SOMMAIRE

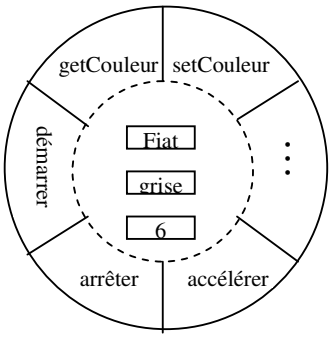
1 LANGAGE ET PROGRAMMATION ORIENTÉS OBJETS

Java[1] est un langage de programmation orienté objets. Un **objet** est une représentation simplifiée d'une entité du monde réel : entité concrète (ex : ma voiture) ou non (ex : la date d'aujourd'hui). Un objet se caractérise par son **état** et son **comportement**. Un objet stocke son état dans des variables appelées **champs** (ou **attributs**) et présente son comportement au travers de fonctionnalités appelées **méthodes**.

Exemple d'objet : *maVoiture = { valeur des attributs : Fiat, grise, 6 l/100km, …; méthodes : démarrer, arrêter, accélérer, …}*

Typiquement, l'état est **encapsulé** au cœur de l'objet et n'est accessible depuis l'extérieur de l'objet, en consultation ou modification, que via les méthodes offertes par cet objet.

Exemple : l'objet maVoiture



On interagit avec un objet en lui envoyant un **message** de demande d'activation d'une de ses méthodes.

Exemple d'envoi d'un message à un objet : *maVoiture.accélérer( );*

Une méthode particulière, appelée **constructeur**, permet d'initialiser l'état de l'objet à sa création.

Une **classe** sert à regrouper sous une même catégorie et un même nom générique les objets partageant le même type d'état et le même comportement. Une classe est donc un type d'objet ; on dit réciproquement qu'un objet est une **instance** de classe.

Exemple de classe : *Automobile = { attributs : marque, couleur, consommation, …; méthodes : démarrer, arrêter, accélérer, …}*

Les classes peuvent être organisées en hiérarchies. Chaque classe hérite alors des attributs et méthodes des classes situées sur sa ligne hiérarchique ascendante.

Réaliser un programme dans un langage orienté objets, c'est :

- modéliser par des classes les objets que l'on a à manipuler
- écrire ces classes dans le langage de programmation orienté objets
- créer des objets instances de ces classes
- communiquer avec ces objets en invoquant leurs méthodes.

[1] Java est né en 1995 dans l'entreprise californienne Sun Microsystems. Il a été conçu par James GOSLING & al.

2 PROGRAMME JAVA

2.1 Style de programmation

Il est largement admis que le respect de la syntaxe et un programme qui fonctionne ne sont pas des critères de qualité suffisants pour un programme. L'expérience de la programmation a ainsi conduit à établir des recommandations de style. Les respecter n'est pas techniquement obligatoire mais contribue à améliorer la lisibilité des programmes. Ces conventions visent en fait une finalité économique : contribuer à produire des programmes plus facilement maintenables pendant toute leur durée de vie.

Dans les chapitres suivants, les exemples respectent les conventions de style préconisées par java.sun.com

2.2 Identificateurs

Quelques règles et recommandations pour nommer les identificateurs :

- Le caractère minuscule-majuscule des caractères est discriminant (règle du langage Java)
- Les identificateurs d'objets, de champs et de méthodes commencent toujours par une minuscule (par convention)
- Les identificateurs de classe (et d'interface) commencent toujours par une majuscule (par convention)
- Les identificateurs de constantes sont tout en majuscules (par convention)
- Les identificateurs de méthode commencent par un verbe (recommandation)

2.3 Commentaires

2.3.1 Commentaires de documentation

Les commentaires de documentation visent à décrire la spécification du code, sans considération d'implémentation. Ils sont destinés à des développeurs et utilisateurs qui n'auraient pas nécessairement les codes sources sous la main.

Ces commentaires peuvent être automatiquement extraits par l'utilitaire `javadoc` pour en constituer un fichier de documentation en HTML.

La documentation d'une classe devrait inclure a minima :

- le nom de la classe, sa fonctionnalité et ses grandes caractéristiques, son auteur, sa version
- une documentation pour chaque constructeur et chaque méthode

La documentation d'une méthode devrait inclure a minima :

- le nom de la méthode, sa fonctionnalité, une description de chaque paramètre et de la valeur retournée, les exceptions susceptibles d'être lancées par cette méthode

Syntaxe :

```
/**
 *  commentaires
 */

/** commentaire  */
```

Principales balises de commentaires prédéfinies :

```
@author
@version
@param
@return
@see
@throws
```

2.3.2 Commentaires d'implémentation

Les commentaires d'implémentation visent à expliciter (sans paraphraser) le code et les choix d'implémentation. Le choix des identificateurs et la clarté de la programmation devrait permettre de minimiser le nombre de tels commentaires.

Syntaxe :

```
/*
 * commentaires
 */

/* commentaire */

// commentaire
```

2.4 Fichier JAVA

2.4.1 Nom et contenu d'un fichier java

- Chaque fichier source java contient une seule classe publique. L'organisation recommandée est la suivante :
    1) Les commentaires de début. C'est un commentaire de documentation comprenant typiquement : l'intitulé de la classe et un descriptif sommaire de sa fonction, l'auteur, la version
    2) Les instructions de paquetage
    3) Les instructions d'import
    4) Les déclarations des classes
- Un fichier devrait contenir moins de 2000 lignes et des lignes de moins de 80 caractères, voire 70 pour les commentaires de documentation (recommandation)
- Un fichier Java porte le nom de la classe publique qu'il contient ; les fichiers source Java ont pour suffixe `.java` ; les fichiers Java compilés ont pour suffixe `.class` (par convention). Les suffixes `.java` et `.class` sont imposés par les outils.

2.4.2 Compilation, exécution, génération de la documentation

Depuis la ligne de commande :

- Compilation : `javac fileName.java`
- Exécution : `java fileName // fichier du main`
- Génération de la documentation : `javadoc *.java`

  Pour avoir une documentation « programmeur » complète (y compris les membres privés) de toutes les classes et de leur interaction, il faut lancer cette commande avec l'option `-private`.

2.5 Paquetages

Un paquetage (*package* en anglais) est une bibliothèque de classes organisées de façon hiérarchique.

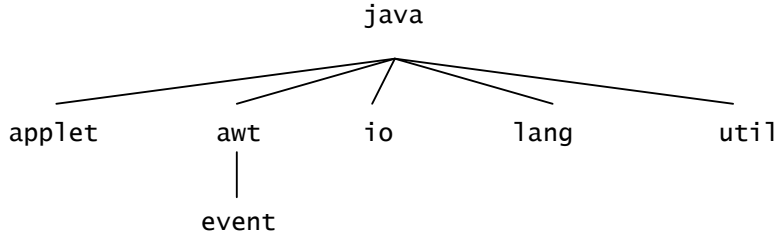
Le nom d'un paquetage est nécessairement le nom de son répertoire. Le nom d'un paquetage est toujours en lettres minuscules (par convention).

Différents paquetages peuvent être rassemblés et compactés dans un fichier d'archive .zip ou .jar

2.5.1 Paquetages standard

La variable d'environnement CLASSPATH indique à la machine virtuelle Java où trouver les classes et bibliothèques de classes dont elle a besoin.

Hiérarchie partielle des paquetages standard :



Paquetages les plus immédiatement utiles	
Paquetage	**Contenu**
java.io	Classes relatives aux entrées/sorties
java.lang	Classes fondamentales : types basiques (classes enveloppes, String, StringBuffer, …), classe Math, …
java.util	Classes utilitaires diverses : collections (ArrayList, Vector, …), classe Random, gestion des dates et des heures, …

2.5.2 Affectation d'une classe à un paquetage

Pour qu'une classe appartienne à un paquetage, il faut déclarer le paquetage d'appartenance en tête de fichier source.

Syntaxe :

```
package packageName ;
```

2.5.3 Importation d'une classe d'un paquetage

Pour utiliser une classe définie dans un paquetage autre que celui de la classe d'utilisation, cette classe doit être importée en tête de fichier source.

Syntaxe :

```
import packageName.ClassName ;
import packageName.* ;  // forme déconseillée car peu informative
```

Exemples :

```
import java.util.ArrayList;
import java.util.Iterator;
import java.io.*;       // forme déconseillée car peu informative
```

Le paquetage java.lang est importé automatiquement.

3 CLASSES et INTERFACES

3.1 Classes standard

Syntaxe :

```
public class ClassName {
      Fields
      Constructors
      Methods
            // accessor methods …
            // mutator methods …
}
```

Exemple :

```
/**
 * Class RationalNumber - Represents a rational number num / den
 *
 * @author AM
 * @version 1.0
 */
public class  RationalNumber {
   // FIELDS
   private int numerator ;
   private int denominator ;
   private boolean reduced ;

   // CONSTRUCTORS
   /** Construct a rational number initialized to 0/1 */
   public RationalNumber( ) {
      numerator = 0 ;
      denominator = 1 ;
      reduced = true ;
   }

   // METHODS - ACCESSORS
   /** Get the numerator of this rational number */
   public int getNumerator( ) {
      return numerator ;
   }
   /** Get the denominator of this rational number */
   public int getDenominator( ) {
      return denominator ;
   }
   /** Return true iff the numerator/denominator form is irreducible
    */
   public boolean isReduced( ) {
      return reduced ;
   }
```

```
   // METHODS - MUTATORS

   /** Set this rational number to _numerator/_denominator */
   public void setNumber(int _numerator, int _denominator) {
      numerator = _numerator ;
      denominator = _denominator ;
      reduced = false ;
   }

   /** Set this rational number to a irreducible
    *  numerator/denominator form
    */
   public void reduce( ) {
      int divisor = greatestCommonDivisor(numerator, denominator );
      numerator = numerator / divisor ;
      denominator = denominator / divisor ;
      reduced = true ;
   }

   // METHODS - UTILITIES

   /** Return the greatest common divisor of the two integers
    *  x and y
    */
   private int greatestCommonDivisor(int x, int y) {
      int divisor ;
      // … à implémenter …
      return divisor ;
   }
} // end class RationalNumber
```

3.2 Classes abstraites et sous-classes

Une classe qui possède au moins une méthode abstraite (i.e. sans corps) est abstraite et doit être déclarée avec le modificateur abstract.

Les classes abstraites ne peuvent pas être instanciées. Pour pouvoir créer des objets, il faut créer une sous-classe dans laquelle toutes les méthodes abstraites seront définies : cette sous-classe, non abstraite, pourra alors être instanciée.

3.2.1 Classes abstraites

Syntaxe :

```
public abstract class ClassName {
      Fields
      Constructors
      Methods
}
```

Exemple :

```
/**
 * Abstract Class Progression
 *
 * Represents a real linear progression of order 1 with constant
 * coefficients
 *
 * @author AM
 * @version 1.0
```

```
 */

public abstract class Progression {
   // FIELDS
   private double u0;   // the first term of the progression

   // CONSTRUCTORS
   /** Define a progression, the first term of which is u0 */
   public Progression(double u0) {
       this.u0 = u0 ;
   }

   // METHODS
   /** Get the first term of this progression */
   public double getu0() {
       return u0 ;
   }
   /** Given a current term u(n), return the next term u(n+1) */
   public abstract double nextTerm(double un) ;

   /** Return the term of rank n */
   public double termOfRank(int n) {
       double u = u0 ;      // u(n)
       double v ;           // u(n+1)
       for(int i = 1 ; i <= n ; i++ ) {
            v = nextTerm(u) ;
            u = v ;
       }
       return u ;
   }
} // end class Progression
```

3.2.2 Sous-classe

Syntaxe :

```
public class SubclassName extends SuperclassName {
      Fields
      Constructors
      Methods
}
```

Exemple :

```
/**
 * Class ArithGeomProgression
 *
 * Represents a real arithmetic-geometric progression with constant
 * coefficients :   u(n+1) = a*u(n) + b
 *
 * @author AM
 * @version 1.0
 */
public class ArithGeomProgression extends Progression {
   // FIELDS
```

```
    private double a;    // multiplying coefficient
    private double b;    // additive coefficient

    // CONSTRUCTORS

    /** Define an arithmetic-geometric progression
     *  @param u0  first term of the progression
     *  @param a   multiplying coefficient
     *  @param b   additive coefficient
     */
    public ArithGeomProgression (double u0, double a, double b) {
        super(u0);
        this.a = a;
        this.b = b;
    }

    // METHODS

    /** Given a current term u(n), return the next term u(n+1) */
    public double nextTerm(double u) {
        return a*u + b ;
    }
} // end class ArithGeomProgression
```

3.3 Interfaces

Remarque liminaire. Le terme « interface » a une sémantique multiple. Dans le cadre de la programmation en Java, il peut prendre, selon le contexte, trois sens distincts (même si conceptuellement apparentés) :

- au sens informatique commun : une interface est une « jonction entre deux éléments (matériels ou logiciels) permettant l'échange d'information » ; par exemple, l'interface homme-machine.
- au sens programmation objet : l'interface d'une classe est la partie visible publique de cette classe ; elle se définit typiquement par la liste des membres (attributs, constructeurs, méthodes) non privés associés à leurs commentaires de documentation
- au sens spécifique du Langage Java : une interface est une sorte de classe abstraite possédant des caractéristiques particulières et qui définit un protocole de comportement ; c'est le sens qui est défini dans cette section.

C'est son contexte d'utilisation qui donne sens au mot interface.

Les interfaces (au sens du Langage Java) sont des sortes de classes abstraites sans aucun détail d'implémentation et qui possèdent un degré d'abstraction supérieur à celui des classes abstraites. Les interfaces :

- ont des méthodes qui sont toutes implicitement abstraites (elles n'ont pas de corps) – à noter qu'il n'est pas recommandé d'ajouter le modificateur `abstract` puisqu'il est implicite ;
- ne possèdent pas de champs, à l'exception éventuelle de constantes de classe (implicitement `static final`) assignées ;
- ses membres sont implicitement publics si l'interface est publique.

Une interface spécifie la signature minimale d'une classe qui l'implémente. Une interface est donc un contrat à respecter. Quand une classe implémente une interface, elle s'engage à se conformer à cette interface. Une classe peut implémenter plusieurs interfaces.

Une classe définit un type et son implémentation ; une interface définit un type sans son implémentation.

3.3.1 Interface

Syntaxe :

```
public interface InterfaceName {
      constantAttributes
      abstractMethods
}
```

Exemple :

```
public interface Measurable {
      double size() ;

      /** @return -1, 0, 1 if this is <, = or > than x  */
      int isLargerThan(Measurable x) ;
}
```

3.3.2 Classe implantant une interface

Syntaxe :

```
public class ClassName implements InterfaceName1, InterfaceName2, … {
      …
}
```

Exemple :

```
public class Square implements Measurable {
      private double width ;

      public Square(double _width) {
            width = _width ;
      }

      public double size() {
            return width*width ;
      }

      public int isLargerThan(Measurable x) {
            if (this.size() < x.size()) {
                  return -1 ;
            }
            if (this.size() > x.size()) {
                  return 1 ;
            }
            return 0 ;
      }
}
```

3.4 Enumération

Une classe `enum` permet de représenter un ensemble de constantes nommées dont le type est la classe elle-même. Ces classes contribuent à accroître la lisibilité des programmes.

Syntaxe :

```
public enum EnumName {
      // List of named constants.
      // Possibility of other fields and methods.
}
```

Exemple :

```
public enum Direction {
      NORTH, SOUTH, EAST, WEST;
}
public class Test {
      private Direction dir ;
      public setDir(Direction d){
            dir = d ;
      }
      public boolean isNorthSouth() {
            return (dir == Direction.NORTH) ||
                   (dir == Direction.SOUTH) ;
      }
      …
}
```

Sous leur forme la plus simple, les classes `enum` sont souvent utilisées sous la forme d'une classe interne (inner class).

Exemple :

```
public class MyClass {
      private Direction dir ;
      ...
      private enum Direction {
            NORTH, SOUTH, EAST, WEST ;
      }
      public void setNorth() {
            dir = Direction.NORTH ;
      }
      ...
}
```

Nota :

- il est possible d'associer à chaque constante énumérée une valeur d'un type quelconque
- il est possible d'utiliser un type énuméré dans une instruction `switch`

3.5 Classes imbriquées (Nested classes)

Ce concept n'est pas vital pour un débutant.

Une classe peut être définie à l'intérieur d'une autre classe. On parle alors de classe imbriquée. On en distingue quatre catégories :

- les classes membres déclarées `static` (*static nested classes*) : membres statiques de la classe englobante ;
- les classes internes (*inner classes*) : membres non statiques de la classe englobante ;
- les classes locales (*local classes*) : classes définies à l'intérieur d'une méthode ;
- les classes anonymes (*anonymous classes*) : classes locales sans nom.

Exemple :

```
public class OuterClass {
      ...
      public static class StaticNestedClass {
            ...
      }
      private class InnerClass {
            ...
      }
      ...
}
// Exemple de référence à la classe StaticNestedClass :
//    OuterClass.StaticNestedClass
```

Quel intérêt peut-il y avoir à déclarer des classes imbriquées ? Potentiellement un regroupement logique, une encapsulation accrue, un code plus lisible et plus facilement maintenable. En pratique, la plupart des classes imbriquées sont des classes internes. Quant aux classes anonymes, elles peuvent rendre le code difficile à lire et doivent donc être limitées à des classes très petites (d'une ou deux méthodes) : voir exemple typique en section 11.4.2.

3.6 Contrôle d'accès par les modificateurs

Le contrôle d'accès à un membre d'une classe peut être précisé dans leur déclaration par des mots-clés appelés modificateurs. Ces modificateurs permettent de spécifier :

- la visibilité du champ ou de la méthode : `private, protected` ou `public`
- le lieu de stockage : dans la classe (modificateur `static`) ou dans l'objet (par défaut)
- le caractère modifiable (par défaut) ou non (modificateur `final`)

Modificateur*	Commentaires et exemple
`private`	Un membre déclaré `private` n'est accessible qu'à l'intérieur de sa classe de définition. Modificateur d'accès recommandé pour les champs.
`protected`	Un membre déclaré `protected` n'est accessible que depuis sa classe de déclaration, une sous-classe, ou une classe du même paquetage. Modificateur d'accès recommandé pour les méthodes des super-classes.

