Programmation Orientée Objet en JAVA

Plan général

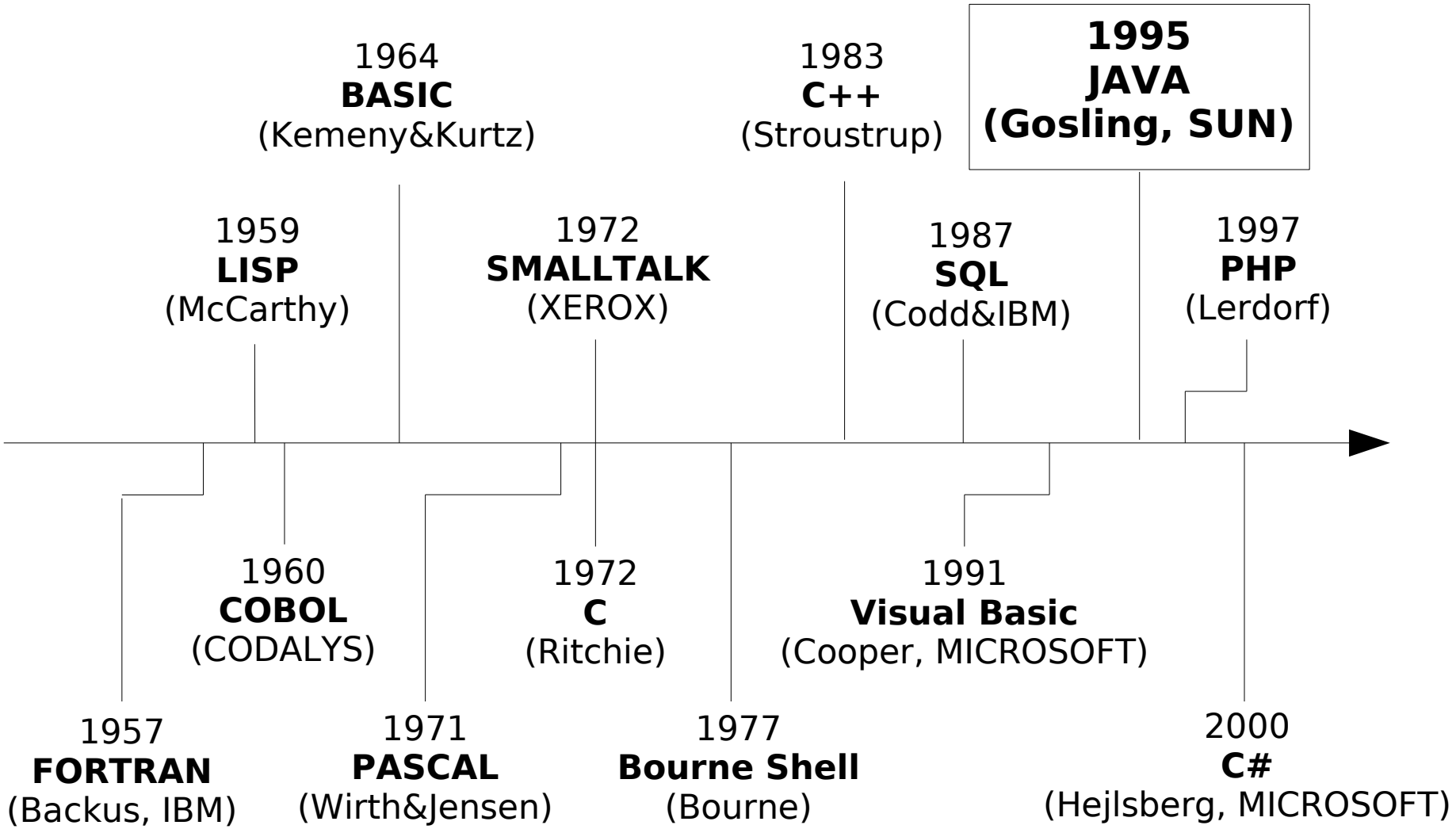
1. Introduction : historique et intérêt de JAVA
2. Premiers pas : généralités sur JAVA
3. Objets et classes
4. Héritage
5. Paquetages
6. Exceptions
7. Interfaces graphiques
8. Généricité
9. Les flux

1. Introduction à JAVA

Qu'est-ce que JAVA ?

- Un langage de programmation orienté objet
- Une architecture de *machine virtuelle*
- Un ensemble d'API (Interfaces de Programmation d'Applications) variées
- Un ensemble d'outils, le JDK ou SDK (JAVA ou Software Development Kit)

Historique des langages



Historique de JAVA

Un langage en constante et rapide évolution:

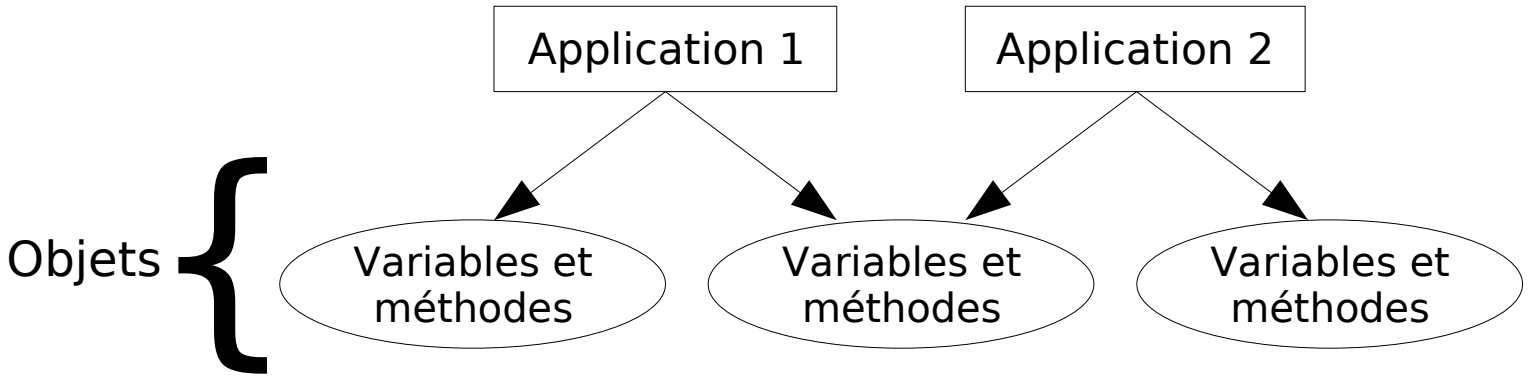
- 1991 : lancement du *green project*
- 1993 : contrôleur multimédia en Oak
- 1995 : première présentation de JAVA
- 1995 : Netscape annonce le support de JAVA
- 1996 : 1.0.2 (212 classes, 8 paquetages)
- 1997 : 1.1.5 (504 classes, 23 paquetages)
- 1998 : 1.2 (1520 classes, 59 paquetages)
- 2000 : 1.3 (1595 classes, 77 paquetages)
- 2002 : 1.4 (2175 classes, 103 paquetages)
- 2004 : 1.5 (2656 classes, 131 paquetages)
- 2006 : 6 (3777 classes, 203 paquetages)

Caractéristiques de JAVA

1. Orienté objet
2. Code compilé portable
3. Code compilé robuste et sécurisé
4. Multi-thread et distribué

JAVA est orienté objet

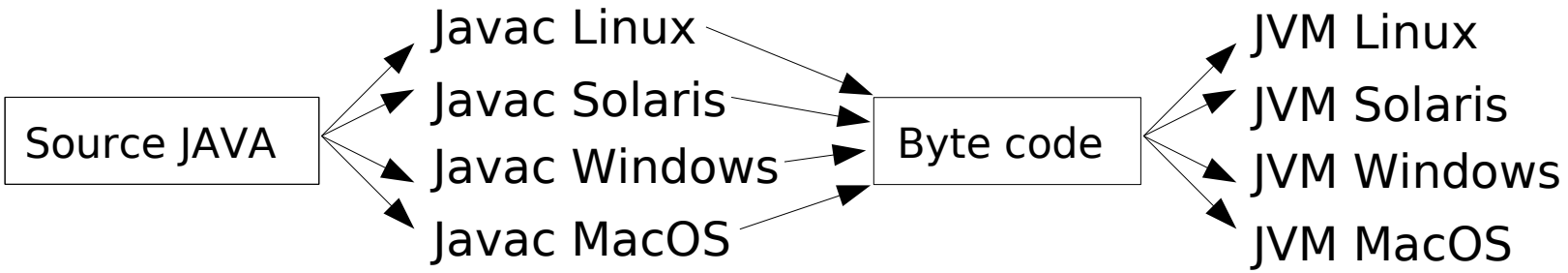
La programmation objet systématise la modularité et permet la réutilisation des composants logiciels



- En JAVA, tout est *classe* (pas de structures ou fonctions) sauf les types primitifs (`int`, *`float`*, *`double`*...) et les tableaux; les *objets* sont des *instances* de ces classes.
- Mécanisme d'*héritage* (simple pour les classes, multiple pour les interfaces)
- Les objets se manipulent via des *références*

JAVA est portable

- Le compilateur JAVA (*javac*) génère du *byte code* et non de l'*assembleur*
- Le byte code est exécuté sur une machine virtuelle, la JVM (Java Virtual Machine)
- C'est la JVM qui doit être réécrite pour chaque plateforme
- La taille des types primitifs (`int`, `float`, `double` etc.) est indépendante de la plateforme
- JAVA est accompagné d'une bibliothèque standard



JAVA est robuste et sûr

- Il a été conçu pour les systèmes embarqués (terminaux multimédia, téléphones, PDA, satellites...)
- Il gère seul sa mémoire grâce au *garbage collector*
- Pas d'accès direct aux ressources de la plateforme d'éxécution (en particulier la mémoire)
- Gestion des exceptions
- Compilateur contraignant (retourne des erreurs si une exception n'est pas gérée, si une variable n'est pas initialisée etc.)
- Contrôle des conversions de types (*cast)* à l'exécution
- Plusieurs couches de sécurité et code certifié par une clé

JAVA est multi-thread et distribué

Un *thread* est une tâche d'un programme s'exécutant séquentiellement : avec plusieurs threads on peut exécuter plusieurs tâches en même temps

- API thread (`java.lang.Thread` etc.)
- Accès concurrent à des objets gérés par *moniteur*
- API réseau (`java.net.Socket`, `java.net.URL` etc.)
- Applet (application téléchargée à chaque utilisation)
- Servlet (applet exécutée sur le serveur, non le client)
- RMI (Remote Method Invocation)
- JavaIDL (Interface Definition Language : Corba)

Inconvénients de JAVA

- Défauts de jeunesse : stabilité et compatibilité ascendante imparfaites, les environnements de programmation commencent seulement à apparaître
- Les JVM sont (très très) lentes (solutions : JIT «*Just In Time JAVA*», conversion en C ou assembleur)
- JAVA est très gourmand en mémoire
- Certaines API sont décevantes
- JAVA était jusque fin 2006 un langage propriétaire (Sun Microsystems)

Différences avec C++

En surface, JAVA est en fait un C++ simplifié:

- Pas de structures, d'unions ou de types énumérés
- Pas de `typedef`
- Pas de préprocesseur
- Pas de variables ou fonctions en dehors des classes
- Pas d'héritage multiple de classes
- Pas de *templates*
- Pas de surcharge d'opérateurs
- Pas de pointeurs, seulement des références

Principaux outils utilisés

(entre de nombreux autres possibles)

- Compilateur JAVA du SDK : *javac*
- Interpréteur de *byte code* du SDK : *java*
- Interpréteur d'applets du SDK : *appletviewer*
- Générateur de documentation : *javadoc*
- Environnement de développement : *Eclipse*
- Navigateur Web : *MS Internet Explorer*, *Firefox*

Un premier programme

Dans un fichier appelé *Hello.java* (nom impératif !), écrire le code suivant :

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

Compilation : *javac Hello.java*

- Lance la compilation des classes nécessaires et crée un fichier *Hello.class*

Exécution : *java Hello*

- La JVM recherche dans Hello.class une méthode de signature ***public static void*** *main (****String****[] args)* et l'exécute, si elle n'existe pas : erreur

Une première applet

Dans le fichier *PremApp.java* (nom impératif !), écrire :

```
import java.awt.* ;
public class PremApp extends java.applet.Applet
{ public void init()
 { add(new Label("Hello World")) ; }
}
```

Dans le même répertoire, créer le fichier *PremApp.html* avec :

```
<HTML><BODY>
<APPLET CODE="PremApp.class" WIDTH=350 HEIGHT=100>
</APPLET></BODY></HTML>
```

Compilation : *javac PremApp.java*

- Compile et crée un fichier *PremApp.class*

Exécution : *appletviewer PremApp.html*

- Lance une JVM cherchant *init()* et l'exécute graphiquement

2. Les bases : généralités sur JAVA

Organisation Générale

- L'unité de base d'un programme JAVA est la *classe* :

```
class UneClasse
  { définition des attributs
    définition des méthodes
  }
```

- Les classes sont décrites dans des *fichiers* :
  - Un fichier peu contenir plusieurs classes, mais une seule avec l'attribut **`public`** (à placer devant le mot clé `class`) qui sera visible depuis tous les autres fichiers et qui donnera son nom au fichier
  - Un fichier peut débuter par une déclaration de paquetage (ex : **`package`** `monPaquetage;`) et/ou des importations de classes venant d'autres paquetages (**`import`** `MaClasse;`)

Conventions de codage

Non obligatoires mais à respecter

- Le nom d'une classe commence par une majuscule (exemple : `UneClasse`)
- Le nom d'un méthode commence par une minuscule (exemple : `uneMethode`)
- Chaque mot supplémentaire d'un nom composé d'une classe ou d'une méthode commence par une majuscule (classe : `UnPetitExempleDeClasse`, méthode : `unPetitExempleDeMethode`)
- Les constantes sont en majuscules avec le caractère *souligné* (*underscore*) "_" comme séparateur (exemple : `MIN_VALUE`)

Un exemple introductif

```
import java.lang.Math ; // On veut pouvoir utiliser la classe Math
public class Addition
{ // args est le tableau des chaînes de caractères passées en argument
  public static void main (String [] args)
  { int i, addition=0 ;
    // length est un attribut associé aux instances de tableaux
    for (i=0 ; i<args.length ; i++)
    { // La méthode Integer.parseInt convertit une chaîne en entier
      addition += Integer.parseInt(args[i]) ;
      // out est un objet attribut de la classe System
       // println est une méthode appliquée à l'objet System.out (l'écran)
      // + est ici l'opérateur de concaténation
      System.out.println("Addition "+i+" : "+addition) ;
    }
  }
}
```

compilation : *javac Addition.java*

exécution : *java Addition 4 3*

Addition 0 : 4

Addition 1 : 7

Bases de JAVA

- La syntaxe de JAVA est volontairement très proche de celle de C/C++
- En JAVA, on manipule deux types d'entités :
- Les **variables** qui sont d'un *type primitif* défini
  - Elles sont créées par *déclaration* (ex: "`int i ;`" déclare une variable de nom `i` et de type `int`)
  - On peut leur *affecter* une valeur de leur type (ex: "i=0;") et/ou les utiliser dans une expression (ex: "i = 2*i + 1 ;")
- Les **objets** via des *références*
  - Les objets sont des *instances de classes*
  - Ce sera étudié en détail au chapitre suivant !

Les types primitifs

On retrouve en JAVA la plupart des types du C/C++, avec quelques changements et la disparition des non signés (`unsigned`). Les types ne sont pas des objets, des *wrappers* font la correspondance.

