Java

Introduction et historique

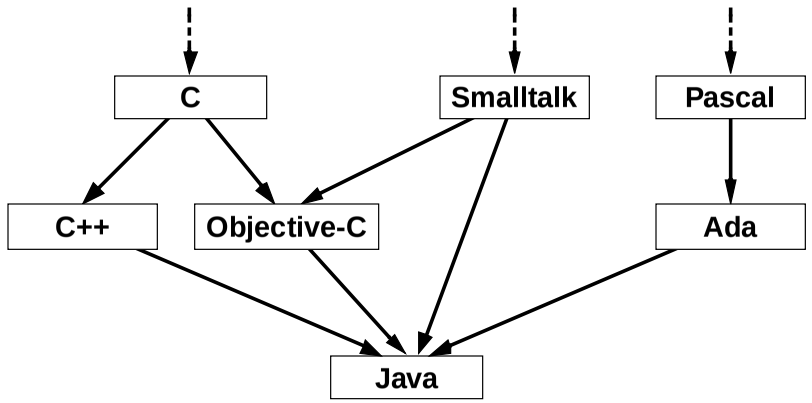
Historique

- **1990-1992, projet Oak**
  - Joy, Gosling (Sun)
  - Langage de programmation pour systèmes embarqués: TV interactive, ...
  - But: avoir une plateforme commune.
  - Projet abandonné en 1992
- **1996, Java**
  - Projet Oak appliqué à WWW. Oak renait de ses cendres.
  - Java, combiné avec un browser spécial Hotjava, apporte le concept d'applet.
  - Avec le dévellopement du web, Java est un jouet qui a pris de l'importance.

Composants liés à Java

- **Java language**
  - langage de programmation.
- **Java bytecode**
  - représentation "machine" d'un programme; indépendant de la plateforme.
- **Java virtual machine**
  - machine virtuelle permettant d'executer du bytecode.
- **Hotjava**
  - browser web ayant servi de base pour demontrer le concept d'applet.

Ancètres directs de Java



Caractéristiques de Java

- **Similaires à Smalltalk**
  - Orienté-objet
  - Héritage simple
  - Toute classe est sous-classe d'`Object`.
  - Garbage collection
  - Réflexivité (partielle; lecture seule)
  - Large bibliothèque de classes standard.
- **Différentes de Smalltalk**
  - Typage statique
  - Types "primitifs"
  - Syntaxe ("à la C/C++")
  - Pas de clôtures (BlockClosure)

Hello World!

- **fichier HelloWorld.java**

```
public class HelloWorld
{
     public static void main(String[] args)
     {
        System.out.println("Hello World!");
     }
}
```

- **Compilation**
  - `javac HelloWorld.java`
- **Execution**
  - `java HelloWorld`

```
Hello World!
```

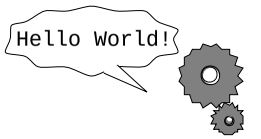

Compilation et interprétation

HelloWorld.java

```
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

HelloWorld.class

```
0 1207  getstatic #7 <Field java.io.PrintStream out>
3 2301  ldc #1 <String "Hello World!">
5 2508  invokevirtual #8 <Method println(java.lang.String)>
8 A3    return
```



Source Java

javac

Bytecode Java

java

Machine virtuelle

Java

Bases du langage

Contenu

- **Types primitifs et tableaux**
- **Classes**
  - Définition de classe
  - Variables et méthodes d'instance
  - Variables et méthodes de classe
  - Héritage, recherche de méthode/variable
  - Classes abstraites
  - Constructeurs
  - Variables et méthodes finales

Types "primitifs"

- **Les types entiers**

```
byte    // ( 8 bits) -128 à 127
short   // (16 bits) -32768 á 32767
int     // (32 bits) -2147483648 à 214748364732 bits
long    // (64 bits) -9223372036854775808 à 9223372036854775807
char    // (16 bits) '\0000' à '\ffff'
```

- **Les types flottants**

```
float   // (32 bits) simple précision
double  // (64 bits) double précision
```

- **Le type booléen**

```
boolean   // true ou false
```


Les tableaux

- **Déclaration**

```
int[]  tableau_entiers;
double tableau_reels[];
long[][] tableau_2D;
```

- **Création**

```
tableau_entiers = new int[10];
tableau_reels   = new double[2 * a];
tableau_2D      = new long[5][4];
```

- **Itération**

```
for (int i=0; i<tableau_entiers.length; i++) {
     tableau_entiers[i] = 2 * i;
}
```


Tableaux multidimensionels

```
int[][] t = new int[5][];

for (int i=0; i<t.length; i++) {
     t[i] = new int[5-i];
}

for (int i=0; i<t.length; i++) {
     for (int j=0; j<t[i].length; j++) {
        t[i][j] = i+j;
     }
}
```

	[0]	[1]	[2]	[3]	[4]
t[0]	0	1	2	3	4
t[1]	1	2	3	4	
t[2]	2	3	4		
t[3]	3	4			
t[4]	4				

- **Remarques**
  - Un tableau multidimensionnel est un "tableau de tableau".
  - Un tableau à deux dimensions n'est pas nécessairement carré.

Définition de classe

```
class Point
{
     // définition de Point
     ...
}
```

Remarques

- Le code de la classe `Point` doit être écrit dans un fichier `Point.java`.
- La déclaration et la définition d'une classe sont combinées.
- Une classe défini un **type**.

Instantiation

```
Point p1;                // déclaration
p1 = new Point();        // instantiation
Point p2 = new Point();  // déclaration avec initialisation
Point p3 = null;
```

Remarques

- Une variable est déclarée avec un type.
- Une variable contient une **référence** sur un objet.
- Un objet est créé en instantiant une classe.
- `new` est un opérateur d'instantiation.
- `null` est une **valeur** qui dénote l'absence d'objet.
- `null` n'a pas de type.
- Contrairement à Smalltalk, `null` n'est pas un objet.

Variables d'instance

```
class Point
{
     int x = 0;
     int y = 0;
     Point origin;
}
```

Exemple

