Affichage des composants à tour de rôle
- JTabbedPane : Affichage d'onglets

```
package coursSSI3.exemples.ihm;
import javax.swing.*;
public class Onglet extends Fenetre{
        JPanel p1, p2, p3;
        public Onglet() {
                JTabbedPane onglets = new JTabbedPane();
                setSize( 300, 200 );
                JPanel panel = new JPanel();
                onglets.addTab("Option",p1=new JPanel());
                onglets.addTab("Resultats",p2=new JPanel());
                onglets.addTab("Aide",p3=new JPanel());
                panel.add(onglets);
                p1.add(new JButton("Options"));
                p2.add(new JButton("Resultats"));
                p3.add(new JButton("Aide"));
                getContentPane().add(panel);setVisible(true);
        }
        public static void main(String[] args) {
                Onglet o = new Onglet();
        }
}
```

Listing 72 – coursSSI3/exemples/ihm/Onglet.java

Traiter les événements

- L'utilisateur utilise le clavier et la souris pour intervenir
- Le système d'exploitation engendre des événements
- Le programme doit lier des traitements à ces événements
  - **bas niveau**, générés directement par des actions élémentaires de l'utilisateur
  - **« logiques »** de plus haut niveau, plusieurs actions élémentaires correspondants à une action complète

Exemples d'événements

- De bas niveau :
  - appui sur un bouton de souris ou une touche du clavier
  - relâchement du bouton de souris ou de la touche
  - déplacer le pointeur de souris
- Logiques :
  - frappe d'un A majuscule
  - clic de souris
  - lancer une action (clic sur un bouton par exemple)
  - choisir un élément dans une liste
  - modifier le texte d'une zone de saisie

Événements engendrés

- La frappe d'un A majuscule engendre 5 événements :
- 4 événements de bas niveau :
  - appui sur la touche Majuscule
  - appui sur la touche A
  - relâchement de la touche A
  - relâchement de la touche Majuscule
- 1 événement logique :
- frappe du caractère « A » majuscule

Classes d'événements

- Les événements sont représentés par des instances de sous-classes de java.util.EventObject
- Liés directement aux actions de l'utilisateur : KeyEvent, MouseEvent
- De haut niveau : FocusEvent, WindowEvent, ActionEvent, ItemEvent, ComponentEvent fenêtre ouverte, fermée, icônifiée ou désicônifiée, composant déplacé, retaillé, caché ou montré déclenchent une action choix dans une liste, dans une boîte à cocher

Écouteurs

- Chacun des composants graphiques a ses observateurs (ou écouteurs, listeners)
- Un écouteur doit s'enregistrer auprès des composants qu'il souhaite écouter, en lui indiquant le type d'événement qui l'intéresse (par exemple, clic de souris)
- Il peut ensuite se désenregistrer
- Relation écouteurs - écoutés
  - Un composant peut avoir plusieurs écouteurs (par exemple, 2 écouteurs pour les clics de souris et un autre pour les frappes de touches du clavier)
  - Un écouteur peut écouter plusieurs composants

ActionEvent

- Cette classe décrit des événements de haut niveau très utilisés qui correspondent à une action de l'utilisateur qui va le plus souvent déclencher un traitement (une action) :
  - clic sur un bouton
  - return dans une zone de saisie de texte
  - choix dans un menu
- Ces événements sont très fréquemment utilisés et ils sont très simples à traiter

Interface ActionListener

- Un objet ecouteur intéressé par les événements de type « action » (classe ActionEvent) doit appartenir à une classe qui implémente l'interface java.awt.event.ActionListener
- Définition de ActionListener :

```
public interface ActionListener
extends EventListener {
void actionPerformed(ActionEvent e); }
```

- Inscription comme ActionListener
  - On inscrit un tel écouteur auprès d'un composant nommé composant par la méthode composant.addActionListener(ecouteur) ;
  - On précise ainsi que ecouteur est intéressé par les événements ActionEvent engendrés par composant

Message déclenché par un événement

- Un événement unActionEvent engendré par une action de l'utilisateur sur bouton, provoquera l'envoi d'un message actionPerformed à tous les écouteurs : ecouteur.actionPerformed(unActionEvent) ;
- Ces messages sont envoyés automatiquement à tous les écouteurs qui se sont enregistrés auprès du bouton

