Algorithmes et langage Java

Tatiana AUBONNET
Conservatoire National des Arts et Métiers

Chapitre 1
Introduction aux concepts et à la programmation objet

Sommaire

- Organisation
- Bibliographie
- Le concept d'objet
- Présentation et les premières notions de Java
  - Qu'est-ce que Java
  - Les API
  - Les outils
- Compilation et exécution
- Un premier exemple avec BlueJ

Organisation

- 27 h de cours / travaux pratiques (10 séances de 3h /2h)
- A travers les concepts de langage Java on introduira :
  - Conception et programmation objets
  - API (*Application Programming Interface)*
  - Un ensemble d'outils JDK (*Java Development Kit*) …
- Contrôle
  - Un projet de programmation en java réalisé en binôme.
    - Les 3 notions suivantes doivent être utilisées dans le projet : héritage, surcharge, exceptions.
  - Soutenances du projet : mercredi 29 avril 2008

Programme de ce cours

- Introduction aux concepts et à la programmation objet
- Les classes :
  - Définir et utiliser une classe
  - Principe d'encapsulation
  - Créer et utiliser une instance de classe
  - Surcharge d'une méthode
- Manipulation d'un tableau d'objets
  - Définir un tableau
  - Longueur d'un tableau et dépassement des bornes
  - Un tableau d'objets
  - Tableaux à deux dimensions
- L'héritage et les interfaces
  - Étendre une classe
  - Ce qu'est une interface
  - Implémenter une interface
  - Redéfinir une méthode
- Comprendre les exceptions
  - Définir sa propre exception
  - Un exemple où on lance soit même une exception
- Interfaces graphiques, **flux des données**

Bibliographie

- **James Rumbaugh**

  Object-Oriented Modeling and Desing

  Prentice Hall, 1991
- **I. Jacobson**

  Object-Oriented Software Engineering

  Addison-Wesley
- **Irène Charon**

  Le langage Java : concepts et pratique (3ème édition revue et augmentée)

  Éditions Hermès, parution janvier 2006
- **J. Gosling, B. Joy, G. Steele**

  The Java Language Specification
- **K. Arnold, J. Gosling**

  The Java Programming Language
- **David J. Barnes & Michael Kölling**

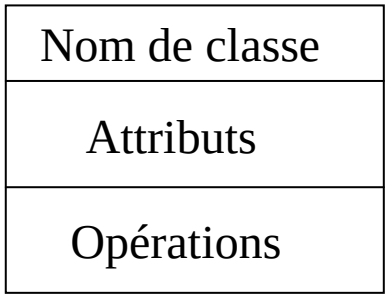
  Objects First with Java. A Practical Introduction using BlueJ
  Pearson Education, 2003

Références

- Site Web :
  - http://www.sun.com/: *le site java de Sun*
  - http://java.sun.com/developer/onlineTraining/Programming/BasicJava1 : *cours "de base" partie 1*
  - http ://www.javasoft.com : *Site officiel Java (JDK et doc.)*
  - http://www.infres.enst.fr/~charon/coursJava/index.html
  - http ://www.javaworld.com : *Info sur Java*
  - http//www.gamelan.com : *applications, applets, packages, ...*
  - http ://www.jars.com : *idem*
  - http ://www.blackdown.com : *Java pour linux*
  - http ://www.bluej.org/doc/documentation.html : *BlueJ*

Le concept d'objet

- Un objet est identifié par son **nom** et caractérisé par un ensemble **d'attributs** et **d'opérations**
- Représentation



Exemple

Voiture
Prix Couleur Puissance
Commander Livrer

- Un **objet réunit** en une seule entité :
  - **les données**
  - **les traitements**

Notion de Classe et d'Instance

- **Classe**
  - On constate que certains objets ont des caractéristiques communes :
    - attributs
    - opérations
  - Au lieu de les représenter individuellement chaque objet, on utilise **le concept de classe**

- **Instance**
  - "**Une instance est un objet crée à partir d'une classe**. La classe décrit la structure de l'instance (opérations et attributs), tandis que l'état courant de l'instance est défini par les opérations exécutées sur l'instance" (I. Jacobson)

L'encapsulation

- **Le concept d'encapsulation** permet de séparer :
  - **l'aspect externe d'un objet**
    - *défini lors de la phase de modélisation*,
    - qui forme son interface (nom des attributs et nom des opérations)
  - **de l'aspect interne**
    - précisé lors de la phase de programmation
    - façon dont sont réalisés les attributs et les opérations

Encapsulation et attribut

- Permet d'atteindre 2 objectifs :
  - modélisation sans se soucier des problèmes d'implantation
  - l'implantation offrant la flexibilité du langage de programmation pour coder les méthodes

Encapsulation et opération

- La structure externe d'un opération est appelée sa signature. Elle est publique.

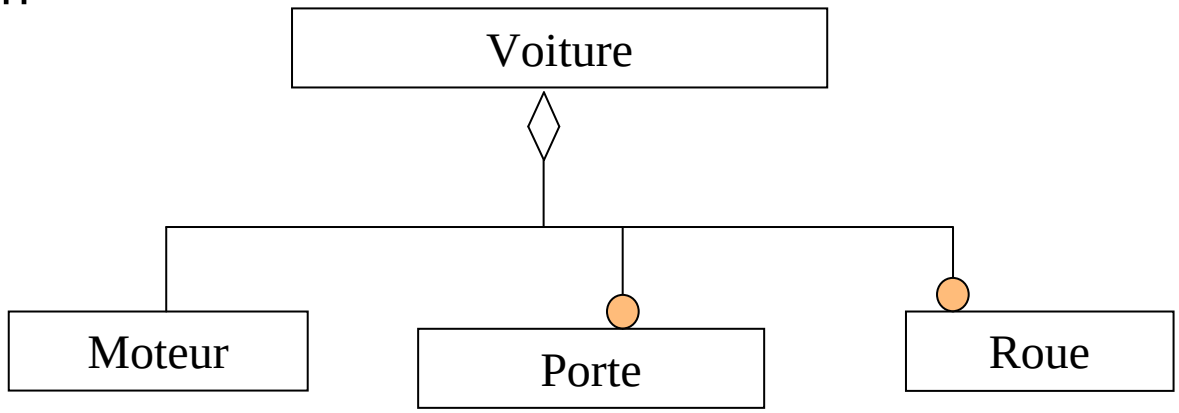
- La partie de programme (ou code) associée, qui est **appelé méthode** est cachée.

Relations statiques : Héritage

- **La notion d'héritage** peut se définir comme suit :
  - Si un classe *X* hérite d'une classe *Y*, les opérations et les attributs de la classe *Y* appartiennent également à la classe *X*.
  - La classe Y est souvent désignée comme *super-classe* et la classe X comme *sous-classe.*
- Le mécanisme d'héritage permet à une nouvelle classe de recevoir des attributs et des opérations précédemment définis.
- Un autre intérêt de l'héritage est de faciliter les mises à jour (par une modification de super classe).
- Le verbe être

Relations statiques : Agrégation

- **L'agrégation** caractérise une relation entre classes dans laquelle une classe est composée d'une ou plusieurs autres classes.
- Représentation



- La relation d'agrégation limite toute modification d'un objet
- Le verbe avoir

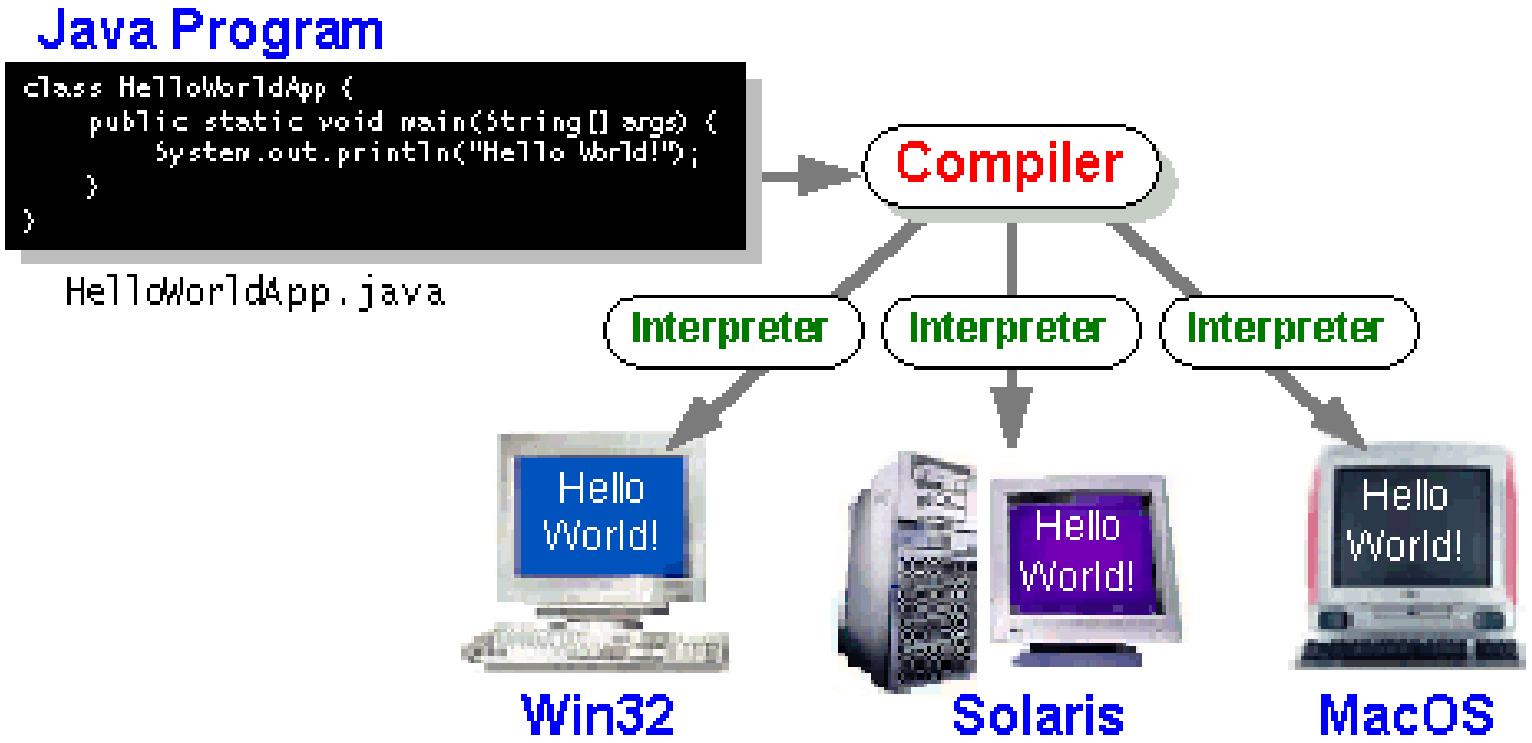
Terminologie et représentation

- Lorsque l'on utilise l'approche objet dans les 3 phases :
  - **analyse/conception**
    - on s'intéresse à l'aspect externe de l'objet (attributs, opérations).
  - ***programmation***
    - on s'intéresse à l'aspect interne de l'objet : il faut *programmer ses méthodes* et définir la façon la plus appropriée pour représenter ses attributs.
  - ***production***
    - lors de l'exécution de l'application, des structures d'objets sont créées dynamiquement
    - ces structures appelées *instances* sont identique à celle de la classe à partir de laquelle elles ont été créées.

Java – langage objet, compilé et interprété

- Langage objet
- compilé et interprété
  - compilation ⟶ bytecode (indépendant de l'architecture)
  - exécution : "nécessite une machine virtuelle Java" (spécifique à un système) qui interprète le bytecode pour l'exécuter)

Comment cela fonctionne t'il ?



L'interprète est une machine virtuelle

Qu'est-ce que Java ?

- Une architecture de *Virtual Machine*
- Un ensemble d'API (*Application Programming Interface*) variées
  - nouvelle méthode de travail
- Un ensemble d'outils (le JDK *Java Development Kit*)

Le « Java Development Kit »

- Java est un langage de programmation.
- Le « Java Development Kit » est une boite à outils :
  - il fournit un compilateur java
  - il fournit une machine virtuelle
  - il fournit un ensemble de bibliothèques d'outils pour faciliter la programmation.

Applets et applications

- Deux types de développements sont possibles :
  - ***les Applets***

    il s'agit d'un programme s'exécutant par exemple au sein d'un navigateur web. Une applet ne peut s'exécuter indépendamment et doit être logée et activée par une autre application.

  - ***les applications***

    il s'agit de programme standard et indépendant.

Les différences avec C

- Pas de structures
- Pas de types énumérés
- Pas de *typedef*
- Pas de variables ni de fonctions en dehors des classes
- Pas de pointeurs, seulement des références

Les premières notions (1)

- Tout programme écrit en Java se fonde sur l'utilisation de classes.
- Une classe est constituée d'un ***ensemble d'attributs*** et d'un ***ensemble de méthodes*****.**
- Un programme construit en général *des instances de classe*, une instance de classe est appelé objet.

Les premières notions (2)

- Il est possible de ranger les classes selon des ensembles appelés ***packages* (paquetages).**
  - Un package regroupe un ensemble de classes sous un même espace de 'nomage'.
  - Les noms des packages suivent le schéma : `name.subname ...`
  - Une classe `Watch` appartenant au package `time.clock` doit se trouver dans le fichier `time/clock/Watch.class`
  - Les packages permettent au compilateur et à la JVM de localiser les fichier contenant les classes à charger.
  - L'instruction `package` indique à quel package appartient la ou les classe(s) de l'unité de compilation (le fichier).