Type	Wrapper	Taille (octets)	Valeur minimale	Valeur maximale
byte	Byte	1	-128 Byte.MIN_VALUE	127 Byte.MAX_VALUE
short	Short	2	-32768 Short.MIN_VALUE	32767 Short.MAX_VALUE
int	Integer	4	-2147483648 Integer.MIN_VALUE	2147483647 Integer.MAX_VALUE
long	Long	8	-9223372036854775808 Long.MIN_VALUE	9223372036854775807 Long.MAX_VALUE
float	Float	4	-1.40239846E-45 Float.MIN_VALUE	3.40282347E38 Float.MAX_VALUE
double	Double	8	4.9406564584124654E-324 Double.MIN_VALUE	1.7976931348623157E308 Double.MAX_VALUE
char	Character	2	! Character.MIN_VALUE	? Character.MAX_VALUE
boolean	Boolean	1 Pas d'ordre ici	false Boolean.FALSE	true Boolean.TRUE

Les références

Les objets se manipulent en JAVA via des *références*

- Les références sont en réalité des *pointeurs* (au sens des langages C/C++) bien cachés
- On *déclare* une référence comme pour les types primitifs

```
Integer entier ;
```

- La déclaration ne réserve pas la place mémoire pour l'objet, elle sera allouée sur demande explicite par l'opérateur ***new***

```
entier = new Integer() ;
```

- La valeur `null` désigne une référence non allouée
- La libération de la mémoire est gérée par la JVM

Les tableaux

Les tableaux se manipulent en JAVA comme des objets

- Les tableaux sont déclarés comme des références

```
int t [] ; // t sera une référence à un tableau d'entiers
int [] t ; // même chose
int [] t1, t2 ; // t1 et t2 sont des tableaux d'entiers
int t1[], n, t2[] ; // même chose pour t1 et t2, et n est entier
Integer ti [] ; // ti est un tableau d'objets de type Integer
```

- On crée un tableau comme on crée un objet : avec **new**

```
t = new int[5] ; // t fait référence à un tableau de 5 entiers
ti = new Integer[3] ; // ti: tableau de 3 objets Integer
ti[0] = new Integer() ; // on est pas dispensé d'allouer les objets
```

- On peut initialiser les tableaux à la déclaration

```
int t[] = {1,n,4} ; // t[0] vaut 1, t[1] vaut n... plus besoin de new
```

- Les tableaux sont indexés à partir de zéro comme C/C++

Le type String

JAVA possède un vrai type chaîne de caractères

- Déclaration et initialisation similaires aux types primitifs

```
String chaine="une chaine" ;
String chaine2=new String("une chaine") ; // Même chose
String chaine3=chaine ;  // Utilisation d'autres chaînes
```

- Les valeurs des chaînes ne sont pas modifiables

```
String chaine="une chaine" ;
chaine="salut" ; // En réalité il y a ici création d'un nouvel objet
```

- Nombreuses possibilités de manipulation

```
c=2+chaine+"s" ; // Donne 2saluts (+: opérateur de concaténation)
int n=c.lenth() ; // Donne 7 (méthode longueur)
char x=c.charAt(2) ; // Donne a (caractère situé à la position 2)
boolean b=c.equals("autre") ; // Donne false (comparaison)
int p=c.compareTo("autre") ; // Equivaut à strcmp() du C/C++
```

etc.

Opérateurs (entre autres...)

Les opérateurs sont ceux de C/C++

- Arithmétiques :

  +,-,*,/,%

- Incrémentation :

  ++,-- ,+=,-=,/=,*=, %=

- Relationnels :

  == (egal),!= (différent),>,<,>=,<=

- Booléens :

  && (ET), || (OU), ! (NON)

- Binaires :

  & (ET), | (OU), ^ (XOU), ~ (complément), >>, <<

Structures de contrôle

Les structures de contrôle sont celles de C/C++

- On appelle *suite* une suite d'une ou plusieurs instructions consécutives (ex : `i = 0 ; j = i ;`)
- On appelle *bloc* une instruction seule (ex : `i = 0 ;`) ou une suite d'instructions placées entre `{` et `}`
  (ex : `{i = 0 ; j = i ; }`)
- On retrouve les structures conditionnelles :
  - `if ... else ...`
  - `switch`
- On retrouve les structures de boucles :
  - `for`
  - `while`
  - `do...while`

Structure conditionnelle if (1/2)

L'instruction `if` teste une condition booléenne, si le résultat est vrai, le bloc d'instructions suivant la condition est exécuté, sinon il ne l'est pas :

```
if (condition) bloc1
```

Optionnellement, le premier bloc sera suivit du mot clé `else` et d'un second bloc exécuté seulement si la condition est fausse :

```
if (condition) bloc1 else bloc2
```

Structures conditionnelle if (2/2)

```
// Exemple de code et d'execution
public class TestIf
{ public static void main (String[] args)
  { int a = 2, b = 3, max ;

    if (a < b)
      max = b ;
    else
    { if (a == b) // Ce "if" n'a pas de "else"
      { System.out.println("Les nombres sont egaux !") ;
        System.exit(0) ; // Pour sortir du programme
      }
      max = a ;
    }
    System.out.println("maximum = "+max) ;
  }
}
---
maximum = 3
```

Structure conditionnelle switch (1/2)

L'instruction `switch` cherche le résultat d'une *expression* (de type `char` ou `int`) dans une liste de cas et exécute la suite correspondante puis toutes les suites suivantes :

```
switch (expression)
{case cas1:suite1 case cas2:suite2 ... case cas_n:suite_n}
```

Optionnellement le dernier cas peut être le mot clé `default` si aucun cas ne correspond :

```
switch (expression)
{case cas1:suite1 case cas2:suite2 ... case cas_n:suite_n
default:suite}
```

Pour sortir du `switch` à la fin d'une suite d'instructions, utiliser `break;` en dernière instruction de cette suite

Structure conditionnelle switch (2/2)

// Exemple de code et d'execution

```
public class TestSwitch
{ public static void main (String[] args)
  { int a = 1 ;

    switch (a)
    { case 0  : System.out.println("NUL") ; break ;
      case 1  : System.out.println("MOYEN") ;
      case 2  : System.out.println("GRAND") ; break ;
      default : System.out.println("Inconnu") ;
    }
    System.out.println("Fin") ;
  }
}
---
MOYEN
GRAND
Fin
```

Structure de répétition for

L'instruction `for` exécute une *instruction1* puis répète les instructions du *bloc* suivant l'instruction `for` tant que sa *condition* est vraie, à la fin de chaque répétition elle exécute son *instruction2* :

```
for (instruction1;condition;instruction2) bloc
```

// Exemple de code et d'execution

```
public class TestFor
{ public static void main (String[] args)
  { for (int i=0;i<3;i++)
    System.out.println("i vaut "+i) ;
  }
}
---
i vaut 0
i vaut 1
i vaut 2
```

Structure de répétition while

L'instruction `while` évalue une *condition* puis répète les instructions du *bloc* tant que cette condition est vraie

`while (condition) bloc`

// Exemple de code et d'execution

```
public class TestWhile
{ public static void main (String[] args)
  { int i = 0 ;
    while (i<3)
    { System.out.println("i vaut "+i) ;
      i++ ;
    }
  }
}
---
i vaut 0
i vaut 1
i vaut 2
```

Structure de répétition do...while

L'instruction `do...while` exécute une *suite* puis évalue une *condition* et répète *bloc* tant que *condition* est vraie

```
do suite while (condition) ;
```

// Exemple de code et d'execution

```
public class TestDoWhile
{ public static void main (String[] args)
  { int i = 0 ;
    do
    { System.out.println("i vaut "+i) ;
      i++ ;
    }
    while (i<3) ;
  }
}
---
i vaut 0
i vaut 1
i vaut 2
```

3. Objets et classes

La programmation objet systématise la **modularité** et permet la **réutilisation** des composants logiciels

La **modularité** consiste à structurer un programme pour les décomposer en parties, en modules

- Plus simples que le programme complet
- Les plus indépendants possibles

La **réutilisation** consiste à partager des composants logiciels (objets ou modules) entre plusieurs programmes

Ces deux concepts sont centraux à la programmation objet : ils sont fondés sur une vue unifiée des types de données et des traitements et non sur la décomposition des traitements seuls

Objets

Objet = état + comportement

Un objet est une abstraction d'une donnée définie par :

- Une **identité** unique
- Des **attributs** valués (variables) qui déterminent l'état de l'objet
- Des **opérations** (méthodes) qui manipulent l'objet et définissent son comportement
- Un **cycle de vie** (l'objet naît, vit, meurt)

Attributs et opérations sont les **propriétés** de l'objet

Classes

La définition d'une classe décrit les propriétés des objets de cette classe

- Une classe décrit le **type** des objets correspondants

Une classe comporte un ou plusieurs *constructeurs* permettant de créer des objets de cette classe

- Une classe est une **fabrique** d'objets

Une classe peut posséder des propriétés globales (statiques)

Objets et classes

JAVA est un langage de programmation objet **fortement typé**

- Tout objet a un type non modifiable
- Le type d'un objet détermine à la fois son état et son comportement

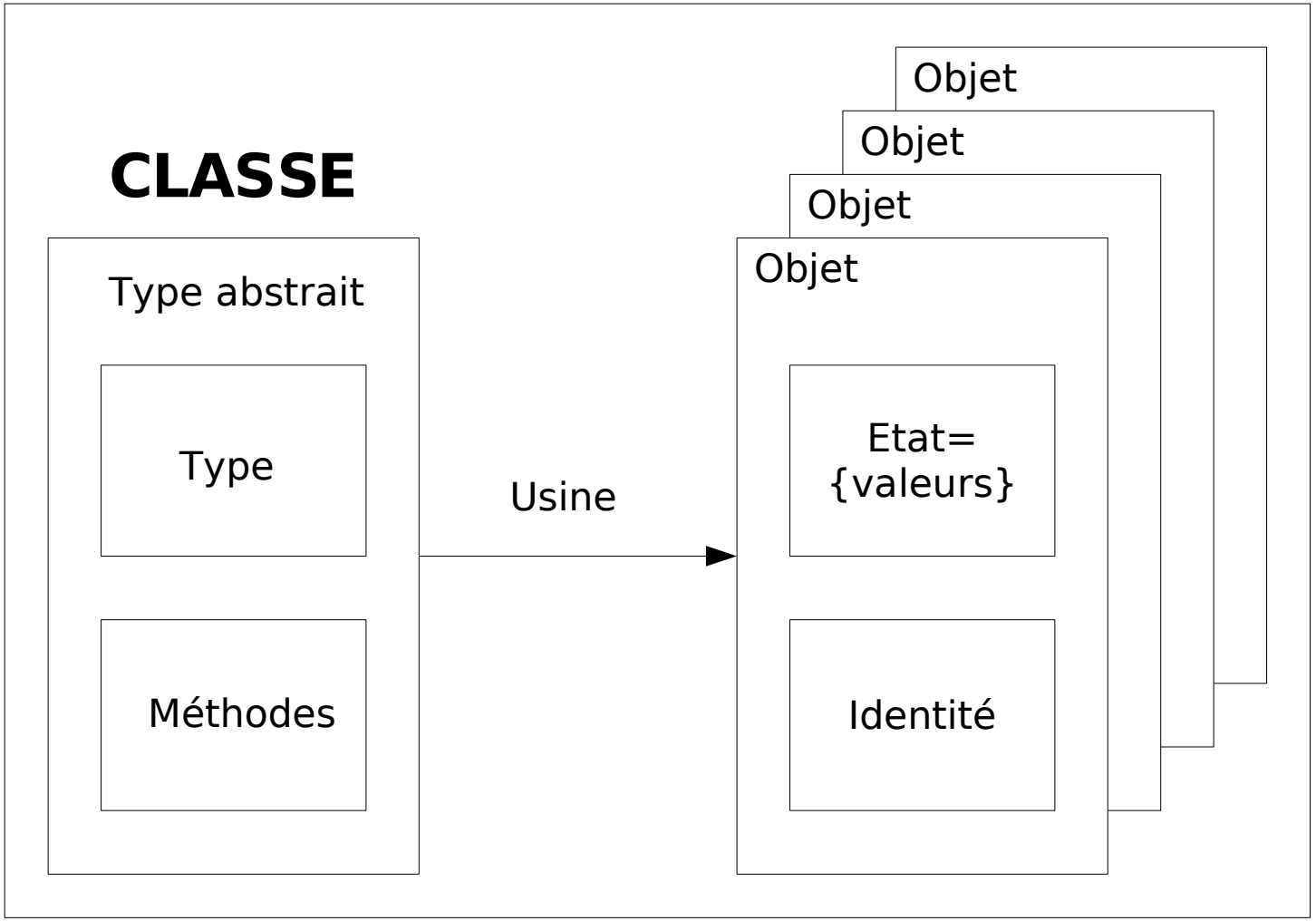
Une classe regroupe une catégorie d'objets de même type

- Une classe décrit le type de ses objets

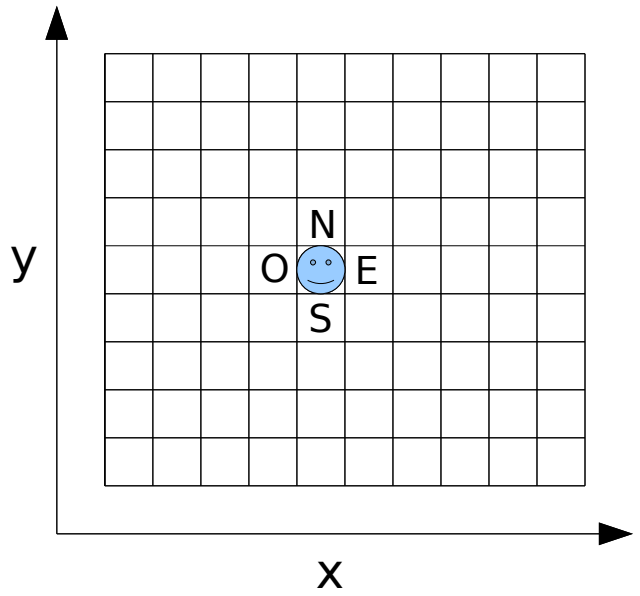
En JAVA, il n'existe pas d'objet créé statiquement

- Une classe sert à construire dynamiquement des objets

Classe/type/objet/valeur



Exemple : un robot



Soit un robot élémentaire :

- Etat
  - Position
  - Orientation du déplacement
  - ...
- Comportement
  - Avancer
  - Tourner à droite
  - ...

Exemple : la classe Robot

Le canevas général pour une classe appliqué à *Robot* est le suivant (non obligatoire mais à respecter) :

```
class Robot
{ //définition des constantes
  ...
  //définition des variables d'état des instances
  ...
  //définition des constructeurs
  ...
  //définition des méthodes sur les instances
  ...
} //fin de la classe Robot
```

Propriétés d'une classe

Les propriétés **dynamiques** se rapportent à une instance particulière de cette classe

- Variables d'instance (ou attributs)
- Méthodes (ou opérations)

Les propriétés **statiques** (`static`) se rapportent à la classe et sont communes à tous les objets de la classe

- Constantes
- Attributs statiques (ou variables de classe)
- Méthodes statiques (dont les constructeurs)

La classe définit ce qui est **visible** (attributs/méthodes) et par qui (autres classes)

- Quatre niveaux : `public`, `protected`, paquetage, `private`

La classe définit ce qui est **modifiable** ou non (`final`)

Attributs (état)

Un attribut est soit une valeur (appartenant à un type prédéfini) soit une référence (désignant un objet)

Un attribut peut être **statique** (précédé de `static`) :

- Il n'existe qu'un exemplaire de l'attribut
- L'initialisation a lieu à la première référence à la classe
- Il est accessible à l'aide du nom de la classe

Un attribut peut être **dynamique** (par défaut) :

- Chaque instance a son propre exemplaire
- Il existe des valeurs par défaut mais on préfèrera l'initialiser lors de la déclaration ou avec les constructeurs

Exemple : état d'un robot

```
class Robot
{ // Définition des constantes
  static final int NORD  = 0 ;
  static final int EST   = 1 ;
  static final int SUD   = 2 ;
  static final int OUEST = 3 ;

  // Définition des variables d'état des instances
  String nom ; // Initialisé à NULL par défaut
  int x, y ;   // Initialisé à 0 par défaut
  int orientation = NORD ;
  ...
```

Méthodes (comportement) 1/2

Une méthode en JAVA est soit une procédure, soit une fonction, soit un constructeur prenant 0, 1 ou plusieurs arguments (que l'on appelle aussi des paramètres)

Une **fonction** retourne une valeur ou une référence d'objet (par **`return`** *`expression`* dans le corps) et se définit ainsi :

```
typeRetour nomFonction (typeP1 p1, typeP2 p2 ...)
{ corps de la fonction }
```

Une **procédure** est une suite de traitements et se définit ainsi :

```
void nomProcedure (typeP1 p1, typeP2 p2 ...)
{ corps de la procédure }
```

Un **constructeur** s'applique à une classe, invoque sa fabrique d'objets, sert à initialiser l'objet fabriqué et se définit ainsi :

```
NomClasse (typeP1 p1, typeP2 p2 ...){ corps }
```

Méthodes (comportement) 2/2

Une méthode peut être **statique** (si elle est précédée de **`static`** ou est un constructeur) :

- Elle est invocable à l'aide du nom de la classe :
  `NomClasse.nomMethode(arguments)`
- Elle ne peut accéder aux variables non statiques

Une méthode peut être **dynamique** (par défaut) :

- Elle est invocable depuis un objet d'une autre classe à l'aide du nom d'une référence :
  `objetReveceur.nomMethode(arguments)`
- Elle est invocable depuis un objet de la classe courante par
  `nomMethode(arguments)`
- Dans le corps, on pourra désigner l'objet receveur par **`this`**
- La méthode peut accéder/modifier les attributs

Exemple : comportement d'un robot

```
// définition des méthodes sur les instances
void avancer ()
{ switch (orientation)
   { case NORD  : y++ ; break ;
     case EST   : x++ ; break ;
     case SUD   : y-- ; break ;
     case OUEST : x-- ; break ;
   }
}
void tourner (int n)
{ orientation = (orientation + n)%4 ; }
void tournerADroite () { tourner(1) ; }
void tournerAGauche () { tourner(3) ; }
void faireDemiTour  () { tourner(2) ; }
...
```