Interface Action

- Cette interface hérite de ActionListener
- Elle permet de partager des informations et des comportements communs à plusieurs composants
- Par exemple, un bouton, un choix de menu et une icône d'une barre de menu peuvent faire quitter une application
- Elle est étudiée à la fin de cette partie du cours

Conventions de nommage

- Si un composant graphique peut engendrer des événements de type TrucEvent sa classe (ou une de ses classes ancêtres) déclare les méthodes {add|remove}TrucListener()
- L'interface écouteur s'appellera TrucListener

Écouteur MouseListener

- Des interfaces d'écouteurs peuvent avoir de nombreuses méthodes
- Par exemple, les méthodes déclarées par l'interface MouseListener sont :
  - `void mouseClicked(MouseEvent e)`
  - `void mouseEntered(MouseEvent e)`
  - `void mouseExited(MouseEvent e)`
  - `void mousePressed(MouseEvent e)`
  - `void mouseReleased(MouseEvent e)`

Adaptateurs

- Pour éviter au programmeur d'avoir à implanter toutes les méthodes d'une interface « écouteur », AWT fournit des classes (on les appelle des adaptateurs), qui implantent toutes ces méthodes
- Le code des méthodes ne fait rien
- Permet au programmeur de ne redéfinir dans une sous-classe que les méthodes qui l'intéressent
- Dans java.awt.event : KeyAdapter, MouseAdapter, MouseMotionAdapter, FocusAdapter, ComponentAdapter, WindowAdapter

Exemple de code d'un écouteur

- Un exemple d'écouteur sur un bouton

```
package coursSSI3.exemples.ihm;

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

class EcouteurBouton extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
                ((JButton) e.getSource()).setText("CLIC!");
        }

        public void mouseClicked(MouseEvent e) {
                ((JButton) e.getSource()).setText(
                  "x:"+e.getX()+" Y:"+e.getY());
                }
}
```

Listing 73 – coursSSI3/exemples/ihm/EcouteurBouton.java

Tester les touches modificatrices

- Pour tester si l'utilisateur a appuyé sur les touches Maj, Ctrl, Alt pendant qu'il cliquait sur la souris ou qu'il appuyait sur une autre touche, on utilise les constantes static de type int suivantes de la classe InputEvent : SHIFT_MASK , CTRL_MASK, ALT_MASK, META_MASK, ALT_GRAPH_MASK
- d'autres constantes pour tester le bouton de la souris : BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK
- Exemple d'utilisation : `if ((e.getModifiers() & SHIFT_MASK) == 0)`
- Pour les boutons de la souris, on peut aussi utiliser des méthodes static de la classe SwingUtilities telles que isLeftMouseButton

Classe de l'écouteur

- Soit C la classe du container graphique qui contient le composant graphique
- Plusieurs solutions pour choisir la classe de l'écouteur de ce composant graphique :
  - classe C
  - autre classe spécifiquement créée pour écouter le composant :
  - classe externe à la classe C (rare)
  - classe interne de la classe C
  - classe interne anonyme de la classe C (fréquent)

Solution 1 : classe C

- Solution simple
- Mais peu extensible :
  - si les composants sont nombreux, la classe devient vite très encombrée
  - de plus, les méthodes « callback » comporteront alors de nombreux embranchements pour distinguer les cas des nombreux composants écoutés

Exemple solution 1

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TestEcouteur1 extends JFrame
  implements ActionListener {
 private JButton b1 = new JButton("B1"), b2 = new JButton("B2");
 public void actionPerformed(ActionEvent e) {
  Object source = e.getSource();
  if (source == b1) System.out.println("Clic sur bouton 1");
  else if (source == b2) System.out.println("Clic sur bouton 2");
 }
 public TestEcouteur1() {
  JPanel panel;
  getContentPane().add(panel = new JPanel());
  panel.add(b1);panel.add(b2);
  b1.addActionListener(this);
  b2.addActionListener(this);
  pack();setVisible(true);
 }
 public static void main(String[] args) {
  TestEcouteur1 t = new TestEcouteur1();
 }
}
```

Listing 74 – coursSSI3/exemples/ihm/TestEcouteur1.java

Remarque sur les boutons

- On peut associer à chaque bouton (et choix de menus) une chaîne de caractères par `bouton.setActionCommand("chaine");`
- Cette chaîne
  - par défaut, est le texte affiché sur le bouton
  - permet d'identifier un bouton ou un choix de menu, indépendamment du texte affiché
  - peut être modifiée suivant l'état du bouton
  - peut être récupérée par la méthode `getActionCommand()` de la classe `ActionEvent`

Solution 2 : classe interne

- Solution plus extensible : chaque composant (ou chaque type ou groupe de composants) a sa propre classe écouteur
- Le plus souvent, la classe écouteur est une classe interne de la classe C

Exemple

