Programmation en langage Java

Intitulé Cours : Programmation en langage Java

Révision : J1
Date : 6/8/01

Protections Juridiques

AVERTISSEMENT

Introduction

Objectifs

Ce cours a pour vocation de vous permettre d'assimiler les aspects essentiels du langage Java, d'en comprendre les principes techniques, organisationnels et conceptuels.

Les objectifs prioritaires concernent l'acquisition des concepts fondamentaux de la "programmation objet" avec Java:

- notions de package, de classe, d'instance
  variables membres, encapsulation,
  méthodes (paramètres, résultats, exceptions)
  constructeurs

- typage,
  héritage,
  polymorphisme,
  concept d'"interface"

Sortent du périmètre de ce cours :

- Les aspects avancés du langage comme la programmation concurrente (*threads*).

- L'utilisation détaillée des librairies

- Les librairies d'extension

Ces points sont abordés dans le cours "maitrise de la programmation en langage Java", dans les cours Java spécialisés et dans le *java tutorial* en ligne sur `http://java.sun.com`


Prérequis

Pour pouvoir tirer un maximum de bénéfices de ce cours il est vivement conseillé:

- De connaître la programmation d'un point de vue théorique et pratique : ce cours est conçu pour des programmeurs familiers avec la programmation structurée dans des langagea comme "C", COBOL, FORTRAN, Pascal, etc.

- D'avoir des notions sur les technologies du Web internet (usage d'un navigateur, adresses URL, langage HTML)

- De connaître un minimum d'anglais (il faut pouvoir lire la documentation)

Phases

Le cours est décomposé en plusieurs parties :

1. Un chapitre d'introduction aux concepts fondamentaux de la programmation Objet. Les principes de cette programmation seront revus tout au long du cours.

1. Un chapitre d'introduction aux mécanismes fondamentaux de Java. C"est un exposé théorique dont l'objectif est de vous amener à refléchir sur les options fondamentales de la technologie Java.

2. Un groupe de chapitres concernant les bases techniques: syntaxe, introduction aux types, expressions, structures de contrôles.

3. Un groupe de chapitres consacrés aux aspects fondamentaux de la programmation objet en Java : classes, instances, héritage, packages, exceptions.

4. Un groupe de chapitres permettant de mettre en oeuvre les notions acquises en utilisant des librairies standard : introduction aux entrées/sorties, introduction à la programmation graphique.
   Ces chapitres permettent de poser des questions importantes sur la conception des programmes en Java.

5. La description des aspects fondamentaux de Java est complétée par une présentation du mécanisme des "interfaces" .
   Bien que briévement introduits dans le cours et présents en annexe dans le support, la description systématique des classes membres et classes locales ne font pas directement partie de l'exposé : leur apprentissage repose sur votre travail personnel après assimilation des aspects fondamentaux qui seront traités ici en priorité.

6. Des annexes optionelles présentent les librairies standard et l'utilisation de Java dans le cadre du Web.

7. Des annexes de référence globales complètent les annexes propres aux chapitres.

En fin de certains chapitres on trouvera une annexe de référence intitulé "Compléments": bien qu'elles ne fassent pas partie directement de l'exposé de l'animateur ces annexes vous seront utiles pour approfondir vos connaissances.Les sujets et corrigés d'exercices vous seront présentés dans des documents fournis par l'animateur. Des sujets et des corrigés d'exercices "express" sont également présents dans le texte des chapitres.