`public`	Un membre déclaré `public` est accessible depuis n'importe quelle classe. Modificateur d'accès recommandé pour les méthodes
`static`	• Un champ déclaré `static` est une *variable de classe* et n'est pas dupliqué dans les différentes instances ; un champ non `static` est une *variable d'instance* et est dupliqué dans chaque instance. • Une méthode déclarée `static` est une *méthode de classe* et est invoquée sans référence à un objet particulier mais en faisant référence à sa classe ; une méthode non `static` est une *méthode d'instance* et est invoquée sur un objet particulier *Voir également chapitre 14.*
`final`	• Une variable déclarée `final` ne peut plus être modifiée après qu'elle a été initialisée. Cela permet en particulier de déclarer des champs constants. *Exemple :* `final int SIZE = 10 ;` • Une méthode déclarée `final` ne peut pas être redéfinie dans une sous-classe. • Une classe déclarée `final` ne peut pas être sous-classée.

* Nota : à défaut de modificateur `private, protected` ou `public,` la visibilité est dite « *package access* » : le membre de la classe est visible depuis toutes les classes du même paquetage.

Exemple : `public static final double GRAVITY = 9.81 ;`

4 INSTANCE DE CLASSE

Avant de pouvoir être utilisé, un objet – ou instance de classe – doit être :

1) déclaré `ClassName objectName ;`
2) créé puis initialisé `objectName = new constructorCall ;`

Exemples :

```
GregorianCalendar cal ;
cal = GregorianCalendar() ;
Timer myTimer = new Timer( ) ;
```

La déclaration d'un objet ne crée pas l'objet mais uniquement une variable pointeur destinée à mémoriser l'adresse future de l'objet ! La création de l'objet en lui-même (i.e. l'instance proprement dite) est réalisée par l'opérateur `new`. L'appel au constructeur qui suit l'opérateur `new` a pour rôle d'initialiser proprement l'objet ainsi créé. In fine, l'adresse de l'objet ainsi créé et initialisé, adresse renvoyée par l'opérateur `new`, est stockée, grâce à l'opérateur d'affectation =, dans la variable préalablement déclarée.

Quelques précisions importantes :

- *Déclaration*. La déclaration `ClassName objectName;` crée une variable de nom `objectName` de type référence (i.e. un pointeur) et l'initialise automatiquement à `null`. `null` est une constante littérale prédéfinie de Java qui peut être assignée à toute variable de type référence et qui identifie un pointeur nul. Une variable de valeur `null` ne pointe vers aucun objet. Toute tentative de référer à un objet à partir d'un pointeur null engendre une erreur (de compilation ou d'exécution selon le contexte).
- *Gestion dynamique de la mémoire*. L'opérateur `new` crée l'instance de classe dans une zone mémoire générale appelée « tas » qui contient toutes les données allouées à l'exécution et n'ayant pas de portée locale. L'objet ainsi créé conserve donc une portée globale et reste « vivant » tant qu'on dispose d'une référence (i.e. un pointeur) permettant d'y accéder. Quand un objet n'est plus référencé, il devient définitivement inaccessible et « meurt » : son espace mémoire est automatiquement désalloué (la récupération automatique de cet espace mémoire est réalisée par un processus appelé ramasse-miettes (*garbage collector*)).

  Une façon de détruire un objet est donc d'assigner la valeur `null` aux variables qui y font référence directement ou indirectement.

5 TYPES DE DONNEES

Java connaît deux types de données :

- Les types primitifs. Les données de type primitif sont stockées directement dans les variables et ont une sémantique de valeur (i.e. c'est la valeur qui est copiée quand une variable est affectée à une autre variable).
- Les types objet (ou types référence). Les données de type objet sont stockées sous forme de référence à cet objet (et non pas en tant qu'objet même). Quand une variable de type objet est affectée à une autre variable, seule la référence est copiée, pas l'objet lui-même.

Java utilise le système Unicode-4, en interne, pour coder les caractères, les chaînes de caractères et les identificateurs. Chaque caractère Unicode est codé sur 16 bits, ce qui permet de représenter de nombreux alphabets (grec, cyrillique, hébreu, arabe, …) ainsi que les symboles mathématiques et techniques. Bien qu'Unicode soit utilisé par certains systèmes d'exploitation (Windows NT ou Vista par exemple), la plupart des systèmes d'exploitation utilisent des systèmes de codage 8 bits, de sorte que des transcodages sont nécessaires et des problèmes d'incompatibilité peuvent survenir à l'affichage.

5.1 Types primitifs

Type	**Description**	**Exemples de littéraux**	
Nombres entiers :	Types signés :		
`byte`	octet (entier 8 bits)	`127`	`-128`
`short`	entier court (16 bits)	`32767`	`-32768`
`int`	entier (32 bits)	`2147483647`	`-2147483648`
`long`	entier long (64 bits)	`5123456789L`	`-55L`
Nombres réels :	Types signés :		
`float`	réel simple précision	`43.889F`	`341.159E-2F`
`double`	réel double précision	`45.63`	`-2.4E107`
Autres types :	Types non signés :		
`char`	caractère (un seul ; 16 bits)	`'?'`	`'\u00F6'`
`boolean`	valeur booléenne	`false`	`true`

- Un nombre sans point décimal est interprété comme un entier de type `int` mais immédiatement converti en `byte`, `short` ou `long` lors de l'affection. On peut déclarer un littéral de type `long` en l'affectant du suffixe `L`.
- Un nombre avec un point décimal est interprété comme un réel de type `double`. On peut déclarer un littéral de type `float` en l'affectant du suffixe `F`.
- Un caractère littéral peut être écrit sous forme d'un unique caractère Unicode entre apostrophes ou d'une valeur Unicode précédée par '\u'.

- Les deux littéraux booléens sont `true` et `false`.
- Les variables d'un type primitif ne référant pas à des objets, elles n'ont pas de méthodes associées.

5.2 Types objet (ou types référence)

Tous les types autres que les types primitifs sont des types objet. Ils incluent les classes (non abstraites) des bibliothèques standard Java (tels que le type `String`, par exemple) ainsi que les classes définies par l'utilisateur.

Une variable d'un type objet contient une référence (ou pointeur) sur cet objet. Les affectations et passages de paramètres ont une sémantique de référence (i.e. c'est la référence qui est copiée et non l'objet lui-même). Après l'affectation d'une variable à une autre, les deux variables réfèrent au même objet : il s'agit donc d'une copie de surface (par opposition à une copie profonde qui dupliquerait les objets). Les deux variables sont alors dites alias pour le même objet.

5.2.1 Classes enveloppes

Les classes enveloppes permettent de transformer un type simple en objet (cette opération est appelée, en anglais, *boxing*).

Classe enveloppe	Type primitif
`Boolean`	`boolean`
`Byte`	`byte`
`Character`	`char`
`Double`	`double`
`Float`	`float`
`Integer`	`int`
`Long`	`long`
`Short`	`short`

Exemples :

```
// convert primitive type → wrapper type
Integer k = new Integer(456);

// convert wrapper type → primitive type
double x = k.doubleValue( ) ;
```

Les classes enveloppes sont utiles quand une méthode n'accepte pas les types simples mais requiert des objets. Toutefois, depuis la plate-forme 5.0 (voir chapitre 16), les valeurs de type primitif sont, en cas de besoin, implicitement converties en l'objet de la classe enveloppe correspondante (*autoboxing*), et réciproquement (*unboxing*). Auparavant, la conversion devait être explicite.

Les objets de telles classes sont immuables (*immutable*), c'est-à-dire non modifiables une fois créés. En conséquence, une méthode ne peut pas avoir de paramètres de sortie de tels types.

Les classes enveloppes présentent surtout l'intérêt d'offrir des méthodes de conversion type primitif ↔ `String`, ainsi que des utilitaires pour des données de type primitif.

Exemple :

```
// Convert String type → primitive type
double x = Double.parseDouble("  3.14159 ") ;
```

5.2.2 Chaîne de caractères non modifiable (String)

Bien que les chaînes de caractères de type `String` soient des objets, il n'est pas nécessaire d'avoir recours à l'opérateur `new` pour les créer.

Les caractères d'une chaîne de caractères `chaine` sont indicés de 0 à `chaine.length()-1`

Une chaîne de caractères littérale est notée entre apostrophes.

L'opérateur + est l'opérateur de concaténation de chaînes. Il produit une nouvelle chaîne sans éléments physiquement partagés avec ses opérandes.

La comparaison de deux chaînes (caractère par caractère) nécessite de faire appel à la fonction `equals()`, l'opérateur == ne donnant le résultat intuitivement attendu que dans un cas particulier.

Les objets de type `String` sont immuables (*immutable*), c'est-à-dire non modifiables une fois créés. En conséquence, une méthode ne peut pas avoir de paramètres de sortie de type `String`.

La classe `String` appartient au paquetage `java.lang`

Exemple :

```
String chaine = "Exemple n° "  + 1 + " de chaine littérale" ;
System.out.println("Longueur de \"" + chaine + "\" ="
                   + chaine.length()
                  ) ;
```

5.2.3 Chaîne de caractères modifiable (StringBuffer)

Ces chaînes de caractères, contrairement aux chaînes de type `String`, peuvent être modifiées.

L'opérateur `new` est indispensable pour créer une chaîne de type `StringBuffer`.

La classe `StringBuffer` appartient au paquetage `java.lang`

Exemple :

```
StringBuffer chaine ;
chaine = new StringBuffer("Meeting at 6 pm !" ) ;
chaine.setCharAt(11, '5') ; // meeting at 5 and not 6 !
System.out.println(chaine) ;
```

5.2.4 Tableau

Un tableau est une collection de taille *fixe* d'éléments de *même type*. Chaque élément est repéré par son indice qui permet un accès direct à l'élément.

A tout tableau est automatiquement associée une classe dérivée d'`Object` et partagée par tous les tableaux ayant le même type d'éléments. L'attribut d'instance `length` (qui est un champ public et non une méthode) contient le nombre d'éléments du tableau. Les éléments d'un tableau `array` sont indicés de 0 à `array.length - 1`

Quand on tente d'accéder à un élément dont l'indice est hors de la plage permise, une exception de type `IndexOutOfBoundsException` est levée.

Il est à noter qu'un tableau de caractères n'est pas un objet de type String et réciproquement.

Syntaxe (pour le cas d'un tableau mono-dimensionnel) :

```
// Declaring an array variable
TypeOfElements[] arrayVariable ;

// Creating an array object
arrayVariable = new TypeOfElements[numberOfElements] ;

// Referring to an element
arrayVariable[integerIndex]
```

Exemple :

```
// Declaring an array variable
private double[] marks ;

// Creating an array object
static final int NB_OF_STUDENTS = 48 ;
marks = new double[NB_OF_STUDENTS];

// Using an array object
for (int i = 0; i < marks.length; i++ ) {
     marks[i] = Math.random() * 20.0 ;
     System.out.println("marks[" + i + "] = " + marks[i] ) ;
)
```

Exemple :

```
static final int NROW = 10 ;
static final int NCOL = 20 ;
String[][] matrix = new String[NROW][NCOL] ;
```

Il est possible de créer *et* d'initialiser un tableau *à la déclaration*.

Exemple :

```
int[ ] t = { 150, -300, 40, 500 } ;
System.out.println(t[0] + " " + t[3]) ;              // 150 500

String[ ][ ] s = {  {"Mr. ", "Mrs. ", "Ms. "},       // row 0
                    {"Smith", "Jones"}               // row 1
                 } ;
System.out.print(s[0][2] + s[1][1]) ;                // Ms. Jones
```

La classe `java.util.Arrays` offre diverses méthodes utilitaires applicables à des tableaux (comparaison, tri, …).

Depuis la plate-forme 5.0 (voir chapitre 16), Java offre une boucle `for` simplifiée pour parcourir les éléments d'un tableau.

Exemple :

```
// for each element of my array of doubles, print this element
for (double e : myArrayOfDoubles) {
      System.out.println(e) ;
}
```

5.2.5 Collection

Une *collection* est un objet représentant un groupe d'objets. Une collection est un agrégat qui regroupe des éléments multiples en une seule entité. Aucune hypothèse n'est faite a priori sur le type des éléments, ni sur l'ordre des éléments, ni sur la possibilité d'éléments dupliqués.

Plus formellement, en Java, une collection d'éléments de type `E` est une classe qui implémente l'interface `Collection<E>`. Le paquetage standard `java.util` offre une variété de collections (*collections framework*) qui répondent aux besoins les plus fréquents, par exemple : `ArrayList<E>, Vector<E>, LinkedList<E>, Stack<E>, Hashtable<K,V>, … .`

5.2.5.1 Généricité

Cette notion n'existe en Java que depuis la plate-forme 5.0 (voir chapitre 16). Cette version 5.0 introduit une révision majeure par rapport à la précédente, notamment en matière de gestion des collections. L'apparition de types génériques en est probablement l'évolution la plus marquante.

La généricité permet de communiquer au compilateur le type des éléments d'une collection au moment de la création de l'objet et non pas en le fixant a priori dans la définition de classe.

Exemple de classe générique :

```
// Defining a generic collection.
// The formal parameter T is the generic type of the elements.
public class MyCollection<T> implements List<T> {
        …
      // T can be used here as any type
        …
}

// Creating a collection of Integers
MyCollection<Integer> collectionOfIntegers ;
CollectionOfIntegers = new MyCollection<Integer>();

// Creating a collection of Strings
MyCollection<String> collectionOfStrings ;
CollectionOfStrings = new MyCollection<String>();
```

Tout comme les déclarations de type peuvent être génériques, les déclarations de méthodes peuvent être aussi génériques, c'est-à-dire paramétrisées par un ou plusieurs paramètres.

Exemple de méthode générique :

```
// Defining
public static <T> void fromArrayToCollection(T[] a,
                                              Collection<T> c){
      for (T o : a) {
            c.add(o);
      }
}
// Using it
String[] a1 = {“data1”, “data2”, “data3”, “data4”, “data5”} ;
Collection<Object> c1 = new ArrayList<String>();
fromArrayToCollection(a1, c1);
```

Nota. Il est à signaler que les tableaux et les génériques ne font pas encore bon ménage !

Exemple :

```
Vector<Integer>[] sheets = new Vector<Integer>[NBOFSHEETS];
```

produit l'erreur classique "`Generic array creation`". Dans cet exemple, une solution pourrait consister à passer par une classe intermédiaire simplement définie par :

```
class VectorOfIntegers extends Vector<Integer> { }
```

Le type ? est appelé type joker (*wildcard type*). Il permet de représenter tout type.

Exemple :

```
public void printAll(Collection<?> c) {
      for (Object o : c) {
            System.out.println(o);
      }
}
```

Il est possible de borner le type joker. Un caractère joker avec une borne supérieure est spécifié :

```
<? extends Type>
```

et représente tous les sous-types de *Type*. Un caractère joker avec une borne inférieure est spécifié :

```
<? super Type>
```

et représente tous les types qui sont des super-types de *Type*.

5.2.5.2 Exemple de collection : ArrayList

Un objet de type `ArrayList<E>` est une collection de taille *variable* d'éléments de type `E`. Chaque élément est repéré par son indice qui permet un accès direct à l'élément.

Une liste-tableau supporte les principales fonctionnalités suivantes :

- ajout d'un élément en fin de tableau (méthode `add(E)`)
- accès à un élément d'indice donné (méthodes `get(int)` et `add(int, E)`)
- suppression d'un élément d'indice donné (méthode `remove(int)`)
- consultation du nombre d'éléments (méthode size())

Les éléments d'un objet collection de type `ArrayList<E>` sont indicés de 0 à `collection.size( ) – 1`

La classe `ArrayList<E>` doit être importée du paquetage `java.util` par :

```
import java.util.ArrayList ;
```

Exemple :

```
ArrayList<Integer> myCollectionOfIntegers ;
myCollectionOfIntegers = new ArrayList<Integer>() ;
for (int i = 0 ; i < 10 ; i++) {
      listOfIntegers.add(i, 0) ;
}
```

5.2.5.3 Parcours d'une collection – Itérateur

Le parcours d'une collection est l'opération qui consiste à explorer cette collection élément par élément.

Deux façons de faire pour parcourir une collection :

- Soit utiliser un itérateur.

  Un itérateur est un objet de type `Iterator<E>`. Les trois fonctionnalités les plus utiles pour créer et gérer un itérateur sont les méthodes : `iterator()` (de la classe `Collection`), `hasNext()` et `next()` (de la classe `Iterator`). A noter que la suppression d'éléments de la collection en cours de parcours relève de la méthode `remove()` de l'itérateur et non de celle de la collection.

  La classe `Iterator<E>` doit être importée du paquetage `java.util` par :

  ```
  import java.util.Iterator ;
  ```

  Les itérateurs sont la façon de faire historique pour parcourir une collection. Depuis la version 5.0 ils sont génériques.

- Soit utiliser une boucle `for` généralisée.

  Une telle boucle permet de s'affranchir des itérateurs et de parcourir très simplement une collection quelconque. Elle n'est offerte que depuis la version 5.0.

Exemple de parcours avec itérateur :

```
Iterator<Integer>  it = myCollectionOfIntegers.iterator() ;
while ( it.hasNext() ) {
    // call it.next() to get the next object of the collection
    // and do something with that object
}
```

Exemple de parcours sans itérateur :

```
// for each element of my collection of Integers, print it
for (Integer v : myCollectionOfIntegers) {
     System.out.println(v) ;
}
```

5.3 Transtypage

Le transtypage (en anglais : *cast / casting*) est une opération qui consiste à convertir le type d'une expression. La conversion de type se réalise en faisant précéder l'expression par le type cible entre parenthèses. Cette proposition s'applique à certains types primitifs et aux types objets liés par une relation d'héritage. Les conversions incorrectes provoquent une erreur à la compilation ou à l'exécution. A noter qu'un transtypage ne change pas le type des variables concernées (le type d'une variable est défini une fois pour toute dès sa création) : l'opération de transtypage est simplement un changement de point de vue qu'on demande au compilateur.