```
package coursSSI3.exemples.ihm;
import java.awt.*;import java.awt.event.*;
import javax.swing.*;
public class TestEcouteur2 extends JFrame  {
 class EcouteurBouton implements ActionListener {
  public void actionPerformed(ActionEvent e) {
   String commande = e.getActionCommand();
   if (commande.equals("b1"))System.out.println("Clic b1!");
   else if (commande.equals("b2"))System.out.println("Clic b2!");
  }
 }
 private JButton b1 = new JButton("B1"), b2 = new JButton("B2");
 public TestEcouteur2() {
  JPanel panel;getContentPane().add(panel = new JPanel());
  panel.add(b1);panel.add(b2);
  ActionListener eb = new EcouteurBouton();
  b1.addActionListener(eb); b1.setActionCommand("b1");
  b2.addActionListener(eb); b2.setActionCommand("b2");
  pack();setVisible(true);
 }
 public static void main(String[] args) {
  TestEcouteur2 t = new TestEcouteur2();}
}
```

Listing 75 – coursSSI3/exemples/ihm/TestEcouteur2.java

Solution 3 : classe interne anonyme

- Si la classe écoute un seul composant et ne comporte pas de méthodes trop longues, la classe est le plus souvent une classe interne anonyme
- L'intérêt est que le code de l'écouteur est proche du code du composant
- Rappel : une classe interne locale peut utiliser les variables locales et les paramètres de la méthode, seulement s'ils sont final

Classe anonyme

```
package coursSSI3.exemples.ihm;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import coursSSI3.exemples.animaux.Chien;

public class ClasseAnonyme {

        public static void main(String[] args) {
                Chien c1 = new Chien();
                c1.aboyer();
                Chien c2 = new Chien() {
                        public void aboyer() {
                                System.out.println("Grr");
                        }
                };
                c2.aboyer();
        }
}
```

Listing 76 – coursSSI3/exemples/ihm/ClasseAnonyme.java

Exemple JButton

```
package coursSSI3.exemples.ihm;
import java.awt.*;import java.awt.event.*;
import javax.swing.*;
public class TestEcouteur3 extends JFrame  {
  private JButton b1 = new JButton("Bouton 1"),
      b2 = new JButton("Bouton 2");
  public TestEcouteur3() {
    JPanel panel;
    getContentPane().add(panel = new JPanel());
    panel.add(b1);panel.add(b2);
    b1.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
          System.out.println("Clic sur b1 !");} });
    b2.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
          System.out.println("Clic sur b2 !");} });
    pack();setVisible(true);
  }
  public static void main(String[] args) {
    TestEcouteur3 t = new TestEcouteur3();
  }
}
```

Listing 77 – coursSSI3/exemples/ihm/TestEcouteur3.java

Fermer une fenêtre

- Pour terminer l'application à la fermeture de la fenêtre, on ajoute dans le constructeur de la fenêtre un écouteur :

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FermetureFenetre1 extends Fenetre {

  public FermetureFenetre1() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e){
        System.out.println(
         "L'utilisateur␣tente␣de␣fermer␣la␣fenetre");
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    FermetureFenetre1 f = new FermetureFenetre1();
  }

}
```

Listing 78 – coursSSI3/exemples/ihm/FermetureFenetre1.java

Fermer une fenêtre

```
package coursSSI3.exemples.ihm;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FermetureFenetre2 extends Fenetre {
        public FermetureFenetre2() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); }

        void windowClosing(WindowEvent e){
                System.out.println(
                  "L'utilisateur tente de fermer la fenetre");}

        void windowClosed(WindowEvent e) {
                System.out.println("La fenetre est fermee");}

        public static void main(String[] args) {
                FermetureFenetre2 f = new FermetureFenetre2();}
}
```