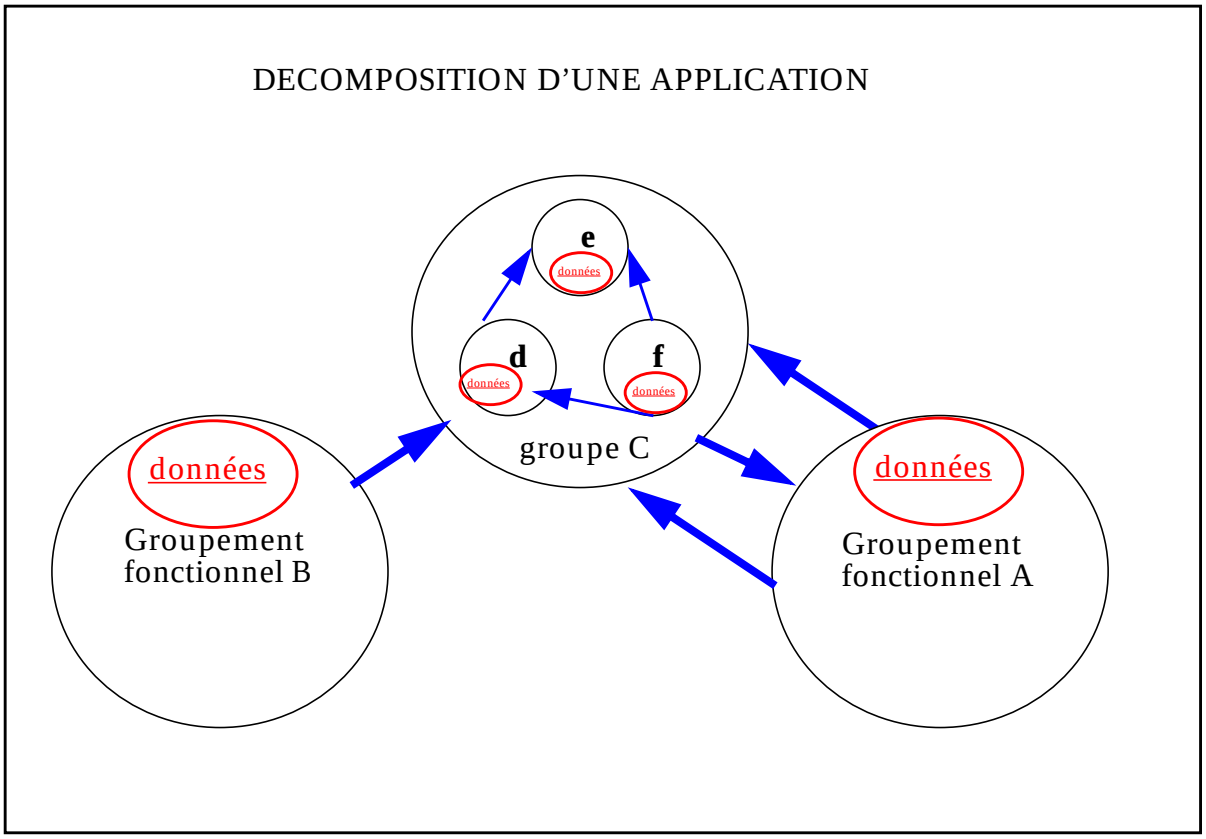
Table des Matières


Introduction à la programmation Objet 1

Présentation

Ce chapitre constitue une introduction aux concepts de la programmation objet.

Les leçons du passé : l'approche modulaire



Depuis longtemps la programmation évolue tout en mettant en avant les mêmes principes :

- Décomposition : un logiciel doit être décomposé en sous-systèmes, chacun décomposé en sous-systèmes, etc.
  Chaque unité logicielle (nous dirons provisoirement un “composant”) ne doit traiter qu’un minimum de problèmes bien délimités.

- Encapsulation : pour contrôler ce qui se passe dans un “composant” il faut éviter que les autres composants soient amenés à connaître (et modifier de manière indue) les données qu’il utilise en propre.

- Minimisation des accès : corollaire du précédent. Les manières d’agir sur un tel composant (et indirectement sur les données qu’il gère) doivent être définies et délimitées précisément.
  Par ailleurs moins un composant a de modes d’accès plus il sera facile à tester, plus l’application sera testable et robuste.

- Contrôle des types : plus tôt on contrôle la nature des données qui sont utilisées en tout point d’un programme, plus robuste sera l’application.
  Nous allons voir que cette notion de type a évolué pour, non seulement caractériser des données, mais aussi des composants eux-mêmes.