Surcharge des méthodes

En JAVA, l'identifiant d'une méthode est défini par sa signature :

- Nom choisi par le programmeur
- Liste des types des arguments
- *Ne prend pas en compte le type retour*

Deux méthodes différentes peuvent avoir le même nom si elles se distinguent par leurs signatures

L'interpréteur choisit la bonne méthode en fonction

- Du nombre d'arguments
- Du type des paramètres figurant dans l'invocation
- *Le type de retour n'est pas pris en compte*

Exemple : surcharge de avancer()

```
void avancer (int pas)
{ switch (orientation)
  { case NORD  : y+= pas ; break ;
    case EST   : x+= pas ; break ;
    case SUD   : y-= pas ; break ;
    case OUEST : x-= pas ; break ;
  }
}
```

Création d'objets

La primitive `new` invoque la **fabrique** d'objets de la classe

- Elle crée l'objet et initialise ses variables d'instance
- Elle rend en résultat une référence sur l'objet créé
- Cette référence peut être affectée à une variable
- Elle invoque ensuite une méthode particulière sur l'objet : le **constructeur**

Un constructeur par défaut «()» est toujours disponible

- Il porte le nom de sa classe et n'a pas de type retour (même pas `void`)
- Il peut être redéfini pour effectuer les initialisations voulues par le programmeur
- Comme toute méthode, il peut être surchargé

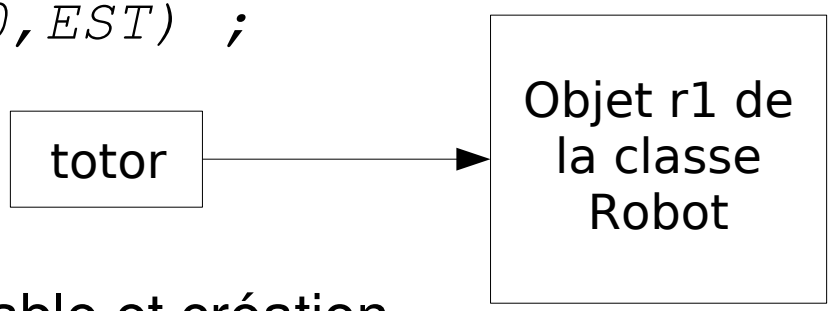
Exemple : constructeurs du robot

```
// définition des constructeurs
Robot ()
{ nom = "anonyme" ; }

Robot (String n)
{ nom = n ; }

Robot (String n, int x0, int y0, int o)
{ nom = n ;
  x = x0 ;
  y = y0 ;
  if ((o >= NORD) && (o <= OUEST))
    orientation = o ;
  else
    System.out.println("Erreur !") ;
}
```

Exemple : création des robots

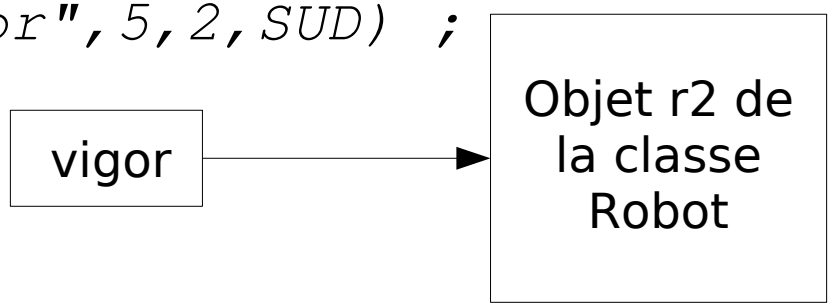
```
Robot totor ; // initialisé à NULL
```


```
totor = new Robot("Totor",0,0,EST) ;
```

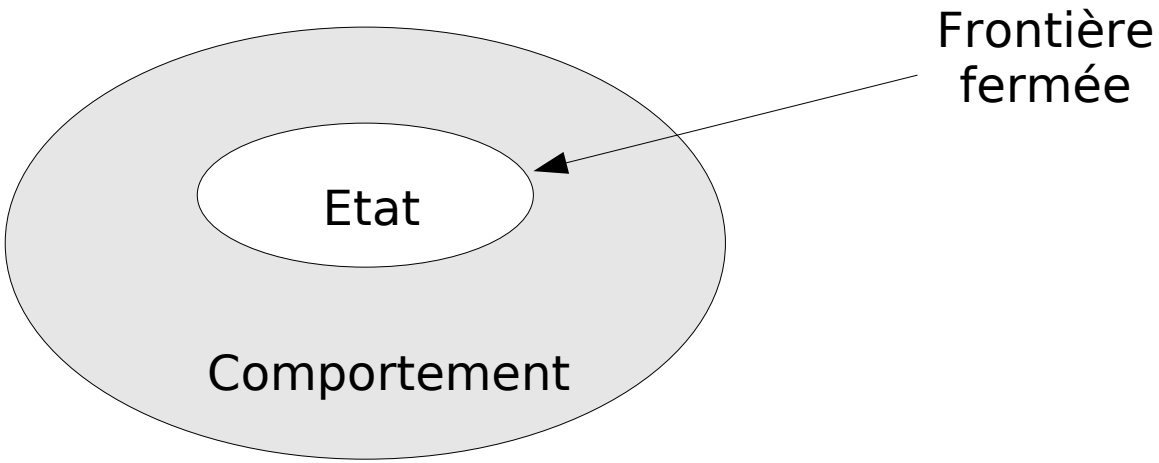


On peut aussi mêler déclaration de variable et création dynamique d'objet à l'initialisation de la référence

```
Robot vigor = new Robot("Vigor",5,2,SUD) ;
```



Encapsulation stricte



L'**encapsulation** consite à

- Masquer l'état d'un objet (les variables d'instances)
- Ne laisser accéder qu'au comportement (les méthodes)

Il est possible de définir sélectivement des méthodes particulières pour accéder à l'état de façon contrôlée : les **accesseurs** (exemple : setVariable(), getVariable())

Encapsulation contrôlée

En JAVA, l'encapsulation est optionnelle

Elle concerne toutes les propriétés d'un objet, aussi bien l'état que le comportement

Le concepteur d'un objet trace la frontière à sa guise

- Elle est contrôlée par des mots clés, des **modificateurs**
- Les principaux modificateurs sont `public` et `private`

En absence de modificateurs :

- Les objets et les classes du même module (paquetage) ont accès au contenu – pas les autres
- Les fichiers sources JAVA d'un même répertoire forment un module par défaut

Exemple : accesseurs pour Robot

```
class Robot
{ ...
  // Définition des variables d'état des instances
  public String nom ; // Initialisé à null par défaut
  private int x, y ;  // Initialisé à 0 par défaut
  private int orientation = NORD ;
  ...
  // Méthodes accesseurs
  public int getOrientation()
  { return Orientation ; }

  public void setOrientation(int o)
  { if ((o >= NORD) && (o <= OUEST))
      orientation = o ;
    else
      // Traitement de l'erreur
  }
  ...
```

4. Héritage

L'héritage consiste à définir un type d'objet similaire à un type existant avec des propriétés supplémentaires.

L'héritage de classe porte à la fois sur l'état et le comportement:

- Une *sous-classe* hérite de toutes les **variables** d'instance de sa *super-classe*
- Une classe hérite de toutes les **méthodes** de sa super-classe
- Une classe hérite de toutes les **propriétés statiques** de sa super-classe

L'héritage est transitif: une sous-classe peut être la super-classe d'une autre classe.

Extension/spécialisation

Une sous classe se définit par **extension** et/ou **spécialisation** d'une classe existante

class SousClasse **extends** *Superclasse*

- (extension) on peut ajouter de nouveaux champs (on ne peut pas redéfinir les champs, mais on peut les masquer par d'autres de mêmes noms), et de nouvelles méthodes
- (spécialisation) On peut redéfinir des méthodes de sa super-classe

Une classe peut **interdire**

- Sa propre dérivation (par le mot clé *final*)
- La redéfinition de certaines fonctions (*final*)
- L'utilisation de certains membres (*private*), qui sont hérités quand même (présents mais pas mentionables)

Exemple : super classe Point

```
class Point
{ // Variables
  private int x, y ;
  // Méthodes
  public void deplace(int dx, int dy)
  { x += dx ; y += dy ;
  }
  public void affiche()
  { System.out.print ("Je suis en "+x+" "+y) ;
  }
  // Constructeurs
  Point(int x, int y)
  { this.x = x ; this.y = y ;
  }
}
```

Exemple : sous-classe PointColore

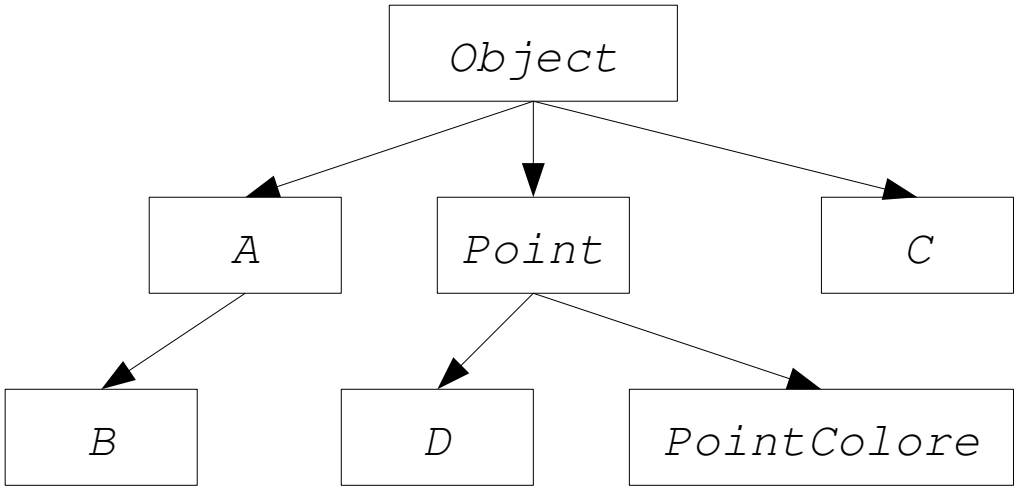
```
class PointColore extends Point // Hérite de Point
{ // Variables
  protected int couleur ;
  // Méthodes
  public void colore(int couleur)
  { this.couleur = couleur ;
  }
  public void affiche() // Redéfinition de affiche()
  { println() ;
    super.affiche() ; // appel à affiche() de la super-classe
    System.out.println("Je suis de couleur "+couleur) ;
  }
  // Constructeurs
  PointColore(int x, int y, int c)
  { super(x,y) ; // Appel au constructeur de la super-classe
    couleur = c ;
  }
}
```

Exemple : un test

```
public class Test
{ public static void main(String [] args)
  { Point p = new Point(1,2) ;
    PointColore pc = new PointColore(3,4,1) ;
    p.affiche() ;
    pc.affiche() ;
    p.deplace(1,2) ;
    pc.deplace(1,2) ;
    pc.colore(2) ;
    p.affiche() ;
    pc.affiche() ;
  }
}
---
Je suis en 1 2
Je suis en 3 4 Ma couleur est 1
Je suis en 2 4
Je suis en 4 6 Ma couleur est 2
```

Hierarchies de classes

- En JAVA, on ne peut dériver via **`extends`** que d'une seule classe, c'est l'*héritage simple* (à comparer avec l'*héritage multiple* de C++)
- Toute classe hérite implicitement de la super-classe prédéfinie `Object`
- La hiérarchie des classes est donc un arbre



La super-classe `Object`

La classe *`Object`* dont toute classe hérite en JAVA possède des méthodes soit à utiliser telles quelles, soit à redéfinir

- La méthode *`toString`* retourne un objet de type **`String`** contenant par défaut le nom de la classe concernée et l'adresse de l'objet en hexadécial, c'est cette méthode qui est invoquée lors du passage d'une référence en paramètre de *`System.out.println()`*
- La méthode `equals` qui se contente par défaut de comparer les adresses des deux objets concernés

Ces méthodes peuvent (et souvent doivent) être redéfinies dans les sous classes

Héritage et typage

En JAVA, on peut tester le type exact d'un objet par

(<expressionObjet> **instanceof** *ClasseX)*

qui rend `true` si l'objet est de la `ClasseX`, `false` sinon

- Un objet d'une sous-classe est aussi objet de sa (ou de ses) super-classes (*polymorphisme*)
- Si la classe B hérite de A, une référence à un objet de la classe A accepte un objet de la classe B

```
PointColore pc = new PointColore() ;
Point p = new Point() ;
p = pc ; // correct, toujours défini
pc = p ; // incorrect, que vaut pc.couleur ?
```

Redéfinition de méthodes

Une méthode dans la super-classe peut être redéfinie dans une sous-classe

- La nouvelle définition remplace celle héritée pour les objets de la sous-classe
- Le remplacement est effectif même quand l'objet de la sous-classe est référencé par une variable typée par la super-classe (*polymorphisme*)

```
PointColore pc = new PointColore(1,2,3) ;
Point p = new Point() ;
p = pc ;
p.affiche(); // Donne : Je suis en 1 2 Ma couleur est 3
```

- L'ancienne définition n'est plus accessible de l'extérieur de l'objet
- L'ancienne définition n'est accessible que par l'objet lui-même grâce au mot clé **`super`**

Construction des objets dérivés

En JAVA, le constructeur de la sous-classe doit prendre en charge l'intégralité de la construction de l'objet

Les constructeurs ne sont pas hérités

- Un constructeur d'une sous-classe peut appeler le constructeur de sa super-classe, mais il doit s'agir de la première instruction du constructeur et il est désigné par le mot clé **`super`**
- L'initialisation en cascade est possible en remontant la hiérarchie des classes

Héritage et visibilité

La visibilité des propriétés d'une super-classe à partir de ses sous-classes est définie selon quatre modes:

- Aucun modificateur ("friendly" ou "paquetage") : accessibles par les classes, y compris les sous-classes héritant, qui font partie du même module (package), inaccessibles par les autres
- **`public`** : accessibles par toutes les classes et sous-classes
- **`protected`** : accessibles par toutes les classes du même module et les sous-classes héritant dans les autres modules
- **`private`** : inaccessibles par toutes les autres classes y compris les sous-classes héritant

Les classes abstraites

JAVA permet de définir partiellement une classe, avec des méthodes dont on décrit l'interface mais pas l'implémentation

- Pour faciliter une approche progressive de la conception d'un programme
- Pour permettre de définir des types incomplets qui serviront de point de départ ou de modèle pour les types complètement implémentés

Une classe sera

- **abstraite**, si son implémentation n'est pas complète
- **concrète** si on peut créer un objet de cette classe et l'utiliser

Méthode abstraite

Une méthode abstraite définit seulement la signature d'une méthode:

- Son nom
- Son type de retour
- Ses arguments

- Elle reste virtuelle mais permet de prévoir des opérations similaires sur des objets de types différents
- On parle également de méthode différée pour insister sur le fait qu'**elle est destinée à être définie ou redéfinie dans les sous-classes**
- Une méthode abstraite est toujours déclarée **`public`**, précédée du mot clé **`abstract`** et contient les noms de ses arguments (même s'ils sont inutiles), par exemple :

```
abstract public int mAbstraite(double arg1) ;
```

Classe abstraite

Une classe est abstraite si elle contient au moins une méthode abstraite ou hérite d'une classe abstraite sans définir toutes les méthodes abstraites de sa super-classe

- Elle est déclarée optionnellement (mais il faut toujours le faire) par le mot clé **`abstract`**
- Elle sert de racine à un sous-arbre d'héritage dans lequel les sous-classes apporteront l'implémentation des méthodes abstraites
- Elle joue le rôle de prototype
- Elle peut définir un état (variables) et un comportement concret (méthodes complètes)
- **On ne peut pas créer un objet d'une telle classe**