```
Point p1 = new Point();
Point p2 = new Point();

p1.x = 10;
p1.y = 5;
p2.x = 2 * p1.x;
p2.y = p1.x + p1.y;
```


Méthodes d'instance

```
class Room
{
     int  size = 0;
     Room next;

     int getSize()
     {
        return size;
     }

     void setSize(int size)
     {
        this.size = size;
     }

     Room getNextRoom()          { return next; }
     void setNextRoom(Room next) { this.next = next }
}
```

Appel de méthode

Appel de méthode

```
// On déclare une cuisine et un salon
Room kitchen = new Room();
Room living  = new Room();

// la cuisine fait 10 m2
kitchen.setSize(10);
// la cuisine donne dans le salon
kitchen.setNextRoom(living);

// le salon a une surface double de la cuisine
living.setSize(kitchen.getSize() * 2);
// le salon mène à la cuisine
living.setNextRoom(kitchen);

// la surface de la cuisine est la même que la pièce voisine
kitchen.setSize(kitchen.getNextRoom().getSize());
```


Variables de classe

```
class Point
{
     static int NbrPoints = 0;
     int x = 0, y = 0;

     int getNbrPoints() { return NbrPoints; }
}
```

Exemple

```
Point p1 = new Point();
Point p2 = new Point();

Point.NbrPoints = 2;
p1.getNbrPoints()   → 2
p2.getNbrPoints()   → 2
```


Méthodes de classe

```
class Point
{
     static Point Origin = new Point();
     int x = 0, y = 0;

     static Point getOrigin() { return Origin; }
     static void setOrigin(Point origin)
     {
        Origin = origin;
     }
}
```

Exemple

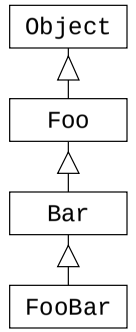
```
Point p = new Point();

p.x = 1; p.y = 2;
Point.getOrigin()    → Point(0,0)
Point.setOrigin(p);
Point.getOrigin()    → Point(1,2)
```


Héritage

Remarques

- Une sous-classe désigne sa super-classe au moyen du mot-clé `extends`.
- Seul l'héritage **simple** est possible.
- Toutes les classes descendent d'`Object`.
- Par défaut, une classe est une sous-classe directe d'`Object`.



Exemple

```
class Foo
{ ... }

class Bar extends Foo
{ ... }

class FooBar extends Bar
{ ... }
```

Héritage (this et super)

```
class A
{
     void foo() { ... }
     void bar() { ... }
}

class B extends A
{
     void foo()    { super.bar(); }
     void bar()    { this.foo(); }
     void foobar() { foo(); }
}
```

❑ Remarque

➪ `this` et `super` sont respectivement équivalent à self et super en Smalltalk.

Héritage (variables d'instance)

```
class A {
     int x = 0;
     int getX()      { return x; }
}

class B extends A {
     int x = 1;
     int getXinB()   { return x; }
     int getThisX()  { return this.x; }
     int getSuperX() { return super.x; }
}
```

❑ Exemple

```
B b = new B();
b.getX();          → 0
b.getXinB();       → 1
b.getThisX();      → 1
b.getSuperX();     → 0
```


Héritage (recherche de méthodes)

```
class Foo {
     static int classFoo()  { return 1; }
     int instanceFoo()      { return 2; }
     static int classCall() { return this.classFoo(); }
     int instanceCall()     { return this.instanceFoo(); }
}
class Bar extends Foo {
     static int staticFoo() { return 10; }
     int instanceFoo()      { return 20; }
}
```

❑ Exemple

```
Bar b = new Bar();
b.instanceCall(); → 20
b.classCall();    → 1
```

❑ Remarques

➪ Les méthodes de classe (`static`) sont recherchées **statiquement** (compilation).

➪ Les méthodes d'instance sont recherchées **dynamiquement** (exécution).

Classes et méthodes abstraites

```
abstract class Shape
{
     double sumArea(Shape other)
     { return this.area() + other.area(); }
     abstract double area();
}
```

❑ Remarques

➪ Une classe qui contient une méthode abstraite **doit** être déclarée abstraite.

➪ Une classe abstraite ne peut pas être instantiée.

➪ Le compilateur détecte lorsqu'une classe abstraite n'est pas déclarée abstraite.

➪ Le compilateur détecte lorsqu'une classe abstraite est instantiée.

Contrôle d'accès

Qualificateur

`private`	accessible uniquement depuis la même classe.
`protected`	accessible depuis la classe et ses sous-classes.
`public`	accessible depuis n'importe où.
par défaut	*abordé plus tard* (à considérer comme `private`)

Exemple

```
class Foo {
     protected int    x = 0;
     private   double r = 1.5;
     private void   setRadius(double r) { this.r = r; }
     public  double getRadius()         { return r; }
}
```

Remarques

- Permet de limiter/étendre l'accessibilité des variables et des méthodes.
- Pour les classes, la signification est différente (*abordé plus tard*).

Constructeurs (1)

```
class Circle
{
     double x, y, r;
}
```

Exemple

```
Circle c = new Circle();

c.x = 2.0;
c.y = 2.0;
c.r = 1.0;
```

Problème

- Comment initialiser les variables d'instance d'un objet?
- Solution: les constructeurs

Constructeurs (2)

```
class Circle
{
     double x, y, r;

     public Circle(double x, double y, double r)
     {
        this.x = x; this.y = y; this.r = r;
     }
}
```

Exemple

```
Circle c = new Circle(2.0, 2.0, 1.0);
```

Remarques

- Un constructeur n'a pas de valeur de retour.
- Un constructeur porte **obligatoirement** le même nom que la classe.
- Un constructeur **n'est pas** une méthode.

Constructeurs (3)