Syntaxe :

```
(newType) expression
```

Exemple :

```
int n = 87 ;
char c = (char) (n / 2) ;
```

Mais certaines classes offrent des méthodes spécifiques plus adéquates, notamment :

- pour les conversions réel → entier :
  - la méthode `Math.round(`*`realValue`*`)` permet de convertir un réel en l'entier le plus proche
  - la méthode `Math.floor(`*`realValue`*`)` associée à une conversion en entier fournit le plus grand entier inférieur ou égal à l'argument.
    Exemple : `(int)Math.floor(15.7)` vaut `15`
  - la méthode `Math.ceil(`*`realValue`*`)` associée à une conversion en entier fournit le plus petit entier supérieur ou égal à l'argument.
    Exemple : `(int)Math.ceil(15.7)` vaut `16`
- pour les conversions en ou à partir de `String` :
  - la méthode `toString()`, dont hérite tout objet, permet de convertir un objet en `String`
  - la méthode `valueOf(`*`primitiveTypeExpression`*`)`, de la classe `String`, renvoie la représentation `String` de la valeur de son argument
  - la méthode `valueOf(`*`string`*`)`, dont dispose toute classe enveloppe, renvoie une instance de cette classe enveloppe contenant la valeur représentée par l'argument de type `String`

Exemples :

```
// Convert String type → primitive type
double x = Double.parseDouble("3.14159") ;

// Convert String type → wrapped type
Double pi = Double.valueOf("3.14159") ;

// Convert wrapped type → primitive type
int k = Integer.valueOf("1789").intValue() ;

// Convert primitive type → String type
String piStr = String.valueOf(3.14159) ;

// Convert primitive type → String type
Double pi = new Double(3.14159) ;
String s = pi.toString() ;
```

Autoboxing / unboxing. Depuis la plate-forme 5.0 (voir chapitre 16), les valeurs de type primitif sont, en cas de besoin, implicitement converties en l'objet de la classe enveloppe correspondante (*autoboxing*), et réciproquement (*unboxing*). Auparavant, la conversion devait être explicite.

6 OPERATEURS ET EXPRESSIONS

Java possède un nombre considérable d'opérateurs arithmétiques et logiques. Les expressions sont constituées d'opérandes et d'opérateurs. Les opérandes peuvent être de différentes natures : variable, constante, appel de fonction.

Dans une expression, tous les opérateurs doivent être explicites.

Quand plus d'un opérateur apparaît dans une seule expression, alors des règles de précédence sont utilisées pour déterminer l'ordre d'application. S'il est nécessaire d'altérer l'ordre normal d'évaluation, des parenthèses peuvent être utilisées. A précédences égales, l'expression est évaluée de gauche à droite.

Recommandations de style :

- Utiliser les parenthèses pour isoler les sous-expressions afin d'éviter les problèmes de précédence d'opérateur et rendre le code plus lisible pour tout programmeur.
- Insérer un espace avant et après chaque opérateur autre que le point

6.1 Opérateurs et expressions arithmétiques

Quelques opérateurs arithmétiques binaires		
Opérateur	**Fonction**	**Champ d'application**
/	division réelle	Réels
/	division entière	Entiers
%	modulo (reste de la division entière) *	Entiers

* Nota 1. En Java comme dans la plupart des langages de programmation, *A % B* est toujours du signe de A (alors qu'en mathématiques le reste de la division euclidienne est toujours positif, le quotient se calculant en conséquence). Pour Java : A/B = sgn(A)*sgn(B)*|A|/|B| ; A%B = A – (A/B)*B

	-7 % 5	-7 / 5
En java	-2	-1
En math	3	-2

* Nota 2. En Java, l'opérateur modulo % peut aussi être appliqué sur des réels. Le résultat est alors le reste obtenu après soustraction de l'opérande droit à l'opérande gauche un nombre entier de fois. Toutefois, à cause des arrondis de calculs, le résultat n'est pas toujours celui qu'on attendrait (par exemple : 64.5 % 6.45 donne 6.449999999999998 et non 0.0). Grande prudence, donc, avec cet opérateur appliqué sur des réels.

Quelques opérateurs arithmétiques unaires		
Opérateur	**Fonction**	**Champ d'application**
++	Incrémentation	Entiers ou réels
--	décrémentation	Entiers ou réels

6.2 Opérateurs et expressions booléennes

Quelques opérateurs booléens binaires	
Opérateur	**Fonction**
&&	ET logique conditionnel *
||	OU logique conditionnel *
^	OU logique exclusif

* Avec ces opérateurs, l'évaluation de l'expression logique s'arrête dès qu'il est possible d'inférer la valeur finale de l'expression ; tous les opérandes ne sont donc pas nécessairement évalués.

Opérateur booléen unaire	
Opérateur	**Fonction**
!	Négation logique

6.3 Opérateurs relationnels

Quelques opérateurs relationnels (ou de comparaison)	
Opérateur	**Fonction**
==	égal à
!=	non égal à
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à

7 METHODES

7.1 Définition et appel d'une méthode ordinaire

Définition d'une procédure	Définition d'une fonction

public void *procName*(formalParams)
{
declarations
statements
}
	public *fctType fctName*(formalParams)
	{
	declarations
	statements
	return *expression ;*
	}

Appel d'une procédure	Appel d'une fonction
Appel interne à la classe de définition : *procName*(*actualParams*) ; ***Appel externe à la classe de définition :*** ***- méthode d'instance*** *objectName.procName*(*actualParams*); ***- méthode de classe*** *ClassName.procName*(*actualParams*);	***Appel interne à la classe de définition :*** *variable = fctName*(*actualParams*); ***Appel externe à la classe de définition :*** ***- méthode d'instance*** *variable =* *objectName.fctName*(*actualParams*); ***- méthode de classe*** *variable =* *ClassName.fctName*(*actualParams*);

Déclaration d'un paramètre formel (*formal parameter*)	Spécification d'un paramètre effectif (*actual parameter*)
Paramètre d'entrée (type primitif ou objet) : *parameterType parameterName* ***Paramètre de sortie de type primitif :*** Impossible (utiliser une fonction ou passer un paramètre de type objet) ***Paramètre de sortie de type objet :*** *parameterType parameterName*	***Paramètre d'entrée (type primitif ou objet) :*** *Expression* ***Paramètre de sortie de type primitif :*** Impossible ***Paramètre de sortie de type objet :*** *objectName*

- Les paramètres spécifiés dans la définition d'une méthode sont appelés *paramètres formels* ; ils doivent être précédés de leur type. Les paramètres apparaissant dans la forme d'appel sont appelés *paramètres effectifs*. S'il y a plusieurs paramètres, ils sont séparés par des virgules.
- Tous les paramètres de type primitif sont *passés par valeur* (c'est une copie de cet argument qui est transmise à la méthode, elle ne dispose pas de l'original). Tous les paramètres de type objet sont *passés par référence* (c'est une référence, i.e. un pointeur, qui est transmis à la méthode, et pas l'objet lui-même).

- Par défaut, les méthodes sont des *méthodes d'instance* : elles sont invoquées sur une instance de classe. Il existe aussi des *méthodes de classe* qui peuvent être invoquées sans instance. On spécifie une *méthode de classe* en la déclarant `static`
- La *signature*[1] d'une méthode est constituée de son nom et de la liste des types des paramètres. Dans une même classe, deux méthodes ne peuvent pas avoir la même signature ; elles peuvent toutefois porter le même nom (cela s'appelle la surcharge).
- Dans toute méthode d'instance ou constructeur, on peut référer à l'objet courant (i.e. l'objet dont la méthode ou le constructeur est en cours d'appel) par le mot clé `this`. On peut ainsi référer à tout membre (champ ou méthode) de l'objet courant par un identifiant de la forme `this.`*`membre`*

7.2 Constructeur

Un constructeur est une méthode particulière dont l'invocation est explicitement associée à la création d'une instance de classe et dont la finalité est d'initialiser proprement l'état de cet objet lors de sa création.

La définition d'un constructeur suit des règles syntaxiques spécifiques : le nom d'un constructeur est obligatoirement le nom de sa classe d'appartenance ; l'en-tête de sa définition ne lui spécifie pas de type.

Syntaxe de définition d'un constructeur d'une classe de nom ClassName :

```
public ClassName(formalParameters) {
      // … Fields initializations …
}
```

Un constructeur est typiquement utilisé en association avec l'opérateur `new`.

Exemple de syntaxe d'utilisation d'un constructeur d'une classe de nom ClassName :

```
ClassName object = new ClassName(actualParameters) ;
```

Une même classe peut comporter plusieurs constructeurs sous condition qu'ils n'aient pas les mêmes paramètres. Dans ce cas, pour éviter de la duplication de code, un constructeur peut en appeler un autre avec la syntaxe : `this(`*`actualParameters`*`)`.

Exemple :

```
public class Point {
    private int x, y;
    public Point( int x, int y ) {  // A first constructor
        this.x = x;
        this.y = y;
    }
    public Point() {                // A second constructor
        this( 0, 0 );
    }
    // …
}
```

[1] A noter que cette définition, adoptée par Java, ne prend pas en considération le type de la méthode. En ce sens, la « signature » correspond à ce qui est parfois appelé par ailleurs « profil » de la méthode.

7.3 Méthode main

La méthode `main`, comme en C ou C++, est le point d'entrée d'une application au lancement de l'exécution.

Syntaxe :

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

L'argument `args` capte les éventuels arguments de la ligne de commande, le premier argument de la ligne de commande étant stocké dans `args[0]` (et non dans `args[1]` comme en C/C++)

Exemple :

Considérons une application dont la classe principale – celle contenant la méthode `main` *– s'appelle* `Game`*. Supposons que l'application dépende de deux paramètres dont les valeurs doivent être fournies au lancement. Supposons également que le fichier source* `Game.java` *ait été compilé en un fichier exécutable* `Game.class`*. Le lancement de l'application par la commande :*

```
java Game Daniel 20
```

attribuera à `args` *le tableau* `{"Daniel", "20"}`

8 INSTRUCTIONS

Sur le plan syntaxique, les instructions (*statements*) peuvent être regroupées en 3 grandes catégories :

- les instructions simples : instructions d'affectation, instructions d'entrée / sortie, instructions d'appel à une procédure. *Une instruction simple se termine toujours par un point-virgule*. Un point-virgule seul définit une instruction vide.
- les instructions de contrôle : instructions de sélection, instructions de boucle, instructions de rupture de séquence.
- les blocs d'instructions. Un bloc d'instructions – aussi appelé instruction composée – est une séquence d'instructions enserrée *entre accolades*. Le bloc { } définit une instruction vide.

8.1 Instruction d'affectation

Syntaxe :

```
variable = expression ;
```

Le symbole = est l'opérateur d'affectation. La valeur de l'expression située en partie droite est affectée à la variable spécifiée en partie gauche. Le type de l'expression doit s'apparier au type de la variable. L'impact physique est différent selon que l'expression est de type primitif ou objet (voir chapitre 5).

8.2 Instructions de contrôle

8.2.1 Instructions de sélection

8.2.1.1 if-else

Première forme :

```
if (expression) {
      statements
}
```

Deuxième forme :

```
if (expression) {
      statements
} else {
      statements
}
```

Exemple :

```
if (field.size() == 0) {
     System.out.println("The field is empty.");
}
```

Exemple :

```
if (number < 0) {
      reportError();
} else {
      processNumber(number);
}
```

Exemple :

```
if (n < 0) {
      handleNegative();
} else if (n == 0) {
      handlezero();
} else {
      handlePositive();
}
```

8.2.1.2 switch

Première forme :

```
switch (expression) {
case value1 :
         statements ;
         break ;
case value2 :
         statements ;
         break ;
default :
         statements ;
         break ;
}
```

Deuxième forme :

```
switch (expression) {
case value1 :
         /* falls through */
case value2 :
         /* falls through */
case value3 :
         statements ;
         break ;
case value4 :
         /* falls through */
case value5 :
         statements ;
         break ;
default :
         statements ;
         break ;
}
```

- Une instruction `switch` peut avoir un nombre quelconque de labels `case`
- L'instruction `break` met fin à l'instruction `switch`. A défaut d'instruction `break`, l'exécution se poursuit sur les instructions des labels suivants. La deuxième forme ci-dessus exploite ce comportement : chacune des trois premières valeurs conduira à l'exécution de la partie *`statements`* associée à *`value3`* ; chacune des deux valeurs suivantes conduira à l'exécution de la partie *`statements`* associée à *`value5`* ;
- La clause `default` est optionnelle, mais recommandée. A défaut, il se peut qu'aucun cas ne conduise à une exécution.

Exemple :

```
switch (day) {
case 1 :
      dayString = "Monday" ;
      break ;
case 2 :
      dayString = "Tuesday" ;
      break ;
case 3 :
      dayString = "Wednesday" ;
      break ;
case 4 :
      dayString = "Thursday" ;
      break ;
```

```
case 5 :
      dayString = “Friday” ;
      break ;
case 6 :
      dayString = “Saturday” ;
      break ;
case 7 :
      dayString = “Sunday” ;
      break ;
default :
      error();
      break;
}
```

Exemple :

```
switch (winterMonth) {
case 11 :
      numberOfDays = 30 ;
      break ;
case 12 : /* falls through */
case  1 :
      numberOfDays = 31 ;
      break ;
case  2 :
      if (isLeapYear()) {
         numberOfDays = 29 ;
      } else {
         numberOfDays = 28 ;
      }
      break ;
default :
      error();
      break;
}
```

8.2.2 Instructions de boucle

8.2.2.1 while

La boucle `while` exécute un bloc d’instructions aussi longtemps que la valeur d’une expression donnée est `true`. L’expression est testée avant chaque exécution du corps de boucle, si bien que le corps de boucle peut être exécuté 0 fois.

Syntaxe :

```
while (expression) {
      statements
}
```

Exemple :

```
int i = 0 ;
while (i < text.size()) {
      System.out.printl(text.get());
      i++;
}
```

Exemple :

```
while (iter.hasNext()) {
      processObject(iter.next());
}
```

8.2.2.2 do-while

La boucle `do-while` exécute un bloc d'instructions aussi longtemps que la valeur d'une expression donnée est `true`. L'expression est testée après chaque exécution du corps de boucle, si bien que le corps de boucle est toujours exécuté au moins 1 fois.

Syntaxe :

```
do {
  statements
} while (expression);
```

Exemple :

```
do {
   input = readInput();
   if (input == null) {
      System.out.printl("Try again");
   }
} while (input == null);
```

8.2.2.3 for

La boucle `for` s'exécute aussi longtemps qu'une condition s'évalue à `true`. Avant que la boucle ne démarre, une instruction d'initialisation est exécutée exactement 1 fois. La condition est évaluée avant chaque exécution du corps de boucle (comme pour une boucle `while`). Une instruction de mise à jour des variables est exécutée après chaque exécution du corps de boucle.

Syntaxe :

```
for (initialization; condition; update) {
     statements
}
```

Exemple :

```
for (int i = 0; i < text.size(); i++ ) {
    System.out.println(text.get( )) ;
}
```

8.2.2.4 for each

Depuis la plate-forme 5.0 (voir chapitre 16), Java offre une boucle `for` améliorée pour parcourir les éléments d'un tableau ou d'une collection (se reporter aux sections respectives).

Syntaxe :

```
// for each element of the collection, process element
for (TypeOfElements element : collectionOfElements) {
      // … process element …
}
```

8.3 Instructions d'entrée / sortie

Java offre un nombre considérable de fonctions et procédures d'entrée / sortie, tant en mode texte que graphique. Nous nous limitons dans cette section aux principales instructions d'entrées/sorties en mode texte ; pour des entrées/sorties en mode graphique, voir chapitre 11.

Les méthodes d'entrée / sortie primaires appartiennent aux classes du paquetage `java.io`.

Depuis la plate-forme 5.0 (voir chapitre 16), la classe `Scanner` du paquetage `java.util` offre :

- des fonctionnalités de base pour lire facilement des données depuis tout flux d'entrée ;
- des sorties formatées comme en C.

En complément, la classe `java.util.Formatter` permet de spécifier des formats d'entrée ou de sortie.

8.3.1 Notion de flux

Les flux ou flots (*streams* en anglais) généralisent la notion de périphérique d'entrée / sortie. Un flux peut être vu comme une interface intelligente unidirectionnelle reliant la machine Java et le périphérique, et qui permet, en lecture aussi bien qu'en écriture, de s'affranchir des spécificités du périphérique concerné. Réaliser une entrée / sortie consiste ainsi à :

1) créer un flux du type approprié (sauf s'il existe déjà, ou est prédéfini (`System.in`, `System.out` ou `System.err`))
2) lire ou écrire, selon le cas, dans ce flux
3) et souvent, dans le cas d'une lecture, convertir la donnée lue dans le type cible souhaité

Pour une première approche, nous nous limiterons à des flux à accès séquentiel.

8.3.2 Entrée / sortie terminal

Les solutions suivantes ne s'affranchissent pas des erreurs de saisie. Pour gérer ce type d'erreur, se reporter au chapitre Exceptions (chapitre 10).

8.3.2.1 Affichage écran

Syntaxe d'un affichage écran en mode texte :

```
System.out.print(string);
System.out.println(string);

System.out.print(primitiveTypeExpression);
System.out.println(primitiveTypeExpression);
```

8.3.2.2 Affichage formaté

Depuis la plate-forme 5.0, Java offre également une méthode `printf` permettant des sorties formatées comme en C (voir classe `java.util.Formatter` pour la spécification des formats).

Exemples :

```
System.out.println("count = " + count);

System.out.printf("%s %5d \n", user, total);
```

Fonctionnement du `printf` et codes de formatage :

Un `printf` réclame une chaîne de caractères en premier paramètre. Cette chaîne peut être suivie de n variables, n≥0. C'est cette chaîne, qui peut contenir des codes de formatage, qui est imprimée. Un code de formatage commence par le caractère %. A chaque fois qu'un code de formatage est rencontré, `printf` imprime la variable suivante dans la liste de ses paramètres.

Principaux codes de formatage	*Commande l'affichage d'une valeur de type :*
`%c`	Caractère
`%d`	Entier
`%e`	Réel (en notation avec exposant)
`%f`	Réel
`%s`	Chaîne de caractères

Le code de formatage peut inclure une spécification de format du champ d'affichage.

Exemples de codes de formatage	*Champ d'affichage spécifié pour le paramètre du type considéré :*
`%6d`	Champ de 6 caractères
`%-6d`	Champ de 6 caractères avec justification à gauche
`%8.2f`	Champ de 8 caractères dont 2 à droite de la virgule

8.3.2.3 Lecture clavier

Exemple :

```
Scanner input = new Scanner(System.in);
String s1 = input.next();        // read a string (a word)
String s2 = input.nextLine();    // read the rest of the line
int n     = input.nextInt();     // read an integer
double r  = input.nextDouble();  // read a double
input.close();
```

8.3.3 Entrée / sortie fichier

Opérer sur un fichier consiste à :

1) créer un flux approprié aux opérations souhaitées
2) lire ou écrire dans le fichier
3) fermer le fichier

Les solutions suivantes ne s'affranchissent pas des erreurs d'entrée / sortie. Pour gérer ce type d'erreur, se reporter au chapitre Exceptions (chapitre 10).