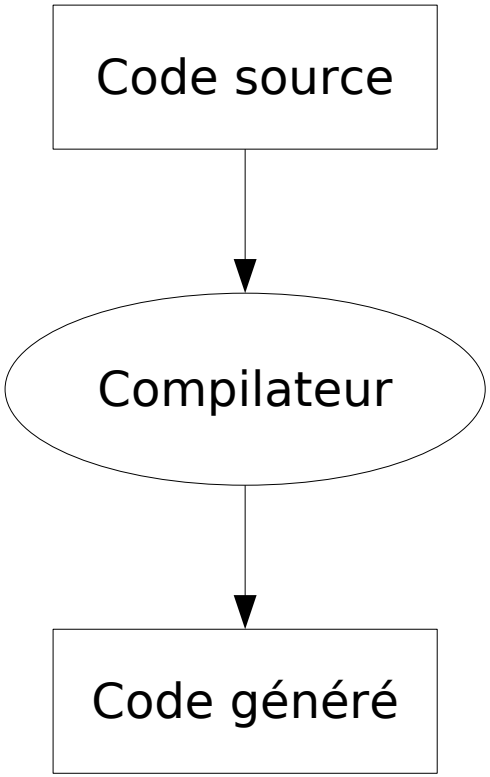
Classe abstraite : exemple (1/2)

```
abstract class Affichable // Classe abstraite
{ abstract public void affiche() ;// Methode abstraite
}

class Entier extends Affichable // Classe concrete
{ private int valeur ;
  public void affiche() // Definition de la methode abstraite
  { System.out.println("Entier = "+valeur) ; }
  public Entier(int n)
  { valeur = n }
}

class Flottant extends Affichable // Classe concrete
{ private float valeur ;
  public void affiche() // Definition de la methode abstraite
  { System.out.println("Flottant = "+valeur) ; }
  public Flottant(float n)
  { valeur = n }
}
```

Classe abstraite : exemple (2/2)

```
public class TestAffichable
{ public static void main (String [] args)
  { Affichable [] tab = new Affichable [3] ;
    tab[0] = new Entier(25) ;
    tab[1] = new Flottant(1.25) ;
    tab[2] = new Entier(42) ;

    for (int i=0 ; i<3 ; i++)
    tab[i].affiche() ;
  }
}

---

Entier = 25
Flottant = 1.25
Entier = 42
```

Liaison statique/dynamique



Liaison statique

- Le code (instructions machines) de l'appel est géré directement par le compilateur (cas de C/C++)

Laison dynamique

- Le compilateur génère du code capable de construire le bon appel à l'exécution (e.g. une table de sauts, cas de JAVA)
- L'interpréteur détermine directement le bon appel à l'exécution

Les interfaces

Une **interface** est un comportement abstrait que les classes peuvent se proposer d'implémenter (elle se présente comme une classe avec **`interface`** à la place de **`class`**)

```
public interface MonInterface
{ void f(int n) ; void g() ; }
```

- Ce comportement est spécifié par une collection de déclarations de méthodes qui seront automatiquement déclarées comme **`public abstract`**
- Une interface ne définit aucun corps de méthode ni aucune variable
- Une interface peut définir des constantes déclarées automatiquement comme **`static final`**
- Une interface n'est pas une classe et n'est pas instanciable (on ne crée pas d'objets avec)

Implémentation d'une interface

Lors de la définition d'une classe, on peut préciser qu'elle implémente une interface à l'aide du mot clé **`implements`**

```
class MaClasse implements MonInterface
{//MaClasse doit définir les methodes f et g prévues
 //dans MonInterface
}
```

- Cette implémentation fournit une implémentation du comportement abstrait défini par l'interface
- Plusieurs classes peuvent fournir des implémentations différentes
- Une classe peut implémenter plusieurs interfaces : `class MaClasse implements Int1, Int2,... {...}`
- Elle doit alors implémenter toutes les méthodes de ses interfaces

Interfaces et polymorphisme

Le **polymorphisme** signifie que chaque objet est susceptible de répondre avec **sa propre méthode** à l'invocation d'un comportement donné

On peut définir des variables de type interface

```
MonInterface i ;// reference a un objet d'une classe
                // implementant MonInterface
```

On peut alors affecter à `i` n'importe quelle référence à un objet d'une classe implémentant `MonInterface`

```
i = new MaClasse() ;
```

A travers `i` on pourra manipuler des objets de classe quelconque, pas forcément liées par héritage, si elles implémentent l'interface `MonInterface`

Interfaces : exemple (1/2)

```
interface Affichable // Interface
{ abstract public void affiche() ;// Methode abstraite
}

class Entier implements Affichable
{ private int valeur ;
  public void affiche() // Definition de la methode abstraite
  { System.out.println("Entier = "+valeur) ; }
  public Entier(int n)
  { valeur = n }
}

class Flottant implements Affichable
{ private float valeur ;
  public void affiche() // Definition de la methode abstraite
  { System.out.println("Flottant = "+valeur) ; }
  public Flottant(float n)
  { valeur = n }
}
```

Interfaces : exemple (2/2)

```
public class TestAffichable
{ public static void main (String [] args)
  { Affichable [] tab = new Affichable [3] ;
    tab[0] = new Entier(25) ;
    tab[1] = new Flottant(1.25) ;
    tab[2] = new Entier(42) ;

    for (int i=0 ; i<3 ; i++)
    tab[i].affiche() ;
  }
}
```

```
---

Entier = 25
Flottant = 1.25
Entier = 42
```

Hiérarchies d'interfaces

On peut définir une interface comme une généralisation d'une autre avec **extends**, on parle là aussi d'héritage

```
interface I1
{ void f(int n) ;
  static final int MAX = 100 ;
}
interface I2 extends I1
{ void g() ; }
```

En fait la définition de I2 est totalement équivalente à

```
interface I2
{ void f(int n) ;
  static final int MAX = 100 ;
  void g() ;
}
```

L'héritage multiple est ici possible (mais trivial)

Quelques interfaces "standard"

- L'interface *Comparable* sert à pouvoir trier toute classe où une notion d'ordre total existe : on doit définir une méthode *compareTo* pour comparer deux objets
- L'interface *Collection* est la racine des interfaces des conteneurs de données (listes, ensembles...), et propose d'implémenter des méthodes telles que *size, add, contains*...
- L'interface *Iterator* déclare *hasNext, next, remove*... : si une classe implémente cette interface, on sait comment parcourir ses instances
- Les interfaces *Serializable* et *Clonable* ne déclarent rien mais servent de marqueur pour indiquer les classes pouvant être sauvegardées/échangées ou copiées champ-à-champ
- etc.

Classes abstraites vs interfaces

Les classes abstraites

- Permettent de partager une partie d'implémentation (variables d'instances, méthodes définies)
- Utiles comme racines de sous-arbres d'héritage
- Comme il n'y a pas d'héritage multiple, on ne pourra pas hériter d'autre chose :-(

Les interfaces

- Une classe peut en implémenter plusieurs
- On peut utiliser des variables de type interface
- Très utiles en terme de génie logiciel
- Rien n'est implémenté, tout est à faire :-(

5. Paquetages

Un **paquetage** est un regroupement logique d'un ensemble de classes sous un identificateur commun :

- Il est proche de la notion de bibliothèque de C/C++
- Il facilite le développement et la cohabitation de grands logiciels en répartissant les classes dans différents paquetages
- Il permet de créer un *espace de nommage :* le risque d'avoir deux classes de même nom est limité aux seules classes d'un même paquetage
- Il définit un niveau de visibilité supplémentaire : sans le mot clé **`public`**, une classe n'est visible que par les classes du même paquetage

Attribution d'une classe à un paquetage

L'attribution d'un nom de paquetage se fait au début du fichier source (toutes les classes d'un même fichier source appartiennent au même paquetage) :

```
package nom ;
```

- Ici *nom* est le nom du paquetage
- Le nom du paquetage est soit un simple identificateur, soit une suite d'identificateurs séparés par des points
- Les points représentent une organisation en sous paquetages (par exemple *nom.sousnom*) qui devra correspondre à l'arborescence des fichiers (la classe publique *MaClasse* du paquetage *nom.sousnom* sera dans le fichier nom/sousnom/MaClasse.java)
- Si on indique pas de paquetage, les classes d'un fichier appartiennent au paquetage "*anonyme*"

Espace de nommage

- On peut toujours identifier une classe publique en la "*qualifiant*" par son nom de paquetage

```
nom.sousnom.MaClasse m = new nom.sousnom.MaClasse();
```

- Un nom simple n'est utilisable que dans son paquetage

```
package nom.sousnom ;
...
MaClasse m = new MaClasse(); // Notation OK
```

- Mais l'importation permet l'utilisation des noms simples

```
import nom.* ; // Importe les classes de nom
               // mais pas des sous-paquetages !!!
import nom.sousnom.* ;
...
MaClasse m = new MaClasse(); // Notation OK
```

- Si conflit de noms : erreur (il faudra qualifier les noms)

6. Les exceptions

Un programme peut avoir à faire face à de multiples circonstances exceptionnelles qui vont compromettre la poursuite de son exécution :

- Données incorrectes (désordre, mauvais type...)
- Résultat inattendu (overflow...)
- Fin de fichier prématurée (informations incomplètes)
- Dépassement de taille prévue (tableaux trop petits...)
- etc.

Il s'agît de repérer et de traiter ces cas sans pour autant déranger le code de base

JAVA propose pour cela un mécanisme très souple de **gestion des exceptions**

Gérer les problèmes : niveau utilisateur ?

Supposons que l'on souhaite ne manipuler que des points ayant des coordonnées non négatives :

```
if ((x>=0)&&(y>=0))
  p = new Point(x,y) ;
else // Gestion coordonnée négative
{ System.out.println("Mauvaises coordonnees !") ;
  System.exit(-1) ;
}
if ((x+dx)>=0) && (y+dy>=0))
  p.deplace(dx,dy) ;
else // Gestion coordonnée négative
{ System.out.println("Mauvais deplacement !") ;
  System.exit(-1) ;
}
```

- Code vite fastidieux
- Les tests seront rapidement oubliés, délaissés...

Gérer les problèmes : niveau concepteur ? (1/2)

```
class Point
{ // Variables, une coordonnée ne doit pas être négative !
  private int x, y ;

  // Méthodes
  public void deplace(int dx, int dy)
  { if (((x+dx)<0) || ((y+dy)<0)) // Gestion coordonnée négative
    { System.out.println("Mauvais deplacement !") ;
      System.exit(-1) ;
    }
    x += dx ; y += dy ;
  }
  // Constructeurs
  public Point(int x, int y)
  { if ((x<0) || (y<0)) // Gestion coordonnée négative
    { System.out.println("Mauvaises coordonnees !") ;
      System.exit(-1) ;
    }
    this.x = x ; this.y = y ;
  }
}
```

Gérer les problèmes : niveau concepteur ? (2/2)

Bien mieux, mais le traitement du problème des coordonnées négatives dans cet exemple :

- Est associé à sa détection, cela résulte en des programmes peu lisibles car le traitement principal est moins visible
- Est figé par le concepteur de la classe `Point` alors qu'un utilisateur pourrait vouloir gérer les problèmes différemment

Erreur ou exception

- Nécessité de traiter les cas anormaux "standards" (division par zéro, débordement de tableaux etc.)
- Nécessité d'un mécanisme de déclenchement d'une erreur (les exceptions) pour :
  - Se concentrer sur l'essentiel
  - Dissocier la détection d'une anomalie de son traitement
  - Alléger le source du programme (éviter les tests fastidieux)
  - Transférer la responsabilité du traitement de l'erreur à l'appelant

Exceptions : exemple 1

```
public class ExceptionStandard1
{ public static void main (String [] args)
  { int x = Integer.parseInt(args[0]) ;
    int y = Integer.parseInt(args[1]) ;

    int z = x / y;

    System.out.println("z = x / y = "+z) ;
  }
}

---

> java ExceptionStandard1 6 0
java.lang.ArithmeticException
at ExceptionStandard1.main (ExceptionStandard1.java:6)
```

try/catch : présentation

Dans l'exemple 1, une *exception standard* correspondant à la division par zéro a été levée

Pour gérer une éventuelle exception, il faut :

- Un bloc particulier, dit "bloc `try`" englobant les instructions dans lesquelles une exception peut être levée

```
try
{ // instructions
}
```

- Faire immédiatement suivre ce bloc de *gestionnaires d'exception,* chacun introduit par le mot-clé **`catch`**

```
catch (ExExemple ee) // gestionnaire pour ExExemple
{ // instructions à executer en cas d'exception ExExemple
}
```

Exceptions : exemple 2

```
public class ExceptionStandard2
{ public static void main (String [] args)
  { int x = Integer.parseInt(args[0]) ;
    int y = Integer.parseInt(args[1]) ;
    try
    { int z = x / y ;
      System.out.println("z = x / y = "+z) ;
    }
    catch (ArithmeticException ae)
    { System.out.println("Division par zero !") ;
    }
  }
}

---

> java ExceptionStandard2 6 0
Division par zero !
```

Exceptions : exemple 3

```
public class ExceptionStandard3
{ public static void main (String [] args)
  { try
    { int x = Integer.parseInt(args[0]) ;
      int y = Integer.parseInt(args[1]) ;
      try
      { int z = x / y ;
        System.out.println("z = x / y = "+z) ;
      }
      catch (ArithmeticException ae)
      { System.out.println("Division par zero !") ;
      }
    }
    catch (ArrayIndexOutOfBoundsException aioobe)
    { System.out.println("On veut deux arguments !") ;
    }
  }
}
---
> java ExceptionStandard3 6
On veut deux arguments !
```

Exceptions : exemple 4

```
public class ExceptionStandard4
{ public static void main (String [] args)
  { try
    { int x = Integer.parseInt(args[0]) ;
      int y = Integer.parseInt(args[1]) ;
      try
      { int z = x / y ;
        System.out.println("z = x / y = "+z) ;
      }
      catch (ArithmeticException ae)
      { System.out.println("Division par zero !") ; }
    }
    catch (ArrayIndexOutOfBoundsException aioobe)
    { System.out.println("On veut deux arguments !") ; }
    catch (NumberFormatException nfe)
    { System.out.println("Mauvais format de nombre !") ; }
  }
}
---
> java ExceptionStandard4 6 4.5
Mauvais format de nombre !
```

try/catch/finally : présentation

Le déclenchement d'une exception provoque le branchement inconditionnel au gestionnaire qui convient, à quelque niveau qu'il se trouve

JAVA permet d'introduire après tous les gestionnaires d'un bloc `try`, un bloc introduit par le mot clé **`finally`** qui sera toujours exécuté :

- Soit à la fin naturelle du bloc `try` sans qu'aucune exception n'ait été déclenchée
- Soit après le gestionnaire s'il n'a pas fini l'exécution

```
try       { /* instructions */ }
// suite de gestionnaires
finally { /* instructions à toujours éxecuter */ }
```

Exceptions : exemple 5

```
public class ExceptionStandard5
{ public static void main (String [] args)
  { int x = Integer.parseInt(args[0]) ;
    int y = Integer.parseInt(args[1]) ;
    try
    { int z = x / y ;
      System.out.println("z = x / y = "+z) ;
    }
    catch (ArithmeticException ae)
    { System.out.println("Division par zero !") ; }
    finally
    { System.out.println("Fin !") ; }
  }
}
---
> java ExceptionStandard5 6 0
Division par zero !
Fin !
> java ExceptionStandard5 6 3
z = x / y = 2
Fin !
```

Redéclenchement d'une exception : *throw*

Dans un gestionnaire d'exception, il est possible de demander qu'une exception soit retransmise à un niveau englobant grâce à l'instruction **`throw`**

```
try
{ // instructions
}
catch (ExException ee)
{ // instructions du gestionnaire
 throw ee ;
}
```

Cette possibilité est particulièrement utile lorsqu'on ne peut résoudre localement qu'une partie du problème posé

Exceptions : exemple 6

```
public class ExceptionStandard6
{ public static void main (String [] args)
  { int x = Integer.parseInt(args[0]) ;
    int y = Integer.parseInt(args[1]) ;
    try
    { int z = x / y ;
      System.out.println("z = x / y = "+z) ;
    }
    catch (ArithmeticException ae)
    { System.out.println("Division par zero !") ;
      throw ae ; // On "relance" l'exception au bloc
                 // try/catch englobant (mais où est-il ?)
    }
  }
}
---
> java ExceptionStandard5 6 0
Division par zero !
java.lang.ArithmeticException
   at ExceptionStandard1.main (ExceptionStandard1.java:6)
```

Bloc *try/catch* par défaut

```
public class ExceptionStandard6
{ public static void main (String [] args)
  { try  // Block try/catch par défaut
    { int x = Integer.parseInt(args[0]) ;
      int y = Integer.parseInt(args[1]) ;
      try
      { int z = x / y ;
        System.out.println("z = x / y = "+z) ;
      }
      catch (ArithmeticException ae)
      { System.out.println("Division par zero !") ;
        throw ae ; // On "relance" l'exception au bloc
                   // try/catch englobant
      }
    }
    catch (Throwable t)
    { System.err.println(t.printStackTrace()) ;
    }
  }
}
```

Création d'une exception

JAVA permet de définir ses propres exceptions, celles-ci dérivent obligatoirement de la super-classe **`Exception`**

```
class ErreurCoord extends Exception
{}
```

- Pour lancer une exception il sufit de fournir à l'instruction **`throw`** un objet du type de l'exception :

```
throw new ErreurCoord() ;
```

- Une méthode ou un constructeur susceptible de lancer une exception sans la traiter lui-même doit préciser quelle exception il peut lancer avec **`throws`**

```
Public point(int x, int y) throws ErreurCoord
```