Les différents langages de programmation qui sont apparus au cours de dernières décennies ont chacun fait progresser ces concepts fondamentaux. Les langages “à objets” (Smalltalk, C++, Eiffel, Java) proposent des modes d’organisation évolués qui ont profité de ces avancées successives.

historique : sous-programmes, séparation code/données

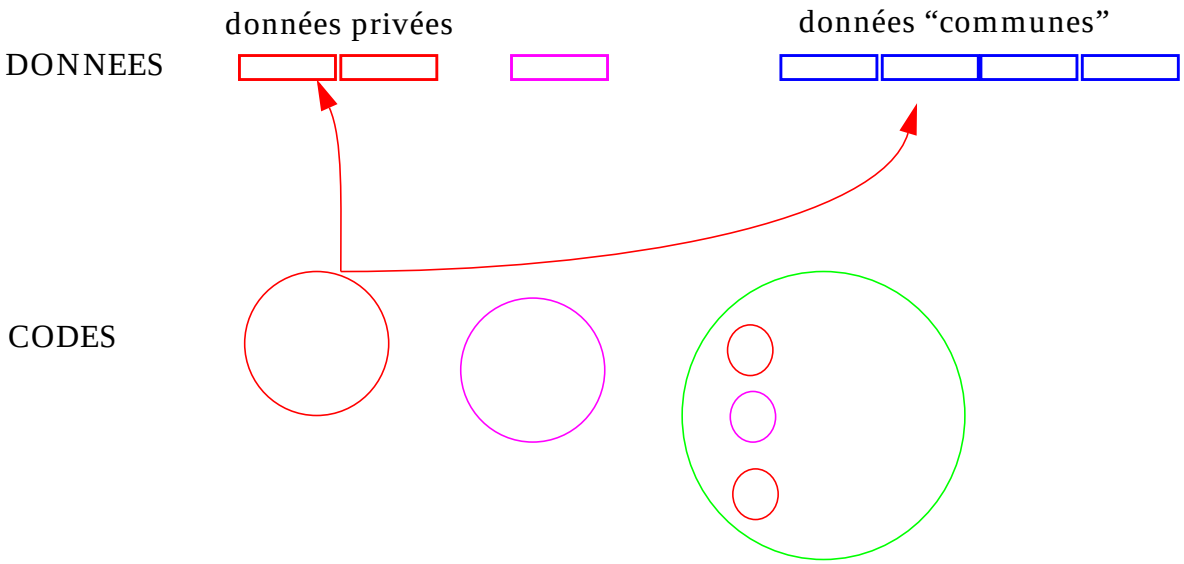
Un des premier moyens introduits pour structurer les codes a été la mise en place de routines.

```
LDA #$0
BSR toto
...
toto: STA $10
        ..
      RTS
```

En un point du programme se trouve un code appelé depuis différents endroits. Ce code correspond à un service précis.

En cas d'erreur ou de modification seul cette routine sera modifiée : elle centralise la gestion d'un problème élémentaire.

Par ailleurs dans "l'image" binaire de l'application les données sont regroupées dans des endroits précis (distincts du code). Un "sous-programme" pourra soit partager les données qu'il accède avec d'autres parties du programme, soit réserver des données à son usage exclusif. Les compilateurs de langages évolués gèrent ces accès de manière transparente pour le programmeur.



historique : types simples

Un des avantages des langages de haut-niveau est de libérer le programmeur des contraintes de gestion du bas niveau des données: tel octet en mémoire doit-il être interprété comme représentant un caractère ou bien un nombre “court” représenté sur un octet? Ce nombre est-il signé ou non? etc.

En “typant” les variables on gère leur allocation (tel type de nombre est implicitement sur 4 octets, etc.) et leur interprétation. Selon les langages, on peut permettre de vérifier au moment de la compilation que les opérations qui utilisent ces données sont licites (par ex. telle fonction ne peut prendre qu’un caractère en paramètre, pas un nombre “flottant”).

En COBOL zone élémentaire

```
 01 zone-elementaire PIC 9(6)
* numerique entier de 6 caracteres usage “display”
 01 zone-entiere PIC 9(6) COMP
* usage binaire dependant machine
```

En C ;

```
int valeur ; /* “mot memoire” */
double solde ; /* en virgule flottante */
```

Le compilateur qui transforme le code “lisible” par un être humain en code machine, génère un code tel qu’au moment de l’exécution un emplacement mémoire particulier soit réservé pour le stockage de la variable. Il y a plusieurs manières d’allouer ces emplacements en mémoire:

- Les variables dont le type et le nombre sont connus au moment de la compilation sont allouées *statiquement* (d’où le nom: données statiques). En réalité il y a aussi un autre mode d’allocation (dit *automatique*) que nous verrons ultérieurement.