Les premières notions (3)

- Java est livré avec un grand ensemble de classes.
- Tous les fichiers sources doivent avoir l'extension ".java".
- L'ensemble des paquetages livrés par Java forme ce qu'on appelle l'API *(Application Programming Interface*), celle-ci s'élargit continuellement.
  - Un descriptif en ligne par Sun Microsystems : http ://www.javasoft.com

Les premières notions (4)

Entre autres, on peut trouver dans l'API les paquetages ci-dessous (Les *core* API 1.0.2 et 1.1) :

- Le paquetage ***java.lang*** contient les classes les plus centrales du langage. Il contient la *classe Object* qui est la super-classe ultime de toutes les classes : Types de bases, Threads, ClassLoader, Exception, Math, ...
- Le paquetage ***java.util*** définit un certain nombre de *classes utiles* et est un complément de java.lang.
- Le paquetage ***java.awt*** (awt pour Abstract Window Toolkit) contient des classes pour fabriquer des *interfaces graphiques*.
- Le paquetage ***java.applet*** est utile pour faire des applets, *applications utilisables à travers le Web.*
- Le paquetage ***java.io*** contient les classes nécessaires aux *entrées-sorties*.
- Le paquetage ***java.net*** fournit une infrastructure pour la programmation réseau : Socket (UDP, TCP, multicast), URL, ...

Les *core* API 1.1

- **java.beans** : composants logiciels
- **java.sql** (JDBC) : accès homogène aux bases de données
- **java.security** : signature, cryptographie, authentification
- **java.rmi** : *Remote Method Invocation*
- **java.idl** : interopérabilité avec CORBA

Les autres API

- **Java Server : ** jeeves / servlets
- **Java Commerce*** : JavaWallet
- **Java Management** (JMAPI) : gestion réseau
- **Java Média** : 3D, Média Framework, Share, Animation*, Telephony

(*) Feront partie, à terme, des *core* API

Compilation et exécution

- Lorsque le programmeur ne range pas ses programmes en paquetages, et suppose que vous utilisez le JDK (Kit de développement de Java de Sun).
- Pour compiler, il faut :
  - se mettre dans le répertoire contenant le fichier source
  - utiliser la commande **javac** suivie du nom du fichier source : cela crée un fichier pour chaque classe contenue dans le fichier compilé. Ces fichiers ont pour nom le nom de la classe correspondante, suivi de l'extension .class.
- Pour exécuter, il faut :
  - avoir un fichier contenant une classe contenant une méthode main taper java suivi du nom (sans extension) de la classe contenant *le main*.

Un premier exemple (1)

- **Écrire "bravo"**

  - se trouve dans le fichier
    - Premier.java
  - est compilé par
    - javac Premier.java,
  - ce qui crée un fichier
    - Premier.class
  - est exécuté avec
    - java Premier

Le point d’entrée d ’une application

- Pour débuter une application on doit fournir un point d'entrée. Lorsqu'une classe sert d'application elle doit fournir ce point d'entrée qui est une méthode statique portant le nom de « main ».

- Ce point d'entrée doit respecter la syntaxe suivante :

```
public static void main( String [ ] args )
{
   // Corps du point d'entrée
}
```

- Où « args » correspond à la liste des arguments passé depuis la ligne de commande.

Un premier exemple

```
class Premier
{ public static void main(String [ ] arg)
{ System.out.println("bravo");
}
}
```

- La fonction main se présente toujours comme ci-dessous. Elle est obligatoirement définie à l'intérieur d'une classe.
  - **public** : est visible de partout, y compris les autres paquetages
  - **static :**qu'il s'agit d'un attribut ou d'une méthode de classe
  - **String** : il s'agit d'une classe définie dans java.lang. La fonction main a pour paramètre un tableau de chaînes de caractères.
  - **System** : classe de java.lang qui fournit quelques fonctionnalités systèmes, de façon indépendante de la plate-forme.
  - **out** : instance de la classe java.io.PrintStream
  - **println("bravo")** : on utilise ainsi une méthode de la classe PrintStream.

Conversion de types (1)

- La conversion de type (« ***cast*** ») est une technique fondamental de la programmation :
  - Pour convertir un type en un autre on respecte la syntaxe suivante :

    (type de convervion) type_a_convertir

Conversion de types (2)

- **conversion implicite**: une conversion implicite consiste en une modification du type de donnée effectuée automatiquement par le compilateur.

```
int x;
x = 8.324;
```

x contiendra après affectation la valeur 8

- **conversion explicite**: (*opération de cast*) consiste en une modification du type de donnée forcée.

```
int x;
x = (int)8.324;
```

x contiendra après affectation la valeur 8

Les types primitifs

- boolean(true/false), byte (1 octet), char (2 octets), short (2 octets), int (4 octets), long (8 octets), float (4 octets), double (8 octets).
- Les variables peuvent être déclarées n'importe où dans un bloc.
- Conversions de type:
  - Les cas de conversion permis (implicites) :
    - byte --> short --> int --> long --> float --> double
  - Les affectations non implicites doivent être *castées* (sinon erreur à la compilation).
    - Opérateur de *cast* : (type) un par type

Exemples (conversion forcée par un affectation)

```
int i = 258;
long l = i;   // ok, i est une variable entère
byte b = i;   // error : Explicit cast needed to convert int to byte
byte b = 258; // error : Explicit cast needed to convert int to byte
byte b = (byte) i; // ok, utilisation de l'opérateur de cast (perte d'info)
```

Les structures de contrôle et expressions

- Essentiellement les mêmes qu'en C
  - if,
  - switch,
  - for,
  - while,
  - do while

Exercice

- Écrire un programme qui calcule la somme des 100 premiers entiers et indique à l'écran le résultat.

Sachant que :

- la syntaxe de base de java est quasiment la même que celle du C
- si n est une variable de type int, l'instruction :
  System.out.println(n);
  provoque l'écriture du contenu de la variable n
- l'instruction : System.out.println("voila " + n);
  provoque l'écriture du mot voila suivi du contenu de la variable n

Un corrigé

```
class Somme
{
  public static void main(String[] arg)
  {
    int i, somme = 0;

    for (i = 1; i <= 100; i++) somme += i;

    System.out.println("Voila la somme des 100 " +
                       "premiers entiers : " + somme);
  }
}
```

Java est robuste

- A l'origine, c'est un langage pour les applications embarquées.
- Gestion de la mémoire par un *garbage collector.*
- Mécanisme d'exception.
- compilateur contraignant (erreur si exception non gérée, si utilisation d'une variable non affectée, ...).
- Tableaux = objets (taille connue, débordement $\rightarrow$ exception).
- Seules les conversions sûres sont automatiques.
- Contrôle des *cast* à l'execution.

Notion de garbage collector

- Un garbage collector gère la désallocation de la mémoire :
  - le programmeur alloue la mémoire dont il a besoin,
  - le programmeur ne désalloue pas la mémoire :
    - plus de risque de fuite mémoire,
    - plus de risque d'utiliser un pointeur désalloué.

- Le concept de garbage collector apparaît très tôt au sein des langages de programmation, par exemple dès 1969 dans LISP.

Java est sécurisé

- Indispensable avec le code mobile.
- Pris en charge dans l'interpréteur.
- Trois couches de sécurité :
  - *Verifier* : vérifie le *byte code*.
  - *Class Loader* : responsable du chargement des classes.
  - *Security Manager* : accès aux ressources.
- Code certifié par une clé.

Java est multi-thread

- Intégrés au langage et aux API :
  - classes d'interfaces "Thread" et "*Runnable"* permettent la programmation *multithread*
    - `java.lang.Thread, java.lang.Runnable`
- Accès concurrents à objet gérés par un *monitor.*
- Implémentation propre à chaque JVM.

Java est distribué

- API réseau (java.net.Socket, java.net.URL, ...).
- Chargement / génération de code dynamique.
- Applet.
- Servlet.
- *Protocole / Content handler.*
- *Remote Method Invocation.*
- JavaIDL (CORBA).

Chapitre 2
Classes, attributs, méthodes

Sommaire

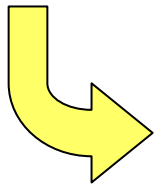
- Définir et utiliser une classe
  - Créer et utiliser une instance de classe
  - Types primitifs et types par référence
  - Utiliser un constructeur
  - Utiliser variables et méthodes statiques d'une classe
- Surcharge d'une méthode
- Les constantes

Notion de « class »

- Tout code java est défini dans des classes (ou des interfaces)
- Champs (membres) : grosso modo
  - attribut (variable)
  - et méthode (fonction)
- Les classes contiennent deux types bien distincts de champs :
  - champs de class, ou statique
  - champs d'instance
- Une variable ou une méthode de classe joue approximativement le rôle de variables ou des méthode globale.

Définir une classe (1)

- Pour pouvoir manipuler des objets, il est essentiel de définir des classes. Cette définition avec Java se fait de la manière suivante :



```
class nom_de_la_class
{
        corps de la classe
}
```

Définir une classe (2)

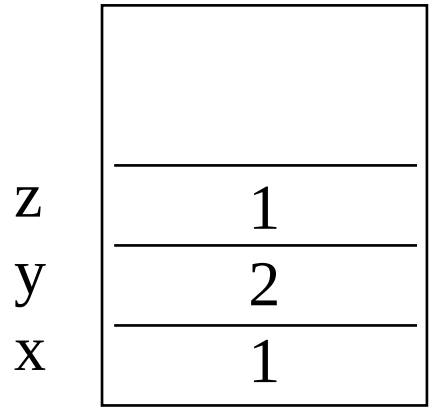
Exemple :

```
class Point
{ // les attributs
int x, y;
// Méthode que translate le point
void translate (int h, int k)
{
        x += h;
        y += k;
}
}
```

Types de données

- Deux types : **primitif et Référence (Object)**

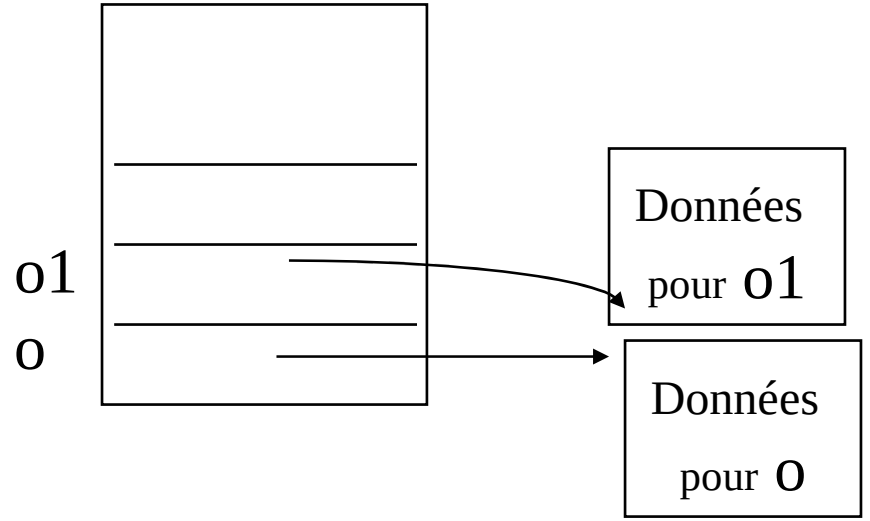
- **Type primitif** :
  - int x = 1;
  - int y = 2;
  - int z = x;



- **Type référence**
  - Object o = new Object();
  - Object o1 = new Object();

Une espace mémoire

est alloué à la variable o et o1



Les types primitifs

Les types primitifs sont :

- le type booléen **boolean**
  - qui n'est pas un type entier
  - valeurs ***false*** et ***true***
- le type caractère **char**
- les types entiers **byte**, **short**, **int** et **long**
- les types nombres flottants **float et double**

Type référence

- Ce sont de variables destinées à contenir des adresses d'objets ou de tableaux.
- Les données non primitives sont manipulées "par référence" alors que les données primitives sont manipulées par valeur.

- ***Terminologie :***

  Ex : Point p = **new** Point ( );

  - On dira que p est :
    - De type référence d'une instance de Point
    - De type Point
  - On dira que l'instance créée est :
    - L'instance référencé par p
    - L'objet p

  - **new** joue le rôle de malloc en C, la place mémoire est attribuée.

Initialisation des attributs

- Les attributs sont initialisés par défaut :
  - Types primitifs
    - **Int** : **0**
    - **Boolean** : **false**
    - **Char** : **‘\0’**
- Variable de type "par référence" : **null** (minuscule)

Accéder aux données et aux méthodes

- De l'extérieur de sa classe ou d'une classe héritée, un attribut ou une méthode de classe pourront être utilisés précédés du nom de sa classe :
  - **nom_d'une_classe.nom_de_l'attribut**
  - **nom_d'une_classe.nom_de_méthode_de_classe**

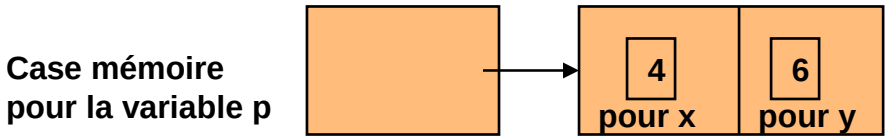
Dans une classe

…

```
Point p;

p = new Point ( );

p.x = 1;

p.y =2;

p. translate ( 3, 4);
```

méthode associée à la référence p

On dit que l'on instancie la classe Point : on crée ainsi un objet



Définir une classe

Exemple :

```
class Point
{ // les attributs
int x, y;
// Méthode que translate le point
void translate (int h, int k)
{
        x += h;
        y += k;
}
}
```

Instancier une classe

```
class EssaiPoint
{
   public static void main(String[] arg)
{

      int abs = 1;

      Point p;

      p = new Point();

      p.x = abs;
      p.y = 2;

      System.out.println("coordonnees : " + p.x + " " + p.y);
      p.translate(3, 5);
      System.out.println("coordonnees : " + p.x + " " + p.y);
   }
}
```

On obtient à l'exécution ?

Constructeur

- Méthode qui sert à « construire » les objets
- Automatiquement appelé quand on instancie une classe
- Toute classe possède au moins un constructeur.
- Même nom que la classe
- Si pas de constructeur, le compilateur en ajoute automatiquement un.

Définir un constructeur

- ***Un constructeur est une méthode*** qui est effectuée au moment de la création (par un **new**) d'une instance de la classe.

- Un constructeur ***ne retourne pas de valeur et ne mentionne pas void*** au début de sa déclaration.

- Par ailleurs, un constructeur ***doit posséder le même nom que celui de sa classe***. Il peut posséder des arguments qui seront initialisés de façon classique.