Exemple : création d'une exception (1/2)

```
// Définition de notre exception ErreurCoord
class ErreurCoord extends Exception
{}

// Définition de la classe Point
class Point
{  // Variables, une coordonnée ne doit pas être négative !
   private int x, y ;

   // Méthodes
   public void affiche()
   { System.out.print ("Je suis en "+x+" "+y) ;
   }
  // Le constructeur peut lever une exception ErreurCoord
   public Point(int x, int y) throws ErreurCoord
   { if ((x<0) || (y<0))
        throw new ErreurCoord() ;
     this.x = x ; this.y = y ;
   }
}
```

Exemple : création d'une exception (2/2)

```
// Classe de test pour notre classe Point
public class TestPoint
{ public static void main(String args[] )
  { try
    { Point a = new Point (1,4) ;
      a.affiche() ;
      a = new Point (-3,5) ;
      a.affiche() ;
    }
    catch (ErreurCoord ec)
    { System.out.print ("Erreur construction") ;
      System.exit(1) ;
    }
  }
}
---
> java TestPoint
Je suis en 1 4
Erreur construction
```

Transmission d'information

Il est possible et souvent utile de transmettre de l'information au gestionnaire d'exception, cela est possible :

- Par le biais de l'objet fourni dans l'instruction **`throws`** : l'objet fournit à cette instruction sert à choisir le bon gestionnaire mais il est aussi récupéré sous la forme d'un argument et peut transmettre de l'information
- Par l'intermédiaire du constructeur de l'objet exception : puisque cet objet hérite de la super-classe *`Exception`* on peut utiliser par exemple un de ses constructeurs prennant un argument de type **`String`** et utiliser sa méthode *`getMessage`*

Exemple : transmission par *throws* (1/2)

```
// Définition de notre exception ErreurCoordInfo
class ErreurCoordInfo extends Exception
{ public int abs, ord ;
  public ErreurCoordInfo(int abs, int ord)
  { this.abs = abs ; this.ord = ord ; }
}

// Définition de la classe PointInfo
class PointInfo
{ // Variables, une coordonnée ne doit pas être négative !
  private int x, y ;
  // Méthodes
  public void affiche()
  { System.out.print ("Je suis en "+x+" "+y) ; }
  // Le constructeur peut lever une exception ErreurCoordInfo
  public PointInfo(int x, int y) throws ErreurCoordInfo
  { if ((x<0) || (y<0))
      throw new ErreurCoordInfo(x,y) ; // Infos au constructeur
    this.x = x ; this.y = y ;
  }
}
```

Exemple : transmission par *throws* (2/2)

```
// Classe de test pour notre classe PointInfo
public class TestPointInfo
{ public static void main(String args[] )
  { try
    { PointInfo a = new PointInfo (1,4) ;
      a.affiche() ;
      a = new PointInfo (-3,5) ;
      a.affiche() ;
    }
    catch (ErreurCoordInfo ecf)
    { System.out.print ("Erreur construction") ;
      System.out.print ("Coordonnees "+efc.abs+" "+efc.ord) ;
      System.exit(1) ;
    }
  }
}
---
> java TestPointInfo
Je suis en 1 4
Erreur construction
Coordonnees -3 5
```

Exemple : transmission par constructeur (1/2)

```
// Définition de notre exception ErreurCoordInfo2
class ErreurCoordInfo2 extends Exception
{ public ErreurCoordInfo2(String message)
   { super(message) ; }
}

// Définition de la classe PointInfo2
class PointInfo2
{ // Variables, une coordonnée ne doit pas être négative !
   private int x, y ;
   // Méthodes
   public void affiche()
   { System.out.print ("Je suis en "+x+" "+y) ; }
  // Le constructeur peut lever une exception ErreurCoordInfo2
   public PointInfo2(int x, int y) throws ErreurCoordInfo2
   { if ((x<0) || (y<0))
        throw new ErreurCoordInfo2("Erreur constr. "+x+" "+y) ;
     this.x = x ; this.y = y ;
   }
}
```

Exemple : transmission par constructeur (2/2)

```
// Classe de test pour notre classe PointInfo2
public class TestPointInfo2
{ public static void main(String args[] )
  { try
    { PointInfo2 a = new PointInfo2 (1,4) ;
      a.affiche() ;
      a = new PointInfo2 (-3,5) ;
      a.affiche() ;
    }
    catch (ErreurCoordInfo2 ecf2)
    { System.out.println (ecf2.getMessage()) ;
      System.exit(1) ;
    }
  }
}
---
> java TestPointInfo2
Je suis en 1 4
Erreur constr. -3 5
```

Choix du gestionnaire d'exception

- Lorsqu'une exception est lévée dans un bloc *try*, on recherche parmi les gestionnaires celui qui correspond à l'objet passé à throw
- On sélectionne le premier gestionnaire qui est soit du type exact de l'objet, soit d'un type de base (polymorphisme)

```
class ErreurPoint extends Exception {...}
class ErreurConst extends ErreurPoint {...}
class ErreurDepla extends ErreurConst {...}

try
{ /* Les 3 types d'exceptions peuvent être levées */ }
catch (ErreurConst ec)
{ /* Pour les exceptions ErreurConst et ErreurDepla */ }
catch (ErreurPoint ep)
{ /* Pour les exceptions ErreurPoint (autres déjà traitées) */ }
```

7. Bases de la programmation graphique

En mode texte, c'était le programme qui sollicitait l'utilisateur ; à travers les interfaces graphiques c'est l'utilisateur qui pilotera l'application

JAVA propose plusieurs bibliothèques standards pour la programmation graphique

AWT (Abstract Windowing Toolkit)

- Premier paquetage graphique, ancien, mieux supporté
- Seul paquetage possible pour les applets (mais...)

Swing

- Basé sur AWT, plus récent, mieux optimisé
- Plus riche (plus de widgets), plus beau !

Swing : les différents concepts

Réaliser une interface graphique fait intervenir plusieurs composants (toujours des classes) avec Swing :

- **Top-level** : c'est la racine de l'application graphique (JFrame, JDialog ou JApplet)
- **Conteneurs intermédiaires** : ils contiennent d'autres composants et structurent l'application graphique (JPanel, JScrollPane, JSplitPane, JToolBar...)
- **Composants atomiques** (JButton, JList, JLabel, JMenu...)
- **Layout-managers** : ils servent à décrire le placement des composants
- **Evénements** : ils permettent d'associer un traitement à une action

Une première fenêtre

```
import javax.swing.* ; // Import du paquetage Swing
public class PremFen
{ public static void main (String args[])
   { JFrame fen = new JFrame() ; // Création objet fenêtre
     fen.setSize(300,150) ; // Sinon créée avec taille nulle
     fen.setTitle("Ma premiere fenetre !") ; // Titre
     fen.setVisible(true) ; // Pour la rendre visible
   }
}
---
> java PremFen
```

Une première fenêtre personnalisée

```
import javax.swing.* ; // Import du paquetage Swing
class MaFenetre extends JFrame // Hérite de JFrame
{ public MaFenetre() // Constructeur
  { setSize(300,150) ;
    setTitle("Ma seconde fenetre !") ;
  }
}
public class PremFen2
{ public static void main (String args[])
  { JFrame fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
---
> java PremFen2
```

Les événements (1/2)

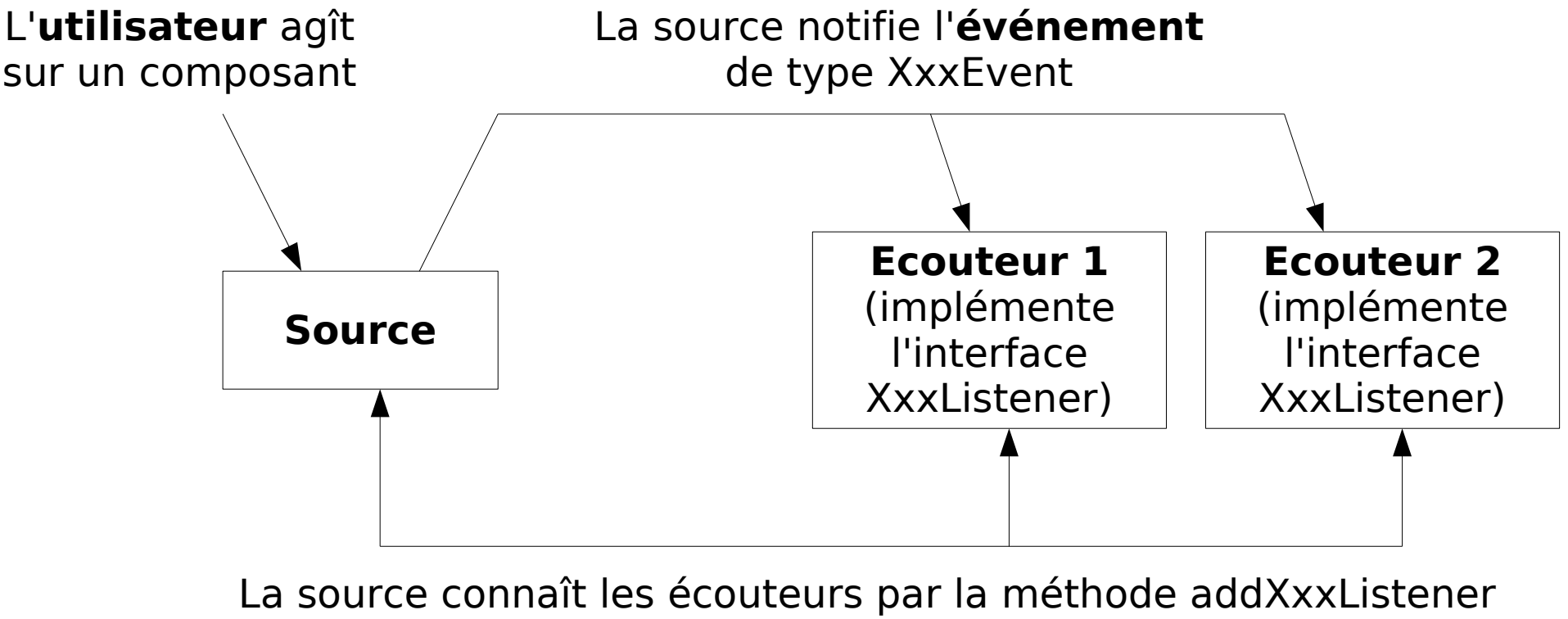
Les événements constituent la caractéristique principale d'une interface graphique

Ce sont des *objets* généralement créés par des composants atomiques qu'on aura introduit dans la fenêtre (boutons, menus, boîtes de dialogue...)

Les événements font entrer en relation deux types d'objets :

- **Les sources** : ce sont les objets ayant donné naissance à l'événement (bouton, fenêtre...)
- **Les écouteurs** : ce sont des objets associés aux sources dont la classe implémente une *interface* correspondant à une *catégorie d'événements*

Les événements (2/2)



Un premier événement : clic dans une fenêtre (1/3)

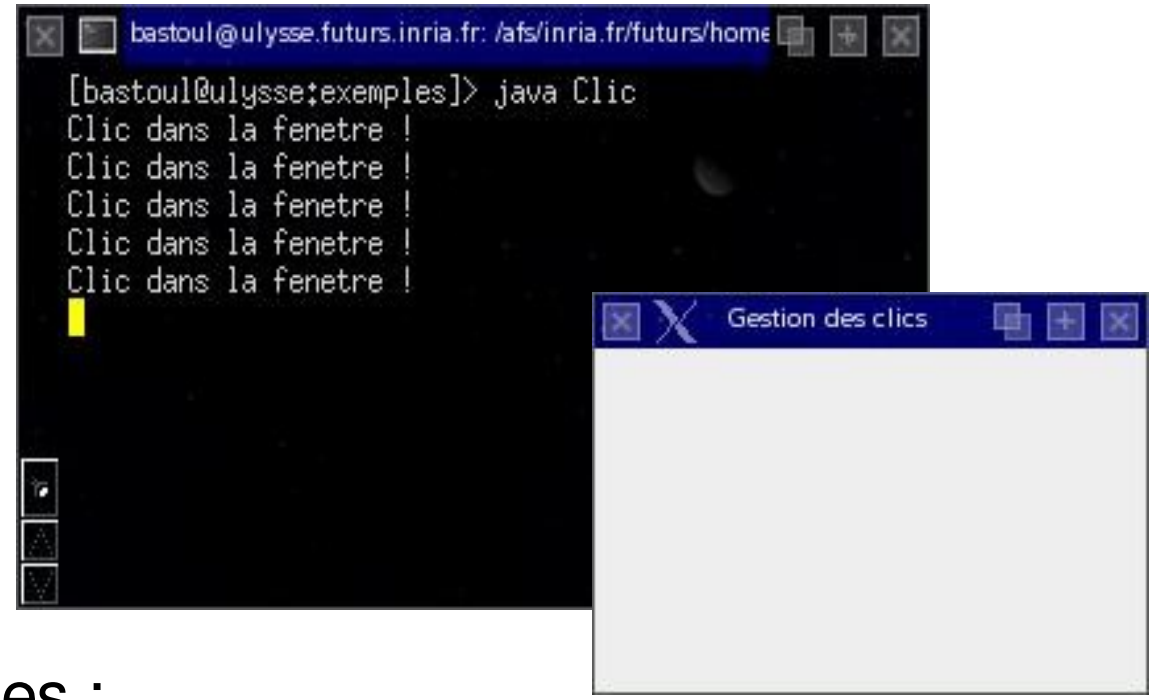
Nous souhaitons écrire un message à chaque fois qu'un clic survient dans notre fenêtre :

- La source est la fenêtre elle-même,
- La classe d'événements liée à la souris est *MouseEvent*
- L'interface *MouseListener* que l'écouteur doit implémenter comporte cinq méthodes correspondant chacune à un événement particulier
  - mousePressed (bouton pressé sur un composant)
  - mouseReleased (bouton relaché sur un composant)
  - mouseEntered (la souris entre dans un composant)
  - mouseExited (la souris sort d'un composant)
  - **mouseClicked** (bouton cliqué sur un composant)

Un premier événement : clic dans une fenêtre (2/3)

```
import javax.swing.* ;     // Pour JFrame
import java.awt.event.* ; // Pour MouseEvent et MouseListener
class MaFenetre extends JFrame implements MouseListener
{ public MaFenetre()
  { setSize(300,150) ;
    setTitle("Gestion des clics") ;
    addMouseListener(this) ; // Elle est son propre écouteur
  }
  public void mouseClicked (MouseEvent ev)     // gestion du clic
  { System.out.println("Clic dans la fenetre !") ; }
  public void mousePressed (MouseEvent ev) {} // méthode vide
  public void mouseReleased(MouseEvent ev) {} // méthode vide
  public void mouseEntered (MouseEvent ev) {} // méthode vide
  public void mouseExited  (MouseEvent ev) {} // méthode vide
}
public class Clic
{ public static void main (String args[])
  { JFrame fen = new MaFenetre() ; // Création objet fenêtre
    fen.setVisible(true) ; // Pour la rendre visible
  }
}
```

Un premier événement : clic dans une fenêtre (3/3)



Remarques :

- Nous pouvions aussi créer un objet *ObjetEcouteur* et en passer la référence à la méthode `AddMouseListener`
- Nous pouvions utiliser les attributs de l'événement :

```
System.out.println("clic "+ev.getX()+" "+ev.getY()) ;
```

Ecouteurs/adaptateurs

Avoir à implémenter toutes les méthodes de l'interface dans l'écouteur n'est pas toujours très pratique

Pour faciliter les choses, JAVA dispose toujours d'une classe qui implémente toutes les méthodes d'une interface donnée avec un corps vide : l'**adaptateur**

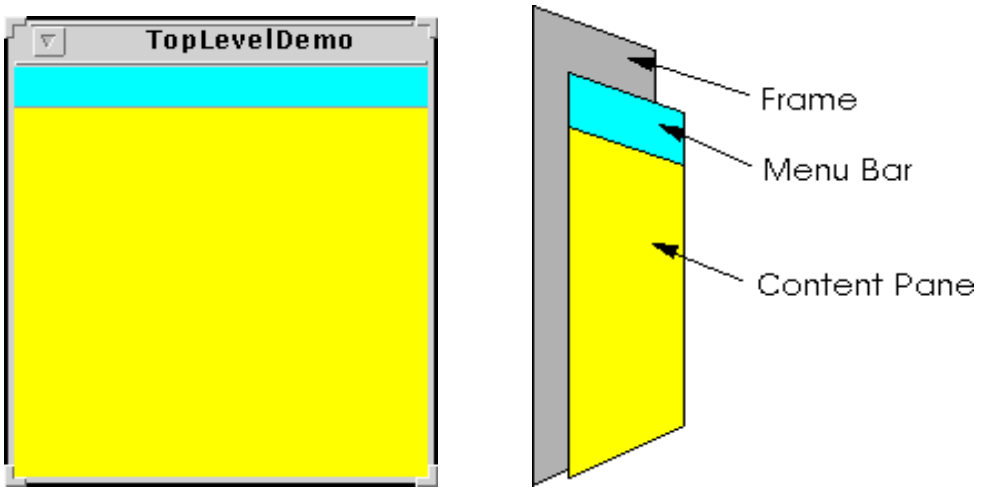
- Pour XxxEvent, l'adaptateur est XxxAdapter
- Si on veut comme dans l'exemple que la classe soit aussi l'écouteur il faudra utiliser une **classe anonyme**

```
class MaFenetre extends JFrame // Pas d'héritage multiple !
{ ...
  addMouseListener(new MouseAdapter()
                   { public void mouseClicked(MouseEvent ev)
                     {...}
                   }) ;
  ...
}
```

Les composants (1/2) où les placer ?