```
class Circle
{
     double x, y, r;
     public Circle(double x, double y, double r)
        { this.x=x; this.y=y; this.r=r; }
     public Circle(double x, double y)
        { this.x=x; this.y=; r=0.0;}
     public Circle(Circle c)
        { this.x=c.x; this.y=c.y; this.z=c.z; }
     public Circle()
        { this.x = this.y = this.z =0.0; }
}
```

Remarques

- Une classe peut définir plusieurs constructeurs avec des **arguments différents**.
- Le constructeur est appelé par `new`.
- Le choix du constructeur est donné par le **type** et le **nombre** d'arguments.
- Par défaut, un constructeur **sans arguments** est défini.

Constructeurs (4)

```
class Circle
{
     double x, y, r;
     public Circle(double x, double y, double r)
     {
        this.x=x; this.y=y; this.r=r;
     }
     public Circle(double x,double y) { this(x,y,0.0); }
     public Circle(Circle c)          { this(c.x,c.y,c.z); }
     public Circle()                  { this(0.0,0.0,0.0); }
}
```

Remarques

- `this(...)` permet de déléguer l'initialisation à un autre constructeur.
- Ce mécanisme permet d'écrire moins de code.
- `this(...)` doit désigner un constructeur qui existe dans la **même classe**.
- `this(...)` doit être la **première** instruction du constructeur.

Constructeurs (5)

```
class ColorCircle extends Circle
{
     int color;
     public ColorCircle(double x,double y,double r,int color)
     {
        super(x, y, z);
        this.color = color;
     }
     public ColorCircle(ColorCircle c)
     {
        super(c);
        this.color = c.color;
     }
}
```

Remarque

- Une classe **n'hérite pas** des constructeurs de sa super-classe.
- `super(...)` permet d'invoquer un constructeur de la **super-classe**.
- Pour le reste `this(...)` et `super(...)` ont les même limitiations.

Initialiseurs statiques

```
class Calendar
{
     static Hashtable monthes = new Hashtable();
     static {
        monthes.put("jan", "January");
        monthes.put("feb", "February");
        monthes.put("mar", "March");
        ...
     }
     ...
}
```

Remarques

- `static {...}` est un initialiseur statique.
- Une classe peut avoir un ou plusieurs initialiseurs statiques.
- Les initialiseurs statiques servent notament à initialiser les variables de classe.
- Les initialiseurs statiques sont exécutés lorsque la classe est **chargée**.
- Les initialiseurs statiques sont exécutés dans le **contexte de la classe**.

Variables finales

```
class Account
{
     static final int MAX_POINTS = 2;
     static int InstanceCounter = 0;
     final int id = InstanceCounter++;
     void Foo()
     {
        final int foo = 10 * id;
        ...
     }
}
```

Remarques

- `final` signifie qu'une variable ne peut pas être redéfinie (constante).
- Une variable de classe finale doit être initialisée.
- Une variable d'instance finale doit être initialisée.
- Devrait être utilisé autant que possible.

Méthodes finales

```
class Foo
{
     final void foo() { ... }
     void bar()       { ... }
}

final class Bar extends Foo
{
     void foo() { ... }  // INTERDIT car foo() est finale
     void bar() { ... }
}

class FooBar extends Bar { ... } // INTERDIT car FooBar est finale
```

- **Remarques**
  - Interdit la redéfinition d'une méthode ou d'une classe entière.
  - Ne doit être utilisé que dans certains cas précis. (p.ex., sécurité pour les applets)
  - N.B.: Limite l'extensibilité d'une classe et, par conséquent, la réutilisabilité.

Java

Concepts avancés

Contenu

Typage (1)

```
class Employee { ... }
class Secretary extends Employee { ... }
class Engineer extends Employee { ... }

Employee yumi = new Secretary("Yumi");
```

- **Remarques**
  - Une classe défini un type.
  - Le type d'une variable est différent du type de l'objet qu'elle référence.
    e.g., la variable `yumi` est de type `Employee`; l'objet est de type `Secretary`.
  - Une variable d'un type donné ne peut contenir que des objets du même type ou de l'un de ses sous-types. (polymorphisme)
  - Le type d'une variable ne dépend pas de l'objet qu'elle référence.

```
Secretary julia = new Secretary("Julia");

yumi  = julia;   // OK.
julia = yumi;    // ERREUR à la compilation.
```

Typage (2)

```
Employee yumi   = new Secretary("Yumi");
Employee xavier = new Engineer("Xavier");
Secretary on_duty;

on_duty = yumi;               // ERREUR à la compilation.
on_duty = (Secretary) yumi;
on_duty = (Secretary) xavier; // ERREUR à l'exécution.
```

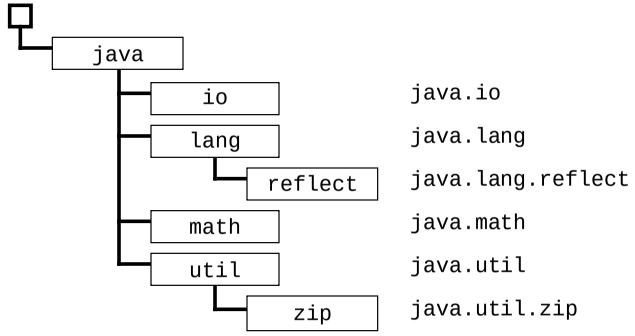
Remarques

- Le "*type casting*" permet de restreindre le type d'un objet.
  e.g., le type de la variable `yumi` (`Object`) est convertit en type `Secretary`.
- Restreindre un type doit être **explicite**.
- Restreindre un type n'est autorisé que dans certains cas particuliers.
  e.g., `xavier` ne peut être convertit en `Secretary` puisqu'elle contient un objet de type `Engineer`. (`Engineer` n'est pas une sous-classe de `Secretary`)

Paquetages (1)

Remarques

- Les classes sont regroupées en paquetages, selon leurs affinités.
- Un paquetage permet de réduire les problèmes de conflit avec les noms de classe.
- Un paquetage donne une dimension supplémentaire pour l'encapsulation.



Paquetages (2)

`java.io`

- Regroupe différents types de classes gérant les entrées/sorties.

`java.lang`

- Classes Java de base. e.g., `Object`, `String`, `System`

`java.lang.reflect`

- Classes permettant de gérer la réflexion. e.g., `Method`, `Constructor`, `Field`

`java.util`

- Différentes structures de données utiles. e.g., `Vector`, `Hashtable`

Paquetages (3)