- Il y a des variables pour lesquelles on ne connaît pas à l’avance le nombre (on sait qu’on aura besoin de *N* valeurs mais *N* n’est pas connu au moment de la compilation). Ces variables seront allouées au moment de l’exécution dans le **tas** : c’est une partie de la mémoire gérée par l’exécution et dans laquelle on alloue (et on désalloue) des emplacements au fur et à mesure des besoins.

historique : types composites

Une évolution de la notion de type a été de gérer en même temps un regroupement de données (*structure*, *enregistrement*,...).

Ce sont des types définis par le programmeur.

Ils ont un aspect conceptuel et un aspect intéressant la gestion de mémoire:

- aspect conceptuel : on regroupe un ensemble de données qui concourent à une même définition. Un type “compte” est défini par un numéro de compte et un solde
- aspect physique : les données sont des blocs contigus en mémoire et peuvent être gérés globalement (par ex. dans certaines utilisations des fichiers à accès direct tous les blocs ont la même taille et on peut facilement accéder au Nième élément).

En COBOL

```
 01 zone-compte
          05 numero-compte PIC 9(6)
          05 solde-compte PIC 9(10)V9(2)
...
* modification dans la procedure division
          MOVE zone-entiere TO numero-compte
          MOVE zone-flottante TO solde-compte
* le nom des zones elementaire doit être unique
*on les designe sans faire reference a la zone englobante
* voir syntaxe en langage C ci-dessous
```

En C

```
struct compte {
         int numero;
         double solde ;
         } ;

struct compte unCompte ;/* déclaration de variable */
         unCompte.numero = 10 ;
         unCompte.solde = 0.10 ;
```

historique : données d'un sous-programme

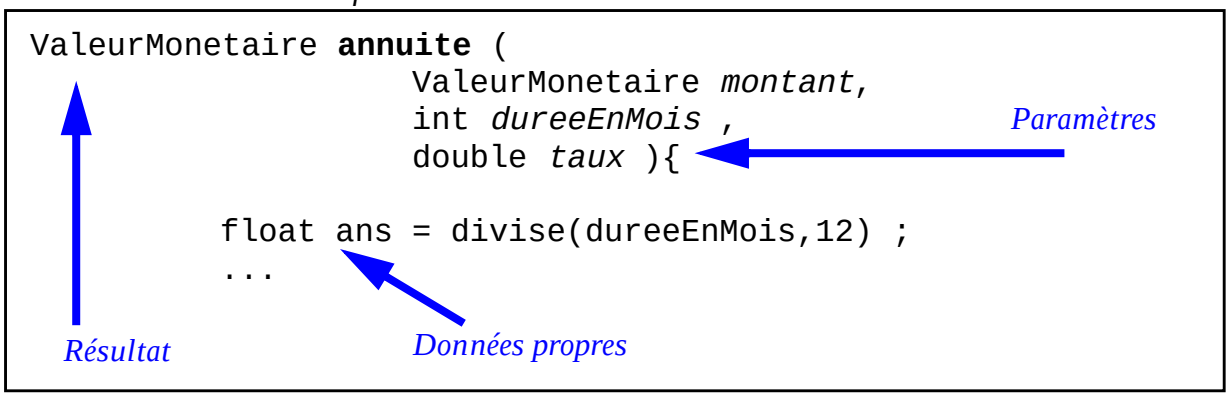
Un sous-programme est susceptible d'utiliser :

- Des données statiques "communes"
- Des données statiques "privées"
- Des arguments d'appel: c'est à dire des données qui lui sont passées par le programme d'appel pour paramétrer son comportement.
- Un résultat: c'est à dire des données qui sont récupérées par le progamme appelant comme résultat de l'exécution du sous-programme.
- Des données propres à chaque invocation du sous-programme.

Paramètres, résultat et données locales sont alloués dans la *pile d'exécution* (voir ci-après) leur valeur est temporaire (les données locales sont, en général, valides uniquement pendant la durée d'exécution du sous-programme)

historique : modèle des appels de fonction

Pour modéliser les appels de sous-programme on a fait souvent appel à un *modèle fonctionnel*.



On définit une fonction qui prend un certain nombre de paramètres typés et qui rend un résultat. Dans de nombreux langages c'est l'ordre des paramètres qui permet de les distinguer: il y a un nom utilisé dans la définition de la fonction pour les variables passées en paramètres, mais ce nom n'est pas utilisé par le programme appelant.