Il est possible de placer toute sortes de composants dans une fenêtre (textes, boutons...), il s'agît de définir **où** le placer et **comment** le placer

Les composants *top-level* (comme `JFrame`) possedent une barre de menu et un contenu (le *content-pane* de type `Container`) qui contient tous les composants visibles



La méthode `getContentPane` donne la référence au contenu où on pourra ajouter un composant avec la méthode `add`

Les composants (2/2) comment les placer ?

La disposition des composants dans une fenêtre est gérée par un gestionnaire de mise en forme (le *layout-manager*)

- Il en existe plusieurs utilisant des règles spécifiques pour disposer les composants
- Par défaut JAVA utilise `BorderLayout` avec lequel en l'absence d'informations un composant occupe toute la fenêtre
- pour choisir un gestionnaire, il faut appliquer la méthode `setLayout` à l'objet *contenu* de la fenêtre : pour utiliser par exemple `FlowLayout` (affichant les composants comme les mots d'un texte) on peut invoquer

```
getContentPane().setLayout(new FlowLayout()) ;
```

Un premier composant : un bouton (1/5)

Pour utiliser un composant bouton il s'agît tout d'abord de le créer et de le placer :

- On crée un objet bouton en utilisant le constructeur de la classe *JButton* à qui on donne le label du bouton

```
JButton monBouton ;
monbouton = new JButton("Un bouton")) ;
```

- On place ce bouton dans le contenu de la fenêtre en récupérant une référence vers ce contenu et en ajoutant le bouton avec la méthode add

```
Container contenu = getContentPane() ;
contenu.add(monBouton) ;
// ou directement getContentPane().add(monBouton) ;
```

- On indique comment disposer ce bouton avec le gestionnaire de mise en forme de notre choix

```
getContentPane().setLayout(new FlowLayout()) ;
```

Un premier composant : un bouton (2/5)

```
import javax.swing.* ; // Pour JFrame
import java.awt.* ;    // Pour Container et FLowLayout
class FenBouton1 extends JFrame
{ private JButton monBouton ; // Attribut bouton

  public FenBouton1()
  { setSize(300,150) ;
    setTitle("Premier bouton") ;
    monBouton = new JButton("Un bouton") ; // Création du bouton
    getContentPane().setLayout(new FlowLayout()) ; // Forme
    getContentPane().add(monBouton) ; // Ajout du bouton
  }
}
public class Bouton1
{ public static void main (String args[])
  { JFrame fen = new FenBouton1() ; // Création objet fenêtre
    fen.setVisible(true) ; // Pour la rendre visible
  }
}
```

Un premier composant : un bouton (3/5)

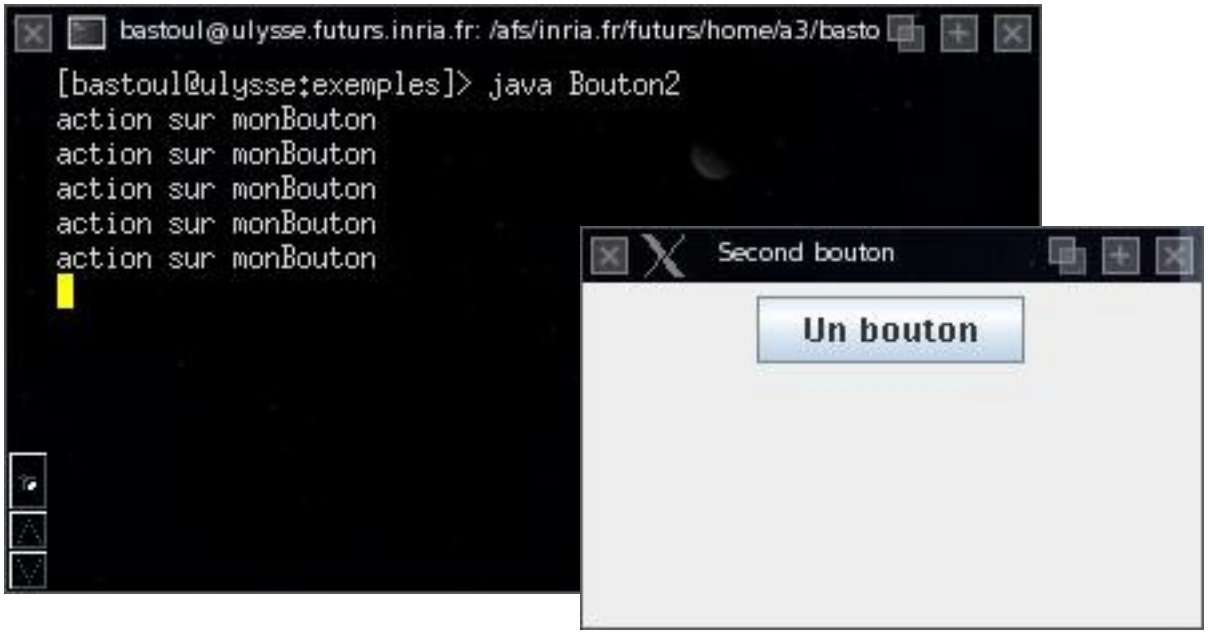
Le bouton dans cet exemple ne fait rien, l'intérêt d'un bouton sera d'utiliser le seul événement qu'il puisse créer, l'événement `Action`, il faudra donc

- Créer un écouteur qui sera un objet d'une classe implémentant l'interface `ActionListener`
- Associer cet écouteur au bouton par la méthode `addActionListener`

Un premier composant : un bouton (4/5)

```
import javax.swing.* ;      // Pour JFrame
import java.awt.* ;         // Pour Container et FLowLayout
import java.awt.event.* ;   // Pour ActionEvent et ActionListener
class FenBouton2 extends JFrame implements ActionListener
{ private JButton monBouton ; // Attribut bouton
  public FenBouton2()
  { setSize(300,150) ; setTitle("Second bouton") ;
    monBouton = new JButton("Un bouton") ;
    getContentPane().setLayout(new FlowLayout()) ;
    getContentPane().add(monBouton) ;
    monBouton.addActionListener(this) ; // Lien avec l'écouteur
  }
  public void actionPerformed (ActionEvent ev)
  { System.out.println("action sur monBouton") ; }
}
public class Bouton2
{ public static void main (String args[])
  { JFrame fen = new FenBouton2() ; // Création objet fenêtre
    fen.setVisible(true) ;          // Pour la rendre visible
  }
}
```

Un premier composant : un bouton (5/5)



Remarques :

- Si on cherche à ajouter le bouton sur la fenêtre et non sur son contenu, une erreur surviendra
- Pour avoir l'esprit tranquille avec les importations de paquetages, toujours importer les trois utilisés dans cet exemple pour la programmation graphique

Gérer plusieurs composants (1/3)

En général une fenêtre disposera de plusieurs composants du même genre (ou pas) et on souhaitera un traitement différent d'un même type d'événements en fonction du composant qui l'a crée

JAVA offre différentes possibilités pour cela :

- Créer une classe écouteur pour chaque composant dont l'implémentation de l'interface sera différente
- Utiliser la méthode `getSource`, qui fournit une référence sur l'objet ayant déclenché l'événement, pour identifier l'émetteur et agir en conséquence
- Utiliser les attributs de l'évènement, par exemple la méthode `getActionCommand` permettra pour un bouton de récupérer son label et de l'identifier

Gérer plusieurs composants (2/3)

```
import javax.swing.* ;      // Pour JFrame
import java.awt.* ;         // Pour Container et FLowLayout
import java.awt.event.* ;   // Pour ActionEvent et ActionListener
class FenBouton3 extends JFrame implements ActionListener
{ private JButton bouton1, bouton2 ; // Attributs boutons
  public FenBouton3()
  { setSize(300,150) ; setTitle("Deux boutons") ;
    bouton1 = new JButton("Bouton 1") ;
    bouton2 = new JButton("Bouton 2") ;
    getContentPane().setLayout(new FlowLayout()) ;
    getContentPane().add(bouton1) ;    // Placement des boutons
    getContentPane().add(bouton2) ;
    bouton1.addActionListener(this) ; // Lien des  boutons
    bouton2.addActionListener(this) ;
  }
  public void actionPerformed (ActionEvent ev)
  { if (ev.getSource() == bouton1)
    // Ou encore if (ev.getActionCommand() == "Bouton 1")
    System.out.println("action sur bouton1") ;
    else
    System.out.println("action sur bouton2") ;
  }
}
```

Gérer plusieurs composants (3/3)

```
public class Bouton3
{ public static void main (String args[])
  { JFrame fen = new FenBouton3() ; // Création objet fenêtre
    fen.setVisible(true) ;          // Pour la rendre visible
  }
}
```

Éléments de base

Top-level:

- JFrame
- JDialog
- JApplet

Conteneurs

- JPanel
- JScrollPane
- JSplitPane
- JTabbedPane
- JToolBar
- Etc.

Composants atomiques

- JButton
- JCheckBox
- JList
- JSlider
- JTextField
- JLabel
- Etc.

Layout :

- BorderLayout
- BoxLayout
- CardLayout
- Etc.

Bases du dessin en JAVA

JAVA permet de dessiner sur n'importe quel composant en utilisant des fonctions de dessin

Pour éviter que le dessin ne disparaisse après un réaffichage de la fenêtre (par exemple après un déplacement) il faut le placer dans une méthode particulière du composant appelée `paintComponent`

- Ceci n'est pas indispensable, mais le plus pratique
- Il est possible mais pas recommandé de dessiner directement dans la fenêtre, dans sa méthode `paint` (et non `paintComponent` réservé aux composants)
- Pour faire un dessin seul, on le placera dans un composant panneau (un objet de la classe `Jpanel` ou dérivé)

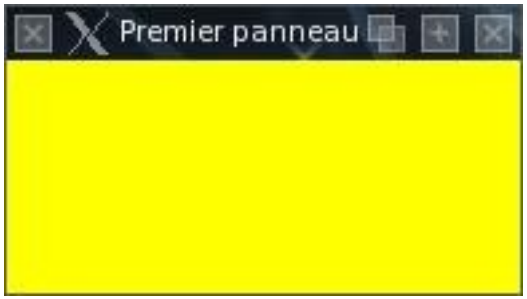
Création d'un panneau (1/2)

Un panneau est un *conteneur* : il doit être attaché à un autre conteneur (par exemple le *contenu* de la fenêtre) et peut contenir soit d'autres conteneurs, soit des composants atomiques (boutons, textes...)

- Par défaut un panneau n'a pas de bordure et sa couleur est celle du conteneur (il est invisible)
- On peut utiliser sa methode `setBackground` recevant un objet de type `Color` (il existe quelques couleurs constantes comme `Color.yellow` pour le jaune)
- Comme les composants, il s'attache au *contenu* par la méthode `add`

Création d'un panneau (2/2)

```
import javax.swing.* ;
import java.awt.* ;
class FenPanneau1 extends JFrame
{ private JPanel monPanneau ; // Attribut panneau
  public FenPanneau1()
  { setSize(300,150) ;
    setTitle("Premier panneau") ;
    monPanneau = new JPanel() ; // Création du panneau
    monPanneau.setBackground(Color.yellow) ; // Invisible sinon
    getContentPane().add(monPanneau) ; // Ajout du panneau
  }
}
public class Panneau1
{ public static void main (String args[])
  { JFrame fen = new FenPanneau1() ;
    fen.setVisible(true) ;
  }
}
```



Dessin dans un panneau

Pour créer un dessin permanent dans un composant, il faut redéfinir sa méthode *paintComponent*

- Puisqu'il s'agît de redéfinir une méthode de *JPanel*, il faut créer un objet qui hérite de *Jpanel*
- La méthode à redéfinir possède l'entête suivant :

  public void *paintComponent (Graphics g)*

  son argument est appelé un *contexte graphique*. c'est elle qui dispose des méthodes de dessin et des paramètres courants (police, couleur de fond etc.)
- La méthode *paintComponent* de *Jpanel* dessinait le composant, il est nécessaire de l'appeler **avant** de réaliser ses propres dessins par

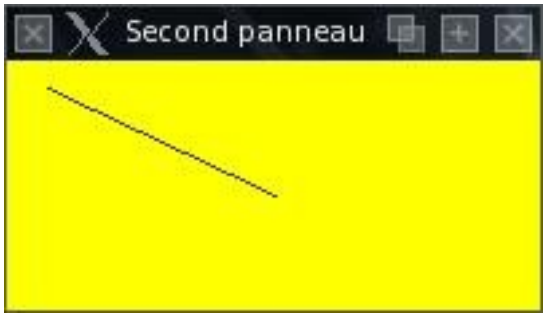
  super.*paintComponent(g) ;*

Dessin dans un panneau : exemple (1/2)

```
import javax.swing.* ;
import java.awt.* ;
class Panneau extends JPanel     // Notre propre panneau
{ public void paintComponent(Graphics g)
  { super.paintComponent(g) ;   // Appel de la fonction du père
    g.drawLine(15,10,100,50) ; // Trait depuis coords 15, 10
                               // Jusqu'à coords 15+100, 10+50
  }
}
class FenPanneau2 extends JFrame
{ private Panneau monPanneau ; // Attribut panneau
  public FenPanneau2()
  { setSize(300,150) ;
    setTitle("Second panneau") ;
    monPanneau = new Panneau() ; // Création du panneau
    monPanneau.setBackground(Color.yellow) ; // Invisible sinon
    getContentPane().add(monPanneau) ; // Ajout du panneau
  }
}
```

Dessin dans un panneau : exemple (2/2)

```
public class Panneau2
{ public static void main (String args[])
  { JFrame fen = new FenPanneau2() ;
    fen.setVisible(true) ;
  }
}
```



Remarques :

- Dans cet exemple, le dessin est défini au début du programme et il n'est pas possible de le modifier
- Aucun gestionnaire de mise en forme n'a été spécifié, *BorderLayout* est donc utilisé par défaut et le panneau occupe toute la fenêtre

Forcer le dessin

Pour changer un dessin en cours d'exécution il faudra forcer l'appel à *paintComponent*

- C'est l'appel à la méthode *repaint* qui force l'appel à *paintComponent*
- La méthode *paintComponent* devra réagir en fonction du contexte

Pour dessiner à la volée (au fur et à mesure des actions de l'utilisateur) :

- La permanence des dessins n'est plus assurée
- Il faut récupérer le contexte graphique du composant à redessinner grâce à la méthode *getGraphics*
- Il faudra libérer les ressources occupées par le contexte graphique par la méthode *dispose*

Forcer le dessin : exemple (1/3)

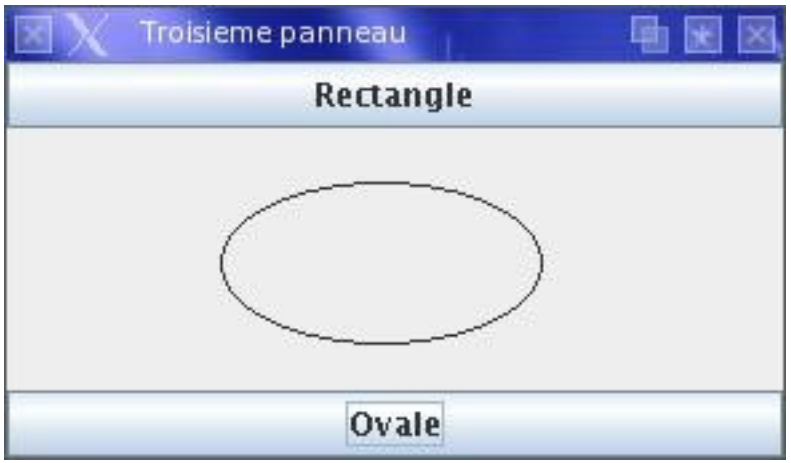
```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class Panneau extends JPanel    // Notre propre panneau
{ private boolean rectangle = false, // Notre "contexte"
                  ovale     = false ;

  public void setRectangle()
  { rectangle = true ; ovale = false ; }

  public void setOvale()
  { rectangle = false ; ovale = true ; }

  public void paintComponent(Graphics g)
  { super.paintComponent(g) ;  // Appel de la fonction du père
    if (ovale)                 // Dessin en fonction du contexte
      g.drawOval(80,20,120,60) ;
    if (rectangle)
      g.drawRect(80,20,120,60) ;
  }
}
```

Forcer le dessin : exemple (2/3)

```
class FenPanneau3 extends JFrame implements ActionListener
{ private Panneau monPanneau ;                  // Attribut panneau
  private JButton rectangle, ovale ;            // Attributs boutons
  public FenPanneau3()
  { setSize(300,170) ;
    setTitle("Troisieme panneau") ;
    Container contenu = getContentPane() ;
    monPanneau = new Panneau() ;                // Creation du panneau
    contenu.add(monPanneau) ;
    rectangle = new JButton("Rectangle") ;      // Bouton rectangle
    contenu.add(rectangle,"North") ;            // Voir BorderLayout
    rectangle.addActionListener(this) ;
    ovale = new Bbutton("Ovale") ;              // Bouton ovale
    contenu.add(ovale,"South") ;                // Voir BorderLayout
    ovale.addActionListener(this) ;
  }
  public void actionPerformed(ActionEvent ev)
  { if (ev.getSource()==rectangle) monPanneau.setRectangle() ;
    if (ev.getSource()==ovale) monPanneau.setOvale() ;
    monPanneau.repaint() ; // Pour redessiner le panneau
  }
}
```

Forcer le dessin : exemple (3/3)

```
public class Panneau3
{ public static void main (String args[])
  { JFrame fen = new FenPanneau3() ;
    fen.setVisible(true) ;
  }
}
```



Quelques méthodes de dessin

- *drawLine(x1,y1,x2,y2)* trace une ligne, les paramètres étant les coordonnées des deux sommets
- *drawRect(x,y,largeur,hauteur)* trace un rectangle, les deux premier paramètres sont les coordonnées du coin supérieur gauche
- *drawOval(x,y,largeur,hauteur)* même chose que pour le rectangle mais dessine un ovale
- *drawRoundRect(x,y,largeur,hauteur,diamh,diamv)* trace un rectangle aux bords arrondis, les deux derniers paramètres sont le diamètre horizontal et vertical de l'arc aux quatre coins
- Et aussi *drawArc*, *drawPolygon*, *drawPolyLine* etc.