Utilisation d'un constructeur

```
class NouveauPoint
{
 // les attributs
      int x, y;
// constructeur de la classe
NouveauPoint (int abs, int ord)
{
       x = abs
      y  = ord
}
// Méthode que translate le point
void translate (int h, int k)
{
       x += h;
      y += k;
}
}
```

Utilisation d'un constructeur - suite

```
class EssaiNouveauPoint {
   public static void main(String[] arg) {
      NouveauPoint p = new NouveauPoint(1, 2);
      NouveauPoint q;

      q = p;
      p.translate(3, 5);
      System.out.println("Abscisse de p : " + p.x);
      System.out.println("Abscisse de q : " + q.x);
   }
}
```

On obtient à l'exécution ?

Créer et utiliser une instance de classe (1)

```
class Ecriture
{
  String chaine ="Encore une fois ";
  void ecrire(String autreChaine)
  {
    System.out.print(chaine);
    System.out.println(autreChaine);
  }
}

class PourFeliciter
{
  public static void main(String[] arg)
  {
    Ecriture ecrivain;

    ecrivain = new Ecriture();
    ecrivain.ecrire("bravo");
    ecrivain.chaine="et pour finir ";
    ecrivain.ecrire("au revoir");
  }
}
```

A l'execution, on obtient :

Encore une fois bravo et pour finir au revoir

Créer et utiliser une instance de classe (2)

```
class Ecriture
{
String chaine = "Encore une fois ";
void ecrire(String autreChaine)
{
 System.out.print(chaine);
System.out.println(autreChaine);
}
}
```

- L'attribut ***chaine*** est de type référence (adresse) et devra être une référence d'un objet de la classe *String*.
- La méthode *ecrire*, lorsqu'elle est invoquée, écrit à l'écran la chaîne de caractères référencée par chaine suivie de la chaîne passée en paramètre à cette méthode.

Créer et utiliser une instance de classe (3)

- Un programmeur a besoin d'écrire "Encore une fois ", suivi du mot "bravo" → la classe Ecriture peut me rendre facilement ce service.
  - Pour que le programme puisse être exécuté, le programmeur a l'obligation de créer une classe contenant une méthode main.
  - Choisissons ici de donner le nom PourFeliciter a la classe avec main.
  - Lorsque, après compilation, nous enverrons la commande :

    java PourFeliciter

    les instructions de la méthode main seront exécutées. Pour utiliser la classe Ecriture, instancions celle-ci, ce qu'il fait par l'instruction :

    ecrivain = **new** Ecriture();

Créer et utiliser une instance de classe (4)

- Disposant maintenant de **l'objet ecrivain** on peut utiliser la méthode ecrire de cet objet.
- Notez la façon de faire cela : nom de l'objet (ou plutôt de sa référence), suivi d'un point, suivi de l'appel proprement dit de la méthode.
- On peut aussi accéder aux attributs de ecrivain avec la même syntaxe : nom de l'objet, suivi d'un point, suivi du nom de l'attribut, comme le montre l'instruction A.

```
class PourFeliciter
{
  public static void main(String[] arg)
  {
    Ecriture ecrivain;

    ecrivain = new Ecriture();
    ecrivain.ecrire("bravo");
    ecrivain.chaine = "et pour finir "; (A)
    ecrivain.ecrire("au revoir");
  }
}
```

Accéder aux données et aux méthodes

- De l'extérieur de sa classe ou d'une classe héritée, un attribut ou une méthode de classe pourront être utilisés précédés du nom de sa classe :
  - **nom_d'une_classe.nom_de_l'attribut**
  - **nom_d'une_classe.nom_de_méthode_de_classe**

Données et méthodes statiques

- Une méthode ou un attribut auxquels n'est pas appliqué le modificateur **static** sont dits d'instance.
- Un attribut déclaré **static** existe dès que sa classe est chargée, en dehors et indépendamment de toute instanciation.
- Une méthode, pour être de classe, ne doit pas manipuler, directement ou indirectement, d'attributs non statiques de sa classe.
  - En conséquence, une **méthode de classe ne peut utiliser** dans son code
    - aucun attribut non statique
    - ni aucune méthode non statique de sa classe ;
    - une erreur serait détectée à la compilation.

Données et méthodes statiques : exemple

- une classe, UnPetitEcrivain, qui possède un attribut et une méthode de classe
- une classe qui utilise UnPetitEcrivain.

```
class UnPetitEcrivain {
 static String identificateur = "Sophie";
 static void ecrire(String chaine)
{ System.out.println(identificateur + " vous dit " + chaine); }
}
public class Statique
{
public static void main(String [  ] arg)

{ System.out.println("L'ecrivain est " +
                                        UnPetitEcrivain.identificateur);
     UnPetitEcrivain.ecrire("bonjour");
}
}
```

On obtient à l'exécution ?

Utiliser la référence this

`This` représente, au moment de l'exécution, **l'instance de la classe** sur laquelle le code est entrain de s'appliquer

```
System.out.println(this) ;
```

```
class TresSimple
{
int n;
TresSimple (int v)
{
        n = v;
}

}
```

```
class TresSimpleBis
{
int n;
TresSimple (int n)
{
        this.n = n;
}

}
```

Attribut

Variable locale de la méthode

this.n désigne l'attribut n !

Cette classe est équivalente à celle de gauche

Utiliser une classe possédant un constructeur explicite

```
class Incremente
{
  int increment;
  int petitPlus;

  Incremente(int increment, int petit)
  {
    this.increment = increment;
    petitPlus = petit;
  }

  int additionne(int n)
  {
     return (n + increment + petitPlus);
  }
}

class Constructeur
{
  public static void main(String[] arg)
  {
    Incremente monAjout = new Incremente(10, 1);
    System.out.println(monAjout.additionne(5));
  }
}
```

- **this.increment représente l'attribut increment de la classe.**

- **L'instruction :**
  this.increment = increment initialise l'attribut increment de la classe avec la valeur donnée en argument du constructeur. Le même problème ne se pose pas avec l'attribut petitPlus et l'argument petit qui ont des noms différents.

- **On obtient à l'exécution ?**

Les constantes

- Un attribut constant peut être défini par le modificateur **final**.
- Un attribut déclaré final ne peut être affecté qu'une seule fois, soit au moment de sa définition, soit dans le constructeur de la classe.
- Ainsi :
  - **final int** MAX=2;
    déclare la constante MAX qui vaut 2. C'est néanmoins une constante d'instance car, si on initialise l'attribut dans le constructeur, l'initilisation peut être différente selon l'instance .
  - Si on déclare l'attribut par : **final static int** MAX=2;
    alors c'est un attribut de classe, donc indépendant de l'instance de la classe. C'est alors une ***constante de classe***.

Surcharge de méthodes

- Une classe peut définir **plusieurs méthodes de même nom** mais **différent par** :
  - **les types d'arguments différents**
  - **l'ordre des types de leurs d'arguments**
- La bonne méthode est choisie pour qu'il y ait correspondance sur les paramètres de l'appel et les arguments de la méthode. Une différence sur le type de la valeur de retour ne suffirait pas.
- Ex : La classe Surcharge possède trois méthodes portant le **nom *opération*** mais avec un jeu d'arguments différents, globalement ou par leur ordre : on utilise en cela la surcharge.

Utilisation de la Surcharge

```
class Surcharge {
int n =1;
double x = 3.5;
Surcharge( ) { }
Surcharge(int n, double x) {
 this.n=n;
 this.x=x;
}
 int operation(int p) {
return 10*p+n; }
double operation(double y, int p) { return x*p+y; }
double operation(int p, double y) { return
      (double)n/p+y;
   }
}
```

```
class EssaiSurcharge {
 public static void main(String[ ] argv) {
int n;
double x;
surcharge = new Surcharge( );
n = surcharge.operation (2);
n = surcharge.operation (1.5, 4);
n = surcharge.operation (4, 1.5);
surcharge = new Surcharge (7, 2.0);
    }
 }
```

On obtient à l'exécution ?

15.5

1.75

27

Chapitre 3
Héritage, instruction **`import`**, tableaux

Sommaire

- L'héritage
  - Etendre une classe
  - Chaînage des constructeurs
  - Redéfinir une méthode
  - Conversion de classe
  - Les classes abstraites
- L'instruction **`import`**
- Manipulation d'un tableau d'objets
  - Définir un tableau
  - Longueur d'un tableau et dépassement des bornes
  - Tableaux à deux dimensions

L'héritage (1)

- Principe fondamental des langages objets
- Toute classe, sauf la classe **java.lang.Object,** hérite d'une seule classe : sa superclasse
- Une classe (autre que la classe **`Object`**) qui n'indique pas sa superclasse hérite automatiquement de la classe **`Object`**

L'héritage (2)

```
class B  extends A
{  ...  }
```

- On dit :
  - A est la superclasse de B
  - B hérite de A
  - B est une sous-classe de A
  - B étend A

- B dispose de toutes les attributs et méthodes (non-privés) de A; s'y ajoutent les attributs et les méthodes qui lui sont propres

L'héritage (3)

```
class Point
 // class Point extends Object
  { … }
```

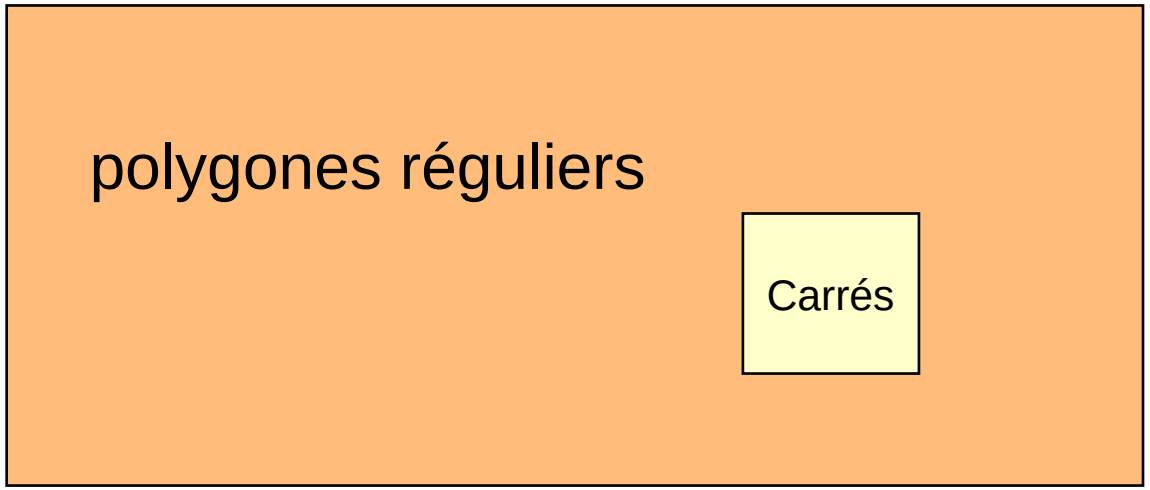
- Une classe ne peut étendre qu'une seule classe : pas d' "héritage multiple"

- Une classe déclarée **`final`** ne peut pas être étendue

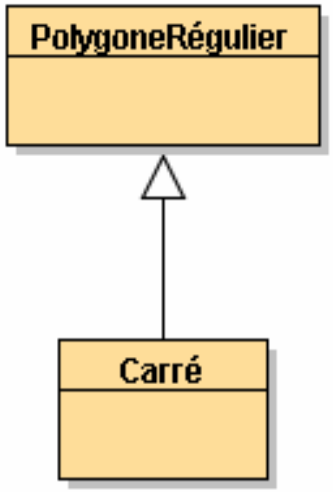
```
final class A
{ … }
```

Exemple de 2 classes avec héritage

- définir une nouvelle classe en ajoutant de nouvelles fonctionnalités à une classe existante
  - ajout de nouvelles données
  - ajout de nouvelles méthodes
- **Les carrés** sont **des polygones réguliers (ce serait l 'idéal...)**

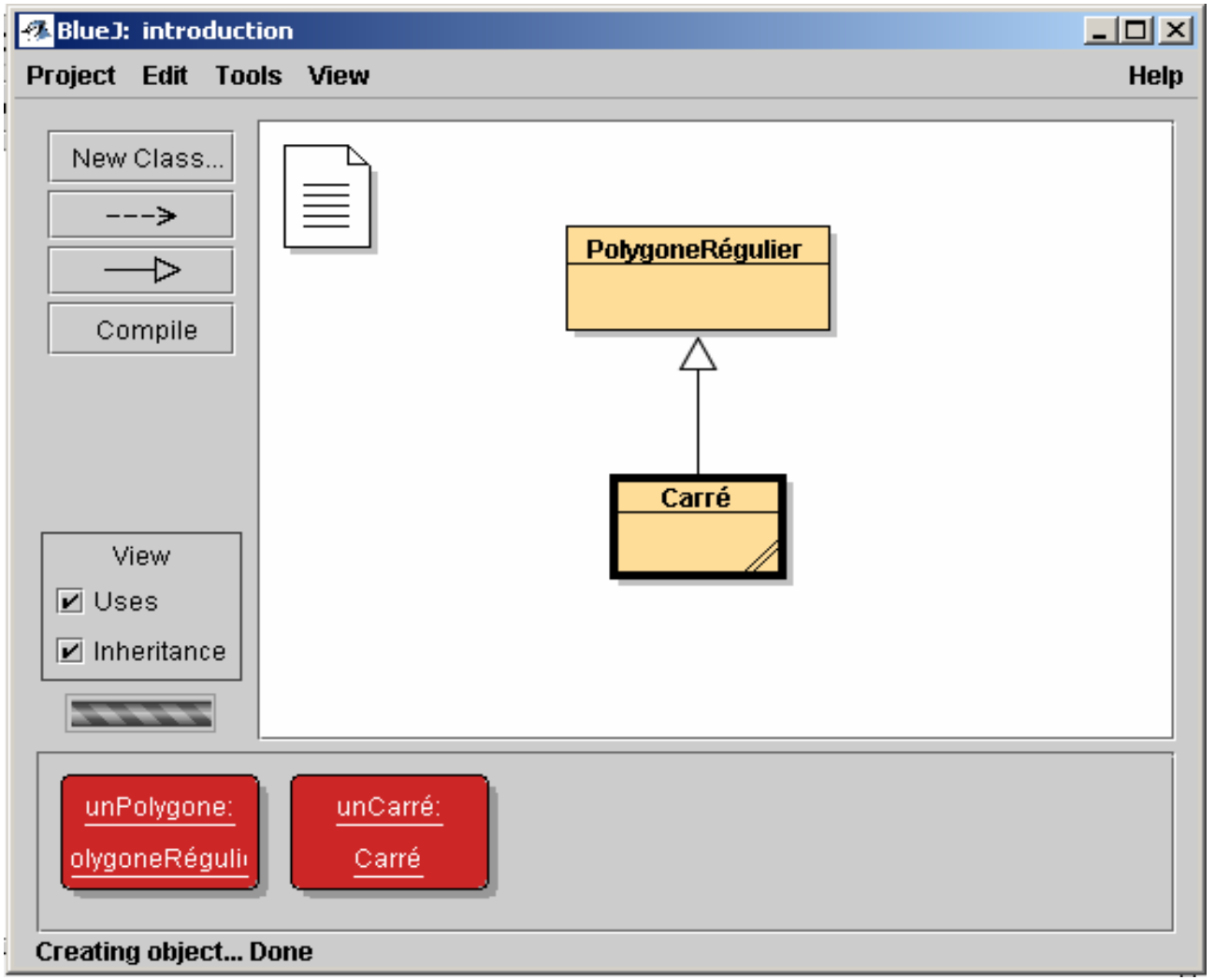


Héritage (2)