8.3.3.1 Ecriture fichier

Depuis la plate-forme 5.0, Java offre également une méthode `printf` permettant des sorties formatées comme en C (voir classe `java.util.Formatter` pour la spécification des formats).

Exemple d'écriture dans un fichier texte :

```
PrintWriter out = new PrintWriter(outputFileName);
for (int i=0; i<10; i++) out.printf("%c%d",'\t',i);
out.close();
```

8.3.3.2 Lecture fichier

Exemple de lecture d'un fichier texte mot à mot :

```
Scanner in = new Scanner(new File(inputFileName));
String s ;
while (in.hasNext()) {
      s = in.next();          // read a string (a word)
      System.out.println(s);
}
in.close();
```

8.3.3.3 Contextes particuliers

Cas1. Ouverture en lecture d'un fichier texte stocké dans le fichier .jar exécuté

```
import java.io.InputStream ;
import java.util.Scanner ;

ClassLoader cl = ClassLoader.getSystemClassLoader();
InputStream is = cl.getResourceAsStream(inputFileName);
Scanner     in = new Scanner(is);
```

Cas2. Ouverture en lecture d'un fichier texte distant identifié par une URL

```
import java.net.URL;
import java.util.Scanner;

URL     url = new URL("http://www.esiee.fr/.../inputFileName");
Scanner in  = new Scanner(url.openStream());
```

Nota. Solution non applicable depuis une applet. En outre, bien s'assurer que toutes les conditions d'accessibilité au fichier sont levées (pas d'interdiction par le pare-feu local notamment !)

Cas3. Ouverture d'une nouvelle page html depuis une applet (de même répertoire)

```
import java.net.URL;
import javax.swing.JApplet;

URL   url = new URL(getDocumentBase(), "myFile.html");
getAppletContext().showDocument(url, "_blank");
repaint();
```

8.3.4 Lecture d'une chaîne de caractères formatée

Il est fréquent d'avoir à lire une chaîne de caractères dont on connaît le format, quel que soit le flux d'entrée (terminal ou fichier). Depuis la plate-forme 5.0, la classe `Scanner` du paquetage `java.util` offre des fonctionnalités très pratiques.

Exemple de lecture d'une chaîne de caractères formatée :

```
String str = "1 XXX 2 XXX yellow XXX blue XXX";
Scanner in = new Scanner(str).useDelimiter("\\s*XXX\\s*");
System.out.println(in.nextInt());
System.out.println(in.nextInt());
System.out.println(in.next());
System.out.println(in.next());
in.close();
// will display :
```

```
//    1
//    2
//    yellow
//    blue
```

9 HERITAGE

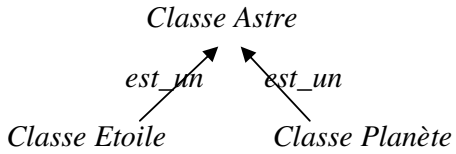
9.1 Notion d'héritage

L'héritage est une technique qui offre de nombreux avantages :

- évite la duplication de code
- permet la réutilisation de code dans un autre contexte
- facilite la maintenance des programmes
- facilite l'extension d'applications existantes.

L'héritage (*inheritance*) est une relation « est_un » (non pas « a_un » !!!) qui permet de définir une classe comme l'extension d'une autre.

Exemple :



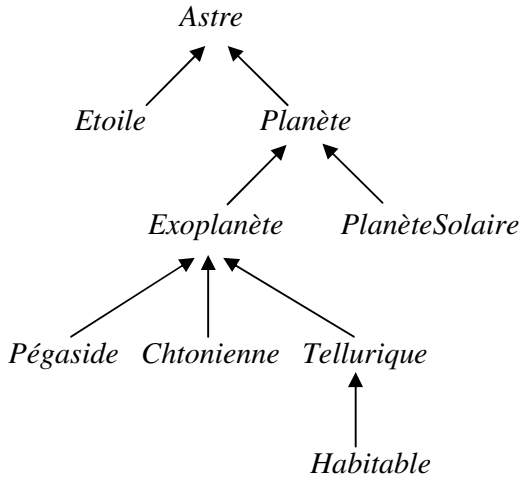
Une étoile est un astre ; une planète est un astre.
La classe Etoile étend la classe Astre = la classe Etoile hérite de la classe Astre.

Une super-classe est une classe qui est étendue par d'autres classes.

Une sous-classe est une classe qui étend (hérite de) une autre classe. Elle hérite l'état et le comportement de tous ses ancêtres, mais peut aussi les redéfinir.

Les classes qui sont liées par des relations d'héritage forment une hiérarchie d'héritage.

Exemple de hiérarchie d'héritage :



Le constructeur d'une sous-classe doit toujours invoquer le constructeur de la super-classe en première instruction (à défaut, Java essaiera d'insérer un appel automatique). La forme d'appel est la suivante :

```
super(actualParameters) ;
```

Toute classe sans super-classe explicite a `Object` comme super-classe.

Comme pour la hiérarchie de classes, les types forment une hiérarchie de types. Le type défini par la définition d'une sous-classe est un sous-type du type défini par la super-classe.

Exemple : Etoile est un sous-type du type Astre

Une variable peut contenir :

- un objet du type déclaré de la variable
- ou un objet de tout sous-type de ce type déclaré

Exemple :

```
Astre  a1 = new Astre();      // correct
Astre  a2 = new Etoile();     // correct (transtypage ascendant)
Astre  a3 = new Planete();    // correct (transtypage ascendant)
Etoile e1 = new Astre();      // erreur
Etoile e3 = (Etoile)a2 ;      // correct (un transtypage descendant
                              // doit être explicite)
```

Un objet d'un sous-type peut-être utilisé partout où un objet d'un super-type est attendu. Ceci s'appelle la substitution.

Java ne permet pas l'héritage multiple : une sous-classe ne peut hériter directement que d'une seule super-classe.

9.2 Qu'hérite une sous-classe ?

Une sous-classe hérite de tous les membres de sa super-classe qui sont accessibles, sauf si ce membre (attribut ou méthode) est redéfini dans la sous-classe.

Une sous-classe hérite de sa super-classe :

- les membres déclarés `public` ou `protected`
- les membres déclarés sans modificateur d'accès, si la sous-classe est dans le même paquetage que la super-classe.

Une sous-classe n'hérite pas de sa super-classe :

- les membres qui sont redéfinis dans la sous-classe
- les constructeurs.

Attention au choix des identificateurs dans la sous-classe : ce choix peut involontairement masquer un membre de la super-classe ! C'est particulièrement le cas pour les attributs : un attribut ayant le même nom qu'un attribut de sa super-classe masque l'attribut de la super-classe, même s'ils sont de types différents.

9.3 Redéfinir une méthode

Pour redéfinir une méthode dans une sous-classe, il faut qu'elle ait la même signature et le même type de retour que la méthode de la super-classe. La méthode redéfinie peut autoriser un accès plus large mais pas moins (ex : une méthode déclarée `protected` dans la super-classe peut être redéfinie `public` dans la sous-classe, mais pas `private`).

L'exemple typique est la méthode `toString` de la classe `Object` : appliquée à un objet, elle produit une chaîne de caractères formée du nom de sa classe et de son code de hachage, mais cette méthode peut être redéfinie dans toute classe.

Si la méthode redéfinie dans la sous-classe a simplement besoin d'ajouter quelques fonctionnalités complémentaires à celles de la méthode de la super-classe, il n'est pas besoin de la réécrire complètement car elle peut appeler la méthode de la super-classe par :

```
super.superclassMethodName(actualParameters)
```

Quelques règles :

- une méthode d'instance ne peut pas redéfinir une méthode de classe, et réciproquement
- une méthode *d'instance* d'une sous-classe ayant les mêmes signature et type qu'une méthode d'instance dans la super-classe redéfinit la méthode de la super-classe : la version de la méthode invoquée sur une instance de la sous-classe (type dynamique car liaison dynamique des méthodes d'instance) est celle de la sous-classe.
- une méthode *de classe* d'une sous-classe ayant les mêmes signature et type qu'une méthode de classe dans la super-classe masque la méthode de la super-classe : la version de la méthode invoquée dépend de si elle est appelée sur la super-classe ou sur la sous-classe (car liaison statique des méthodes de classe).
- une sous-classe ne peut pas redéfinir les méthodes déclarées `final` de sa super-classe
- une sous-classe doit redéfinir les méthodes déclarées `abstract` dans la super-classe, à moins que la sous-classe soit elle-même déclarée `abstract`.

Depuis la plate-forme 5.0 (voir chapitre 16) :

- Il est conseillé d'utiliser l'annotation (ou méta-donnée) `@Override` pour marquer toute méthode qui est supposée redéfinir une méthode héritée d'une classe parent. Cela permet au compilateur de signaler immédiatement une erreur si jamais la redéfinition n'est pas avérée (à cause par exemple d'une faute de frappe dans le nom de la méthode). En outre, en signalant les méthodes redéfinies, l'annotation `@Override` permet d'accroître la lisibilité du code.
- La covariance des types de retour est autorisée : une méthode d'une sous-classe peut retourner un objet dont le type est une sous-classe du type retourné par la méthode de même signature dans la super-classe. Cette caractéristique supprime le besoin de tests et de conversions de types excessifs.

Exemple :

```
@Override
public String toString() {
      // . . .
}
```

Exemple :

Soit une classe `Point` possédant deux attributs `x` et `y` ; soit une classe `ColoredPoint` héritant de `Point` et possédant un attribut spécifique `color`. Où et comment écrire la méthode `equals` sans duplication de code ?

```
/* In the class Point */
@Override public boolean equals( Object o ) {
   if ( o == null ) return false;                        // case null
   if ( this == o ) return true;                         // reflexivity
   if ( this.getClass() != o.getClass() ) return false;// symmetry
   // Now, this and o having the same class, compare the fields
   Point p = (Point)o;
   return ( this.x == p.x ) && ( this.y == p.y ) ;
}

/* In the class ColoredPoint extending Point */
@Override public boolean equals( Object o ) {
   // Verify if this and o have the same class and if their super
   // fields are equal
   if ( !super.equals(o) ) return false;
   // Now compare the specific fields
   ColoredPoint cp = (ColoredPoint)o;
   return this.color.equals(cp.color) ;
}
```

Exemple :

```
public class MotherClass {

    public void normal() {
       System.out.println("the normal method in MotherClass") ;
    }

    public static void hide() {
       System.out.println("the hide method in MotherClass") ;
    }

    public void override() {
       System.out.println("the override method in MotherClass");
    }
} // end MotherClass

public class DaughterClass extends MotherClass {

    @Override
    public static void hide() {
       System.out.println("the hide method in DaughterClass");
    }

    @Override
    public void override() {
       System.out.println("the override method in DaughterClass");
    }

    public static void main(String[] args) {
       DaughterClass o1 = new DaughterClass() ;
       MotherClass   o2 = o1 ;
```

```
        o1.normal();    // call super.normal()
        o2.normal();    // call super.normal()

        o1.hide();      // call this.hide()
        o2.hide();      // call super.hide()     !! liaison statique

        o1.override();  // call this.override()
        o2.override();  // call this.override()  !! liaison dynamique
    }
} // end DaughterClass
```

La méthode `main` *affichera :*

```
the normal method in MotherClass
the normal method in MotherClass
the hide method in DaugterClass
the hide method in MotherClass
the override method in DaugterClass
the override method in DaugterClass
```

9.4 Type statique / dynamique – Liaison statique / dynamique

Cette section reprend et formalise quelques uns des importants concepts introduits dans les sections précédentes.

9.4.1 Type statique / dynamique – Transtypage

Le *type statique* d'un objet est le type spécifié par sa déclaration. C'est le type déclaré. C'est le type à la compilation.

Le *type dynamique* d'un objet est le type (i.e. la classe d'appartenance) de l'objet instancié. C'est le type constaté. C'est le type effectif à l'exécution.

Le type dynamique d'un objet est toujours soit son type statique soit un sous-type de son type statique (sauf exceptions dues aux transtypages).

Exemple :

Soient une classe `A` *et une classe* `B` *sous-classe de* `A`*. Considérons le code suivant :*

```
A a ;
if ( myClass.booleanRandom() ) {
      a = new A() ;
} else {
      a = new B() ;     // transtypage ascendant implicite
}
```

Le type statique de `a` *est* `A` *; le type dynamique de* `a` *sera* `A` *ou* `B` *mais ne pourra être constaté qu'après que la condition aura été évaluée.*

Le *transtypage* (casting) appliqué à une référence n'est que la possibilité d'avoir une vue spécifique de l'objet pointé. Il ne change en rien le type de l'objet sous-jacent ! (qui garde bien évidemment tout au long de sa vie le type avec lequel il a été créé). L'opération de transtypage est purement syntaxique et ne fait qu'indiquer au compilateur le point de vue depuis lequel il doit voir l'objet.

Le *transtypage* vers un sur-type est dit *ascendant*. Le transtypage ascendant est implicite.

Le *transtypage* vers un sous-type est dit *descendant*. Un transtypage descendant doit être explicite. Il permet de forcer la compilation mais ne garantit pas nécessairement l'absence d'erreur d'exécution. Un transtypage n'est garanti correct que s'il respecte la règle suivante :

Soient `T1` et `T2` deux types définis. Soit `o` un objet de type dynamique `T3`. Considérons la déclaration :

```
T1 obj = (T2)o;
```

Ce transtypage est garanti possible, tant à la compilation qu'à l'exécution, si et seulement si `T3` est un sous-type de `T2` et `T2` un sous-type de `T1`.

L'opérateur `instanceof` permet de tester si un objet est d'un type dynamique donné ou de tout sous-type de ce type donné. Il peut donc permettre vérifier la compatibilité des types avant d'opérer un transtypage.

Exemple :

Supposons définies :

```
public class Shape2D { … }
public class Circle   extends Shape2D { … }
public class Triangle extends Shape2D { … }
public class Date { … }
```

Les déclarations suivantes génèreraient une erreur pour incompatibilité de types :

```
Shape2D s = (Shape2D)(new Date()) ; // erreur de compilation
Circle c = new Shape2D() ;          // erreur de compilation
Circle c = (Circle)(new Shape2D())  // erreur d'exécution
Triangle t = new Triangle() ;
Circle c = (Circle)t ;              // erreur d'exécution
```

Les déclarations suivantes sont saines :

```
Shape2D s = new Circle() ;
Circle c = (Circle)s ;
```

```
Shape2D s ;
if( obj instanceof Shape2D ) s = (Shaped2D)obj ;
```

9.4.2 Liaison statique / dynamique

Considérons l'invocation `o.m` où o est un objet de type statique `T` et de type dynamique `T'`, et `m` un message (attribut ou méthode) envoyé à `o`. Si le message `m` exécuté est celui dont le code est accessible depuis la classe `T`, alors la liaison de l'objet au message sera dite *statique* (liaison à la compilation) sinon elle sera dite *dynamique* (liaison à l'exécution).

En java :

- la liaison d'un objet avec ses variables d'instance est statique ;
- la liaison d'un objet avec ses méthodes est dynamique (sauf si méthode `static` ou `final`).

Exemple :

```
public class A {
    public boolean m1(A a) { return true ; }
    public boolean m2(A a) { return true ; }
}

public class B extends A {
    public boolean m1(A a) { return false ; }
    public boolean m3(A a) { return false ; }

    public static void main(String[] args) {
        A a = new B() ;
        System.out.prinln(a.m1(a));
        System.out.prinln(a.m2(a));
        // System.out.prinln(a.m3(a)); // problème
    }
}
```

La méthode `main` *affichera :*

```
false
true
```

Pourquoi ? Le type statique de `a` *est* `A`*, et son type dynamique* `B`*. C'est donc le* `m1` *de B qui est exécutée. Quant à la méthode* `m2`*, elle n'est pas définie dans* `B` *mais dans sa super-classe* `A` *: B hérite donc normalement de* `m2` *et* `m2` *est exécutée.*

La dernière instruction `System.out.prinln(a.m3(a))` *provoquerait une erreur à la compilation. Pourquoi ? Parce que le compilateur ne connaît que le type statique (i.e. le type déclaré) : la méthode* `m3` *n'étant pas définie dans la classe* `A` *ni dans une de ses super-classes, le compilateur ne trouve pas la définition de* `m3` *(bien que dans* `B`*) et génère une erreur. La solution serait le transtypage descendant préalable de* `a` *:* `System.out.prinln(((B)a).m3(a))`*. Ce transtypage serait correct car* `a` *possède déjà, par création, tous les attributs d'une instance de* `B`*.*

9.5 Polymorphisme

Un même appel de méthode peut invoquer des méthodes différentes car la liaison est réalisée dynamiquement (i.e. à l'exécution et non à la compilation), l'identification du type de l'objet associé à l'appel étant dynamique.

Exemple de polymorphisme d'héritage :

```
public class Shape {
      public void draw() { System.out.println( this + ".draw();" ); }
}

public class Circle extends Shape {
      @Override public String toString() { return "circle"; }
}

public class Square extends Shape {
      @Override public String toString() { return "square"; }
}
```

```
public class Picture {
      public static void main(String[] args) {
            Vector<Shape> v = new Vector<Shape>() ;
            v.add( new Circle() );
            v.add( new Square() );
            for (Shape s : v) {
                  System.out.prinln( s.draw() ); // polymorphism
            }
      }
}
```

La méthode `main` *affichera* :

```
circle.draw();
square.draw();
```

9.6 Classes et méthodes particulières

Une classe déclarée `final` ne peut pas être sous-classée.

Une méthode déclarée `final` ne peut pas être redéfinie dans une sous-classe.

Une classe déclarée `abstract` représente un concept abstrait : elle ne peut pas être instanciée et ne peut être que sous-classée.

Une méthode déclarée `abstract` est une méthode dont le corps n'est pas spécifié. Une classe abstraite peut contenir des méthodes non abstraites, mais toute classe contenant une méthode abstraite doit être déclarée `abstract`. Si une sous-classe n'implémente pas toutes les méthodes abstraites de sa super-classe, elle doit être déclarée `abstract`.