```
class HelloWorld
{
     java.util.Vector v = new java.util.Vector();
     java.io.File     f = new java.io.File();
     ...
}
```

Remarques

- Le **nom complet** d'une classe est le nom de la classe préfixé par le nom du paquetage dans lequel elle se trouve.
  e.g., `Vector` est une classe définie dans le paquetage `java.util`.
- Pour accéder à une classe se trouvant dans un paquetage, le nom complet doit être utilisé.
  e.g., `java.util.Vector`

Paquetages (4)

```
import java.util.Vector;
import java.io.File;

class HelloWorld
{
     Vector v = new Vector();
     File   f = new File();
     ...
}
```

Remarques

- `import` permet de rendre une classe directement accessible.
- Une classe importée est accessible sans utiliser son nom complet.
- Un fichier peut contenir autant de clauses `import` que nécessaires.
- Les clauses `import` sont placées au début du fichier.
- L'étendue des clauses `import` est limitée au fichier.

Paquetages (5)

```
import java.util.*;
import java.io.*;

class HelloWorld
{
     Vector v = new Vector();
     File   f = new File();
     ...
}
```

Remarques

- Lorsqu'une clause `import` spécifie le nom d'un paquetage suivi de `.*`, toutes les classes du paquetage sont automatiquement importées.
- Les classes d'éventuels "sous-paquetages" ne sont pas importées.
- En cas de conflit de nom, une erreur de compilation est générée.
- Par défaut, toutes les classes du paquetage `java.lang` sont importées.
- Il est généralement préférable d'importer les classes explicitement.

Paquetages (6)

```
package epfl.poo;

public class Exemple {...}
```

Remarques

- Pour qu'une classe soit définie dans un paquetage, la clause `package` suivie du nom du paquetage doit figurer au début du fichier contenant la classe.
  e.g., la classe `epfl.poo.Exemple`.
- Seules les classes publiques sont exportées par un paquetage.
- La clause `package` doit figurer avant d'éventuelles clauses `import`.

Contrôle d'accès

Qualificateur

`private`	accessible uniquement depuis la classe.
par défaut	accessible depuis toutes les classes du même **paquetage**.
`protected`	accessible depuis le même **paquetage** et toutes les sous-classes.
`public`	accessible depuis n'importe où.

Remarques

- Pour les classes, `public` signifie que la classe est exportée du paquetage dans lequel elle est définie. Les autres qualificateurs n'ont pas de sens.
- Par défaut, une classe n'est visible que dans le paquetage dans lequel elle est définie.

Interfaces (1)

```
interface Scalable
{
     public void scale();
}
```

Remarques

- Une interface défini un protocole (méthodes auxquelles un objet répond).
- Une interface ne possède pas d'implémentation.
- Toutes les méthodes d'une interface sont (implicitement) abstraites.
- Une interface peut aussi définir des constantes (`static` et `final`).

```
interface Rotateable
{
     static final double PI = 3.1415926535;
     public void rotate();
}
```


Interfaces (2)

```
interface Transformable extends Scalable, Rotateable
{}
```

Remarques

- Une interface peut étendre une ou plusieurs autres interfaces.
- Une interface hérite de toutes les méthodes et constantes de sa super-interface.
- Une interface dénote ce qu'un objet peut faire. Par convention, le nom d'une interface se termine généralement par `-able`.

Interfaces (3)

```
interface Drawable { public void draw(GraphicContext gc); }

class DrawableShape
     extends Shape
     implements Drawable
{
     public void draw(GraphicContext gc) { ... }
}
```

Remarques

- Une classe *implémente* une interface. Elle doit alors fournir une implémentation pour **toutes** les méthodes définies par l'interface.
- Une interface est un *contrat* que prend une classe.
- Une interface ne peut pas être instantiée, mais défini un **type** (de la même manière qu'une classe abstraite).

```
Drawable obj = new DrawableShape();
```


Interfaces (4)

```
abstract class GraphicShape
     extends Shape
     implements Drawable, Transformable
{
     public abstract void draw();
     public abstract void rotate();
     public abstract void scale();
}
```

Remarques

- Une classe peut implémenter **plusieurs** interfaces.
- Une classe abstraite peut implémenter une interface avec des méthodes abstraites.
- Lorsqu'une classe abstraite ne contient que des méthodes abstraites, c'est un bon candidat pour devenir une interface.

Exceptions (1)

... est une erreur

- Une exception représente une erreur qui survient dans le système et que ne peut être détectée que durant l'exécution.
- Les exceptions peuvent être manipulées dans un programme.
  - une exception peut être générée par le système. (e.g., dépassement d'index de tableau)
  - une exception peut être générée explicitement par le programmeur.
  - une exception peut être interceptée par le programmeur.

... est un objet

- Une exception est représentée dans le système par un objet.
- Les exceptions sont des instances de la classe `Exception`.

Exceptions (2) - intercepter une exception

```
try {
     ...
     a[c] = 10 * x;
     ...
} catch (Exception e) {
     // traîtement de l'exception
     System.out.println("Exception: " + e);
}
```

Remarques

- Le bloc "`try`" désigne le code dans lequel on suspecte qu'une exception pourrait être levée.
- Si une exception est levée durant l'exécution du bloc "`try`", le bloc "`catch`" est exécuté et désigne le comportement à suivre.
- Si aucune exception n'est levée durant l'exécution du bloc "`try`", alors le bloc "`catch`" n'est jamais exécuté.

Exceptions (3) - lever une exception