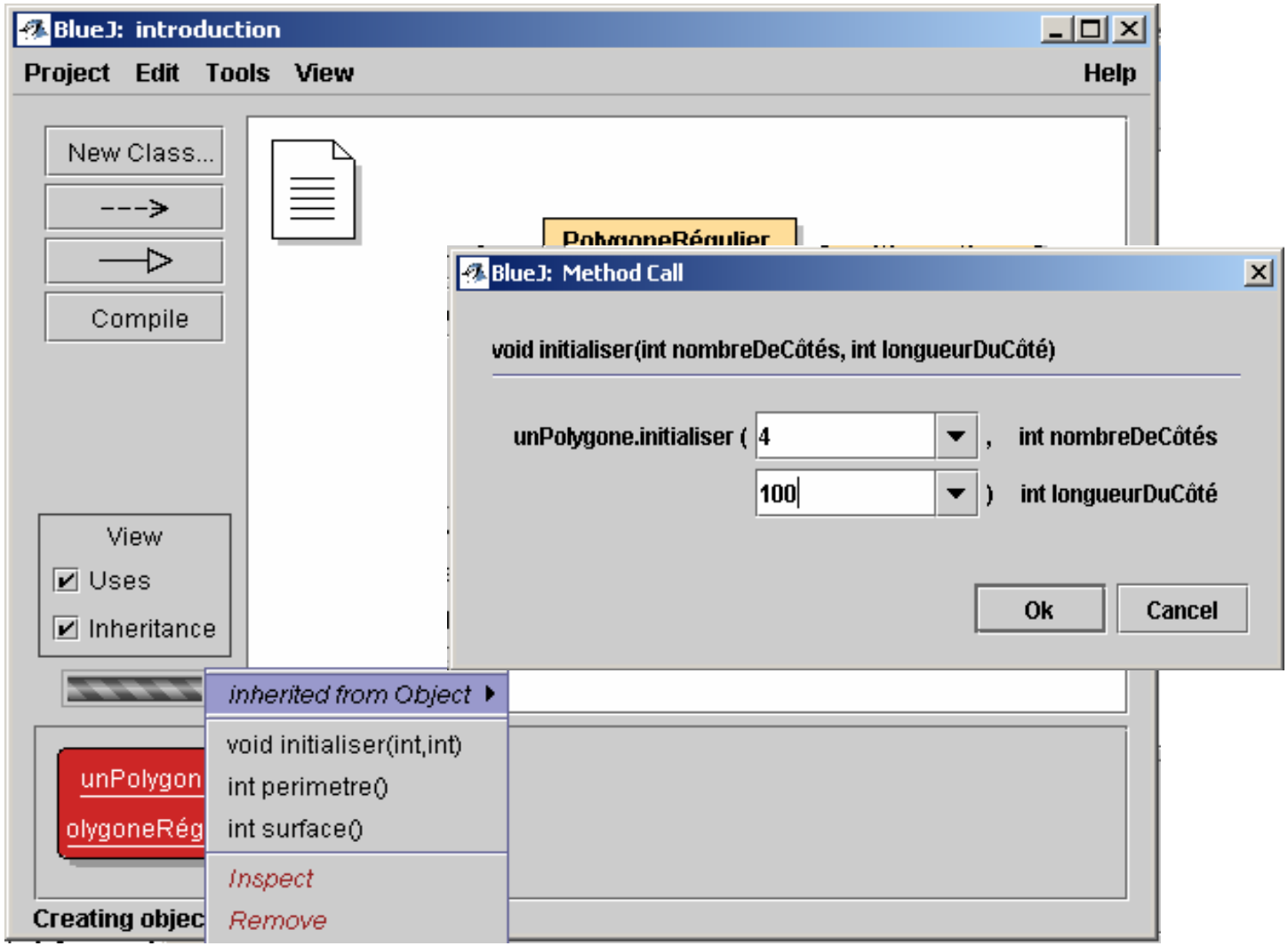



```
public class PolygoneRégulier
{
}
public class Carré extends PolygoneRégulier
{
}
```

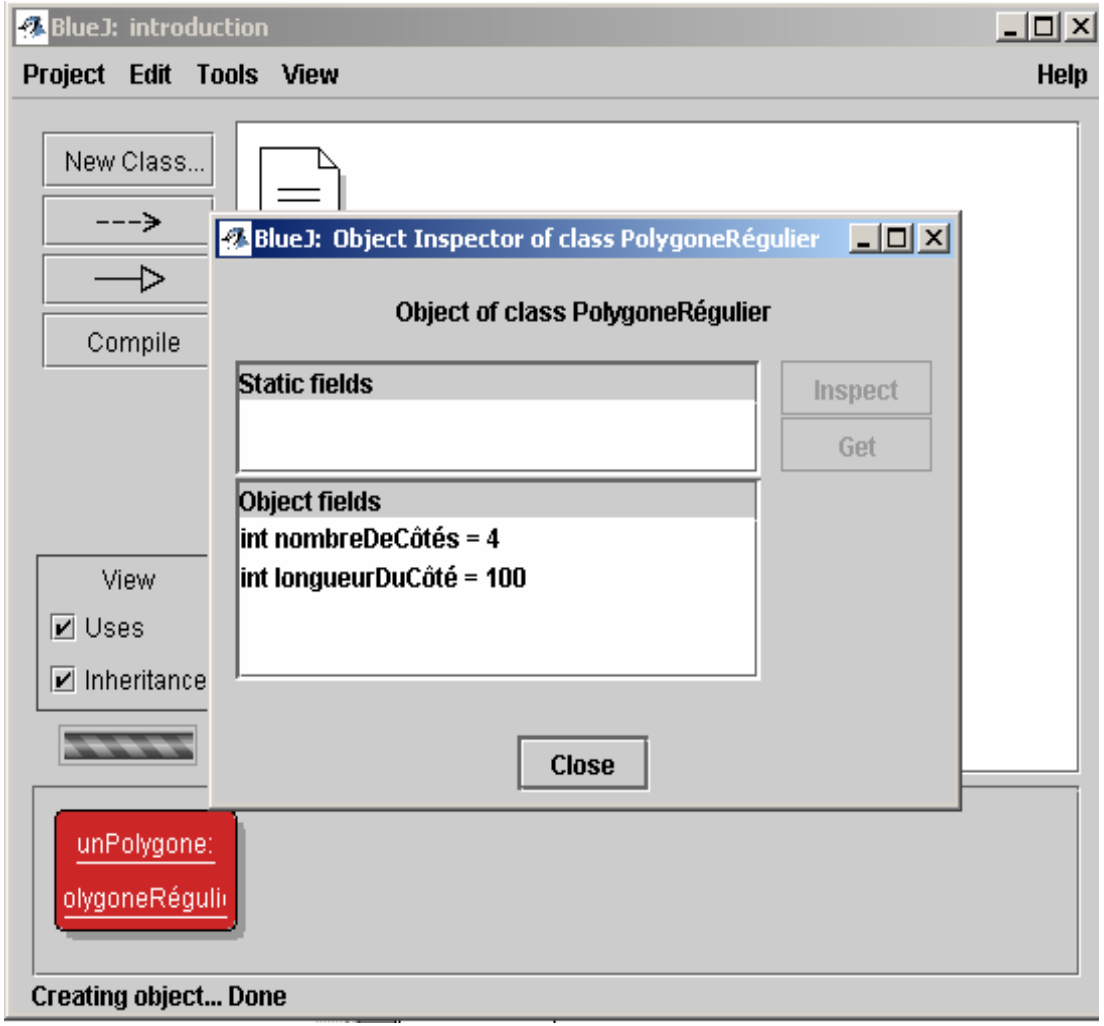
Avec BlueJ : Classes, instances



Avec BlueJ, unPolygone.initialiser(4,100)

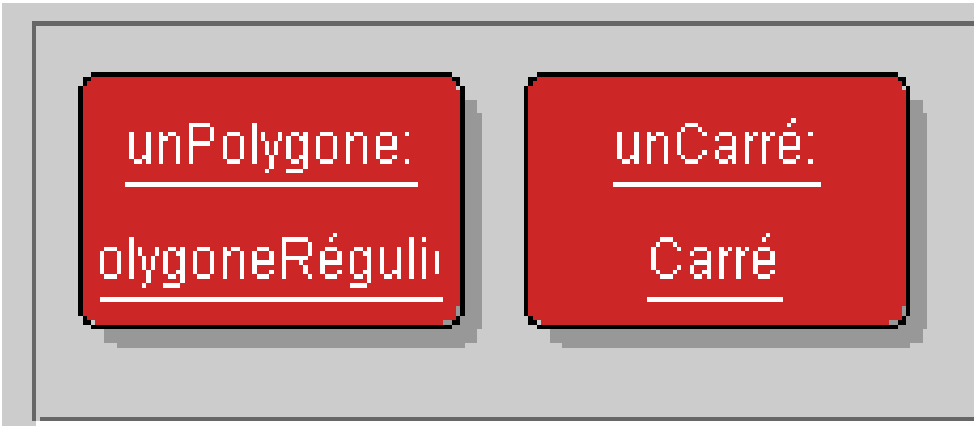


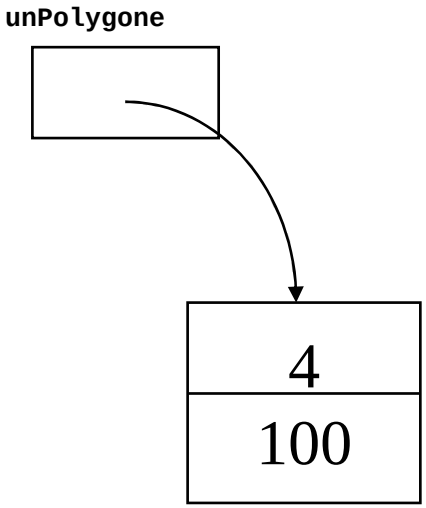
Objet, instance : unPolygone



Objet et référence

- `PolygoneRégulier unPolygone = new PolygoneRégulier();`
- `unPolygone.initialiser(4,100);`





Classe PolygoneRégulier

```
public class PolygoneRégulier
{
   int longueurDuCôté;
   int nombreDeCôtés;
  void initialiser(int nombre, int longueur)
   {
    longueurDuCôté = longueur;
    nombreDeCôtés = nombre;
  }
 int périmètre()
   {
   return longueurDuCôté * nombreDeCôtés ;
  }
 public int surface(){ …..}
}
```

```
// un usage de cette classe
PolygoneRégulier unPolygone = new PolygoneRégulier();
unPolygone.initialiser(4,100);
int y = unPolygone.périmètre();
```

La classe Mammifere

```
class Mammifere {
   int taille;
   Mammifere(int taille) {
        this.taille = taille;
   }
   String description() {
   return "Animal à sang chaud mesurant " + taille + " cm.";
   }
```

La classe Herbivore

```
class Herbivore extends Mammifere {
    boolean ruminant;

    Herbivore(int taille, boolean ruminant) {
        super(taille) ;
        this.ruminant = ruminant;
    }
    String nourriture( ) {
        if (ruminant) return "Il mange des vegetaux et rumine.";
        else return "Il mange des vegetaux et ne rumine pas.";
    }
    public static void main(String[ ] arg) {
        Herbivore herbivore = new Herbivore(200, true);
        System.out.println(herbivore.description( ));
        System.out.println(herbivore.nourriture( ));
    }
}
```

Animal à sang chaud mesurant 200 cm

Il mange des vegetaux et rumine

Constructeur

- Méthode qui sert à « construire » les objets
- Automatiquement appelé quand on instancie une classe
- Toute classe possède au moins un constructeur.
- Même nom que la classe
- Si pas de constructeur, le compilateur en ajoute automatiquement un.

```
class MaClasse
{
  MaClasse( )
  {
     super ( );
  }

     …
}
```

Chaînage des constructeurs

- Tout constructeur, sauf celui de la classe Object, fait appel à un autre constructeur qui est :
  - Un constructeur de la superclasse;
  - Un autre constructeur de la même classe;

- Appel : nécessairement en première ligne du constructeur
- En cas d'absence de cet appel, le compilateur ajoute **super** ( );

Exemple sur les constructeurs

```
class  A {
    A( ) {
        System.out.println("constructeur de A");  }
}
class  B extends A {
    B( ) {
super ( );          System.out.println("constructeur de B");
    }
    B(int r) {
        this( );    // appel au constructeur ci-dessus
        System.out.println("autre constructeur de B"); }
}
class  C extends B {
    C( ) {
        super(3);           // appel au constructeur ayant un paramètre entier de la classe B
        System.out.println("constructeur de C");
}
    public static void main(String[ ] arg) {
        new C( ); }
}
```

```
constructeur de A
constructeur de B
autre constructeur de B
constructeur de C
```

Exercice

```
class A {
        int i;
        A(int i) {
                this.i = i;
        }
}

class B extends A {}
```

Message du compilateur ?