9.7 Interfaces

Une interface (au sens du langage Java, cf section 3.3) est un protocole de comportement qui peut être implanté par toute classe. Une interface est en dehors de la hiérarchie des classes. Une interface se différencie principalement d'une classe abstraite par ces caractéristiques :

- aucune méthode concrète (par opposition à abstraite) ne peut être définie dans une interface ;
- une classe peut implémenter plusieurs interfaces.

Tout comme on peut construire une hiérarchie de classes, on peut construire une hiérarchie d'interfaces. Mais alors qu'une classe ne peut étendre directement qu'une seule super-classe, (pas d'héritage multiple), une interface peut étendre directement plusieurs super-interfaces (héritage multiple possible).

Exemple : `public interface List<E> extends Collection<E> { … }`

Exemple :

```
public interface Predator { boolean chasePrey(Prey p); }
public interface Prey { boolean isChasedBy(Predator p); }
public interface Venomous { … }
public interface VenomousPredator extends Predator, Venomous { … }
public class Pet { … }
public class Cat extends Pet implements Predator { … }
public class Frog implements Predator, Prey { … }
```

10 EXCEPTIONS

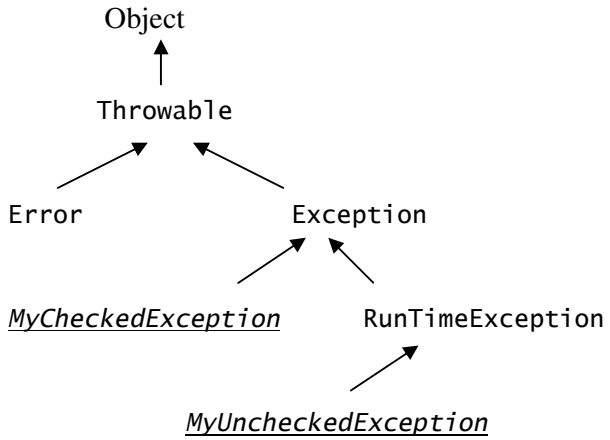
Une exception est un objet fournissant des informations relatives à une défaillance du programme. Une exception est générée dans chacune des situations suivantes :

- condition d'exécution anormale, comme par exemple : tentative de division par zéro, mémoire insuffisante, …
- exécution d'une instruction `throw`

Le paquetage `java.lang` définit nombre de classes d'exception qui couvrent les cas les plus courants.

10.1 Exceptions contrôlées et non contrôlées

Toute exception est une instance de la classe `Throwable` ou une de ses sous-classes :



Java divise les classes d'exception en deux catégories :

- Les exceptions *non contrôlées (unchecked)*. Ces exceptions concernent les cas « imprévus », c'est-à-dire les situations où il ne devrait pas y avoir de défaillance en fonctionnement normal. Elles concernent des situations qui pourraient être raisonnablement évitées (exemple : un indice de tableau hors limite). Ces exceptions indiquent généralement une erreur de programme. Elles ont normalement vocation à conduire à un arrêt immédiat et définitif du programme.
- Les exceptions *contrôlées (checked)*. Ces exceptions concernent les cas « prévus », c'est-à-dire les situations où le programmeur devrait s'attendre à ce qu'une opération puisse échouer. Elles concernent des situations de défaillance qui sont au delà du contrôle du programmeur (exemple : un disque devenu plein rendant impossible une écriture dans un fichier). Ces exceptions ont normalement vocation à être récupérées pour permettre au programme de se poursuivre normalement. La gestion d'une exception contrôlée impose l'utilisation de la clause `throws` et de l'instruction de contrôle `try .. catch`.

Les exceptions *non contrôlées* sont les classes `RuntimeException`, `Error` et leurs sous-classes ; toutes les autres classes et sous-classes sont des exceptions *contrôlées*.

10.2 Classes d'exception standard

10.2.1 La hiérarchie Error

La classe `Error` est la super-classe d'une hiérarchie d'erreurs graves *non contrôlées*. Cette hiérarchie est habituellement réservée aux erreurs d'exécution système. Exemples de sous-classes prédéfinies :

```
NoSuchMethodError
StackOverFlowError
OutOfMemoryError
IllegalAccessError
```

10.2.2 La hiérarchie Exception

La classe `Exception` est la super-classe d'une hiérarchie d'exceptions, exclusion faite de la sous-hiérarchie `RuntimeException`, *contrôlées*. Exemples de sous-classes d'exceptions *contrôlées* prédéfinies :

```
DataFormatException
IOException
      EOFException
      FileNotFoundException
```

10.2.3 La hiérarchie RuntimeException

La classe `RuntimeException` est la super-classe d'une hiérarchie d'exceptions *non contrôlées*. Exemples de sous-classes prédéfinies :

```
ArithmeticException
IllegalArgumentException
      NumberFormatException
IllegalStateException
IndexOutOfBoundsException
NullPointerException
```

10.3 Lancement d'une exception

L'instruction `throw` permet de lancer une exception. Il est recommandé de ne l'appliquer qu'à des exceptions *non contrôlées.*

Syntaxe :

```
throw new ClassOfException("diagnostic message to print") ;
```

Exemple :

```
if (key == null) {
    throw new NullPointerException("null key found in method x");
}
```

Exemple :

```
if (key == 0) {
    throw new IllegalArgumentException("empty key received") ;
}
```

Exemple (dans un constructeur) :

```
if (this.name.length() == 0) {
   throw new IllegalStateException( “name must be provided - ”
                                    + “object creation aborted”
                                   );
}
```

Exemple. L’opération `remove()` *de l’interface* `Iterator` *est spécifiée optionnelle, c'est-à-dire que son implémentation pourrait être dans ce cas :*

```
public void remove() {
      throw new UnsupportedOperationException();
}
```

10.4 Capture d’une exception

La capture d’une exception n’est obligatoire que pour les exceptions que l’on veut contrôler.

Dans le cas des exceptions *contrôlées*, le compilateur renforce ses contrôles à la fois dans la méthode qui lance une telle exception et dans l’appelant de cette méthode.

10.4.1 Clause throws

Une clause `throws` est une déclaration ajoutée à l’en-tête d’une méthode susceptible de lever une exception *contrôlée*. Elle déclare les exceptions *contrôlées* à prendre en compte dans le cadre de cette méthode. Ceci est nécessaire pour que le compilateur puisse savoir quelles exceptions contrôler.

Par convention, les exceptions *non contrôlées* ne devraient pas être incluses dans une clause `throws`.

Syntaxe :

```
throws ExceptionClass1, ExceptionClass2, …
```

Exemple :

```
public void saveToFile(String filename) throws IOException {
      …
}
```

10.4.2 Instruction try .. catch

L’instruction `try .. catch` permet de capturer une exception générée par l’appel d’une méthode. Quand une exception est générée, le contrôle est transféré du code qui a généré l’exception à la clause `catch` qui a capturé l’exception.

La clause `catch` doit avoir un seul paramètre formel. Le type de ce paramètre doit être la classe `Throwable` ou une de ses sous-classes. Une instruction `try` peut contenir plusieurs blocs `catch`. Dans ce cas, l’exception la plus générale doit être traitée en dernier.

Syntaxe :

```
try {
      // protect one or more statement here
} catch (ExceptionClass e) {
      // report and recover from the exception here
} finally {
      // in fine, in any case, always execute these statements
}
```

Le bloc `finally` est optionnel, mais il permet souvent d'éviter la duplication de code.

Exemple (à partir de la plate-forme 5.0) :

```
// Read a double from the standard input stream
Scanner input = new Scanner(System.in);
double x ;
try {
      x = input.nextDouble() ;
} catch(InputMismatchException e) {
      System.err.println("Error - Double waited - " + e);
}
input.close() ;
```

Exemple (à partir de la plate-forme 5.0) :

```
// Create a copy of a text file
Scanner     in  = null ;
PrintWriter out = null ;
try {
    in  = new Scanner(new File(inputFileName));
    out = new PrintWriter(outputFileName);
    while (in.hasNext()) {
        out.println(in.nextLine());
    }
} catch (Exception e) {
    System.err.println("Error in reading/writing file - " + e);
} finally {
    if (in  != null) in.close() ;
    if (out != null) out.close() ;
}
```

Exemple (à partir de la plate-forme 5.0) :

```
// Read and print a text file of doubles
Scanner in = null ;
Try {
      in = new Scanner(new File(inputFileName));
      while(in.hasNext()) {
            System.out.println(Double.parseDouble(in.next())) ;
      }
} catch (FileNotFoundException e) {
      System.err.println("Unable to open the input file." + e);
} catch (NumberFormatException e) {
      System.err.println("A token is not a double." + e) ;
} catch (Exception e) '
      System.err.println("Problem in reading the file " + e) ;
} finally {
      if (in != null ) {
      try {
            in.close() ;
      } catch (Exception e) {
            System.err.println("Error in closing the file."
                              + e) ;
      }
}
```

10.5 Définir une nouvelle classe d'exception

Une des principales raisons conduisant à la définition d'une nouvelle classe d'exception est le besoin d'inclure des informations complémentaires dans l'objet exception afin de permettre le diagnostic et la récupération de l'erreur.

Toute nouvelle classe d'exception sera définie comme sous-classe d'une classe d'exception existante de la hiérarchie `Exception`.

Exemple :

```
public class WrongValueException extends Exception {
      private String key ;
      public WrongValueException(String key) {
            this.key = key;
      }
      public String getKey() {
            return key ;
      }
      public String toString() {
            return “Wrong value “ + key + “ found” ;
      }
}
```

11 ANNEXE 1 – PROGRAMMATION GRAPHIQUE

Références :

Kathy WALRATH, & al. The JFC Swing Tutorial – A guide to Constructing GUIs (2nd ed.). Addison Wesley, 2004.

Bases de Swing et exemples de programmation d'interfaces graphiques :
java.sun.com/docs/books/tutorial/uiswing/mini/index.html

Gestion des événements et exemples de programmation :
java.sun.com/docs/books/tutorial/uiswing/events/api.html

Swing Architecture Overview *java.sun.com/products/jfc/tsc/articles/architecture*

Ce chapitre introduit à la programmation graphique avec la bibliothèque `javax.swing`

11.1 Swing

Il existe deux grands paquetages d'utilitaires Java pour réaliser des interfaces graphiques :

- AWT (Abstract Window Toolkit) : le paquetage primitif. Les composants de ce paquetage s'importent par :

  ```
  import java.awt.* ;
  ```

- Swing : le plus récent et le plus évolué, construit sur AWT, c'est le paquetage recommandé. Les composants de ce paquetage s'importent par :

  ```
  import javax.swing.* ;
  ```

Les deux paquetages Swing les plus communément utilisés sont :

```
javax.swing
javax.swing.event
```

Les classes de javax.swing les plus courantes sont :

```
JFrame
JPanel
JLabel
JTextField
JButton
JOPtionPane
```

Exemple de saisie / affichage dans une boîte de dialogue :

```
// import javax.swing.JOptionPane ;
// use method showInputDialog to prompt for some input
String inputValue = JOptionPane.showInputDialog("Value ?");
// use method showMessageDialog to display a message
JOptionPane.showMessageDialog(null,"A message …")
```

11.2 Conteneurs et composants

Tout objet graphique est composé de différents objets : des conteneurs (containers) et des composants atomiques, ces différents constituants étant organisés en une hiérarchie d'inclusion.

Principales catégories de conteneurs et composants :

- les conteneurs de plus haut niveau

  Pour qu'un objet graphique puisse être affiché à l'écran, la hiérarchie de ses constituants doit nécessairement avoir pour racine l'un des trois conteneurs suivants :

  - Frame (classe `JFrame`) : conteneur le plus général, il fournit la fenêtre principale dans laquelle les autres composants Swing pourront se dessiner.
  - Dialog (classe `JDialog`) : plus limité qu'un Frame, il fournit une fenêtre de dialogue ; la classe `JOptionPane` permet de créer des fenêtres de dialogue simples et standard ; la classe `JDialog` permet de créer des fenêtres de dialogue personnalisées.
  - Applet (classe `JApplet`) : spécialisé pour des interfaces web, il fournit une fenêtre qui sera affichée par un navigateur web.

- les conteneurs intermédiaires, notamment :

  - Panel (classe `JPanel`) : simplifie le positionnement des objets graphiques atomiques.

    Tout conteneur de haut niveau contient un conteneur intermédiaire spécial appelé `ContentPane`. C'est le conteneur intermédiaire principal. Il contient typiquement un ou plusieurs panels.

- les composants atomiques, par exemple :

  - Button (classe `JButton`)
  - Label (classe `JLabel`)
  - Text field (classe `JTextField`)

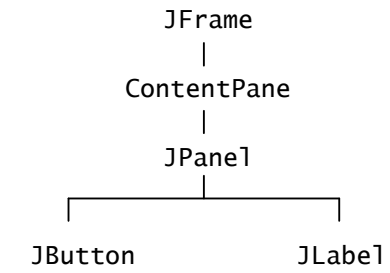
Sauf pour les conteneurs de plus haut niveau, tous les composants (conteneur intermédiaire ou composant atomique) qui commencent par J héritent de la classe `JComponent`.

Pour ajouter un objet à un conteneur, on utilise une méthode add. Cette méthode prend en général un deuxième argument qui spécifie la mise en page.

Exemple :

```
// import javax.swing.* ;
frame  = new JFrame(. . .);
pane   = new JPanel();
button = new JButton(. . .);
label  = new JLabel(. . .);
pane.add(button);
pane.add(label);
frame.getContentPane().add(pane, BorderLayout.CENTER);
frame.pack(); // sizes at or above preferred sizes
```

```
frame.setVisible(true);
// l'objet frame ainsi créé a pour hiérarchie :
```



Exemple :

```
import javax.swing.JApplet ;
public class AppletExample extends JApplet {
      public void init( ) {
            JPanel pane = new JPanel() ;
            pane.add(new JLabel("Hello, World !") ;
            this.setContentPane(pane) ;
      }
}
```

11.3 Gestion de la mise en page

La mise en page (layout) consiste à fixer la taille et la position des composants.

Tout conteneur possède un gestionnaire de mise en page par défaut. S'il ne convient pas, on peut le remplacer.

Les composants peuvent solliciter une taille et un alignement donnés, mais, in fine, c'est le gestionnaires de mise en page du conteneur qui conserve la décision finale.

Il existe 6 gestionnaires de mise en page :

- `BorderLayout` : gestionnaire de mise en page par défaut pour tout `ContentPane` ; définit 5 zones de placement : `NORTH, SOUTH, EAST, WEST, CENTER`
- `FlowLayout` : gestionnaire de mise en page par défaut pour tout `JPanel` ; positionne simplement les composants de gauche à droite, passant à une nouvelle ligne si nécessaire
- `BoxLayout` : gestionnaire de mise en page très flexible ; positionne les composants sur une seule ligne ou colonne en respectant les attentes de composants en matière de taille maximale et d'alignement
- `GridLayout` : gestionnaire de mise en page très simple ; crée un ensemble de composants de même taille et les affiche dans le nombre de lignes et de colonnes spécifié
- `GridBagLayout` : gestionnaire de mise en page le plus sophistiqué et le plus flexible ; aligne les composants en les plaçant dans une grille de cellules, permettant à certains de s'étendre sur plusieurs cellules (les largeurs des rangées et les hauteurs des colonnes, respectivement, peuvent être différentes)

- `CardLayout` : gestionnaire de mise en page pour usage spécial ; permet d'implémenter une zone qui contient différents composants à différents moments ; utilisé en combinaison avec d'autres gestionnaires de mise en page

Exemple de changement du gestionnaire de mise en page par défaut :

```
JPanel pane = new JPanel() ;
pane.setLayout(new BorderLayout());
```

La taille et l'alignement d'un composant peut être spécifié par les méthodes suivantes : `setMinimumSize, setPreferredSize, setMaximumSize, setAlignmentX, setAlignmentY`.

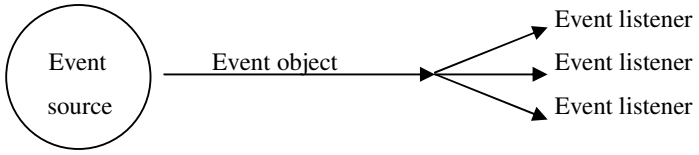
11.4 Gestion des événements

11.4.1 Notion d'événement

A chaque fois que l'utilisateur agit sur le clavier ou la souris, un événement survient, un objet événement est généré.

Un objet événement est un objet contenant des informations sur la source et la nature de l'événement survenu.

Swing offre des interfaces (au sens Java) appelées « écouteurs d'événements » (event listeners) qui permettent de capter les événements de façon sélective afin de pouvoir les traiter.



Exemple d'événement	Type de Listener
Clic d'un bouton graphique ou enfoncement de la touche return lors d'une saisie dans un champ texte	`ActionListener`
Fermeture d'une fenêtre graphique (frame)	`WindowListener`
Clic de la souris lors du passage du curseur sur un composant graphique	`MouseListener`

Afin de faciliter le travail du programmeur, Swing associe généralement à chaque interface Listener une classe adaptateur (Adapter). Si cet adaptateur existe, c'est avec lui que l'on travaille.

Exemple d'interface Listener	Classe Adapter associée
ActionListener	None
WindowListener	WindowAdapter
MouseListener	MouseAdapter
KeyListener	KeyAdapter

11.4.2 Réaliser un gestionnaire d'événement

Réaliser un gestionnaire d'événement (event handler) requiert trois éléments de code :

- déclarer une classe gestionnaire d'événement : cette classe doit implémenter une interface Listener ou étendre une classe Adapter

  Exemple :
  ```
  import java.awt.event.* ;
  public class MyClass implements ActionListener { . . . }
  ```

- définir dans cette classe gestionnaire d'événement les méthodes de l'interface Listener

  Exemple :
  ```
  public void actionPerformed(ActionEvent e) { .. . }
  ```

- définir une instance de cette classe gestionnaire d'événement et l'enregistrer sur le(s) composant(s) concerné(s)

  Exemple :
  ```
  someComponent.addActionListener(instanceOfMyClass) ;
  ```

Le type `ActionListener` permet de définir les gestionnaires d'événement les plus simples et les plus communs. Lorsqu'un événement de ce type est généré (clic d'un bouton graphique ou enfoncement de la touche return lors d'une saisie dans un champ texte), un message `actionPerformed` est envoyé à tous les écouteurs d'événements qui sont enregistrés sur le composant concerné.