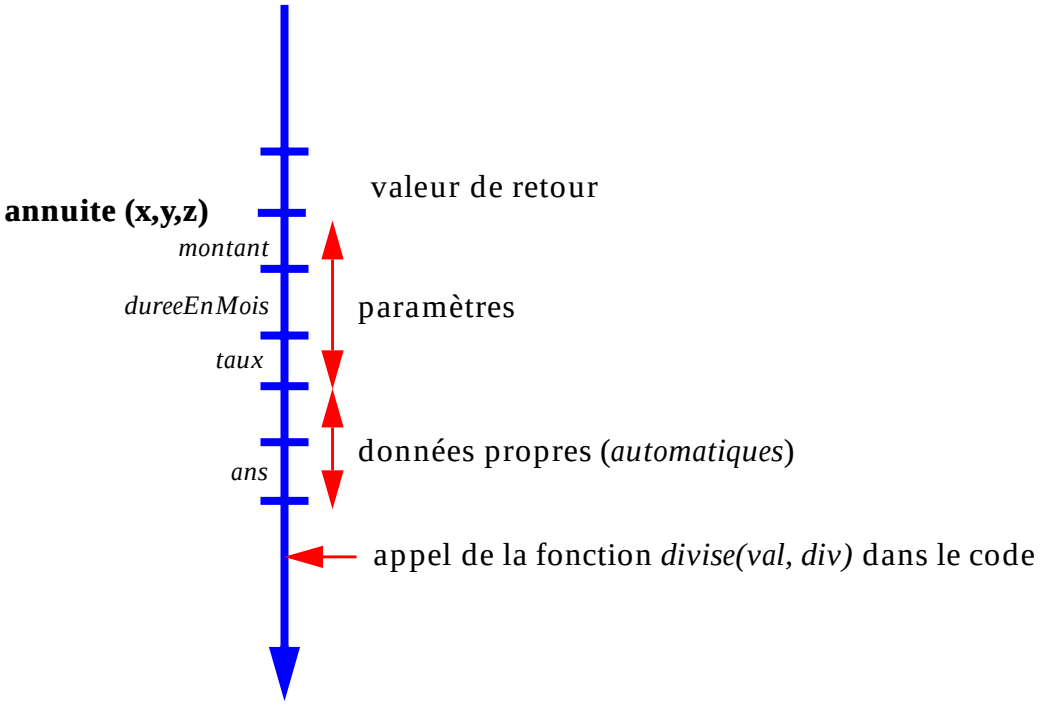
```
/* en C */
resultat = annuite(valeur_courante, 120, 5.6) ;
```

On retrouve dans ce modèle d'organisation le souci d'isolation exprimé précédemment: la fonction a sa vision propre de l'environnement (les paramètres "formels"), a ses propres variables (inconnues de l'extérieur), et rend à l'environnement d'appel un résultat d'un type bien défini.

Dans les cas les plus "purs" la fonction doit s'interdire de modifier des variables de son environnement (elle devient plus indépendante du contexte) et doit s'interdire de modifier ses paramètres d'appel -quand c'est possible-.