Redéfinition d'une méthode d'instance

- On suppose qu'une classe *B* étend une classe *A*
- Redéfinir dans une classe *B une méthode de la classe A :*
  - définir dans B une méthode ayant
    - même nom,
    - mêmes types et nombre d'arguments
    - et même type de retour qu'une méthode contenue dans la classe A.
- L'interpréteur cherche la définition d'une méthode invoquée :
  - à partir de la classe de l'instance concernée,
  - en retenant la première définition rencontrée correspondant à la liste des paramètres d'appel,
  - en remontant de classe en super-classe.

Quelle méthode ?

```
class A {
        void faire( ) {
                System.out.println("niveau a");
        }
}

class B extends A {
        void faire( ) {
                System.out.println("niveau b");
        }
}

class C extends B {
        public static void main(String[ ] arg) {
                A a ;
                a = new A( );
                a.faire();
                a = new B( );
                a.faire();      // 1 → ?
                a = new C( );
                a.faire( );     // 2 → ?
        }
}
```

niveau a

niveau b

niveau b

On obtient à l'exécution ?

l'objet référencé par a est construit comme une instance de B, c'est la méthode faire ( ) redéfinie dans la classe B qui est exécutée

l'objet référencé par a est construit comme une instance de C. La classe C hérite de la méthode redéfinie dans la classe B, qui est exécutée

`final` pour une méthode

- Pour une méthode, **final** indique que la méthode ne peut pas être redéfinie (par les sous-classes de A)

```
Class A
{
    …
    final void meth ( )
{
    …
}

}
```

Pour une classe, **final** indique que la classe ne peut pas être étendue.

Pour une donnée, **final** indique qu'il s'agit d'une constante.

Que signifie : polymorphisme ?

```
class Orateur {
     String action() {
          return "s'exprime";
     }
}

class Grenouille extends Orateur {
     String action() {
          return "coasse";
     }
}

class Fourmi extends Orateur {
     String action() {
          return "croonde";
     }
}

class EssaiOrateurs {
     public static void main(String[ ] arg) {
          Orateur orateur;

          orateur = new Orateur();
          System.out.println("Orateur " + orateur.action());
          orateur = new Grenouille();
          System.out.println("Grenouille "+ orateur.action());
          orateur = new Fourmi();
          System.out.println("Fourmi " + orateur.action());
     }
}
```

Des objets référencés par une même variable peuvent avoir des comportement différents

On obtient à l'exécution ?

Instance de quelle classe ?

```
class A {}
class B extends A {}
class EssaiInstanceof {
        public static void main(String[ ] arg) {
                Object obj = new B( );
                System.out.println("obj est une instance de B  : "
                        + (obj instanceof B));
                System.out.println(" obj est une instance de B  : "
                        + (obj instanceof A));

                System.out.println("obj est une instance de Object  : "
                        + (obj instanceof Object));
                System.out.println("obj est une instance de Integer  : "
                        + (obj instanceof Integer));
        }
}
```

instanceof

On obtient à l'exécution ?

```
obj est une instance de B  : true
obj est une instance de B  :true
obj est une instance de Object  :true
obj est une instance de Integer  : false
```

Une instance de classe est aussi une instance de toutes les classes dont elle hérite.

Conversion de types

- La conversion de type (« ***cast*** »)
  - Pour convertir un type en un autre on respecte la syntaxe suivante :

    (type de convervion) type_a_convertir

  - Attention la conversion est uniquement possible si le type à convertir est lié au type de conversion.

Conversion de classe ou transtypage

- Une classe Chien étend la classe Animal.

  (**class** Chien **extends** Animal)

  - Animal animal = **new** Chien( ); **permis?**

  - Animal animal;

    …

    Chien chien = animal; **permis?**

    Tout objet de la classe Chien est aussi une instance de la classe Animal mais le contraire n'est pas vrai.

  - Animal animal;

    …

    Chien chien = (Chien) animal **permis?**

    Si on tente de référencer un Animal qui n'est pas Chien par une référence de Chien, une **java.lang.ClassCastException** est lancée à l'exécution.

Conversion de classe

```
class Animal
{ }
class Chien extends Animal
{
  int taille = 80;
}

class Conversion
{
  public static void main(String argv[])
    {
      Animal animal;
      Chien chien = new Chien();
      animal = chien;
      chien = (Chien)animal;      //conversion de classe
      System.out.println("un chien mesure : " +chien.taille+" cm");
      animal = new Animal();
      try
      {
        chien = (Chien)animal;    //conversion de classe
      }
      catch (ClassCastException exc)
      {
        System.out.println(exc +     ", tout Animal n'est pas un Chien");
      }
    }
```

```
/*A l'execution on obtient :
un chien mesure : 80 cm
java.lang.ClassCastException : Animal, tout Animal
n'est pas un Chien */
```

`Super` comme une référence pour appeler une méthode redéfinie (1)

- Considérons une classe A étend une classe B
- B redéfinit une méthode *meth()* définit dans A
- Supposons que dans B nous voulons utiliser une méthode *meth()* définit dans A
  - La solution est de faire appel à

    super. meth()

Super comme une référence pour appeler une méthode redéfinie (2)

De l'intérieur d'une classe redéfinissant une méthode, on peut avoir accès à la méthode redéfinie de la superclasse, en utilisant **super.**

```
class A {
    void faire() {
      System.out.println("fait de A");
    }
}
class B extends A{
    void faire() {
      System.out.println("fait de B");
      super.faire();
    }
     void pourquoiPas() {
      super.faire();
      faire();
    }

    public static void main(String[] arg) {
      (new B()).pourquoiPas();
    }
}
```

va renvoyer sur la superclasse

On obtient à l'exécution ?

Méthode abstraite

- Une méthode abstraite n'a que son prototype, c'est-à-dire
  - son type de retour
  - suivi, de son nom,
  - suivi de la liste de ses paramètres entre des parenthèses,
  - suivi d'un point-virgule.
- *Doit alors être déclarée* **abstract**

  abstract float perimetre ( );
- Une méthode abstraite ne peut pas être déclarée **static** ou **private** ou **final**.

Classe abstraite (1)

- Classe qui contient une méthode abstraite

  → **est une classe abstraite**

  Doit être déclarée **abstract**

```
abstract class Forme {
…
}
```

Classe abstraite (2)

- Une classe abstraite ne peut pas être instanciée : il faudra évidemment **l'étendre pour pouvoir l'utiliser**

```
class Ellipse extends Forme { …
}
```

- Une sous-classe d'une classe abstraite sera encore abstraite si elle ne définit pas toutes les méthodes abstraites dont elle hérite.

Classe abstraite (3)

- **Avantage :**
  - Les classes abstraites sont très **utiles pour définir des méthodes dépendant d'autres méthodes qui ne sont pas précisées (pouvoir travailler avec des méthodes déclarées mais non définies, indépendamment du corps de ces méthodes).**

Classe abstraite – exemple - 1

```
abstract class Forme
{
    abstract float perimetre( ); //méthode abstraite
    abstract float surface( ); //méthode abstraite
    void decritEtalement( )   //méthode non abstraite
    {
        float lePerimetre = perimetre();
        if (surface(  )>=lePerimetre * lePerimetre /16)
        System.out.println(this + "s'etale au moins comme un carre");
        else System.out.println(this + " s'etale moins qu'un carre");
    }
```

true si à périmètre fixé, la forme a une surface au moins égale à celle d'un carré

Classe abstraite – exemple - 2

```
class Ellipse extends Forme
{
  private int grandAxe, petitAxe;

  Ellipse(int grandAxe, int petitAxe)
  {
    this.grandAxe=grandAxe;
    this.petitAxe=petitAxe;
  }

  float perimetre() // calcul approché
  {
    ...
  }

  float surface()
  {
    return (float)Math.PI*petitAxe*grandAxe;
  }

  public String toString()
  {
    return ("L'ellipse de grand axe "+grandAxe+
        ", de petit axe "+petitAxe);
  }
```

- ✓ **La classe Ellipse étend la classe Forme et définit les deux méthodes abstraites de celle-ci.**
- ✓ **Elle redéfinit aussi la méthode toString de java.lang.Object.**

Classe abstraite – exemple - 3

```
class Rectangle extends Forme {
   private int longueur, largeur;
   Rectangle(int longueur, int largeur) {
   this.longueur=longueur;
   this.largeur=largeur;
   }
   float perimetre( )
   {
   return 2*(longueur+largeur);
   }
   float surface( ) {
   return longueur*largeur;
   }
   public String toString( )
   { return ("Le rectangle de longueur "+longueur+ " et de largeur
   "+largeur);
   }
}
```

- ✓ **La classe Rectangle étend la classe Forme et définit les deux méthodes abstraites de celle-ci. Elle redéfinit aussi la méthode toString de java.lang.Object.**
- ✓ **Remarque : il existe une autre classe Rectangle définie dans java.awt**

Classe abstraite – exemple - 4

```
class EssaiFormes
{
  public static void main(String[] argv)
  {
    Forme forme;

    (new Ellipse(2,1)).decritEtalement();
    (new Ellipse(4,1)).decritEtalement();
    (new Rectangle(2,1)).decritEtalement();
  }
}
```

- Une instance de la classe Ellipse ou une instance de la classe Rectangle, héritant de la classe Forme,dispose de la méthode decritEtalement().

On obtient à l'exécution :

```
L'ellipse de grand axe 2, de petit axe 1 s'etale au moins comme un carre
L'ellipse de grand axe 4 et de petit axe 1 s'etale moins qu'un carre
Le rectangle de longueur 2 et de largeur 1 s'etale moins qu'un carre
```

Sans classe abstraites

- Sans classe abstraites ⟶
  - remplacer les méthodes abstraites par des méthodes ayant un corps vide et `return`.

Pourquoi serait-il moins bien de définir comme ci-dessous la classe forme ? (1)

```
class Forme {
    double perimetre() {
    return 0;
    }

    double surface(){
    return 0;
    }

    void decritEtalement() {
    double lePerimetre = perimetre();

    if (surface() >= lePerimetre*lePerimetre/16)
        System.out.println(this + " s'etale plus qu'un carre");
    else System.out.println(this + " s'etale moins qu'un carre");
    }
}
```

Pourquoi serait-il moins bien de définir comme ci-dessous la classe forme ? (2)

- rien n'empêche instanciation de cette classe forme qui ne présente pas vraiment une forme
  - si une classe est de nature abstraite, c'est-à-dire n'est pas prévu pour être instanciée → utiliser une classe déclarée ***abstract***
  - par ailleurs on ne peut pas remplacer les classes abstraites par des interfaces, car certaines méthodes peuvent être définies et ce n'est pas le cas des interfaces

L'instruction `importe(1)`

- ***import*** : cette instruction permet dans un fichier source un raccourci sur des noms complets de classe d'un paquetage.

  - Classe dans un paquetage : nom « complet **»**

Exemple : classe `Vector` dans le paquetage `java.util;`

`nom long : java.util.Vector`

  - L'instruction **`import :`** permet d'utiliser les noms courts.
  - Elle n' « importe » rien physiquement.

L'instruction `importe(2)`

- On peut employer l'instruction **`import`** sous deux formes :
  - Premier exemple

```
import coursJava/monPaquet/DansPaquetage;
```

  - permet d'utiliser la classe DansPaquetage sans rappeler le nom complet.
    - Autrement dit, lorsque l'on aurait écrit sans l'instruction import :

```
coursJava.monPaquet.DansPaquetage UneInstance= new
  coursJava.monPaquet.DansPaquetage( );
```

    - pourra être écrit avec l'instruction import :

```
DansPaquetage UneInstance = new DansPaquetage( );
```

L'instruction `importe(3)`

- Deuxième exemple :

  import coursJava / monPaquet / * ;

  - Permet d'utiliser le nom de toute classe faisant partie du paquetage coursJava/monPaquet sans rappeler le nom complet.

L'instruction *importe* - exemple

```
import java.util.Random;
import java.util.Vector;

class TableauAleatoire {
    static void emplit(int[] tableau)
{
    Random alea = new Random
    (System.currentTimeMillis());
       Vector vecteur = new Vector() ;

          for (int i = 0; i < 10; i++)
    vecteur.addElement(new Integer
          (Math.abs(alea.nextInt()) % 100));
    }
}
```

Définition d'un tableau

char[ ] tableau;
déclare le tableau tableau
// en C : char tableau[ ];

Par ailleurs, tableau étant déjà défini :
tableau = new char[2];
alloue un tableau pour deux variables de type char.

Exemple:

```
class TableauA
{
  public static void main(String[] argv)
  {
    char[ ] tableau;
    tableau = new char[2];
    tableau[0] = 'h';
    tableau[1] = 'a';
    System.out.println(tableau);
  }
}
```

On obtient à l'exécution ?

Une autre façon de définir un tableau

une façon de déclarer et d'allouer un tableau pour trois variables de type boolean

```
class TableauB
{
  public static void main(String[] argv)
  {
    boolean tableau[]={true,false,true};
    System.out.println("Deuxieme element de tableau  : "
                       + tableau[1]);
  }
}
```

On obtient à l'exécution ?

Longueur d'un tableau

- On peut connaître la longueur d'un tableau référencé par la variable *tableau* en utilisant :
  - tableau.length;
  - length (longueur) est en quelque sorte un attribut du tableau, attribut que l'on peut uniquement lire.
- L'indice de la dernière case d'un tableau vaut sa longueur moins 1

Longueur d'un tableau et dépassement des bornes

```
class TableauC
{
  public static void main(String[] argv)
  {
    int tableau[] = new int[3];

    System.out.println("Taille du tableau :
                        + tableau.length);
    try
      {
 tableau[tableau.length] = 1;
      }
    catch(ArrayIndexOutOfBoundsException e)
      {
 System.out.println(e + ", bien intercepte!");
      }
  }
}
```

- ✓**tableau[tableau.length] : l'indice tableau.length déborde du tableau puisque le plus grand indice de ce tableau est nécessairement sa longueur mois 1.**
- ✓**En conséquence, une exception du type ArrayIndexOutOfBoundsException est levée.**
- ✓**Si celle-ci n'était pas attrapée, elle se propagerait jusqu'à la fin du programme qui se terminerait donc directement avec un message d'erreur.**
- ✓**Ici, le mécanisme utilisant les mots réservés *catch et try* permet d'intercepter l'exception.**

Un tableau à deux dimensions (1)

Deux méthodes sont proposées :

1. on fait l'allocation en une seule fois en précisant les deux dimensions. On obtient alors un tableau rectangulaire.
2. on alloue d'abord un tableau à une dimension de références vers des tableaux à une dimension (d'**int**);
   - on a ainsi alloue un tableau destiné à recevoir les références des lignes.
   - puis on alloue une par une les lignes du tableau. Les lignes ne doivent pas nécessairement avoir toutes la même longueur.