Listing 79 – coursSSI3/exemples/ihm/FermetureFenetre2.java

- Autres actions (`WindowConstants`) `DO_NOTHING_ON_CLOSE`, `HIDE_ON_CLOSE`, `DISPOSE_ON_CLOSE`

Événements clavier

- Un événement clavier de bas niveau est engendré par une action élémentaire de l'utilisateur, par exemple, appuyer sur une touche du clavier (génère un appel à la méthode keyPressed() de KeyListener)
- Plusieurs actions élémentaires de l'utilisateur peuvent engendrer une seul événement de haut niveau ; ainsi, appuyer et relâcher sur les touches Shift et A pour obtenir un A majuscule, génère un appel à la méthode keyTyped() de KeyListener

KeyEvent

- Cette classe correspond aux événements (evt) engendrés par l'utilisation du clavier
- 3 types d'événements repérés par evt.getID() :
  - KeyEvent.KEY_PRESSED et KeyEvent.KEY_RELEASED sont des événements de bas niveau et correspondent à une action sur une seule touche du clavier
  - KeyEvent.KEY_TYPED est un événement de haut niveau qui correspond à l'entrée d'un caractère Unicode (peut correspondre à une combinaison de touches comme Shift-A pour A)

KeyEvent

- Si on veut repérer la saisie d'un caractère Unicode, il est plus simple d'utiliser les événements de type KEY_TYPED
- Pour les autres touches qui ne renvoient pas de caractères Unicode, telle la touche F1 ou les flèches, il faut utiliser les événements KEY_PRESSED ou KEY_RELEASED

KeyListener

public interface KeyListener extends EventListener void keyPressed(KeyEvent e) ; void keyReleased(KeyEvent e) ; void keyTyped(KeyEvent e) ;

- Si on n'est intéressé que par une des méthodes, on peut hériter de la classe KeyAdapter
- Dans ces méthodes, on peut utiliser les méthodes getKeyChar() (dans keyTyped) et getKeyCode() (dans les 2 autres)

Exemple de KeyListener

```
package coursSSI3.exemples.ihm;
import java.awt.*;import java.awt.event.*;import javax.swing.*;
public class TestClavier extends Fenetre {
  JButton b;
  public class ecouteurK extends KeyAdapter {
    public void keyTyped(KeyEvent e) {
      System.out.println(e.getKeyChar());
      b.add(new Label(
       new Character(e.getKeyChar()).toString()));
      pack();
    }
    public void keyReleased(KeyEvent e) {
      if (e.getKeyCode() == KeyEvent.VK_ESCAPE)
          System.out.println("Escape");
    }
  }
  public TestClavier() {
    JPanel panel;
    getContentPane().add(panel = new JPanel());
    panel.add(b = new JButton("bouton 1"));
    addKeyListener(new ecouteurK()); pack();
  }
  public static void main(String[] args) {
    TestClavier t = new TestClavier(); } }
```

Listing 80 – coursSSI3/exemples/ihm/TestClavier.java

Exemple de menu

```
package coursSSI3.exemples.ihm;
import java.awt.*;import java.awt.event.*;import javax.swing.*;
public class MenuExemple extends JFrame implements ActionListener {
 public MenuExemple() {
  setTitle("Exemple de menu");setSize(320, 240);
  MenuBar barreMenu = new MenuBar();
  setMenuBar(barreMenu);
  Menu menuFichier = new Menu("Fichier");barreMenu.add(menuFichier);
  Menu menuAide = new Menu("Aide"); barreMenu.add(menuAide);
  MenuItem menuOuvrir = new MenuItem("Ouvrir");
  menuOuvrir.addActionListener(this);
  menuFichier.add(menuOuvrir);
  MenuItem menuEnregistrer = new MenuItem("Enregistrer");
  menuEnregistrer.addActionListener(this);
  menuFichier.add(menuEnregistrer);
  MenuItem menuQuitter = new MenuItem("Quitter");
  menuQuitter.addActionListener(this);
  menuFichier.add(menuQuitter);
  this.setVisible(true);
 }
 public void actionPerformed(ActionEvent e) {
  String menuSelectionne = e.getActionCommand();
  if (menuSelectionne.equals("Quitter")) {
   this.setVisible(false);this.dispose();
   System.exit(0);
  } else if (menuSelectionne.equals("Enregistrer")) {
   System.out.println("ENREGISTREMENT");
  } else if (menuSelectionne.equals("Ouvrir"))
```