Une classe gestionnaire d'événement est généralement implantée sous la forme d'une classe interne (inner class) au sein d'une classe graphique.

Exemple :

```
// Ecriture d'un MouseListener
public class myClass extends JPanel {
      . . .
      someObject.addMouseListener(new MyAdapter());
      . . .
      class MyAdapter extends MouseAdapter {
            public void mouseClicked(MouseEvent e) {
                  . . .
            }
      }
}
```

Typiquement, une classe gestionnaire d'événement est implantée sous la forme d'une classe interne anonyme.

Exemple (solution équivalente à celle de l'exemple précédent) :

```
// Ecriture d'un MouseListener
public class myClass extends JPanel {
      . . .
      someObject.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                  . . .
            }
      } ) ;
      . . .
}
```

11.5 Affichage / fermeture d'une fenêtre graphique

L'affichage / le masquage d'une fenêtre graphique de type `JFrame` se commande par la méthode `setVisible`.

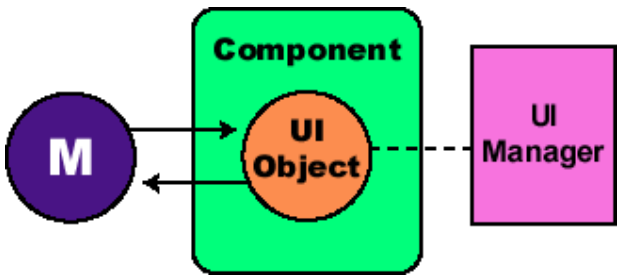
Quand l'utilisateur ferme une fenêtre graphique de type `JFrame`, cette fenêtre n'est en fait, par défaut, que masquée. Quoiqu'invisible, l'objet graphique existe toujours et le programme peut à nouveau le rendre visible. Si l'on désire un comportement différent, il est nécessaire :

- soit d'enregistrer sur cet objet graphique un gestionnaire d'événement de type `WindowListener` et de capter le message `windowClosing`
- soit de spécifier le comportement par défaut en utilisant la méthode `setDefaultCloseOperation`. Le comportement est indiqué en argument par une constante (définie dans l'interface `WindowConstants`). Exemple d'argument : DISPOSE_ON_CLOSE

11.6 Architecture des composants Swing

Objectif : des applications facilement adaptables.

L'architecture des composants Swing est basée sur une adaptation de la très renommée architecture MVC (« Modèle-Vue-Contrôleur » ; voir section 13.2). Dans la communauté Swing, elle est appelée « Architecture à modèle séparable » :



La partie Modèle d'un composant Swing (partie qui représente les données de l'application) est traitée comme un élément séparé, tout comme le préconise la conception MVC ; mais Swing fusionne les parties Vue (qui prend en charge la représentation visuelle des données) et Contrôleur (qui traite les entrées utilisateur) de chaque composant en un seul objet interface utilisateur (User Interface Object).

Il est en général considéré comme une bonne pratique de centrer l'architecture d'une application autour de ses données plutôt qu'autour de son interface utilisateur. Pour supporter ce paradigme[1], Swing définit une interface Modèle séparée pour chaque composant. Cette séparation fournit au programme d'application la possibilité de connecter facilement des composants Swing dans son implémentation modèle propre.

La table suivante montre les interfaces Modèles offertes par les composants Swing :

Component	Model Interface	Model Type
JButton	ButtonModel	GUI
JToggleButton	ButtonModel	GUI/data
JCheckBox	ButtonModel	GUI/data
JRadioButton	ButtonModel	GUI/data
JMenu	ButtonModel	GUI
JMenuItem	ButtonModel	GUI
JCheckBoxMenuItem	ButtonModel	GUI/data
JRadioButtonMenuItem	ButtonModel	GUI/data
JComboBox	ComboBoxModel	Data
JProgressBar	BoundedRangeModel	GUI/data
JScrollBar	BoundedRangeModel	GUI/data
JSlider	BoundedRangeModel	GUI/data

Component	Model Interface	Model Type
JTabbedPane	SingleSelectionModel	GUI
JList	ListModel	Data
JList	ListSelectionModel	GUI
JTable	TableModel	Data
JTable	TableColumnModel	GUI
JTree	TreeModel	Data
JTree	TreeSelectionModel	GUI
JEditorPane	Document	Data
JTextPane	Document	Data
JTextArea	Document	Data
JTextField	Document	Data
JPasswordField	Document	Data

Les interfaces Modèles fournies par Swing relèvent de deux grandes catégories : les types « Etat de l'interface graphique » (GUI), qui définissent le statut visuel des commandes de l'interface graphique ; et les types « Données d'application » (Data), qui représentent des données quantifiables ayant sens pour l'application. Certains modèles relèvent de types partagés (GUI/data).

Il est recommandé d'utiliser la catégorie « Données d'application » (Data) des modèles de Swing car ils renforcent grandement l'adaptabilité et la modularité des applications sur le long terme.

11.7 Applets

Java permet de créer deux types de programmes :

- des applications, qui s'exécutent directement sous le contrôle de la machine virtuelle Java
- des appliquettes (ou applettes ; en anglais : applets) destinées à être exécutées par un navigateur Web

Pour réaliser une applette, le minimum consiste à :

[1] Un paradigme est une vision du monde qui repose sur un modèle.

- créer une sous-classe de la classe `JApplet` (voir section 11.2)
- y créer la méthode `init()`, qui sera lancée par le navigateur
- référencer cette classe dans la balise `<APPLET CODE= …>` d'un fichier HTML

Syntaxe :

```
import javax.swing.JApplet ;
public class AppletClassName extends JApplet {
      // … Fields …
      public void init( ) {
            …
      }
}
```

Syntaxe :

```
<APPLET CODE="AppletClassName.class" WIDTH=anInt HEIGHT=anInt >
</APPLET>
```

12 ANNEXE 2 – THREADS

Références :

Mary CAMPIONE, & al. *The Java Tutorial (3rd ed.)*. Addison Wesley.

Programmation concurrente, processus, threads
java.sun.com/docs/books/tutorial/essential/concurrency/procthread.html

Les threads permettent de programmer des tâches qui, à l'intérieur d'un même programme, s'exécuteront concurremment. Ce sont des processus légers.

La tâche à exécuter par un thread doit être implémentée dans une méthode `run()`. Il existe deux façons de fournir la méthode `run()` :

- définir une sous-classe de la classe `Thread` et y redéfinir la méthode `run()`
- définir une classe qui implémente l'interface `Runnable` et y définir la méthode `run()`

12.1 Créer un thread par extension de la classe Thread

Exemple :

```
// Example of defining a thread as extending Thread
public class SimpleThread extends Thread {

      public SimpleThread(String threadName) {
            super(threadName);
      }

      // The task of this thread is to print 10 times its name
      // at a random rate
      public void run() {
            for (int i = 0; i < 10; i++) {
                  System.out.println(i + " " + getName());
                  try {
                        sleep((long)(Math.random() * 1000));
                  } catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + getName());
      }
} // end class SimpleThread


// Example of running two threads concurrently.
// "Jamaica" and "Fiji" Will be printed randomly 10 times.
public static void main (String[] args) {
      new SimpleThread("Jamaica").start();
      new SimpleThread("Fiji").start();
}
```

12.2 Créer un thread par implémentation de l'interface Runnable

Cette façon de faire s'impose lorsque la classe à créer doit sous-classer une classe quelconque (le cas typique étant une Applet), parce que Java ne supporte pas l'héritage multiple.

Exemple :

```
// Example of defining a thread as implementing Runnable
public class SimpleThread implements Runnable {

      String threadName ;

      public SimpleThread(String threadName) {
            this.threadName = threadName ;
      }

      // The task of this thread is to print 10 times its name
      // at a random rate
      public void run() {
            for (int i = 0; i < 10; i++) {
                  System.out.println(i + " " + threadName);
                  try {
                    Thread.sleep((long)(Math.random() * 1000));
                  } catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + threadName);
      }
} // end class SimpleThread


// Example of running two threads concurrently
// ("Jamaica" and "Fiji" Will be printed randomly 10 times)
public static void main (String[] args) {
      new Thread(new SimpleThread("Jamaica")).start();
      new Thread(new SimpleThread("Fiji")).start();
}
```

12.3 Synchronisation et communication entre threads

12.3.1 Sections critiques et synchronisation

Dans un programme, des segments de code qui accèdent à une même donnée depuis des threads concurrents sont appelés *sections critiques*.

Lorsque deux threads concurrents invoquent des méthodes qui opèrent sur une même donnée partagée (méthodes critiques), il est nécessaire que Java puisse synchroniser ces méthodes afin de garantir un accès sûr à l'information. A cette fin, chacune de ces méthodes doit être déclarée avec le modificateur `synchronized`. A l'exécution, l'accès à la donnée partagée sera ainsi automatiquement verrouillé tant qu'elle sera utilisée par une méthode critique.

Exemple :

get et put sont supposées deux méthodes critiques de l'objet partagé.

```
public synchronized int get() { … }
public synchronized void put(int value) { … }
```

12.3.2 Communication entre threads

Pour que deux threads, processus indépendants qui se déroulent de façon asynchrone, puissent se communiquer des données, il est nécessaire qu'ils sachent se synchroniser autour d'un sas de communication. Le thread émetteur de la donnée est appelé *producteur* ; le thread destinataire de la donnée est appelé *consommateur.*

Le thread producteur de la donnée doit être capable de :
- attendre que l'espace de stockage du sas de communication soit libre ;
- puis y déposer une nouvelle donnée ;
- puis informer les threads consommateurs de la mise à disposition de la donnée.

De façon symétrique, le thread consommateur de la donnée doit être capable de :
- attendre que la donnée soit mise à disposition dans l'espace de stockage du sas de communication ;
- puis lire cette donnée ;
- puis informer les threads producteurs de la disponibilité du sas de communication ou leur accuser réception.

C'est au sas de communication qu'on confie la charge de ces synchronisations. La mise en attente se réalise par la méthode `wait()`, la notification aux autres threads par les méthodes `notify()` ou `notifyAll()`. Ces méthodes sont toutes deux définies dans la classe `Object()`.

Exemple de classe définissant un sas de communication d'un seul élément de type entier (attribut `contents`*). Y noter en particulier la sécurisation apportée par l'usage d'un while et non d'un if afin de prévenir de faux réveils éventuels.*

```
public class CubbyHole {
      private int contents ;
      private boolean available = false ;
      public synchronized int get() {    // called by Consumer
            while ( available == false ) {
                  try {
                        wait() ; // wait for Producer to put value
                  } catch (InterruptedException e) () { }
            }
            available = false ;
            notifyAll();
            return contents ;
      }
      public synchronized void put(int x) { // called by Producer
            while ( available == true ) {
                  try {
                        wait() ; // wait for Consumer to get value
                  } catch (InterruptedException e) () { }
            }
            contents = x ;
            available = true ;
            notifyAll();
      }
}
```

13 ANNEXE 3 – CONCEPTION ET DEVELOPPEMENT EN JAVA

Références :

- Conseils généraux pour une bonne conception :
  David J. BARNES, Mickaël KÖLLING. *Objects First With Java* (4th ed). Chap 7.
  Prentice Hall / Pearson Education, 2008. ISBN 0136060862
  www.bluej.org/objects-first/

www.bluej.org/objects-first/chapters/objects-first-ch7.pdf

Architecture Modèle-Vue-Contrôleur :
java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html

Développement de tests unitaires avec le cadre d'applications `JUnit` :
junit.sourceforge.net/doc/cookbook/cookbook.htm

JavaBeans :
java.sun.com/products/javabeans/
java.sun.com/docs/books/tutorial/javabeans/index.html

Design patterns :
Laurent DEBRAUWER. *Design Patterns - Les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java*. Editions ENI, 2007, 329 p. ISBN 2746038870

Cette section va au-delà du langage Java pour aborder quelques aspects de la conception et du développement de programmes Java.

13.1 Conseils généraux pour une bonne conception des classes

RECHERCHER UN COUPLAGE FAIBLE. Le couplage décrit l'interconnectivité entre les classes. Le couplage est faible quand chaque classe est largement indépendante des autres et communique avec elles via une interface petite et bien définie (l'interface d'une classe est la partie visible publique de cette classe : elle se définit typiquement par la liste des membres (attributs, constructeurs, méthodes) non privés associés à leurs commentaires de documentation).

RECHERCHER UNE COHESION FORTE. La cohésion décrit l'adéquation entre une unité de code et une entité ou tâche logique. La cohésion est forte quand chaque unité de code (méthode, classe, module, …) est responsable d'une entité ou tâche très bien définie.

UTILISER AU MAXIMUM L'ENCAPSULATION. L'encapsulation est une technique de base pour réduire le couplage. Elle contribue à séparer le quoi et le comment, la vue et l'implémentation, la définition et l'utilisation. Principe de base : rendre les attributs privés et utiliser des méthodes d'accès.

DECOUPLER ENTREES, TRAITEMENTS ET SORTIES. Voir en particulier section 13.2

ADOPTER UNE CONCEPTION DIRIGEE PAR LES RESPONSABILITES. L'attribution des bonnes responsabilités aux bonnes classes est l'un des problèmes les plus délicats de la conception orientée objet. La programmation dirigée par les responsabilités est un processus de conception de classes par attribution de responsabilités bien définies à chaque classe. Cette

approche peut être utilisée pour déterminer quelle classe devrait implémenter telle fonctionnalité. Une conception dirigée par les responsabilités contribue à réduire le couplage.

EVITER LA DUPLICATION DE CODE. Eviter qu'un même segment de code ne se retrouve plus d'une fois dans une application.

UTILISER L'HERITAGE A BON ESCIENT. L'héritage est une relation « est_un », pas une relation « a_un ». Dans une hiérarchie d'héritage, les objets doivent avoir une relation « est_un » avec leur classe de base, afin que l'héritage de tous les attributs, propriétés et méthodes de la classe de base garde sens (même si la sous-classe en spécialise ou les complète). Les objets ne relevant pas strictement d'une relation « est_un » doivent se traduire par des classes ou interfaces sans relation d'héritage entre elles.

LIMITER LA TAILLE ET COMPLEXITE. Ne pas mettre trop de choses dans une méthode : une méthode est trop longue si elle fait plus d'une tâche logique. Ne pas tout mettre dans une seule classe : une classe est trop complexe si elle représente plus d'une entité logique.

CHERCHER A FACILITER DES CHANGEMENTS LOCALISES. Réaliser des changements dans une classe ne devrait avoir qu'un minimum d'impact sur les autres classes. La localisation des changements est l'un des buts recherchés d'une bonne conception de classe. Elle est le produit d'un couplage faible et d'une cohésion forte.

13.2 Architecture d'une application interactive : le modèle MVC

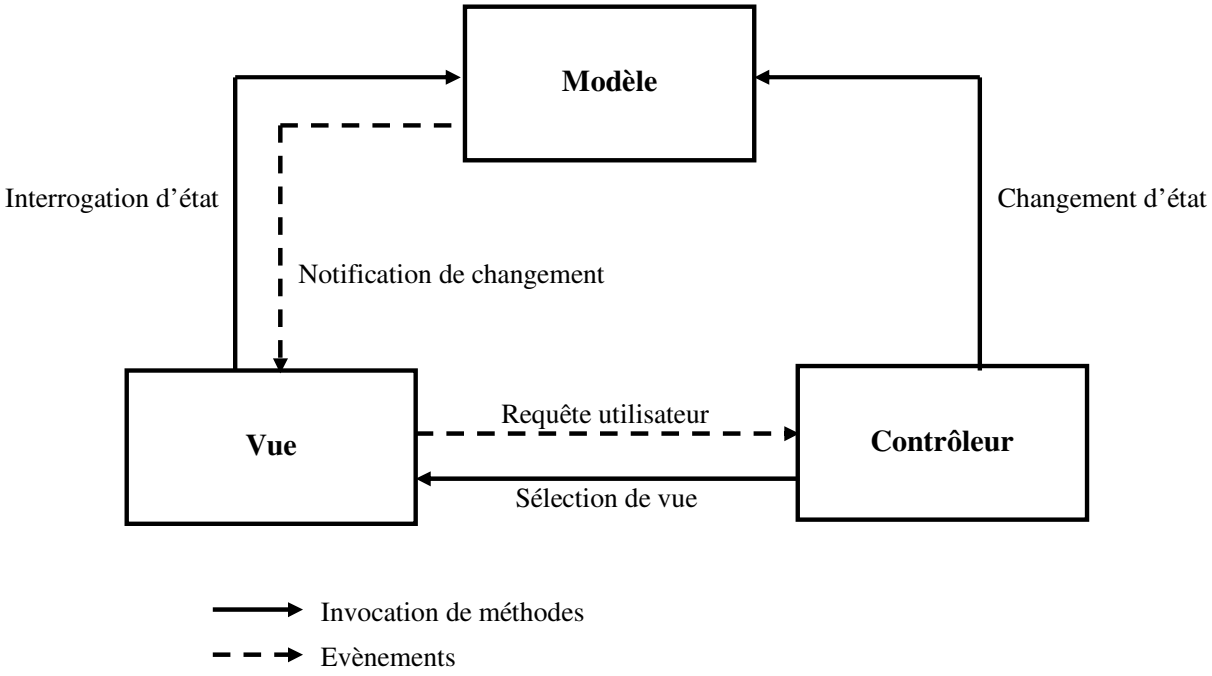
L'architecture *Modèle-Vue-Contrôleur* (Model-View-Controller, *MVC*) est un modèle d'architecture logicielle recommandé – et largement utilisé – pour la conception d'applications interactives. Ce modèle vise à minimiser le degré de couplage entre les objets de l'application en découplant structurellement entrées, traitements et sorties. Le point essentiel consiste à séparer les objets relevant de l'interface utilisateur des objets métier, afin de pouvoir les faire évoluer indépendamment et les réutiliser.