Un tableau à deux dimensions (2)

```
class TableauDouble
{
  public static void main(String[] argv)
  {
    int[][] tableau;

    tableau=new int[2][3];
    for (int i =0;i<tableau.length;i++)
      for (int j =0;j<tableau[i].length;j++
            tableau[i][j]=i+j;
    System.out.println(tableau[0][1]);

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

✓ **tableau=new int [2][ ] : on réattribue un nouvel espace mémoire au tableau ; un ramasse-miettes va libérer l'espace - mémoire alloué au précédent tableau. Le nouveau tableau défini est destiné à recevoir deux références de tableaux d'entiers.**

✓ **tableau[i]=new int [i+2] : le nouveau tableau à deux dimensions ne sera pas rectangulaire.**

Chapitre 4
Interfaces, niveaux de visibilité, exceptions

Sommaire

- Les interfaces
  - Généralités
  - Exemples
- Utilisation de classes, packages
- Les niveaux de visibilité
- Les exceptions
  - Définir sa propre exception
  - Exemples

Les interfaces – généralités (1)

- "Parallèle" aux classes
- **Contient**
  - **des définitions de constantes**
  - **des déclarations de méthodes (prototypes)**
- Une interface correspond à une classe où toutes les méthodes sont abstraites.

Les interfaces généralités (2)

- **`implements`**
  Ce mot est employé dans l'en-tête de la déclaration d'une classe, suivi d'un nom d'interface ou de plusieurs noms d'interfaces séparés par des virgules.

- **S'il y a une clause `extends`, la clause `implements` doit se trouver après la clause `extends`.**

Exemple :

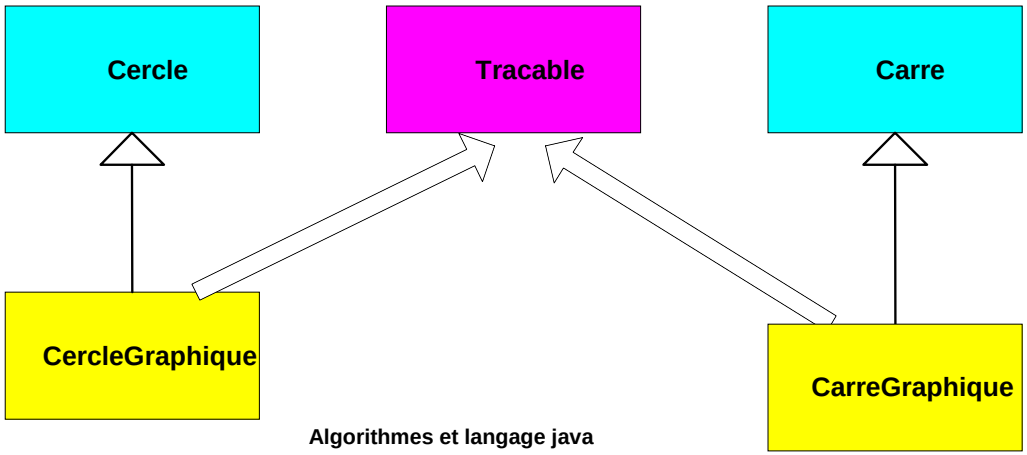
```
class MaClasse extends AutreClasse implements
UneInterface, UneAutreInterface
```

Les interfaces – généralités (3)

- En standard, Java dispose d'un très grand nombre d'interfaces prêtes à l'usage, réparties dans les différents packages:
  - Exemple java.io donne accès à DataInput, ObjectInput et Serializable

Interface sert :

- à regrouper des constantes
- mais surtout sert
  - chaque fois que l'on veut traiter des objets qui ne sont pas nécessairement tous de la même classe
  - qui ont en commun une partie "interface"
  - et que seule cette partie "interface" intervient dans le traitement que l'on veut définir.
- Avantage : lorsqu'on utilise un objet d'une classe implémentant une certaine interface, on a la garantie que cet objet dispose :
  - de toutes les constantes
  - de toutes les méthodes annoncées par l'interface



Règles (1)

- **Les attributs et les méthodes d'une interface sont automatiquement publics**

- **Si une classe A implémente une interface `I`, les sous-classes de A implémentent aussi automatiquement `I`.**

Règles (2)

- Une classe peut implémenter (`implements`) une ou plusieurs interfaces tout en héritant (`extends`) d'une classe.
  - ```
    class A implements I1, I2
    {...
     }
    ```
- Une interface peut hériter (`extends`) de plusieurs interfaces : permet de compenser l'absence d'héritage multiple.
  - ```
    interface I extends I1, I2
    {...
     }
    ```
  - L'interface `I` hérite
    - des constantes définies dans `I1` et `I2`
    - et contient par héritage les méthodes déclarées dans `I1` et `I2`

Interface – Exemple composé -1

```
class Cercle
{
  int rayon;

  Cercle(int rayon)
  {
    this.rayon=rayon;
  }

  double circonference()
  {
     return 2*Math.PI*rayon;
  }
}

class Carre
{
  int cote;

  Carre(int cote)
  {
    this.cote=cote;
  }

  int surface()
  {
     return cote*cote;
  }
```

Une classe pour définir un cercle

Une classe pour définir un carré

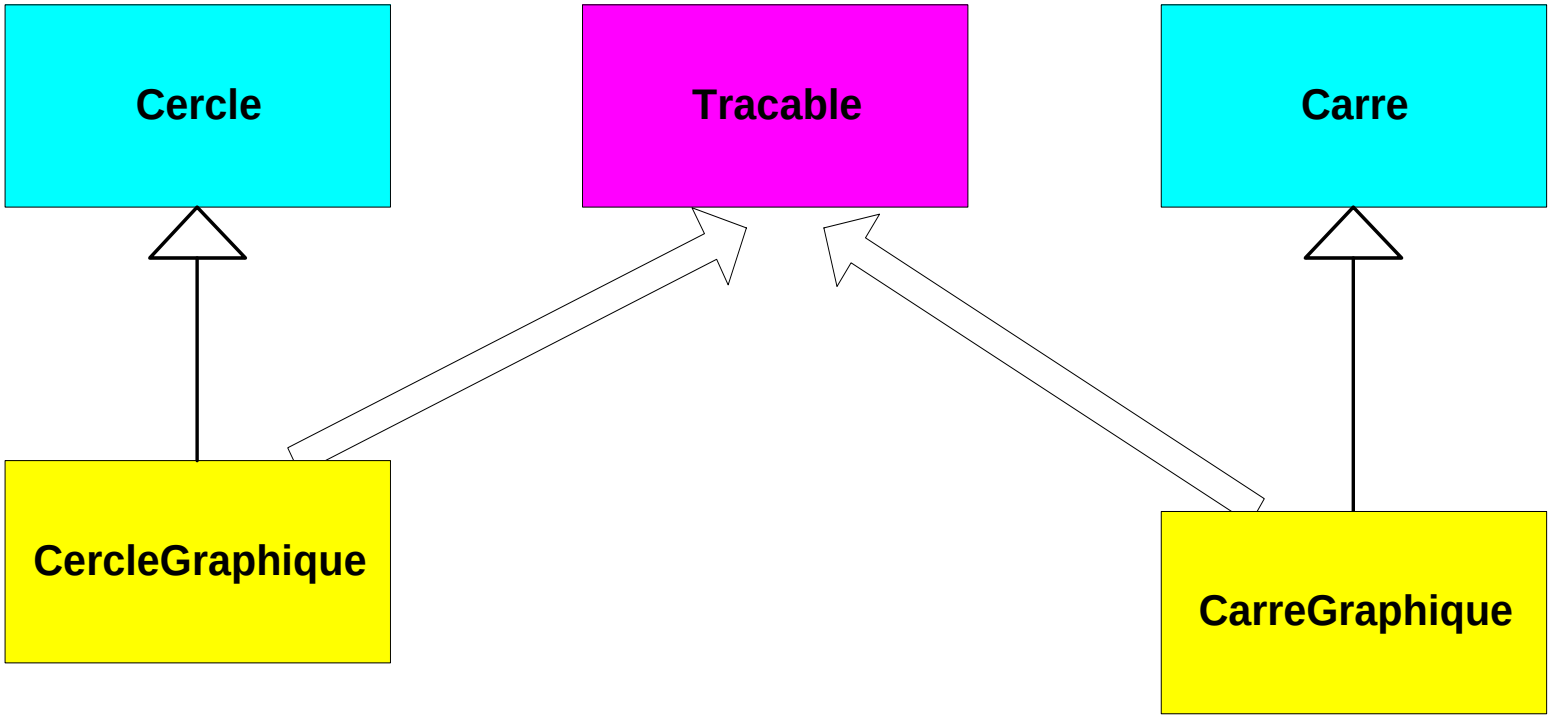
Interface – Exemple composé - 2

Une interface

```
interface Tracable
{
   // constente (s)
   public static final int DIM_MAX=500;

   // prototype(s) de methode(s)
   public void dessine(java.awt.Graphics g);
}
```

- **java.awt.Graphics** : la classe java.awt.Graphics définit, un ensemble de fonctionnalités permettant de dessiner dans une fenêtre graphique.
- Pour dessiner dans une fenêtre, on passe nécessairement par une instance de Graphics.

Une classe CercleGraphique qui étend Cercle et implémente Tracable



Interface – Exemple composé - 3

Une classe CercleGraphique qui étend Cercle et implémente Tracable

```
class CercleGraphique extends Cercle implements Tracable
{
  int x,y;
  java.awt.Color couleur;

  CercleGraphique(int rayon, int x,
                  int y,java.awt.Color couleur)
  {
    super(rayon);
    this.x=x;
    this.y=y;
    this.couleur=couleur;
  }

  public void dessine(java.awt.Graphics g)
  {
    if (getRayon() < DIM_MAX)
      {
 g.setColor(couleur);
 g.drawOval(x-getRayon(),y-
getRayon(),2*getRayon(),2*getRayon());
      }
  }
```

✓**java.awt.Color : on utilise ici une classe de java.awt qui contient ce qui est nécessaire pour travailler avec des couleurs. On aurait pu aussi mettre en début de programme :**

import java.awt.Color;
ou bien : import java.awt.*;

✓**public void dessine(java.awt.Graphics g) : lorsqu'une classe implémente une interface, elle doit définir toutes les méthodes dont les prototypes sont annoncés dans l'interface, et leur attribuer le modificateur de visibilité public.**

✓**DIM_MAX : la constante DIM_MAX définie dans Tracable est connue dans CercleGraphique qui implémente Tracable.**

Interface – Exemple composé - 4

Dessiner un cercle

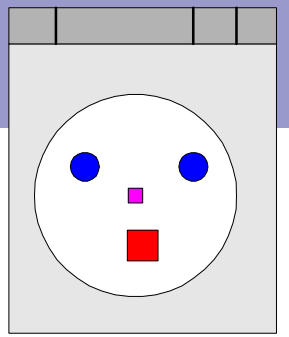
```
class EssaiCercleGraphique extends Frame
{
  CercleGraphique c=new CercleGraphique(50,100,100,Color.red);

  public void paint(Graphics g)
  {
    c.dessine(g);
  }

  public static void main(String[] argv)
  {
    EssaiCercleGraphique monDessin = new EssaiCercleGraphique();

    monDessin.setSize(200,200);
    monDessin.setVisible(true);
  }
}
```

- **Frame** : la classe java.awt.Frame représente une fenêtre principale d'application, pouvant avoir un titre ou une barre de menus, changer de taille, se mettre sous forme d'icône …
- **CercleGraphique(50,100,100,Color.red);** : on invoque le constructeur de `CercleGraphique`.
- **c.dessine(g);** : on utilise la méthode d'instance `dessine` de CercleGraphique.
- **monDessin.setSize(200,200);** : la fenêtre aura pour taille 200 pixels (largeur) sur 200 pixels (hauteur).
- **monDessin.setVisible(true);** : cette instruction est indispensable pour que la fenêtre soit visible.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JPanel;

class EssaiTracable extends JPanel
{
  Tracable[] desFormes = new Tracable[5];

  EssaiTracable()
  {
    setPreferredSize(new Dimension(200, 200));
    desFormes[0]=new CercleGraphique(150,200,200,Color.black);
    desFormes[1]=new CercleGraphique(20,130,150,Color.blue);
    desFormes[2]=new CercleGraphique(20,270,150,Color.blue);
    desFormes[3]=new CarreGraphique(20,200,220,Color.magenta);
    desFormes[4]=new CarreGraphique(30,200,300,Color.red);
  }

  public void paintComponent(Graphics g) // invoquée a chaque fois pour redessiner
  {
    for (int i=0;i<desFormes.length;i++) desFormes[i].dessine(g);
  }

  public static void main(String[] argv)
  {
    EssaiTracable monDessin = new  JFrame();
    monDessin.setContentPane(new EssaiTracable());
    monDessin.pack();                  // dimensionner la fenetre
    monDessin.setVisible(true);
  }
}
```

JPanel fournit un composant graphique pour effectuer des tracés.

JFrame modélise une fenêtre d'appli.



Utilisation de classes

En java :

- pas de # include,
- pas d'édition de liens.

Les classes sont chargées en mémoire pendant la compilation ou pendant l'exécution.