```
try {
     ...
     throw new Exception();
     ...
} catch (Exception e) {
     // traîtement de l'exception
     ...
     throw e;
}
```

Remarques

- Une exception peut être levée explicitement au moyen de l'insttruction `throw`.
- L'instruction `throw` peut être utilisée n'importe où dans le code.
- Lorsque `throw` est rencontré par le programme, l'exécution normale est interrompue et le programme passe directement dans le bloc de traîtement de l'exception.
- Une exception peut être reportée plus loin; `throw e` permet de "faire passer" l'exception `e` plus loin. Ainsi, `e` reste levée en sortant du bloc "`catch`".

Exceptions (4) - reporter une exception

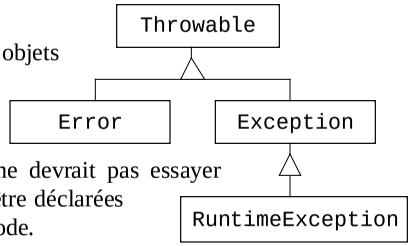
```
void foo() throws Exception
{
     ...
     throw new Exception();
     ...
}
```

Remarques

- Une exception qui est levée dans une méthode mais qui n'est pas capturée doit être déclarée dans la signature de la méthode.
- Une méthode foobar() qui fait appel à une méthode foo() dont la signature signale qu'une exception est levée, doit elle-même (1) intercepter l'exception, ou (2) déclarer cette exception.
- Attention: `throws` dans la signature et `throw` pour lever l'exception.

Exceptions (5) - hierarchie d'exceptions

Classes de base



- La classe `Throwable` représente tous les objets qui peuvent être levés (`throw`) ou interceptés (`catch`).
- Les instances d'`Error` représentent un sérieux problème qu'une application ne devrait pas essayer d'intercepter. Elles n'ont jamais besoin d'être déclarées explicitement dans la signature d'une méthode.
  e.g., `OutOfMemoryError`
- Les instances d'`Exception` représentent un problème qu'une application normale peut raisonnablement avoir besoin d'intercepter.
  e.g., `FileNotFoundException`
- Les instances de `RuntimeException` représentent une erreur générée par la machine virtuelle ou très commune. Elles n'ont jamais besoin d'être déclarées explicitement dans la signature d'une méthode.
  e.g., `NullPointerException`

Exceptions (6) - definir une nouvelle exception

```
class FooException extends Exception
{
     public FooException()           { super(); }
     public FooException(String msg) { super(msg); }
}
```

Remarques

- Pour définir une nouvelle catégorie d'exception, il suffit de créer une sous-classe d'`Exception` (resp. de `RuntimeException` ou d'`Error`).
- Une exception peut avoir ses propres variables d'instance et donc transporter une information supplémentaire.

Exception (7) - intercepter une exception (bis)

```
try {
     ...
} catch(FooException e) {
     ...
} catch(FoobarException e) {
     ...
}
```

Remarque

- Il peut y avoir plusieurs blocs "`catch`"; c'est le premier qui correspond à l'exception qui est exécuté.
- Un bloc "`catch`" correspond à une exception `e`, si `e` est une instance de la classe qui est déclarée dans le bloc ou d'une de ses sous-classes.
  e.g., si `e` est de type `FooException` ou d'un sous-type, alors `e` est interceptée par le premier bloc "`catch`".
- Si aucun bloc "`catch`" ne correspond à une exception, celle-ci reste levée à la sortie de l'ensemble `try-catch`.

Exceptions (8) - classe `Throwable`

Constructeurs

- `public Throwable()`
- `public Throwable(String)`
  Constructeur permettant de créer une nouvelle exception et d'y associer un message.

Méthodes

- `String getMessage()`
  Retourne le message associé à l'exception.
- `String toString()`
  Donne une représentation textuelle de l'exception pouvant être affichée.
- `void printStackTrace()`
  Affiche le contenu de la pile au moment où l'exception a été levée.
- `Throwable fillInStackTrace()`
  Permet de remettre à jour la pile de l'exception. Surtout utile lorsqu'une exception est créée et jetée depuis des endroits différents.

Exception (9) - comportement par défaut

```
public static void _system_(String [] args)
{
     try {
        // initialise le système.
        main(args);
     } catch (Throwable e) {
        e.printStackTrace();
     } finally {
        // libère les ressources du système.
     }
}
```

❑ Remarques

- Par défaut, le système intercepte toutes les exceptions, erreurs, etc.
- Si une exception intervient, le système affiche la pile de l'exception.

```
FooException
     at Foo.foo(Foo.java:8)
     at Foo.bar(Foo.java:15)
     at Foo.main(Foo.java:44)
```


Exceptions (10) - clause `finally`

```
try {
     ...
} catch (Exception e) {
     ...
} finally {
     ...
}
```

❑ Remarques

- La clause `finally` permet de spécifier un bloc de code qui doit être exécuté dans **tous** les cas.
- Si une exception n'est pas interceptée ou alors est relevée dans un des blocs "`catch`", cette exception reste levée après l'exécution du bloc "`finally`".
- La clause `finally` permet notament de fermer ou de libérer des ressources utilisées dans le bloc `try` (e.g., fermer un fichier, libérer une connexion, etc).

Classes intérieures

❑ Visibilité des classes

- Classes publiques (exportées).
- Classes protégées (visibilité limitée au paquetage).
- Classes intérieures.

❑ Classes intérieures (*inner classes*)

- Classes imbriquées (*nested classes*)
- Classes membres
- Classes locales
- Classes anonymes

❑ Différentes règles de visibilité

Classes imbriquées

```
class Outer
{
     static class Nested
     {
        ...
     }
     ...
}
```

❑ Remarques

- Une classe imbriquée est définie dans le contexte d'une classe englobante.
  e.g., `Nested` est imbriquée dans `Outer`.
- Une classe imbriquée peut accéder aux variables statique de la classe englobante.
- Le nom d'une classe imbriquée est préfixé du nom de la classe englobante.
  e.g., `Outer.Nested`.
- Une classe imbriquée peut être déclarée publique, protégée, privée.
- Tout ceci est aussi valable pour les interfaces imbriquées.

Classes imbriquées (exemple)