8. Généricité

La *généricité* permet en JAVA de paramétrer une classe avec un ou plusieurs types de données

Exemple préliminaire :

```
class Generique <T> // Classe générique paramétrée par T
{ private T x ; // Attribut de type paramétré
  public T getX() // Methode au type de retour paramétré
  { return x ;
  }
}
public class Test
{ public static void main (String args[])
  { Generique <Integer> g = new Generique <Integer> () ;
    Integer i = g.getX() ;
    Generique <String> s = new Generique <String> () ;
    String j = s.getX() ;
  }
}
```

Pourquoi la généricité ? (1/2)

Rappel : le *polymorphisme* permet d'affecter à une variable de type A une valeur de type B, si B est un sous-type de A

- Une sous-classe est un sous-type de ses super-classes
- Une classe est un sous-type d'une interface qu'elle implémente
- Une classe A est un sous-type de A

- La solution pour créer des méthodes applicables à plusieurs types différents est de manipuler un type commun dans l'arbre d'héritage (ou de faire une méthode pour chaque type !)
- En dehors des tableaux, les *collections* de JAVA jusqu'à 1.4 (des classes destinées à la manipulation d'objets, par exemple les classes `Vector` ou `ArrayList`), manipulaient des `Object` pour un maximum de flexibilité

Pourquoi la généricité ? (2/2)

La manipulation systématique d'objets instances de la classe `Object` (ou de parents dans l'arbre d'héritage) présente de nombreux inconvénients :

- Il est impossible de limiter à certains types seulement (en particulier pour les collections de JAVA 1.4)
- Il est nécessaire de *caster* les éléments pour utiliser les méthodes qui ne sont pas dans `Object`
- Certaines erreurs ne peux être détectées qu'à l'exécution (il faut attendre ce moment pour savoir si un objet correspond bien au type attendu par un *cast*)
- Pour éviter ces problèmes, il faut écrire un code différent pour chaque type : on perd du temps est c'est MAL (très dangeureux pour la correction de bugs)

Pourquoi la généricité... Exemples (1/3)

Cas d'une collection standard

La classe standard `ArrayList` implémente une liste à la manière d'un tableau, parmis ses méthodes on trouve :

- `Boolean add(Object nouveau)` qui ajoute l'objet `nouveau` à la liste et renvoie une information sur le déroulement de l'opération
- `Object get(int i)` retourne le i-ème objet de la liste (qui commence à 0)
- `int size()` qui retourne le nombre d'objets dans la liste

Cette classe existait avant JAVA 1.5 elle est par conséquent toujours utilisable sans généricité pour des raisons de compatibilité des machines virtuelles

Pourquoi la généricité... Exemples (2/3)

Obligation de *cast*

```
public class DemoCast
{ public static void main (String args[])
  { ArrayList chenil = new ArrayList() ;
    Chien medor = new Chien("Medor") ;
    Chien fifi  = new Chien("fifi") ;
    // La classe ArrayList ne contient que des Object
    chenil.add(medor) ;
    chenil.add(fifi) ;

    // Dans la boucle suivante, le cast (Chien) est indispensable
    // pour accéder à la méthode aboie() qui n'est pas dans
    // la classe Object
     for (int i=0; i<chenil.size(); i++)
     ((Chien)chenil.get(i)).aboie() ;
  }
}
```

Pourquoi la généricité... Exemples (3/3)

Erreur à l'exécution

```
public class DemoErreurExecution
{ public static void main (String args[])
  { ArrayList chenil = new ArrayList() ;
    Chien medor = new Chien("Medor") ;
    Chien fifi  = new Chien("fifi") ;
    Chat  miaou = new Chat("Miaou") ;
    // La classe ArrayList ne contient que des Object, les Chien
   // comme les Chat sont acceptés
    chenil.add(medor) ;
    chenil.add(miaou) ;
    chenil.add(fifi) ;

   // La méthode aboie() n'existe pas pour les Chat, on ne se
   // rendra compte que le cast de l'Object en Chien n'est pas
   // possible qu'à l'exécution
    for (int i=0; i<chenil.size(); i++)
    ((Chien)chenil.get(i)).aboie() ;
  }
}
```

Généricité : utilité

- La généricité permet de paramétrer des *classes* par des types
- La généricité permet de paramétrer des *méthodes* par des types
- Ainsi :
  - Le code n'a pas à être dupliqué
  - Le typage est vérifié à la compilation (on parle de vérification statique)
  - Les cast ne sont plus nécessaires (en fait ils sont faits automatiquement par le compilateur)

Paramétrer une classe ou une interface

- Le nom de la classe ou de l'interface est suivi d'un ensemble de paramètres de la forme `<T1,T2,...,Tn>`
- Les `Ti` sont les paramètres : des types inconnus au moment de la compilation
- Les `Ti` peuvent être des classes (abstraites ou non) ou des interfaces
- Les `Ti` ne peuvent pas être un type primitif (`int`, `float`...)
- On parle de *types génériques* pour les classes ou interface paramétrées

Comment et où se servir des paramètres

Les paramètres de type peuvent apparaître sous la forme Ti dans le code du type générique :

- Pour déclarer :
  - Des variables
  - Des paramètres
  - Des tableaux
  - Ds types de retour de méthodes
- Pour caster avec un paramètre de type
- MAIS PAS pour créer des objets ou des tableaux

Un exemple (1/2)

```
class Couple <T> // La classe Couple est paramétrée par T
{ private T a ;
  private T b ;

  public Couple(T a, T b)
  { this.a = a ;
    this.b = b ;
  }
  public void echange()
  { T temp = a ;
    a = b ;
    b = temp ;
  }
  public String toString() { return (a+" , "+b) ; }
  public T getA() { return a ; }
  public T getB() { return b ; }
  public void setA(T a) { this.a = a ; }
  public void setB(T b) { this.b = b ; }
}
```

Un exemple (2/2)

```
public class TestGenericite
{ public static void main (String args[])
  { // Déclaration et création d'un objet de type Couple dont le
    // paramètre de type vaut String
    Couple<String>c = new Couple<String>("un","deux") ;

    // Idem pour Integer
    Couple<Integer>d = new Couple<Integer>(new Integer(5),
                                           new Integer(6)) ;
    d.echange() ;
    Integer i = d.getA() ;

    // On peut toujours profiter du polymorphisme
    Couple<Object>o = new Couple<Object>(new Integer(5),
                                         new String("one"));
    Object p = c.getA() ;
  }
}
```

Oui, mais...

Soient les déclarations suivantes :

```
Couple<Object> o = new Couple<Object> (new Object(),
                                       new Object());
Couple<Integer>i = new Couple<Integer>(new Integer(5),
                                       new Integer(6)) ;
```

L'affectation suivante est incorrecte :

```
o = i ; // ERREUR !!!
```

Un `Couple<Object>` ne peut recevoir qu'un `Couple<Object>` et de manière générale, un type paramétré ne peut recevoir qu'un objet du même type paramétré

- C'est une forte limitation, la flexibilité de la généricité est clairement amputée
- C'est particulièrement limitant pour les collections

On verra plus loin des solutions !

Méthodes génériques

De même qu'une classe ou une interface, une simple méthode peut être paramétrée par un type

- Toute classe générique ou non peut inclure des méthodes génériques
- Le type de retour de la méthode est précédé par un ensemble de paramètres de la forme `<T1,T2,...,Tn>`
- Les `Ti` ont les mêmes propriétés que pour les classes et interfaces génériques

Utilisation des méthodes génériques

- L'appel à une méthode générique est précédé par le ou les types devant remplacer les paramètres
- Le ou les types sont entourés de < et > par exemple : `<Type>nomMethode(...)`
- Le compilateur peut le plus souvent deviner les types en fonction de l'appel de la méthode

```
// Signature de la méthode :
public <T> T choisir(T a, T b)
// Ici le compilateur sait que leparamètre de type est Integer
Integer i = choisir(new Integer(5), new Integer(6)) ;
```

- Dans les cas litigieux le compilateur choisit au mieux si c'est possible

```
// Ici il choisit Number, super-classe de Integer et Double
Number n = choisir(new Integer(5), new Double(3.14)) ;
```

Exemple de méthode générique

```
import java.util.Random ;
class Utilitaire // Une classe non générique
{ public static <T> T choisir(T a, T b) // Méthode générique
  { int choix = (new Random()).nextInt(2) ; // Nb aléatoire 0 ou 1
    if (choix == 0)
    return (a) ;
    else
    return (b) ;
  }
  public static <T> void print(T a)
  { System.out.println("valeur = "+a) ;
  }
}
public class TestMethodeGenerique
{ public static void main (String args[])
  { Integer i1 = new Integer(5) ;
    Integer i2 = new Integer(4) ;
    Utilitaire.print(Utilitaire.choisir(i1,i2)) ;
  }
}
```

Instanciations avec jokers

Pour offrir une meilleure flexibilité pour les classes et interfaces génériques, JAVA intègre la notion de *joker* (*wildcard* en anglais)

- Il s'agît de ne plus contraindre les types paramétrés à n'être instanciés que par ce type paramétré (voir le problème de `Couple<Object>` et `Couple<Integer>` quelques transparents plus tôt)
- Il suffit d'utiliser comme "type" le caractère "`?`"
- On ne pourra pas utiliser de joker pour créer des objets, ou pour préciser un paramètre de type
- Plusieurs précisions sur la nature du joker seront possibles

Le joker et ses précisions

- `<?>` désigne un type fixé mais inconnu
- `<? extends A>` désigne un type fixé mais *inconnu* qui est `A` ou un sous-type de `A`
- `<? super A>` désigne un type fixé mais *inconnu* qui est `A` ou un sur-type de `A`
- `A` peut être une classe, une interface ou un paramètre de type
- On dit que `A` *contraint* le joker
- Un seul type est possible après `extends` ou `super`

Exemple d'utilisation de joker

En reprenant un exemple de classe `Couple` précédent et en considérant les déclarations suivantes :

```
Couple<Double> d = new Couple<Double> (new Double(),
                                       new Double());
Couple<Integer>i = new Couple<Integer>(new Integer(5),
                                       new Integer(6)) ;
```

L'affectation suivante était incorrecte :

```
d = i ; // ERREUR !!!
```

Elle le devient en modifiant les instructions ainsi :

```
Couple<? extends Number> d =
         new Couple<Double> (new Double(),new Double());
Couple<? extends Number> i =
         new Couple<Integer>(new Integer(),new Integer());
```

Restrictions à l'utilisation du joker

Le compilateur ne sera pas toujours en mesure d'être convaincu de la validité du code, si ce n'est pas le cas il ne l'acceptera pas

Prennons l'exemple d'une affectation

```
T a = b ;
```

- Le compilateur devrait connaître un sous-type de `T` car l'affectation n'est valable que si `b` est d'un sous-type du type de `a` ou si `a` et `b` ont un super-type commun `X` (dans ce cas, `T` devrait être `<? super X>`)
- Si le compilateur connaît un super-type du type de `b` (le type de `b` est `<? extends X>`) l'affectation ne sera acceptée que si `T` est un super-type du type de `b`

FIN

Bon courage pour la suite !

Pour réviser ou aller plus loin je recommande l'excellent livre de Claude Delannoy "Programmer en JAVA"

Les différentes sources suivantes ont été utilisées pour préparer ce cours :

- Programmer en JAVA de Claude Delannoy
- Cours de JAVA de l'IUT d'Orsay de Nicole Polian
- Cours de JAVA de l'Univ. de Nice de Richard Grin
- Tutoriels JAVA http://java.sun.com par SUN

9. Les flux

Un **flux** est un canal de transmission de données à partir d'une *source* vers une *destination* donnée ; on distingue en JAVA les **flux d'entrée** et les **flux de sortie**

- Le flux d'entrée désigne un canal susceptible d'émettre de l'information (exemple : le clavier `in`)

- Le flux de sortie désigne un canal susceptible de recevoir de l'information (exemple : l'écran `out`)

Sources et destinations peuvent être des fichiers, des processus, des URL, des chaînes etc. etc.

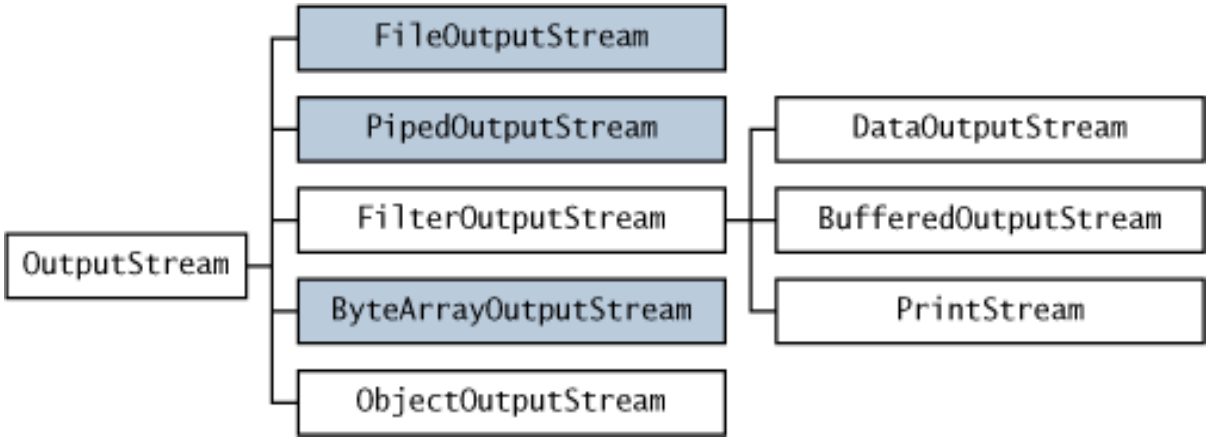
Familles de flux

On distingue deux types de flux en JAVA : les **flux d'octets** et les **flux de caractères**

- Dans le cas du flux d'octets, l'information est transmise par octets sans modifications de la mémoire au flux ou du flux à la mémoire, les deux classes de base (abstraites) sont:
  - `InputStream`
  - `OutputStream`
- Dans le cas du flux de caractères, l'information subit un formatage pour recevoir ou transmettre une suite de caractères, les deux classes de base (abstraites) sont:
  - `Reader`
  - `Writer`

Flux d'octets en écriture

Hiérarchie des classes dédiées aux flux d'octets en écriture : en gris les classes permettant l'écriture, en blanc celles offrant des possibilités de traitement avancées



Ecriture d'un fichier binaire (1/2)

La classe `FileOutputStream` permet de manipuler un flux binaire associé à un fichier en écriture, un de ses constructeurs s'utilise de cette façon :

```
FileOutputStream f = new FileOutputStream("entier.dat") ;
```