Les packages

- servent à faire des arborescences de classes
- permettent au compilateur et à la machine virtuelle java de localiser les classes utiles à l'exécution d'un programme
- permettent de nuancer les niveaux de visibilités

Les packages - exemple

Si la classe A contient une méthode main, celui-ci est lancé par
java –classpath ../; (DOS)

~aubonnet/classes/graph/2D/Circle.java

```
package graph.2D;
public class Circle()
{ ... }
```

~aubonnet/classes/graph/3D/Sphere.java

```
package graph.3D;
public class Sphere()
{ ... }
```

~aubonnet/classes/paintShop/MainClass.java

```
package paintShop;

import graph.2D.*;

public class MainClass()
{
  public static void main(String[] args) {
    graph.2D.Circle c1 = new graph.2D.Circle(50)
    Circle c2 = new Circle(70);
    graph.3D.Sphere s1 = new graph.3D.Sphere(100);
    Sphere s2 = new Sphere(40); // error : class paintShop.Sphere not found
}
```

Niveaux de visibilité (1)

Degré de visibilité	Mot réservé
publique	**`public`**
privé	**`private`**
protégé	**`protected`**
paquetage (par défaut)	

- `private int nbDonnees ;`
- `public void methode ( );`

Niveaux de visibilité (1)

- **Public**
  - s'applique à une classe, une interface ou à un champ (attribut ou méthode) d'une classe.
  - une classe ou une interface publique est visible de partout, y compris les autres paquetages.
  - si ce modificateur n'est pas appliqué à une classe ou une interface, celle-ci n'est visible que de l'intérieur de son paquetage.
  - un champ publique est visible de partout du moment que sa classe est visible.

Niveaux de visibilité (2)

- **Private**
  - s'applique à un champ (attribut ou méthode) d'une classe.
  - visible que depuis sa propre classe.
  - elle n'est visible nulle part ailleurs et en particulier pas dans les sous-classes.

Niveaux de visibilité (3)

- **Protected**
  - s'applique à un champ (attribut ou méthode) d'une classe A.
  - visible de partout dans le paquetage de A et *grosso modo* dans les classes héritant de A dans d'autres paquetages.

Exemple : à l'intérieur d'un même paquetage

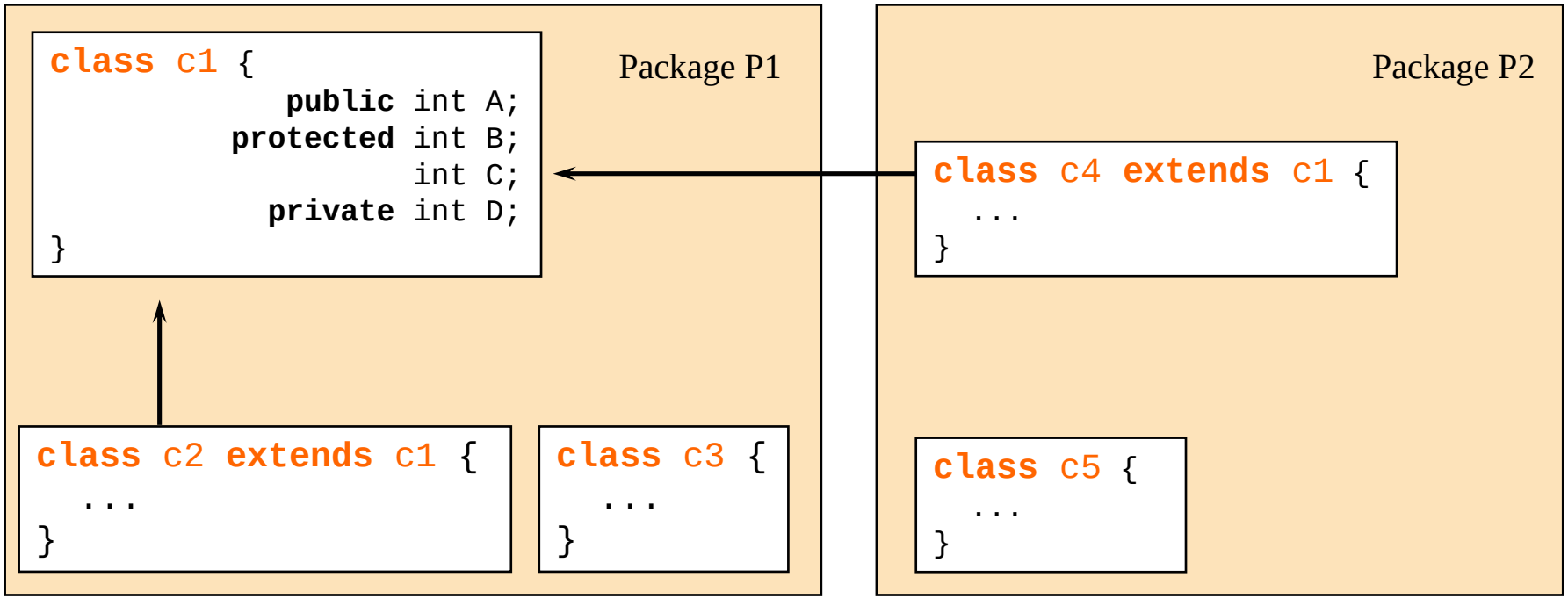
Dans un premier fichier, on a :

```
package coursJava.monPaquet;

public class EssaiPack
{
  public int publique;
  protected int protege;
  int defaut;
  private int prive;
}
```

Dans un second fichier, on peut trouver :

```
package coursJava.monPaquet;
import coursJava.monPaquet.*;

class Connaissance // meme paquetage que EssaiPack
{
  EssaiPack essaiPack = new EssaiPack();//instance de
EssaiPack
  int entier;

  Connaissance()
  {
    entier = essaiPack.publique;
    entier = essaiPack.protege;
    entier = essaiPack.defaut;
    entier = essaiPack.prive;       }
}

class Etend extends EssaiPack
          //classe etendant EssaiPack dans le meme
paquetage
{
  int entier;

  Etend()
  {
    entier = publique;
    entier = protege;
    entier = defaut;
    entier = prive;
  }
```

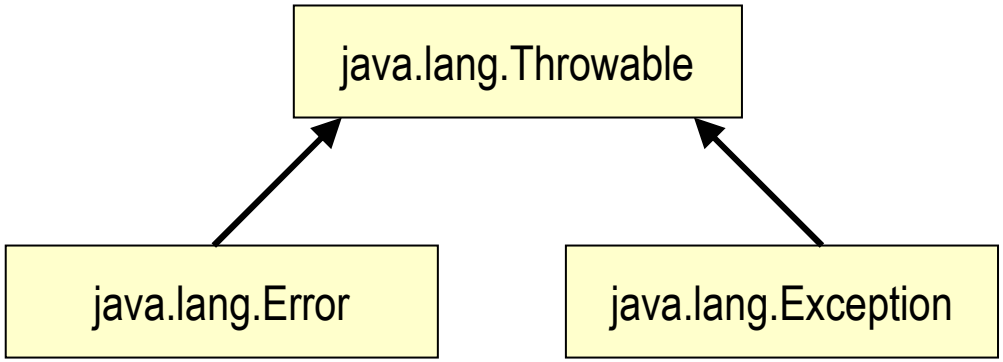
L'encapsulation des champs



	A	B	C	D
Accessible par c2	x	x	x	-
Accessible par c3	x	x	x	-
Accessible par c4	x	x	x	-
Accessible par c5	x	-	-	-

Les erreurs java

- Une « erreur » est un concept proche de celui de l'exception dans la terminologie « java ».
- Une erreur symbolise un problème survenu au sein de la machine virtuelle.
- Les erreurs héritent toutes de « java.lang.Error ».
- Une erreur est interceptée de la même façon qu'une exception.



Quelques exceptions et erreurs standard

- java.lang.ClassCastException
- java.lang.ArrayIndexOutOfBoundsException
  - indique un dépassement d'indice dans un tableau.
  - un tableau A de 4 éléments, on peut utiliser A[0], A[1], A[2] et A[3]. Mais en essayant d'utiliser A[4] on provoque une exception).

Les exceptions (1)

- Une exception correspond à un événement anormal ou inattendu.
- Les exceptions sont des instances de sous-classes
  - des classes **`java.lang.Error`** (pour des erreurs graves, qui devront généralement conduire à l'arrêt du programme)
  - et **`java.lang.Exception`** (pour des événements inattendus, qui seront souvent traités de sorte qu'elle ne provoque pas l'arrêt du programme).
- Un grand nombre d'exceptions sont définies et sont "lancées" par des méthodes de l'API.

Les exceptions (2)

- La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc `catch` acceptant cette exception soit trouvé.
- Si aucun bloc `catch` n'est trouvé, l'exception est capturée par l'interpréteur et le programme s'arrête.
- L'appel à une méthode pouvant lever une exception doit :
  - soit être contenu dans un bloc `try/catch`
  - soit être situé dans une méthode propageant (`throws`) cette classe d'exception

Les exceptions (3)

- Elles permettent de séparer un bloc d'instructions de la gestion des erreurs pouvant survenir dans ce bloc.

```
try {
   ...
instruction_lancant_exeption
   ...              //instructions sont ignorées
}
catch (IOException1 exc1) {
   // instructions effectuées
}
catch (Exception2 exc2){
   // Gestion de toutes les autres exceptions
}
```

Définir sa propre exception (1)

- Si on veut pouvoir signaler un événement exceptionnel d'un type non prévu par l'API, il faut étendre la classe java.lang.Exception ;

- La classe étendue ne contient en général pas d'autre champ qu'un (ou plusieurs) constructeur(s) et éventuellement une redéfinition de la méthode toString.

- Lors du lancement de l'exception, (à l'aide du mot réservé **throw**), on crée une instance de la classe définie.

Définir sa propre exception (3)

throw

- permet de "lancer" une exception lorsqu'un événement exceptionnel s'est produit.
- on lance une instance d'une exception (présence du mot **new).**
  - on écrira par exemple :

if (il_y_a_un_probleme) **throw new** MyException)();

où MyException serait ici une sous classe de java.lang.Exception définie par ailleurs.

Définir sa propre exception (4)

- une classe héritant de la classe Exception dans l'exemple suivant :

```
class ExceptionRien extends Exception
 {  public String toString( )
  {       return("Aucune note n'est valide");
  }
}
```

Lancer une exception - exemple

```
class ExceptionThrow
{ static int moyenne(String[] liste) throws ExceptionRien
  {
    int somme=0,entier, nbNotes=0;
    int i;

    for (i=0;i < liste.length;i++)
      {
    try
      {
        entier=Integer.parseInt(liste[i]);
        somme+=entier;
        nbNotes++;
      }
    catch (NumberFormatException e)
      {
        System.out.println("La "+(i+1)+" eme note n'est "+
                    "pas entiere");
      }
      }
    if (nbNotes==0) throw new ExceptionRien();
    return somme/nbNotes;
  }

  public static void main(String[] argv)
  {
    try
      {
    System.out.println("La moyenne est "+moyenne(argv));
      }
    catch (ExceptionRien e) ...

      {
    System.out.println(e);
      }
```

✓**throws ExceptionRien : la méthode moyenne indique ainsi qu'il est possible qu'une de ses instructions envoie une exception de type ExceptionRien sans que celle-ci soit attrapée par un mécanisme try-catch. Si vous oubliez de signaler par la clause throws l'éventualité d'un tel lancement d'exception, le compilateur vous le rappelera. throw new**

✓**throw new ExceptionRien() : on demande ainsi le lancement d'une instance de ExceptionRien. Une fois lancée, l'exception se propagera. Ici, il y aura sortie de la méthode moyenne, on se retrouve alors à l'appel de la la méthode moyenne dans le main, appel qui se trouve dans un "bloc try" suivi d'un "bloc catch" qui attrape les ExceptionRien(s) : l'instruction de ce "bloc catch" est effectuée. Le programme reprend alors son cours normal (pour se terminer).**

Bloc finally

- On appelle "bloc finally" la clause finally suivi d'un bloc d'instructions.
  - Un "bloc finally" est en général utilisé pour effectuer des nettoyages (fermer des fichiers, libérer des ressources...).
  - On doit toujours utiliser un "bloc finally" en association avec un "bloc try".

Bloc finally (2)

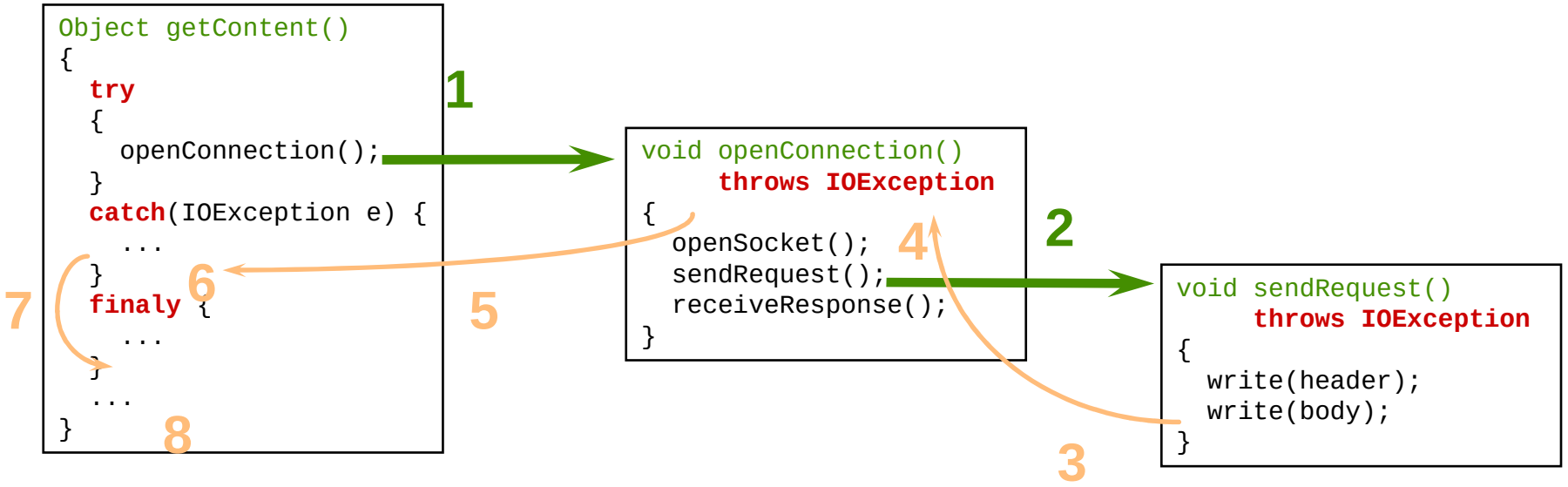
- Un "bloc **finally"** suit:
  - soit un "bloc **try**"
  - soit un "bloc **try**" suivi d'un "bloc **catch**"

- Si un "bloc catch" situé entre le "bloc try" et le "bloc finally" attrape une exception, les instructions du "bloc catch" sont exécutées avant celles du "bloc finally".