```
public class LinkedList
{
     private Node head = null;

     static private class Node
     {
        Object element;
        Node   tail;
        public Node(Object element, Node tail)
           { this.elt=elt; this.tail=tail; }
     }
     static public class EmptyListException
     extends Exception { ... }

     public void insert(Object element)
     {
        head = new Node(element, head);
     }
     public Object removeFirst() throws EmptyListException
     { ... }
}
```


Classes membres

```
class Outer
{
     public class Inner
     {
        ...
     }
     ...
}
```

Remarques

- Une classe membre est définie dans le contexte d'une **instance** de la classe englobante (instance englobante).
  e.g., chaque instance d'`Inner` est définie dans le contexte d'une instance d'`Outer`.
- Une classe membre peut accéder aux variables de l'instance englobante.
- Une classe membre ne peut pas avoir le même nom que la classe englobante.
- Fait appel à une syntaxe particulière pour `this`.

Classes membres (2)

```
class Outer
{
     Object foo;
     class Inner
     {
        Object foo;
        public foobar(Object foo)
        {
           ...
        }
     }
}
```

Dans la méthode `foobar()`

- `foo` fait référence au paramètre de la méthode `foobar()`.
- `this.foo` fait référence à la variable d'instance déclarée dans `Inner`.
- `Outer.this.foo` fait référence à la variable d'instance déclarée dans `Outer`.

Classes membres (exemple)

```
public class LinkedList
{
     private Node head = null;
     private class Enumerator implements Enumeration
     {
        private Node current;
        public Enumerator()
        { current = head; }
        public boolean hasMoreElements()
        { return current != null; }
        public Object nextElement()
        {
           current = current.tail;
           return current.element;
        }
     }

     public Enumeration elements()
     { return new Enumerator(); }
     ...
}
```

Autres classes imbriquées

- **Classe locale**
  - Classe définie à l'intérieur d'un bloc de code.
  - Peut utiliser les variables d'instance de l'instance englobante.
  - Peut accéder aux variables locales du bloc, pour autant qu'elles soient finales.
- **Classe anonyme**
  - Classe sans nom définie dans le contexte d'une expression.
  - Mêmes caractéristiques qu'une classe locale.
- **Remarques**
  - Ces classes imbriquées ont tendance à rendre le code un peu ésotérique.

Java

Les classes de bases

Contenu

- **Aperçu des paquetages standards**
- **Classes de `java.lang`**
- **Classes de `java.util`**
- **Entrées-sorties (`java.io`)**

Aperçu des paquetages standards

- **Les paquetages standards principaux sont:**

`java.applet`	classes de base pour la création d'applets.
`java.awt`	interfaces graphiques et dessin 2D.
`java.beans`	support pour la création de composants logiciels.
`java.io`	support pour les entrées-sorties.
`java.lang`	classes de base.
`java.lang.reflect`	support pour la reflectivité (lecture seule).
`java.math`	support pour les grands entiers.
`java.net`	support pour le connection réseau (TCP/IP).
`java.rmi`	invocation de méthode à distance.
`java.security`	classes permettant de gérer des listes d'accès, etc.
`java.sql`	interface pour gestionnaire de base de donnée.
`java.text`	classes de base pour le traitement de texte.
`java.util`	classes utilitaires diverses (tables associatives, etc).
`java.util.zip`	classes de compression/décompression (zip, gzip, etc).

Le paquetage `java.lang`

Introduction

- paquetage de base - toujours importé.
- paquetage toujours présent quelque soit la machine virtuelle.
- contient les classes décrivant les types primitifs.
- contient les classes décrivant le système.

Classes importantes

- `Object`
- `String`
- Classes "*Wrapper*": `Integer`, `Long`, `Boolean`, `Byte`, `Number`, ...
- `Math`
- `System`

La Classe `Object`

Classe mère

- toutes les autres classes Java descendent d'`Object`
- implictement une classe descend d'`Object`

Méthodes de la Classe `Object`

- Ces méthodes servent à décrirent un objet et devraient être surchargées par les descendants de la classe:

`toString()`	rend une version textuelle de l'objet *appellé implictement si nécessaire.*
`clone()`	fait une copie de l'objet (par défaut "*shallowcopy*").
`equals()`	compare avec un autre objet rend `true` si égal.

La classe `String`

Représente une chaîne de caractères

- suite ordonnée de caractères (Unicode).
- longueur arbitraire.
- **pas** un tableau.

Créé implicitement

- Lorsqu'un chaine de caractères se trouve dans le code source.
- `String a = "toto";`

Objet immuable

- Une instance de String ne peut être modifiée.
- Toutes les opérations créent de nouvelles instance.

La classe `String` (2)

Méthodes de la Classe String

`length()`	rend la longueur de la chaîne. E.g. `"toto".length() -> 4`
`concat()`	construit une nouvelle chaîne qui est la concaténation de deux autres. *Implicitement appellée par l'opérateur* +. E.g. `"toto".concat("ro") -> "totoro"`
`toLowerCase()`	construit une nouvelle chaîne en minuscule.
`subString()`	construit une sous-chaîne.
`startsWith()`	vérifie si une chaîne commence avec une autre.
`indexOf()`	cherche position dans une chaîne d'une sous chaîne / d'un caractère.

Les classes "*Wrapper*"

- **Classes "*Wrapper*"**
  - Enrobent une type primitif dans une classe.
  - Offrent un aspect objet aux types scalaires.
  - Une classe par type primitif: `Integer` (int), `Boolean` (boolean) ...
  - Toutes les classes "*Wrapper*" de types numériques descendent de la classe `Number`.
  - Offrent des méthodes auxilliaires (conversion, tests, etc.)

```
String    s = "1291";
int       i = Integer.parseInt(s);
```

  - Contiennent description des types scalaires pour reflection.

- **Utilisation avec collections**
  - Les collections contiennent des objets (pas de types scalaires).