MVC organise une application interactive en trois grandes composantes : le Modèle, la Vue et le Contrôleur, et découple leurs responsabilités respectives :

- Le MODELE encapsule la fonctionnalité et les données cœur de l'application. Il est indépendant de représentations de sortie spécifiques ou d'un comportement d'entrée. Il a à charge la représentation interne des données de l'application et la logique de gestion de ces données. Il notifie à la composante Vue les changements de données et lui permet de l'interroger sur son état. Il fournit au contrôleur la possibilité d'accéder à certaines fonctionnalités applicatives encapsulées dans le Modèle.
- La VUE affiche les informations à l'utilisateur. Elle récupère du Modèle les données à présenter à l'utilisateur, spécifie leur présentation externe et met à jour l'information affichée. Elle relaie aussi au contrôleur les entrées utilisateur et l'autorise à sélectionner les vues. La séparation du modèle d'une part et de la vue et du contrôleur d'autre part permet des vues multiples d'un même modèle. Quand il y a plusieurs vues du modèle, chaque vue a alors un contrôleur associé.
- Le CONTROLEUR gère les entrées utilisateur. Il reçoit les entrées, généralement sous forme d'événements encodant un mouvement de la souris, l'activation d'un bouton, ou une entrée clavier (ou bien, s'il s'agit d'une application web, les requêtes HTTP GET et POST). Les événements sont traduits en requêtes de service pour le Modèle ou la Vue. Une application a typiquement un contrôleur

pour chaque ensemble de fonctionnalités apparentées. Une application peut avoir un contrôleur séparé par type de client si les interactions utilisateurs varient selon les types de client. L'utilisateur interagit avec le système uniquement au travers des contrôleurs.

L'ensemble Vue + Contrôleur compose l'interface utilisateur.



En pratique, ce schéma de principe laisse place à une grande variété d'implémentations possibles. Une des difficultés pour une implémentation idéale en Java réside notamment dans le fait que les composants graphiques Swing fusionnent les parties Vues et Contrôleur (cf section 11.6). Pour les petits programmes tout particulièrement, la Vue et le Contrôleur sont souvent combinés, mais dans tous les cas le Modèle doit être séparé.

Exemple d'implémentation d'une architecture MVC :

```
import java.util.Observable ;
import java.util.Observer ;
import javax.swing.JApplet ;
import javax.swing.JPanel ;
import java.awt.event.ActionListener ;
import java.awt.event.ActionEvent ;

public class Controller extends JApplet implements ActionListener {
      private View view ;
      private Model model ;

      public void init() {
            model = new Model() ;
            view  = new View(model, this);
            model.addObserver(view);
            this.setContentPane(view);
      }
```

```
        public void actionPerformed(ActionEvent e)   {
              // Process the events
        }
  }

  public class View extends JPanel implements Observer {
        private Model model ;
        // Declare here the JPanel components.

        public View(Model m, ActionListener listener) {
             model = m ;
             makeContentPane(listener);
        }
        public void update(Observable o, Object arg) {
              // Automatically called when observer is notified.
              // Update the view.
        }
        public void makeContentPane(ActionListener listener) {
              // Create and initialize the JPanel components and,
              // when need be, attach them the event listener.
              // Define the layout.
        }
  }

  public class Model extends Observable {
        // Wherever the value of an attribute interesting the
        // View is changed, insert these two statements
        //     setChanged();
        //     notifyObservers();  or  notifyObservers(arg);
  }
```

Chercher à découpler les traitements des entrées-sorties est classique en programmation. Dans cet esprit, on peut noter en particulier que le modèle MVC a une certaine similitude avec la classique architecture trois-tiers (three-tiers) des systèmes distribués. Une architecture trois-tiers est une architecture logicielle client-serveur dans laquelle l'interface utilisateur, la logique de traitement et les données sont trois modules indépendants : l'interface utilisateur est stockée dans le client, le gros de la logique de l'application est stocké dans le ou les serveurs d'application, et les données sont stockées dans un serveur de base de données.

Exemple. Pour des applications web professionnelles reposant sur une solution *Java EE* (voir chapitre 16), le Modèle est assuré par des *EJB* et/ou des *JavaBeans*, le Contrôleur est assuré par des *servlets*, et la Vue par des *JSP* (glossaire en chapitre 17).

13.3 Tests unitaires

13.3.1 Définitions

Le test est une activité qui consiste à déterminer si un segment de code (méthode, classe ou programme) produit bien le résultat attendu. On définit classiquement trois niveaux de tests, qui sont mis en œuvre successivement :

- Le *test unitaire* (unit testing) : il s'agit de tester chaque composante individuelle (méthode, classe) de l'application.
- le *test d'intégration* (integration testing) : il s'agit de tester des groupes de classes en interrelations puis toute l'application dans son ensemble. Les tests appliqués suivent généralement un plan de test (i.e. une approche systématique et méthodique).
- le *test système* (system testing) : il s'agit de tester l'application intégrée dans son environnement d'utilisation complet. Ces tests entrent dans la catégorie des tests boîtes noires, et, en tant que tels, ne devraient pas nécessiter de connaissance de la conception interne du code ou de sa logique.

Les considérations suivantes se limiteront essentiellement aux tests unitaires.

Un *cas de test* est un ensemble de conditions ou variables à partir desquelles un testeur déterminera si une exigence requise est complètement ou partiellement satisfaite. Un cas de test est caractérisé par une entrée connue (pré-condition) et une sortie attendue (post-condition). De nombreux cas de tests peuvent être nécessaires pour vérifier une exigence donnée.

On distingue généralement deux types de cas de test :

- les cas de test *positifs*, qui consistent à tester des cas censés faire réussir la fonctionnalité. Ces tests doivent nous convaincre que l'élément de code testé fonctionne bien comme espéré.
- les cas de test *négatifs*, qui consistent à tester des cas censés mettre en échec la fonctionnalité. Ce sont toutes les situations de valeurs hors limites, d'erreurs, … . Exemples : entrer une valeur hors plage, lire un ensemble vide, … . Dans de tels cas, nous devons nous attendre à ce que le programme traite l'erreur d'une façon contrôlée et appropriée.

La simple correction des erreurs détectées par un test devrait systématiquement améliorer la qualité du programme. Malheureusement, l'expérience montre que tout changement dans le programme peut introduire, de façon inattendue, d'autres erreurs. En particulier des erreurs de régression (regression bugs) : une erreur de régression survient quand une fonctionnalité qui fonctionnait précédemment ne fonctionne plus suite à une modification du programme. Une méthode courante de *test de non-régression* (non-regression testing) consiste à réexécuter les tests réalisés antérieurement et voir si des défauts apparaissent voire réapparaissent. Les tests devant donc être appliqués non pas une seule fois mais à chaque modification du logiciel, il est souhaitable de disposer d'une batterie de tests programmés afin de permettre une automatisation du test.

Il s'agit donc de programmer des cas de test pour chaque méthode non triviale, en veillant à ce que chaque cas de test soit bien séparé des autres.

13.3.2 JUnit

`JUnit`, développé par la communauté open source, offre un cadre d'applications (framework) pour la programmation de tests de régression. Il est utilisé pour implémenter des tests unitaires en Java. La classe `TestCase` permet de définir un cas de test ; la classe `Assert` fournit un ensemble de méthodes facilitant la gestion des erreurs détectées ; la classe `TestSuite` permet d'exécuter une suite de tests ; la classe `TestRunner` est un outil d'exploitation permettant de définir une suite de tests à exécuter et d'afficher les résultats.

Exemple de cas de test :

```
/**
 *  A test case for the class MyClass
 */

import junit.framework.* ;

public class TestMyClass extends TestCase {

    /* The tests need to run against the background of a known
     * set of objects. This set of objects is called a test
     * fixture.
     * Define here an instance variable for each part of the
     * fixture.
     */
    private MyClass myObject;

    public TestMyClass(String name) {
       super(name);
    }

    /**
     * Sets up the test fixture.
     * setUp() is called before every test case method.
     * Initializes the instance variables.
     */
    protected void setUp() {
       myObject = new MyClass( . . . ) ;
    }

    /**
     * Tears down the test fixture.
     * tearDown() is called after every test case method.
     * Releases any permanent resources allocated in setUp
     * (example: closes files, ...)
     */
    protected void tearDown() {
       // . . .
    }

    // Once the fixture is in place, as many Test Cases as
    // necessary can be written.

    /**
     * Tests the method methodX()
     */
    public void testMethodX() {
```

```
        ArgumentType argument = . . . ;
        ResponseType expectedResponse = . . . ;
        String errorMessage = " . . . "   ;
        assertTrue(errorMessage,
                   expectedResponse.equals(MethodX(argument)) ;
    }
} // end class TestMyClass
```

`JUnit` détermine le succès où l'échec d'un test via des instructions d'assertion. Une assertion est simplement une procédure de comparaison entre une valeur espérée et une valeur effective et qui génère un échec en cas de différence. Les procédures d'assertion disponibles sont définies dans la classe `junit.framework.Assert`.

Exemples d'instructions d'assertion :

```
assertTrue( errorMessage, booleanCondition ) ;
assertFalse( errorMessage, booleanCondition ) ;
assertEquals( errorMessage, expectedValue, actualValue ) ;
assertEquals( errorMessage, doubleExpected, doubleActual,
              doubleDelta ) ;
```

Pour exécuter le test, créer une instance de cette classe avec en paramètre le nom de la méthode de test.

Exemple

```
new TestMyClass("testMethodX") ;
```

`JUnit` est intégré dans certains outils de développement. De tels outils offrent alors des fonctionnalités qui permettent de créer facilement des tests.

13.4 Java Beans

Un *Java Bean* (« grain java ») est défini par Sun comme « *un composant réutilisable Java destiné à être manipulé par un outil de développement graphique visuel* ». Les *Java Beans* sont des composants Java (i.e. des classes) respectant certains critères afin de pouvoir être facilement développés, assemblés et réutilisés pour créer des applications sophistiquées. Par exemple, la technologie Java Beans est l'architecture des composants dans la plate-forme Java SE.

Les obligations que doit respecter un Java Bean sont nombreuses mais l'enjeu est d'importance : disposer de composants « *Write One, Run Anywhere* » (WORA) réutilisables.

Les Java Beans sont des classes publiques concrètes qui ont pour caractéristiques communes de :

- supporter des « *propriétés* » : les propriétés sont les attributs discrets nommés d'un Java Bean qui peuvent affecter son apparence ou son comportement. Les propriétés sont pour les Java Beans l'équivalent des champs pour les objets. Par exemple, un bouton graphique pourrait avoir une propriété nommée « label » qui représenterait le texte affiché dans le bouton.

Les méthodes d'accès aux propriétés doivent respecter des conventions de nommage particulières, par exemple :

- pour les propriétés simples (il en existe d'autres : liées, contraintes, indexées, …) :

```
PropertyType getPropertyName();        // simple getter
void setPropertyName(PropertyType);    // simple setter
```

- pour les propriétés booléennes, on utilise typiquement `is` au lieu de `get`

- communiquer entre eux par *événements* : en pratique, on utilise la technique Swing des « *listeners* » (voir section 11.4) ;
- supporter l'*introspection* : à l'exécution et dans l'environnement de développement, on doit être capable de déterminer et d'analyser les propriétés, les événements et les méthodes que supporte un Java Bean.

  A noter que cette propriété d'introspection est déjà présente dans les objets Java puisqu'un objet peut être analysé grâce aux classes `Class`, `Method`, `Field`, …
- supporter la *personnalisation* (« *customization* ») : les outils de développement doivent permettre de modifier l'apparence et le comportement d'un Java Bean en phase de conception ;
- supporter la *persistance* : la persistance d'un objet est la capacité de le sauvegarder sur un support de stockage et de le restaurer à l'identique afin que le programme puisse se terminer sans que cet objet soit perdu. Exemple de persistance : qu'un objet puisse prétendre à être un document Excel à l'intérieur d'un document Word.

  Le mécanisme qui rend possible la persistance est appelé « *sérialisation* » (*serialization*). A cette fin, un Java Bean doit implémenter l'interface `Serializable` ou `Externalizable`.

Exemple de Java Bean :

```
package sunw.demo.simple;

import java.awt.*;
import java.io.Serializable;

/**
 * SimpleBean will be displayed with a green
 * centered rectangle, but its color may be changed later.
 */
public class SimpleBean extends Canvas
                        implements Serializable {

    private Color color = Color.green;

    // Property getter method.
    public Color getColor(){
        return color;
    }

    // Property setter method.
    // Sets new SimpleBean color and repaints.
```

```
    public void setColor(Color newColor) {
        color = newColor;
        repaint();
    }

    public void paint(Graphics g) {
        g.setColor(color);
        g.fillRect(20, 5, 20, 30);
    }

    // Constructor sets inherited properties
    public SimpleBean() {
        setSize(60,40);
        setBackground(Color.red);
    }
} // end class SimpleBean
```

Pour être reconnu par un outil de développement, un Java Bean doit être conditionnné dans une fichier d'archive au format `jar`. Bien que ce ne soit pas obligatoire, il est souhaitable que les Java Beans soient des paquetages (évite notamment les conflits en cas d'homonymie des beans).

13.5 Design patterns

Cette section n'est qu'une toute première introduction aux *modèles de conception* (design patterns)[1].

Le développement orienté objet d'applications professionnelles conduit généralement à rencontrer des problèmes de conception types qui sont familiers à la communauté des développeurs confirmés et pour lesquels des solutions éprouvées existent. Il ne s'agit pas ici de solutions théoriques (algorithmes) mais de solutions orientées objet empiriques dont la solidité et l'efficacité ont fait leur preuve en pratique. Ces solutions sont appelées modèles de conception (design patterns). Un développeur expérimenté se doit de les connaître et de disposer d'un « panier » de modèles de conception qu'il appliquera aussi systématiquement que possible.

Les modèles de conception sont des stratégies de résolution de grands types de problèmes de conception orientée objet. Ils décrivent des formes génériques d'organisation de classes. Ils sont indépendants du langage de programmation objet mis en œuvre. A titre illustratif, l'architecture MVC, décrite à la section 13.2, est un exemple de modèle de conception d'architecture logicielle (architectural design pattern).

Les plus connus des modèles de conception de base sont les 23 modèles fondamentaux décrits par les quatre auteurs qui furent appelés collectivement « The Gang of Four » (« La bande des quatre ») ou tout simplement GoF [2].

On distingue habituellement 3 grandes familles de modèles de conception :

---

[1] L'expression *design pattern* est souvent traduite aussi par *patron de conception*.

[2] Erich GAMMA, Richard HELM, Ralph JOHNSON, John VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995, 416 p. ISBN 0201633612

- Les *modèles de construction* ont pour but d'organiser la création d'objets de façon à gagner en abstraction. Exemple présenté ci-après : le modèle *Singleton*.
- Les *modèles de structuration* facilitent l'organisation de la hiérarchie des classes et de leurs relations. Exemple présenté ci-après : le modèle *Decorator*.
- Les *modèles de comportement* proposent des solutions pour organiser les interactions et pour répartir les traitements entre les objets. Ils tentent de répartir les responsabilités entre chaque classe. Exemple : le modèle *Chain of Responsibility* permet de créer une chaîne d'objets tels que, si l'un d'eux ne peut pas répondre à une requête, il puisse la transmettre à son successeur jusqu'à ce que l'un d'entre eux y réponde.

13.5.1 Exemple : le modèle Singleton

Le modèle de conception *Singleton* permet d'assurer qu'une classe ne possédera qu'une instance au plus et d'offrir une méthode unique retournant cette instance.

Exemple d'implémentation (cas où toutes les informations nécessaires à la création du singleton sont disponibles au moment du chargement de la classe en mémoire)

```
public final class MySingleton {

      // Creating at the class loading
      private static MySingleton singleInstance = new MySingleton() ;

      // Private constructor. Can't be used outside this class
      private MySingleton() { ... }

      /** Get a singleton of this class */
      public static MySingleton getInstance() {
            return singleInstance ;
      }
}

// Creation of a singleton
MySingleton singleton = MySingleton.getInstance();
```

13.5.2 Exemple : le modèle Decorator

Le modèle de conception *Decorator* permet d'ajouter dynamiquement des fonctionnalités à un objet. Ceci se réalise en créant une nouvelle classe *Decorator* enveloppant la classe originale.

Le modèle de conception *Decorator* est une alternative à l'approche par création d'une sous-classe de la classe originale. Il s'impose notamment à une approche par extension de la classe originale dans les cas suivants : la classe originale ne peut pas être étendue par héritage ; le nombre de sous-classes permettant de définir toutes les combinaisons possibles de fonctionnalités est trop important ; l'ajout des fonctionnalités ne peut pas être statique (i.e. réalisé à la compilation) mais doit être dynamique (i.e. réalisé à l'exécution).

Exemple [Wikipedia]. Considérons des fenêtres d'affichage instances d'une classe Window et supposons que cette classe n'offre pas la possibilité d'ajouter des barres de défilement. Créons un décorateur qui puisse ajouter dynamiquement cette fonctionnalité à des objets Window existants.

```
// An interface common to the decorator and the decorated
public interface Window {
      public void draw();
}
// The original class, the instance of which can be decorated
public class SimpleWindow implements Window
      public void draw() { ... }
}
// The abstract decorator
public abstract class WindowDecorator implements Window {
      protected Window decoratedWindow;
      public WindowDecorator( Window decoratedWindow ) {
            this.decoratedWindow = decoratedWindow ;
      }
}
// A first concrete decorator
public class VerticalScrollBarDecorator extends WindowDecorator {
      public VerticalScrollBarDecorator( Window decoratedWindow ) {
            super(decoratedWindow);
      }
      public void draw() {
            drawVerticalScrollBar();
            decoratedWindow.draw();
      }
      public void drawVerticalScrollBar() { ... }
}
// A second concrete decorator
public class HorizontalScrollBarDecorator extends WindowDecorator {
      public HorizontalScrollBarDecorator( Window decoratedWindow ) {
            super(decoratedWindow);
      }
      public void draw() {
            drawHorizontalScrollBar();
            decoratedWindow.draw();
      }
      public void drawHorizontalScrollBar() { ... }
}

// Example: creation of a fully decorated Window instance.
Window decoratedWindow = new HorizontalScrollBarDecorator(
                                 new VerticalScrollBarDecorator(
                                       new SimpleWindow() ));
```

14 ANNEXE 4 – MACHINE VIRTUELLE JAVA (JVM)

Les objectifs premiers de ce chapitre sont :

- comprendre comment s'opère le chargement dynamique des classes en mémoire lors de l'exécution d'un programme Java ;
- comprendre le traitement des champs et blocs `static` au chargement de la classe.