Cette instruction associe `f` à un fichier de nom `entier.dat`, si le fichier existe il est écrasé, sinon il est créé

Les méthodes de `FileOutputStream` sont limitées (envoi d'un seul octet ou d'un tableau d'octets sur le flux)

La classe `DataOutputStream` propose des méthodes pour l'écriture d'un entier (`writeInt`), d'un flottant (`writeFloat`) etc. ; un de ses constructeurs s'utilise ainsi :

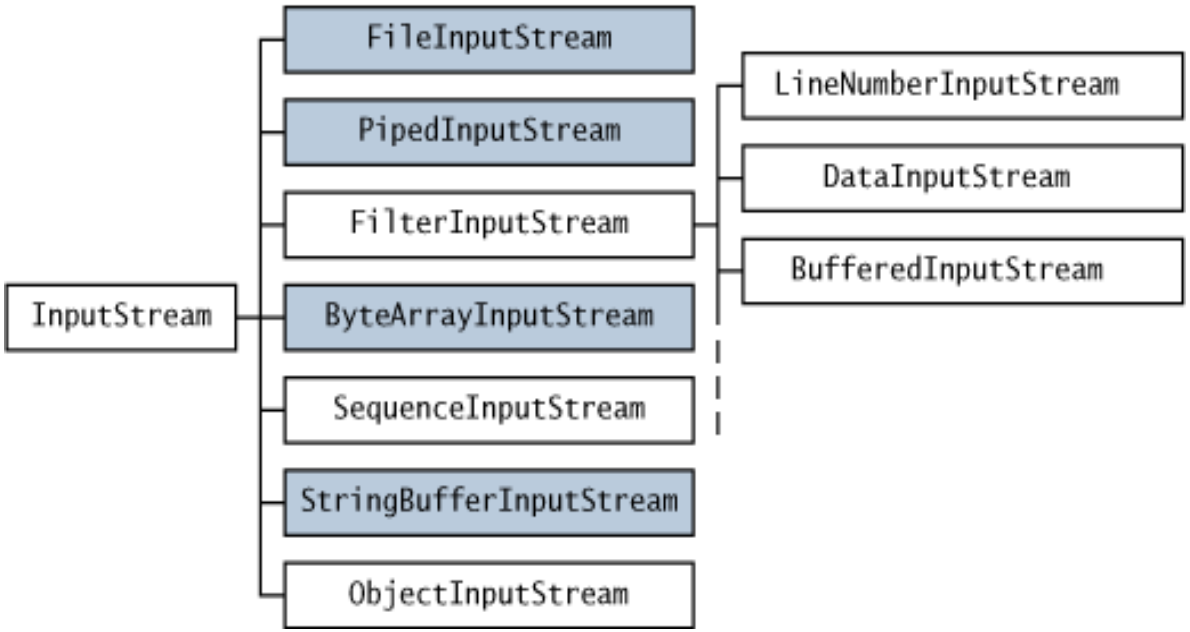
```
DataOutputStream sortie = new DataOutputStream(f) ;
```

Ecriture d'un fichier binaire (2/2)

```
import java.io.* ;    // Pour les classes flux
import java.util.* ; // Pour la classe Random
public class EcritureFichierEntier
{ public static void main(String args[])
  { try
    { FileOutputStream f = new FileOutputStream("entier.dat") ;
      DataOutputStream sortie = new DataOutputStream(f) ;
      int entier = (new Random()).nextInt() ; // Entier aléatoire
      sortie.writeInt(entier) ; // Ecriture de l'entier dans le fichier
      sortie.close() ; // Fermeture du flux
    }
    catch (FileNotFoundException ev) // Levé par FileOutputStream
    { System.out.println("Probleme de creation de fichier !") ;
      System.exit(1) ;
    }
    catch (IOException ev) // Levé par writeInt
    { System.out.println("Probleme d'ecriture !") ;
      System.exit(1) ;
    }
  }
}
```

Flux d'octets en lecture

Hiérarchie des classes dédiées aux flux d'octets en lecture : en gris les classes permettant la lecture, en blanc celles offrant des possibilités de traitement avancées



Lecture d'un fichier binaire (1/2)

La classe `FileInputStream` permet de manipuler un flux binaire associé à un fichier en lecture, un de ses constructeurs s'utilise de cette façon :

```
FileInputStream f = new FileInputStream("entier.dat") ;
```

Cette instruction associe `f` à un fichier de nom `entier.dat`, si le fichier n'existe pas, une exception de type `FileNotFoundException` (dérivée de `IOException`) est levée

Les méthodes de `FileInputStream` sont limitées (lecture d'un seul octet ou d'un tableau d'octets sur le flux)

La classe `DataInputStream` propose des méthodes pour la lecture d'un entier (`readInt`), d'un flottant (`readFloat`) etc. ; un de ses constructeurs s'utilise ainsi :

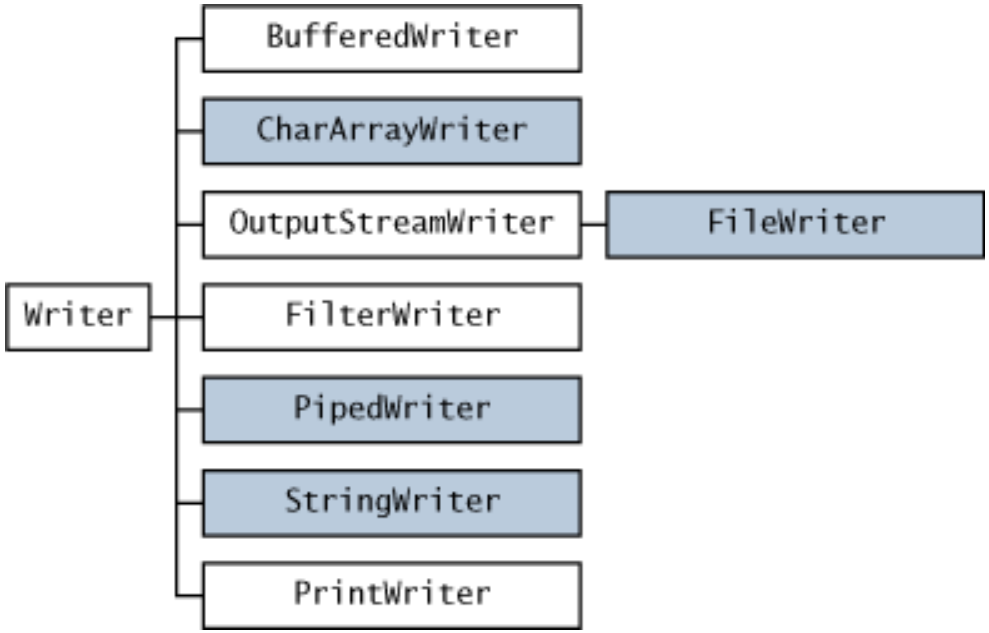
```
DataInputStream sortie = new DataInputStream(f) ;
```

Lecture d'un fichier binaire (2/2)

```
import java.io.* ;    // Pour les classes flux
public class LectureFichierEntier
{ public static void main(String args[])
  { try
    { FileInputStream f = new FileInputStream("entier.dat") ;
      DataInputStream entree = new DataInputStream(f) ;
      int entier = entree.readInt() ; // Lecture de l'entier
      entree.close() ; // Fermeture du flux
      System.out.println("entier :"+entier) ;
    }
    catch (FileNotFoundException ev) // Levé par FileInputStream
    { System.out.println("Pas de fichier !") ;
      System.exit(1) ;
    }
    catch (IOException ev) // Levé par readInt
    { System.out.println("Probleme de lecture !") ;
      System.exit(1) ;
    }
    // NB: readInt lève EOFException quand il arrive en fin de fichier
  }
}
```

Flux de caractères en écriture

Hiérarchie des classes dédiées aux flux de caractères en lecture : en gris les classes permettant l'écriture, en blanc celles offrant des possibilités de traitement avancées



Ecriture d'un fichier texte (1/2)

La classe `FileWriter` permet de manipuler un flux de caractères associé à un fichier en écriture, un de ses constructeurs s'utilise de cette façon :

```
FileWriter f = new FileWriter("carre.txt") ;
```

Cette instruction associe `f` à un fichier de nom `carre.txt`, si le fichier existe il est écrasé, sinon il est créé

Les méthodes de `FileWriter` permettent d'envoyer des caractères, des chaînes ou des tableaux de caractères

Si on souhaite formater avec les méthodes `print` ou `println`, comme lors de l'affichage d'informations, ces méthodes sont fournies par la classe `PrintWriter` ; un de ses constructeurs s'utilise ainsi :

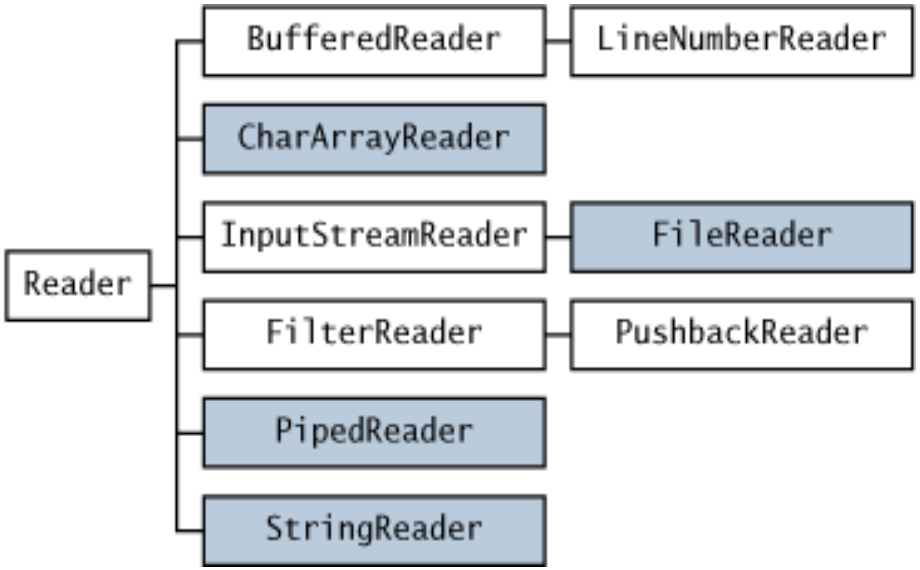
```
PrintWriter sortie = new PrintWriter(f) ;
```

Ecriture d'un fichier texte (2/2)

```
import java.io.* ;    // Pour les classes flux
import java.util.* ; // Pour la classe Random
public class EcritureFichierCarre
{ public static void main(String args[])
  { try
    { FileWriter f = new FileWriter("carre.txt") ;
      PrintWriter sortie = new PrintWriter(f) ;
      int n = (new Random()).nextInt() ; // Entier aléatoire
      sortie.println(n+" a pour carre "+n*n) ; // Ecriture fichier
      sortie.close() ; // Fermeture du flux
    }
    catch (IOException ev) // Pour rattraper toutes les exceptions IO
    { System.out.println("Probleme d'ecriture !") ;
      System.exit(1) ;
    }
  }
}
```

Flux de caractères en lecture

Hiérarchie des classes dédiées aux flux de caractères en lecture : en gris les classes permettant la lecture, en blanc celles offrant des possibilités de traitement avancées



Lecture d'un fichier texte (1/2)

La classe `FileReader` permet de manipuler un flux de caractères associé à un fichier en lecture, un de ses constructeurs s'utilise de cette façon :

```
FileReader f = new FileReader("carre.txt") ;
```

Cette instruction associe `f` à un fichier de nom `carre.txt`, si le fichier n'existe pas, une exception de type `FileNotFoundException` (dérivée de `IOException`) est levée

Les méthodes de `FileReader` permettent seulement de lire des caractères, on devrait donc gérer la fin de ligne

Il n'existe pas de symétrique de `PrintWriter`, mais nous pouvons utiliser la classe `BufferedReader` qui possède une méthode `readLine` pour lire toute une ligne, un de ses constructeurs s'utilise ainsi :

```
BufferedReader entree = new BufferedReader(f) ;
```

Lecture d'un fichier texte (2/2)

```
import java.io.* ;    // Pour les classes flux
public class LectureFichierCarre
{ public static void main(String args[])
  { try
    { FileReader f = new FileReader("carre.txt") ;
      BufferedReader entree = new BufferedReader(f) ;
      String ligne = entree.readLine() ; // Lecture fichier
      System.out.println("Ligne lue: "+ligne) ;
      entree.close() ; // Fermeture du flux
    }
    catch (IOException ev) // Pour rattraper toutes les exceptions IO
    { System.out.println("Probleme d'ecriture !") ;
      System.exit(1) ;
    }
  }
}
```

La classe StringTokenizer (1/2)

La classe `StringTokenizer` permet de découper une chaîne en différentes sous-chaînes (*tokens*) grâce à des caractères séparateurs définis par l'utilisateur

- Cette classe dispose d'un constructeur prennant d'une part la chaîne à étudier et une chaîne définissant les séparateurs

```
StringTokenizer token = new Stringtokenizer(ligne," ") ;
```

- La méthode `countToken` donne le nombre de tokens
- La méthode `nextToken` donne le token suivant s'il existe
- Sur chaque token on pourra utiliser les possibilités de conversion d'une chaîne vers un type primitif

La classe StringTokenizer (2/2)

```
// On lit le fichier généré par l'exemple d'écriture de fichier text
import java.io.* ;    // Pour les classes flux
import java.util.* ; // Pour StringTokenizer
public class LectureFichierCarre2
{ public static void main(String args[]) throws IOException
  { FileReader f = new FileReader("carre.txt") ;
    BufferedReader entree = new BufferedReader(f) ;
    String ligne = entree.readLine() ; // Lecture fichier
    if (ligne != null)
    { StringTokenizer token = new StringTokenizer(ligne," ") ;
      int nombre = Integer.parseInt(token.nextToken()) ;
      for (int i=0;i<3;i++) // On saute le texte "a pour carre"
        token.nextToken() ;
      int carre  = Integer.parseInt(token.nextToken()) ;
      System.out.println(nombre+" "+carre) ;
    }
    entree.close() ; // Fermeture du flux
  }
}
```

La classe File

La classe `File` permet la gestion des fichiers à la manière des commandes système, elle permet de

- Créer, supprimer, renommer un fichier ou un répertoire
- Tester l'existence d'un fichier ou d'un répertoire
- Obtenir des informations relatives aux protections, aux dates de modifications
- Lister les noms de fichiers d'un répertoire
- Etc.

Son principal constructeur s'utilise de la manière suivante :

```
File monFichier = new File("nom.dat") ;
```

Cela crée un objet de type `File`, mais pas de fichier !

Méthodes de la classe File (1/2)

Attributs du fichier

- `public boolean canRead()`
- `public boolean canWrite()`
- `public boolean exists()`
- `public boolean isDirectory()`
- `public boolean isFile()`
- `public long lastModified()`
- `public boolean length()`

Opérations sur les répertoires

- `public boolean mkdir()`
- `public boolean mkdirs()`
- `public String [] list()`
- `public File [] listFiles()`

Méthodes de la classe File (2/2)

Opérations sur les fichiers

- `public boolean createNewFile()`
- `public boolean delete()`
- `public boolean deleteOnExit()`
- `public boolean renameTo(File destination)`

Autres méthodes

- `getName, getParent, toURL, setReadOnly, createTempFile, compareTo, equals, toString`

La classe File : exemple

```
import java.io.* ;    // Pour File
public class ExempleFile
{ public static void main(String args[]) throws IOException
  { File r = new File("REP") ;
    if (r.isDirectory())
    { String [] dir = r.list() ;
      for (int i=0;i<dir.length;i++) // Affichage du contenu de REP
        System.out.println(dir[i]) ;

      File f = new File("REP/"+"toto.txt") ;
      if (f.exists())
      { System.out.println("taille de toto.txt = "+f.length()) ;
        File nouveau = new File("REP/"+"titi.txt") ;
        f.renameTo(nouveau) ; // Renomage de toto.txt
      }
      else
        System.out.println("toto n'existe pas") ;
    }
  }
}
```

Alors, comment on entre des données au clavier ???

```
import java.io.* ;     // Pour les classes flux
public class Clavier  // Une classe spécialement pour lire des infos au clavier
{ public static String lireString()
  { String ligne = null ;
    try
    { InputStreamReader lecteur = new InputStreamReader(System.in) ;
      BufferedReader entree = new BufferedReader(lecteur) ;
      ligne = entree.readLine() ;
    }
    catch (IOException ex)
    { System.exit(0) ; }
    return ligne ;
  }
  public static float lireFloat() // Exemple de méthode pour lire les float
  { float x = 0 ;
    try
    { String ligne = lireString() ;
      x = Float.parseFloat(ligne) ;
    }
    catch (NumberFormatException ex)
    { System.out.println("Erreur format !") ; System.exit(0) ; }
    return x ;
  }
}
```