```
public class Vector ...
    public void addElement(Object obj);
```

```
Vector v = new Vector();
v.addElement(new Integer(10));
v.addElement(new Float(3.4));
v.addElement(new Byte(5));
v.addElement("ceci est du texte");
```


La classe `Math`

- **Classe "vide"**
  - ne contient que des méthodes et des variables statiques.
  - ne peut être instanciée.
- **Fonctions mathématiques**
  - `sin`, `cos`, `abs`, `max`, `log`, etc.
- **Constantes mathématiques**
  - `E`, `PI`

Classe `System`

- **Représente le système**
  - permet de communiquer avec l'extérieur de la machine virtuelle.
- **Sorties et entrées standard**
  - `System.in`, `System.out`, `System.err`
    E.g. `System.out.println("hello world");`
- **Controle de la machine virtuelle**
  - `System.exit()` permet de sortir de la machine virtuelle.

java.lang: Classes Diverses

- **Exception**
  - décrit une exception.
- **Runtime**
  - décrit l'environnement Java.
- **Process**
  - décrit un processus externe à Java.
- **Thread**
  - décrit un "thread" d'exécution.

Le paquetage java.util

- **Introduction**
  - Ce paquetage contient diverses classes auxilliaires: des utilitaires.
  - Ces classes servent souvent de bases aux classes dans d'autres paquetages plus spécialisés.
  - Contient des classes pour la localisation (adaptation selon les pays/languages).
- **Classes & Interfaces Importantes**
  - `Vector`, `Hashtable`, `Stack`.
  - `Enumeration`.
  - `StringTokenizer`.

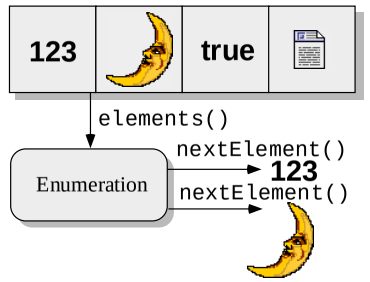
La Classe Vector

- **Décrit un vecteur**
  - collection ordonnée d'objets.
  - contient des objets de type arbitraire.
  - longueur variable et arbitraire.

- **Méthodes**
  - `addElement()` ajoute un élément à la fin.
  - `isEmpty()` vérifie si le vecteur est vide.
  - `elementAt()` rend l'élément a une certaine position.
  - `length()` rend la longueur du vecteur.

L'interface Enumeration

- **Enumeration**
  - permet d'accéder séquentiellement aux éléments d'une collection.
  - ne permet qu'une utilisation.
- **Utilisation**
  - une énumeration est obtenue des collections par l'appel à la méthode `elements()`.



- `hasMoreElements()` permet de savoir s'il reste des éléments.
- `nextElement()` rend l'élément suivant.

Exemple: éléments d'un vecteur

- **Exemple**
  - affiche tous les éléments d'un vecteur

```
Vector v = new Vector();
v.addElement("Un petit peu de texte");
v.addElement(new Integer(10));

Enumeration e = v.elements();
while(e.hasMoreElements()) {
     final Object o = e.nextElement();
     System.out.println(o.toString());
     } // while
```

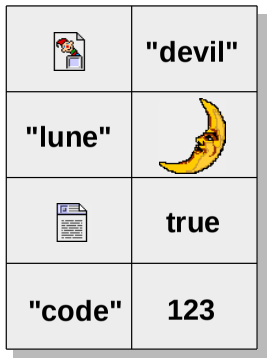
  - Résultat

```
Un petit peu de texte
10
```


La Classe `HashTable`

- **Table associative**
  - équivaut à `Dictionnary` en SmallTalk
  - contient une série d'associations.
  - chaque élément est désigné par une clef.
  - éléments et clefs sont des objets.

- **Méthodes**
  - `put()` associe un objet avec une clef
  - `get()` retourne l'objet associé à une clef
  - `remove()` enlève l'association liée à une clef
  - `keys()` retourne une `Enumeration` des clefs



Paquetage `java.util` - autres classes

- **`StringTokenizer`**
  - permet de découper en éléments lexicaux (tokens) une chaîne de caractères.
  - les éléments lexicaux sont définis pour des séparateurs: espace, ponctuation etc.
  - tokens sont rendus dans une `Enumeration` (`StringTokenizer` est une `Enumeration`).

- **`Properties`**
  - `HashTable` particulière - uniquement du texte
  - Peut être sauvée en format texte (fichier)
  - Utilisée pour stocker les informations du système.

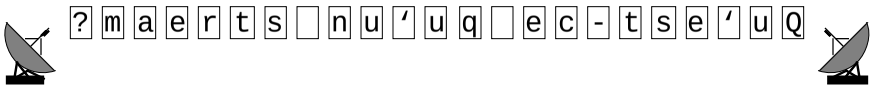
- **`Stack`**
  - sous classe de `Vector` - implémente une pile.
  - deux opérations: `push()`et `pop()`.

`java.io`

- **Description**
  - Classes permettant de gérer les entrées-sorties et la manipulation de fichiers.
  - Les entrées-sorties sont basées sur la notion de flot (en anglais, *stream*).

- **Classes importantes**
  - `Reader`, `Writer`
  - `InputStream`, `OutputStream`
  - `File`

Qu'est-ce qu'un *flot*?

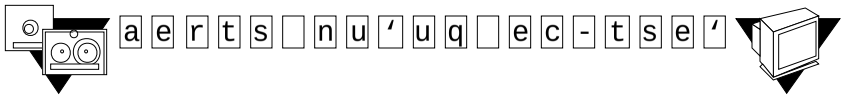


Remarques

- Un "flot" est un flot d'information dont il existe une source et une destination.
- Il s'agit d'une généralisation des entrées-sorties.

Exemple

- Lecture et affichage d'un fichier



Streams vs Reader/Writer

Explications

- En Java, il existe deux types de flots; les `Stream` et les `Reader`/`Writer`.
- Les deux type de flots existent en **entrée** (`InputStream`, `Reader`) et en **sortie** (`OutputStream`, `Writer`).

Streams

- Les paires `InputStream` et `OutputStream` représentent des flots de **bytes**.

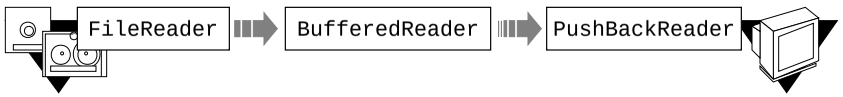
Reader/Writer

- Les paires `Reader` et `Writer` représentent des flots de **caractères** (Unicode).

Composition de flots

Remarques

- `Reader` et `Writer` sont des classes abstraites. Leurs sous-classes définissent des flots ayant différentes propriétés.
- Un flot peut être une **source**.
  e.g., `FileReader`, `StringReader`, `CharArrayReader`
- Un flot peut être une **destination**.
  e.g., `FileWriter`, `StringWriter`, `CharArrayWriter`
- Un flot peut être un **filtre**.
  e.g., `PushBackReader`, `PrintWriter`, `BufferedReader`, `BufferedWriter`
- Les objets flots peuvent être combinés pour former un nouveau flot.
  e.g., `new PushBackReader(new BufferedReader(new FileReader("tt")))`



Exemple: lire un fichier