Une machine virtuelle Java (JVM) est un interpréteur de code Java compilé (*bytecode*) permettant d'exécuter un programme Java sur une machine cible. L'espace mémoire de la JVM comporte plusieurs zones, notamment :

- la zone des méthodes : contient le code des méthodes et constructeurs ainsi que des informations sur la structure de chaque classe (notamment sa table des symboles) ;
- la pile : espace propre à chaque thread, mémorise dynamiquement les contextes d'exécution (variables locales notamment) des méthodes en cours d'exécution ;
- le tas : contient les objets créés par `new` (instances de classe et tableaux) (cf chapitre 4).

C'est la demande d'exécution d'un programme Java qui lance la machine virtuelle Java. La JVM réalise alors les grandes étapes suivantes :

- chargement de la classe principale spécifiée (*main class*) et des classes dont elle dépend immédiatement (au premier rang desquelles toutes ses sur-classes) :
  - création des champs `static`, création de la table des méthodes, …
  - traitement des champs (initialisation) et blocs (exécution) `static`
- exécution de la méthode `main` de la classe principale.

Durant l'exécution d'un code Java, les classes (leur *bytecode*) sont chargées dans la JVM au fur et à mesure des besoins. Dans une hiérarchie d'héritage, le chargement des classes est opéré par ordre hiérarchique descendant (i.e de la super-classe racine vers la classe courante).

Exemple :

```
public class ClassA {

    static { System.out.println("Start loading ClassA"); }

    public  static final int C ;
    private static int n = 100 ;

    static { C = 29979 ; }

    static {
        System.out.println("C = " + C);
        System.out.println("n = " + n);
    }

    public ClassA()  {
        System.out.println("Start ClassA()");
        n = n + 1 ;
        System.out.println("End ClassA()");
    }

    public int getN () { return n ; }
```

```
    static { System.out.println("End loading ClassA"); }
} // end ClassA

public class ClassB extends ClassA {
    static { System.out.println("Start loading ClassB"); }
    private int x = 5 ;
    public ClassB()  {
        System.out.println("Start ClassB()");
        x = x + getN();
        System.out.println(x);
        System.out.println("End ClassB()");
    }
    static { System.out.println("End loading ClassB"); }
} // end ClassB

public class Class0 {
    static { System.out.println("Start loading Class0"); }
    private static int y = initY() ;
    private static int initY() {
        System.out.println(“Init y”);
        return 256 ;
    }
    public static void main(String[] args) {
        System.out.println("Start main Class0");
        ClassB b = new ClassB();
        System.out.println("End main Class0");
    }
    static { System.out.println("End loading Class0"); }
} // end Class0
```

Après compilation de ces trois classes, l’exécution de la commande : `java Class0` *affichera (commentaires ajoutés en marge) :*

```
Start loading Class0
Init Y
End loading Class0
Start main Class0
Start loading ClassA          // Sur-classes de ClassB d’abord
C = 29979
n = 100
End loading ClassA
Start loading ClassB          // puis enfin ClassB.
End loading ClassB
Start ClassA()                // Car super() d’abord.
End ClassA()
Start ClassB()
106                           // 100 + 1 ClassA() + 5 ClassB()
End ClassB()
End main Class0
```

15 ANNEXE 5 – COMMUNICATION ENTRE JAVA ET C (JNI)

Références :

"The Java Native Interface: Programmer's Guide and Specification"

java.sun.com/docs/books/jni/

Tutoriel : java.sun.com/docs/books/jni/html/jniTOC.html

"Java Native Interface Specification v1.5"

java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html

"javah – C Header and Stub File Generator"
java.sun.com/javase/6/docs/technotes/tools/windows/javah.html

On se pose le problème d'appeler un sous-programme C depuis une méthode Java dans un environnement Unix.

La solution met en œuvre l'interface de programmation JNI (Java Native Interface) offert par la plate-forme Java. JNI est un ensemble de fonctionnalités qui permet à un programme Java fonctionnant sur la machine virtuelle Java (JVM) d'appeler ou d'être appelé par un programme écrit dans un autre langage (typiquement en C, C++ ou assembleur).

En permettant l'interfaçage avec des applications ou bibliothèques natives[1], la mise en œuvre de JNI accroît l'interopérabilité des applications Java, mais évidemment au détriment de leur portabilité. Une bonne architecture de l'application devrait donc minimiser le nombre de classes faisant référence à une méthode native.

Par la suite, on supposera que le fichier Java appelant le sous-programme C se nomme *MyClass.java*, et que le fichier C implémentant ce sous-programme se nomme *MyRoutines.c*

Le développement de l'application complète comporte 6 étapes :

1) Ecrire le programme Java et le compiler :

```
javac MyClass.java
```

2) Produire le fichier d'en-tête `MyClass.h` pour le programme C :

```
javah MyClass
```

3) Ecrire le sous-programme C et le compiler :

```
gcc -o MyRoutines -c MyRoutines.c -pedantic -W -Wall -Idir
```

où *dir* est le répertoire d'installation par le JDK des fichiers d'*include* du JNI. Par exemple (ces fichiers étant répartis dans deux répertoires) :

```
gcc  -o MyRoutines -c MyRoutines.c -pedantic -W -Wall
     -I/usr/lib/jdk1.5.0v11/include
```

[1] Une application ou une bibliothèque de sous-programmes est dite en code natif si elle est dépendante d'un environnement hôte (système d'exploitation ou jeu d'instructions du processeur) particulier. Par exemple, un programme C compilé sur un système d'exploitation donné est en code natif car il ne fonctionnera pas sur un autre système d'exploitation.

```
-I/usr/lib/jdk1.5.0v11/include/linux
```

4) Produire une bibliothèque partagée (`lib*.so`) qui sera chargeable par le programme Java :

```
gcc -Wall -shared -o libMyRoutines.so MyRoutines.o
```

5) Avant la toute première exécution, spécifier à la machine virtuelle Java le chemin d'accès à cette bibliothèque :

```
setenv LD_LIBRARY_PATH path
```

où *path* désigne le chemin d'accès. En supposant que celui-ci soit le répertoire courant, la commande sera :

```
setenv LD_LIBRARY_PATH .
```

6) Exécuter le programme Java :

```
java MyClass
```

La façon d'écrire le programme Java et d'implémenter le sous-programme C associé est illustrée ci-après.

15.1 Appel d'une procédure C

On considèrera de bout en bout le même exemple.

15.1.1 Déclaration et appel java

Exemple de classe java

```
public class MyClass {

      // Declaration of an extern class procedure
      native public static void procedure1();

      // Declaration of an extern instance procedure
      native public void procedure2();

      // Test procedure
      public static void main(String[] argv) {
            MyClass.procedure1();
            MyClass o = new MyClasse();
            o.procedure2();
      }

      // Load the library libMyRoutines
      static {
            System.loadLibrary("MyRoutines");
      }

}
```

Commentaires :

- Le modificateur `native` indique qu'il s'agit d'une procédure externe non écrite en java. Le nom de la procédure est arbitraire.
- La méthode `loadLibrary` va charger la bibliothèque *libMyRoutines.so* ; le mot clé `static` qui encapsule le bloc d'instructions dans lequel l'appel est réalisé

permet d'exécuter ce bloc dès le chargement de la classe `MyClass` sans avoir besoin de passer par l'appel d'une méthode intermédiaire.

15.1.2 Prototype C

Le fichier *`MyClass.h`* produit par javah (cf étape 2 précédente) spécifie les prototypes que devront respecter les sous-programmes C à écrire.

Contenu (expurgé) du fichier `MyClass.h` produit par javah

```
/* DO NOT EDIT THIS FILE - it is machine generated */
/* Header for class MYClass */

JNIEXPORT void JNICALL Java_MyClass_procedure1
  (JNIEnv *, jclass);

JNIEXPORT void JNICALL Java_MyClass_procedure2
  (JNIEnv *, jobject);
```

Commentaires :

- `JNIEXPORT` et `JNICALL` sont deux macros prédéfinies par JNI pour faire communiquer java et C
- `void` spécifie qu'il s'agit d'une procédure
- `Java_MyClass_procedure1` est le nom imposé que devra porter la procédure C correspondant à la méthode de nom `procedure1` déclarée dans le programme java ; idem pour `procedure2`
- Deux paramètres sont systématiquement introduits en tête de la liste des paramètres :
    - le premier, de type `JNIEnv *`, est un pointeur vers l'environnement `JNIEnv` qui est lui-même un pointeur vers une structure contenant des fonctions d'interface avec la machine virtuelle java (JVM) ;
    - le second, de type `jclass` s'il s'agit d'une méthode de classe ou `jobject` s'il s'agit d'une méthode d'instance, est un pointeur vers un descripteur de la classe ou de l'objet (une sorte de `this`) d'appel

15.1.3 Implémentation C

Exemple de fichier `MyRoutines.c`

```
#include <stdio.h>
#include "MyClass.h"

JNIEXPORT void JNICALL Java_MyClass_procedure1
      (JNIEnv * penv, jclass cla)
{
      printf("Routine procedure1\n");

}

JNIEXPORT void JNICALL Java_MyClass_procedure2
      (JNIEnv * penv, jobject obj)
{
      printf("Routine procedure2\n");
```

```
}
```

15.2 Echange de données entre Java et C

L'échange de données entre l'application java et le sous-programme C (par les paramètres ou le résultat de la fonction) suppose une compatibilité des types correspondants. Or il n'y a pas de correspondance systématique naturelle entre les types du langage Java et les types du langage C (un caractère, par exemple, est codé sur 16 bits en Java et sur 8 bits en C). Le JNI définit donc un ensemble de types en C (de noms `j*`) qui correspondent aux types de Java.

Le JNI traite les types primitifs et les types objets différemment. La correspondance des types primitifs est directe (voir tableau ci-après), car elle repose sur une représentation machine des données identique, mais ce n'est pas le cas pour les types objets. La transmission d'une chaîne de caractères, d'un tableau, a fortiori d'une instance quelconque de classe, n'est pas directe et nécessite des conversions de type. Un paramètre java de type objet est ainsi transmis au sous-programme C sous forme d'un pointeur sur une structure de données interne à la machine virtuelle Java. L'organisation interne de cette structure est masquée au programmeur mais celui-ci peut néanmoins accéder aux composantes sous-jacentes via les fonctions utilitaires offertes par le JNI dans l'environnement `JNIEnv`.

Type primitif Java	Type primitif côté C	Description
`boolean`	`jboolean`	`unsigned 8 bits`
`byte`	`jbyte`	`signed 8 bits`
`char`	`jchar`	`unsigned 16 bits`
`short`	`jshort`	`signed 16 bits`
`int`	`jint`	`signed 32 bits`
`long`	`jlong`	`signed 64 bits`
`float`	`jfloat`	`32 bits`
`double`	`jdouble`	`64 bits`

Par la suite, on considèrera de bout en bout le même exemple.

15.2.1 Déclaration et appel java

Exemple de classe java

```
public class MyClass {

      // Declaration of an extern instance function
      native public String getMessage(int id, String prompt);

      // Test procedure
      public static void main(String[] argv) {
            MyClass o = new MyClasse();
            System.out.println(o.getMessage(10, "Name: "));
      }

      // Load the library libMyRoutines
      static {
            System.loadLibrary("libMyRoutines");
      }
```

```
}
```

15.2.2 Prototype C

Contenu (expurgé) du fichier `MyClass.h` produit par javah

```
/* DO NOT EDIT THIS FILE - it is machine generated */
/* Header for class MYClass */

JNIEXPORT jstring JNICALL Java_MyClass_getMessage
  (JNIEnv *, jobject, jint, jstring);
```

15.2.3 Implémentation C

Exemple de fichier `MyRoutines.c`

```
#include <stdio.h>
#include "MyClass.h"

JNIEXPORT jstring JNICALL Java_MyClass_getMessage
  (JNIEnv * penv, jobject obj, jint jid, jstring jprompt);
{
      const char * str;
      char buf[128];

      /* Get the C string from the java string */
      str = (*penv)->GetStringUTFChars(penv, jprompt, NULL);

      printf("%d - %s", jid, str);

      /* Don't forget this line !!! */
      (*penv)->ReleaseStringUTFChars(penv, jprompt, str);

      /* Assume the user doesn't type more than 127 char */
      scanf("%s", buf);

      return (*penv)->NewStringUTF(penv, buf);
}
```

Commentaires :

- Les représentations physiques des chaînes de type `jstring` et de type `char*` ne sont pas identiques. Une conversion doit être réalisée. C'est le rôle des fonctions `GetStringUTFChars` et `NewStringUTF`. A noter que, dans tous les cas, la conversion des caractères non codables en ASCII 7 bits (lettres accentuées, …) posera problème.
- La fonction `GetStringUTFChars` réalisant une allocation mémoire dynamique, une désallocation doit être obligatoirement réalisée avant la fin du sous-programme : c'est le rôle de la fonction `ReleaseStringUTFChars`

16 ANNEXE 6 – PLATES-FORMES JAVA

Références :

Java Platform, Standard Edition 6, API Specification : java.sun.com/javase/6/docs/api/

New features and Enhancements J2SE 5.0 :
java.sun.com/j2se/1.5.0/docs/relnotes/features.html

Java Platform, Standard Edition : java.sun.com/javase/

Java Platform, Enterprise Edition : java.sun.com/javaee/

Java Platform, Micro Edition : java.sun.com/javame/

JDK Tools and Utilities : java.sun.com/javase/6/docs/technotes/tools/

Sun Microsystems fournit trois plates-formes Java permettant le développement et l'exécution de programmes Java : *Java SE* (le standard de base), *Java EE* et *Java ME*. A noter que ces noms sont ceux employés pour les versions sorties depuis 2006 : antérieurement, ces plates-formes étaient nommées respectivement *J2SE, J2EE* et *J2ME*.

Java Platform, Standard Edition (Java SE). Cette plate-forme est le framework Java de base permettant de développer ou/et exécuter des programmes Java. La dernière version nommée J2SE est la version 5.0 et porte le nom J2SE 5.0 : il est à souligner que cette version apportait une mise à jour majeure par rapport à la précédente. La version actuelle la plus récente date de décembre 2006 : c'est la version 6 et elle porte le nom **Java SE 6**. La version 7 (Java SE 7) est annoncée comme nouvelle mise à jour majeure mais, à mi 2010, est toujours attendue.

La plate-forme Java SE délivre deux produits logiciels principaux :

Java SE Runtime Environment (ou plus communément *JRE*), qui fournit les librairies, la machine java virtuelle et d'autres composantes nécessaires pour exécuter des programmes Java.

Java SE Development Kit (ou plus communément *JDK*), qui inclut le *JRE* plus des outils de développement, tels que compilateur et débogueur, nécessaires ou utiles pour développer des programmes Java. A noter que le terme JDK, bien que très populaire, est devenu imprécis car d'autres plates-formes Java (Java EE par exemple) comportent elles aussi leur propre kit de développement.

Java Platform, Enterprise Edition (Java EE). Cette plate-forme est un framework Java plus particulièrement destiné aux applications d'entreprises. Elle offre un cadre standard pour développer des applications distribuées.

Java Platform, Micro Edition (Java ME). Cette plate-forme est un framework Java spécialisé dans les applications mobiles. Des plates-formes Java compatibles avec J2ME sont embarquées dans de nombreux téléphones mobiles et PDA.

De façon plus spécifique, on pourrait aussi citer les plateformes **Java Card** (qui fournit un environnement pour le développement d'applications sur cartes à puce) et **JavaFX** (pour créer des *Rich Internet Applications*).

17 ANNEXE 7 – GLOSSAIRE

On trouve facilement sur le web d'excellentes définitions en anglais des termes, sigles, et même produits de référence, informatiques. Parmi les sources possibles, citons par exemple :

- parmi les encyclopédies : www.wikipedia.org (la version anglaise étant la plus complète)
- parmi les moteurs spécialisés dans la recherche dans les encyclopédies, dictionnaires et glossaires : www.answers.com

Voici un glossaire de quelques termes et sigles généraux courants :

Sigle / terme	**Définition**	**Commentaire**
API	Application Programming Interface	Bibliothèque de fonctionnalités communes, courantes et de bas niveau. Constitue une interface entre couches basses et couches applicatives de plus haut niveau. Exemple : java.sun.com/javase/6/docs/api/ spécifie les API standard de Java fournies dans le JDK 6.
Framework	Cadre d'applications	Ensemble cohérent de classes (généralement abstraites) et d'interfaces permettant le développement rapide d'applications. Fournit suffisamment de briques logicielles pour pouvoir produire une application aboutie. Typiquement spécialisé pour un type d'application.
GUI	Graphical User Interface	Interface homme-machine graphique, par opposition à interface en ligne de commande.
IDE	Integrated Development Environment	Programme regroupant un ensemble d'utilitaires logiciels pour le développement de programmes (éditeur de texte, compilateur, débogueur, gestionnaire de versions, aide à la production d'interfaces, …). Généralement dédié à un seul langage de programmation.
JDK	Java Development Kit	(voir chapitre 16)
JFC	Java Foundation Classes	Cadre d'applications graphiques. Ensemble de classes Java permettant de construire des interfaces utilisateur graphiques portables. Contient notamment les composants Swing.
JRE	Java Runtime Environment	(voir chapitre 16)

J2EE Java SE	Java 2 Platform, Enterprise Edition Java Platform, Enterprise Edition	(voir chapitre 16)
J2ME Java ME	Java 2 Platform, Micro Edition Java Platform, Micro Edition	(voir chapitre 16)
J2SE Java SE	Java 2 Platform, Standard Edition Java Platform, Standard Edition	(voir chapitre 16)

Et plus spécifiquement dans le cadre d'applications professionnelles Java distribuées :

Sigle / terme	**Définition**	**Commentaire**
EJB	Enterprise JavaBeans	Technologie Java, offerte par la plateforme Java EE, permettant de créer des composants distribués hébergés dans un serveur d'applications. Dans une architecture MVC, utilisé pour réaliser le Modèle. java.sun.com/products/ejb/
JSP	Java Server Pages	Technologie Java, offerte par la plateforme Java EE, permettant de créer des pages Web dynamiques. Dans une architecture MVC, utilisé pour réaliser la Vue. java.sun.com/products/jsp/
Servlet		Technologie Java, offerte par la plateforme Java EE, permettant d'étendre la fonctionnalité d'un serveur Web (accès à des bases de données, transactions d'e-commerce, …). Dans une architecture MVC, utilisé pour réaliser le Contrôleur. java.sun.com/products/servlet/