Bloc finally : exemple

```
class UtiliseFinally
{
  static int moyenne(String[] liste)
  {
    int somme=0,entier, nbNotes=0;
    int i=0;

    for (i=0;i < liste.length;i++)
      {
    try
      {
        entier=Integer.parseInt(liste[i]);
        somme+=entier;
        nbNotes++;
      }
    finally
      {
        System.out.println("donnee traitee : "+liste[i]);
      }
      }
    return somme/nbNotes;
  }

  public static void main(String[] argv)
  {
    try
      {
    System.out. println("La moyenne est "+moyenne(argv));
      }
    catch (NumberFormatException e)

      {
    System.out.println("Erreur sur vos entiers");
      }
  }
```

Les exceptions : exemple



Chapitre 5. Flux des données

Sommaire

- Les flux des données
  - saisir les données envoyées par le clavier
  - sérialisation

Saisir des données, utiliser des fichiers

- Pour traiter les flux de données héritant directement de la classe Object Il y a quatre classes "mères", abstraites :
  - 2 pour traiter les flux d'octets
    - la classe java.io.InputStream
    - la classe java.io.OutputStream
  - 2 pour traiter de caractères
    - la classe java.io.Reader
    - la classe java.io.Writer

Saisir des données envoyées par le clavier

Si vous voulez lire des données envoyées tout simplement par le clavier il existe deux méthodes

1. On utilise deux classes de java.io :
   - la classe **InputStreamReader**
   - et la classe BufferedReader.

2. On utilise **StreamTokenizer**

Première méthode

- On utilise deux classes de java.io :
  - la classe **InputStreamReader**
  - et la classe BufferedReader.
- La classe **InputStreamReader** a un constructeur qui admet en paramètre un flot d'entrée.
- System.in est une instance de la classe InputStream. Avec une instance de InputStreamReader, on ne peut que lire des caractères un à un
- La classe BufferedReader a un constructeur qui prend en argument une instance de Reader dont hérite la classe InputStreamReader.
  - Cette classe permet de lire une ligne d'un texte, mais en revanche, on ne peut pas lui demander de lire un entier, un double etc...
- On lit ce qui est envoyé par le clavier ligne par ligne et on découpe le contenu de chaque ligne avec un StringTokenizer pour récupérer les entiers attendus.

Exemples

- Les deux programmes d'illustration ont pour objectif :
  - de lire des entiers envoyés par l'intermédiaire du clavier, et d'en faire la somme.
  - leurs cahiers des charges diffèrent uniquement sur la façon d'indiquer la fin de la saisie.

Saisir des données envoyées par le clavier (1)

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

class SaisieClavier
{
  public static void main (String[] argv) throws IOException,
                NumberFormatException
  {
    int somme = 0;
    String ligne;
    StringTokenizer st;
    BufferedReader entree = new BufferedReader
      (new InputStreamReader(System.in));

    ligne = entree.readLine();
    while(ligne.length() > 0)
      {
   st = new StringTokenizer(ligne);
   while(st.hasMoreTokens())
     somme += Integer.parseInt(st.nextToken());
   ligne = entree.readLine();
      }
    System.out.println("La somme vaut : "+somme);
  }
}
```

En rouge les instructions significatives pour la saisie des données au clavier.

Voici une exécution :

$ java SaisieClavier

3 1

2

La somme vaut : 6

L'utilisateur devra taper return deux fois de suite pour interrompre la saisie.

Deuxième méthode

- On utilise ici une instance de **StreamTokenizer** qui est un analyseur syntaxique (plutôt rudimentaire).
- Un constructeur de la classe StreamTokenizer prend en paramètre une instance de Reader.
- La méthode nextToken() de la classe StreamTokenizer retourne ***le type de l'unité lexicale*** suivante, type qui est caractérisé par une constante entière. Cela peut être :
  - TT_NUMBER si l'unité lexicale représente un nombre. Ce nombre se trouve dans le champ nval de l'instance de StreamTokenizer. Ce champ est de type **double**.
  - TT_WORD si l'unité lexicale représente une chaîne de caractères. Cette chaîne se trouve alors dans le champ sval de l'instance du StreamTokenizer.
  - TT_EOL si l'unité lexicale représente une fin de ligne. La fin de ligne n'est reconnue comme unité lexicale que si on a utilisé l'instruction nomDuStreamTokenizer.eolIsSignificant(**true**);
  - TT_EOF s'il s'agit du signe de fin de fichier.

Saisir des données envoyées par le clavier (2)

```
import java.io.*;

class EssaiStream
{
  public static void main (String[] argv) throws IOException
  {
    int somme = 0;
    int type;
    StreamTokenizer entree= new StreamTokenizer
      (new InputStreamReader(System.in));
    String fin=new String("fin");

    System.out.println("Donnez vos entiers, terminez avec fin");
    while(true)
      {
  type=entree.nextToken();
  if (type==StreamTokenizer.TT_NUMBER)
    somme += (int)entree.nval;
  else if ((type == StreamTokenizer.TT_WORD)&&
      (fin.equals(entree.sval)))
    break;
      }
    System.out.println("La somme vaut : " + somme);
  }
```

Voici une exécution :

Donnez vos entiers, terminez avec fin

1 -2 bonjour 4

3 fin 2

La somme vaut : 6

Sérialisation (1)

- La sérialisation introduit dans le JDK permet de **rendre un objet persistant** (il pourra être reconstitué à l'identique) de façon facile, transparente et standard
- Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM.
- Le format utilisé est **indépendant du système d'exploitation**. Ainsi, un objet sérialisé sur un système peut être réutilisé par un autre système pour récréer l'objet.

Sérialisation (2)

- Classe **ObjectOutputStream** : permet d'écrire des objets dans un flux binaire (méthode writeObject)
- Classe **ObjectInputStream :** permet de lire des objets dans un flux binaire (méthode readObject)
- La sérialisation **utilise l'interface Java.oi.Serializable**

Exemple (1)

Une classe dont les attributs pourront être envoyés vers un flux de données :

```
import java.io.*;
import java.util.*;
public MaClasse implements Serializable {
    private String monPremierAttribut;
    private Date monSecondAttribut;
    private long monTroisièmeAttributs;

    // . . . d'éventuelles méthodes . . .
}
```

Exemple (2)

Un attribut ne soit pas sérialisé pour une classe donnée :

```
import java.io.*;
import java.util.*;
public MaClasse implements Serializable {
    private String monPremierAttribut;
    private transient Date monSecondAttribut;
// non sérialisé
    private long monTroisièmeAttributs;

    // . . . d'éventuelles méthodes . . .
}
```

Exemple : les tableaux d'objets sérialisables

```
import java.io.*;

class ClasseEssai implements Serializable {
    String chaine = "bonjour";
    int n = 1;
}

class Serialisation {
  public static void main(String[] arg) {
       ObjectOutputStream sortie = new ObjectOutputStream(new
FileOutputStream("essai.dat"));
       sortie.writeObject(new ClasseEssai());
       sortie.writeObject(new java.util.Date());
       int[] tableau1 = {10, 11, 12};
       sortie.writeObject(tableau1);
       Integer[] tableau2 = {new Integer(110), new Integer(111)};
       sortie.writeObject(tableau2);
       sortie.close();

       ObjectInputStream entree = new ObjectInputStream(new
FileInputStream("essai.dat"));

       ClasseEssai c = (ClasseEssai)entree.readObject();
       System.out.println(c.chaine);
       System.out.println(c.n); entree.close();
    }
}
```

Exemple d'utilisation : LES SERVICES WEB / Appel du service web (1)

1. création du service

```
Service service = new Service();
```

2. création de l'appel du service

```
Call call = (Call) service.createCall();
```

3. Instanciation du nom de la méthode à appeler dans le service web

```
call.setOperationName(new QName("","constraints"));
```

4. création du type datahandler pour l'envoi et le retour

```
QName qnameAttachment = new QName("", "DataHandler");
```

Exemple : LES SERVICES WEB / Appel du service web (2)

5. sérialise l'envoi et fait le mapping à partir d'un DataHandler

```
call.registerTypeMapping(dhSource1.getClass(),  qnameAttachment,
JAFDataHandlerSerializerFactory.class,JAFDataHandlerDeserializerFactory.
class);
```

6. Instanciations de l'adresse où se trouve le service web

```
call.setTargetEndpointAddress( new
java.net.URL("http://siamoi2.cnam.fr:8080/axis/services/Cr
eateInstance"));
```

Exemple : LES SERVICES WEB / Appel du service web (3)

7. ajout des paramètres

```
call.addParameter("op1", qnameAttachment, ParameterMode.IN);
call.addParameter("op2", XMLType.XSD_STRING, ParameterMode.IN);
```

8. ajout du type de retour

```
call.setReturnType(qnameAttachment);
```

9. Invocation du service web

```
DataHandler ret = call.invoke(new Object[]{dhSource1,nameOnto});
```

Les interfaces graphiques

- **Java propose 2 générations de composants graphiques**
  - composants graphiques formant **l'AWT (Abstract Window Toolkit)**
  - composants graphiques de type **swing.**

Quelques avantages des composants Swing

- Les composants Swing sont dans l'ensemble d'une conception un peu différente de celle des composants que l'on trouve dans le paquetage java.awt.
- Un gros avantage est qu'ils n'utilisent **aucun code natif** et cela occasionne un gain de place mémoire (d'où le nom de *composant léger*) ;

  Par ailleurs, tous les composants de java.awt correspondaient à des fenêtres natives.
- On peut par exemple :
  - faire que les boutons présentent une image à la place de texte,
  - avoir des boutons ronds,
  - créer des bords variés pour les composants...

Généralités sur les interfaces graphiques

- **JFrame :** un composant swing construit à partir d'une fenêtre native. On obtient une fenêtre graphique en l'instanciant.

- **JPanel** : la classe javax.swing.JPanel fournit un composant graphique léger qui peut servir de containeur et de lieu pour faire des tracés.

Généralités sur les interfaces graphiques (1)

- Comment tracer une interfacer graphique, y placer des composants, y écrire, y dessiner...
  - Le principe est que l'on travaille avec un ensemble de composants graphiques, la fenêtre principale est un tel composant, et que certains composants peuvent contenir d'autres composants.
  - Beaucoup de types de composants, correspondant à différentes fonctionnalités, sont disponibles : par exemple des boutons, des zones de texte, des zones pour le dessin...
  - Pour ajouter, retirer, dimensionner, positionner les composants, on peut le faire directement en utilisant les méthodes appropriées ; on peut également employer un gestionnaire de répartition, Java prévoit plusieurs tels gestionnaires

Généralités sur les interfaces graphiques (2)

- Pour tracer (des chaînes de caractères, des dessins, des images) dans un composant graphique, on passe par la classe **java.awt.Graphics.**

- Pour tout composant graphique, on peut demander par la méthode getGraphics de cette composante à récupérer une instance de Graphics qui contient une copie du "contexte graphique". Ce contexte contient par exemple la couleur d'avant-plan du composant.

- La plupart des composants graphiques ont une taille par défaut, qui peut d'ailleurs être nulle. Par exemple, un bouton a par défaut la taille nécessaire pour écrire son nom.

- La méthode pack de la classe java.awt.Window, dont hérite par exemple la classe JFrame, permet de dimensionner la taille d'une fenêtre en fonction des tailles de ses différents sous-composants. Il peut être mieux de l'utiliser plutôt que de préciser directement la taille d'un composant.

EssaiBouton.java

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

class Boutons extends JFrame implements ActionListener {
 JButton trace = new JButton("trace");
 JButton efface = new JButton("efface");
 Ardoise ardoise = new Ardoise();  // pour dessiner le cercle

 Boutons()  {
     setLayout(new BorderLayout(5, 5));

     JPanel lesBoutons = new JPanel();  // pour les boutons
     lesBoutons.add(trace);
     lesBoutons.add(efface);
     add(lesBoutons, BorderLayout.NORTH);
     add(ardoise, BorderLayout.CENTER);

     trace.addActionListener(this);
     efface.addActionListener(this);
     pack();
 }
```

ActionListener : l'interface java.awt.event.ActionListener déclare la méthode actionPerformed (ActionEvent) qui est définie dans la classe EssaiBouton.

trace.addActionListener(this); : la méthode addActionListener de la classe java.lang.Button permet que le bouton concerné puisse "recevoir des clics" de l'utilisateur.
L'interface ActionListener ne contient qu'une seule méthode, la méthode actionPerformed. Lorsque l'on clique sur un bouton, si un ActionListener a été attribué à celui-ci, c'est la méthode de l'ActionListener désigné qui est utilisé. Il est essentiel de ne pas oublier cette instruction pour pouvoir utiliser le bouton. Nous dirons que le bouton trace est entendu par this.

Plug-ins Eclipse

- **Plug-ins Eclipse :**
  - **Omondo : Conception UML**

    (ATTENTION : les diagrammes ne sont pas imprimables ni exportables dans cette version). Vous pourrez cependant installer Together pour eclipse, un autre AGL dont la version gratuite est moins restrictive.
  - **Sysdeo : Tomcat**
  - **Lomboz : java**
  - **PHP Eclipse : Développement PHP**
  - **XmlBuddy : Conception XML (ce module ne fonctionne pas dans cette distribution)**

Plug-ins Eclipse (2)

Interfaces graphiques :

- **Plug-in**
  - **VE : Visual Editor**
- **http://www.eclipse.org/vep/WebContent/main.php**

Les points faibles de Java

- **Java est jeune et encore en gestation.**
- **Les JVM sont encore lentes.**
- **Java est gourmand en mémoire**
- **Certaines constructions du langage sont décevantes**
- **Certaines API sont décevantes.**
- **Java est un langage propriétaire (Sun).**

Les points forts de Java

- **Gestion de la mémoire par un *garbage collector.***
- **Mécanisme d'exception.**
- **Tableaux = objets (taille connue, débordement → exception).**
- **Sécurisé**
- **Multi-thread**
- **Distribué (API réseaux, applets)**