```
public static void main(String [] args) throws IOException
{
     Reader in  = new FileReader(args[0]);
     Writer out = new OutputStreamWriter(System.out);
     try {
        while (true) {
           int token = in.read();
           if (token < 0) {
              // fin de flot detectee; sort de la boucle.
              break;
           } else {
              out.write(token);
           }
        }
     } finally {
        out.close();
        in.close();
     }
}
```

Exemple: écrire dans un fichier

```
public static void main(String [] args) throws IOException
{
     final String FILENAME = "arguments";
     Writer      file = new FileWriter(FILENAME);
     PrintWriter out  = new PrintWriter(file);

     try {
        for (int i=0; i<args.length; i++)
           out.print("args[");
           out.print(i);
           out.print("]=");
           out.println(args[i]);
     } finally {
        out.close();
     }
}
```

Remarque

- Lit les arguments de la ligne de commande et les sauve dans un fichier.
- Le `PrintWriter` permet d'utiliser les méthodes `print()` et `println()`.

Créer une nouvelle classe de stream

Méthode

- Il faut créer une nouvelle sous-classe de `Reader` (resp. `Writer`).
- Il faut donner une implementation pour la méthode `read` (resp. `write`).

Exemple: `Rot13Reader`

- Rot13 est une méthode de cryptage archaique utilisée par Jules César.
- Un document codé au moyen de Rot13 consiste à remplacer chaque lettre par la lettre qui est 13 crans plus loin modulo 26.
  E.g., A devient N, B devient O, ..., Z devient M.
- Rot13 a la particularité de se décoder lui-même (i.e., Rot13(Rot13(*texte*)) = *texte*).
- Notre exemple est une classe `Reader` qui code/décode un flot en Rot13.

Exemple: `Rot13Reader`

```
class Rot13Reader extends Reader
{
     private Reader in;

     static private char Rot13(char c)        { ... }

     public Rot13Reader(Reader in)            { this.in=in; }
     public void close() throws IOException   { in.close(); }

     public int read(char buffer[], int offset, int length)
        throws IOException
     {
        final int real_len = in.read(buffer, offset, length);
        for (int i=offset; i<offset+real_len; i++) {
           buffer[i] = Rot13(buffer[i]);
        }
        return real_len;
     }
}
```


Classes utiles (1)

`Reader`

- `BufferedReader`
  Flot dans lequel les caractères sont tamponnés (buffer).
- `CharArrayReader`
  Source. Les caractères sont lus depuis un tableau de caractères.
- `FileReader`
  Source. Les caractères sont lus depuis un fichier.
- `InputStreamReader`
  Conversion. Les caractère sont lus depuis un `InputStream`.
- `LineNumberReader`
  Flot tamponné qui compte les numéros de ligne.
- `PushbackReader`
  Flot avec lequel il est possible de revenir en arrière.
- `StringReader`
  Source. Les caractères sont lus depuis une chaîne de caractères.

Classes utiles (2)

- **`Writer`**
  - `BufferedWriter`
    Flot dans lequel les caractères sont tamponnés (buffer).
  - `CharArrayWriter`
    Destination. Les caractères sont écrits dans un tableau de caractères.
  - `FileWriter`
    Destination. Les caractères sont écrits dans un fichier.
  - `OutputStreamWriter`
    Conversion. Les caractères sont écrits dans un `OutputStream`.
  - `PrintWriter`
    Met à disposition les méthodes `print` et `println`.
  - `StringWriter`
    Destination. Les caractères sont écrits dans une chaîne de caractères.

Manipulation de fichiers

- **classe `File`**
  - Classe permettant de manipuler des fichiers et des répertoires.
  - Une instance de `File` représente un nom de fichier ou de répertoire.
- **Fonctionalités**
  - Tester l'existence d'un fichier ou d'un répertoire.
  - Tester les droits d'un fichier.
  - Effacer/renommer un fichier ou un répertoire.
  - Obtenir la liste des fichiers contenus dans un répertoire.
  - Manipuler le chemin d'un répertoire.
  - Obtenir la taille, date de création/modification d'un fichier ou d'un répertoire.

Exemple: tester l'existence d'un fichier

```
public static void main(String [] args) throws IOException
{
     final String FILENAME = "arguments";
     File file = new File(FILENAME);

     if (file.exists()) {
        throw new IOException("file exists");
     }

     PrintWriter out  = new PrintWriter(new FileWriter(file));

     ...
}
```


Exemple: contenu d'un répertoire

```
public static void main(String [] args) throws IOException
{
     File user_dir = new File(System.getProperty("user.dir"));

     if (user_dir.isDirectory()) {
        String [] list = user_dir.list();

        System.out.println(user_dir.getCanonicalPath()+":");
        for (int i=0; i<list.length; i++) {
           File entry = new File(user_dir, list[i]);

           System.out.print(entry.length());
           System.out.print('\t');
           System.out.print(entry.getName());
           if (entry.isDirectory())
              System.out.print(File.separator);
           System.out.println();
     }
}
```