historique : modèle des appels de fonction

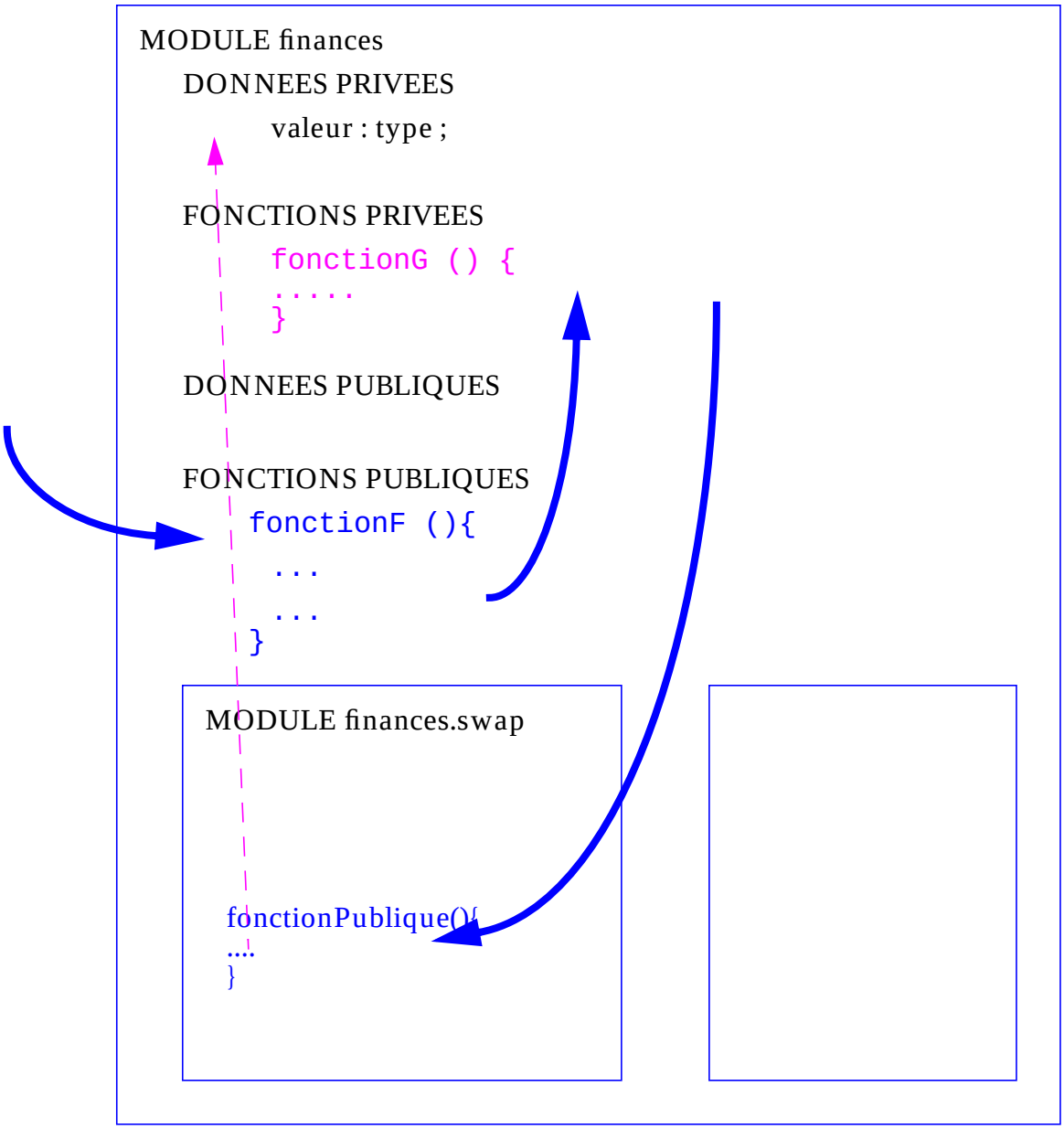
Au modèle conceptuel des fonctions est associé un modèle physique de l'organisation de la mémoire: la *pile des appels*.



Cette organisation permet en outre d'avoir des langages dans lesquels le code des fonctions peut être *réentrant*: l'autonomie de la fonction permet de la rappeler dans des contextes différents et, éventuellement, plusieurs appels peuvent avoir lieu en même temps (cas de tâches qui s'exécutent en parallèle sur plusieurs piles différentes) ou successivement dans la pile (récursivité: la fonction s'appelle elle même).

```
function Fact(n: integer): integer;  (*en Pascal: supposer n>0*)
    var res: integer;
    begin
    if n <= 1 then
        res := 1
    else
        res := n * Fact(n-1);
    Fact := res ; (*le resultat de la fonction *)
    end;
```

historique : modules



historique : modules

Les décompositions successives des systèmes en sous-systèmes va amener une multiplication des fonctions. Il est intéressant de ne pas laisser toutes ces fonctions à un niveau global mais de les hiérarchiser: telle fonction “calcul(x)” a un sens bien précis dans un certain contexte mais pas dans un autre (par exemple dans une grosse application il peut exister deux fonctions “calcul(x)” avec un sens différent et développées par deux équipes différentes).

- Si ces deux fonctions existent dans des “espaces” différents,
- Si on s’organise pour qu’il n’y ait pas ambiguité au moment de l’appel,
- Ou si on fait en sorte qu’elles ne puissent être appelées que dans le même contexte où elles ont leur sens primitif,

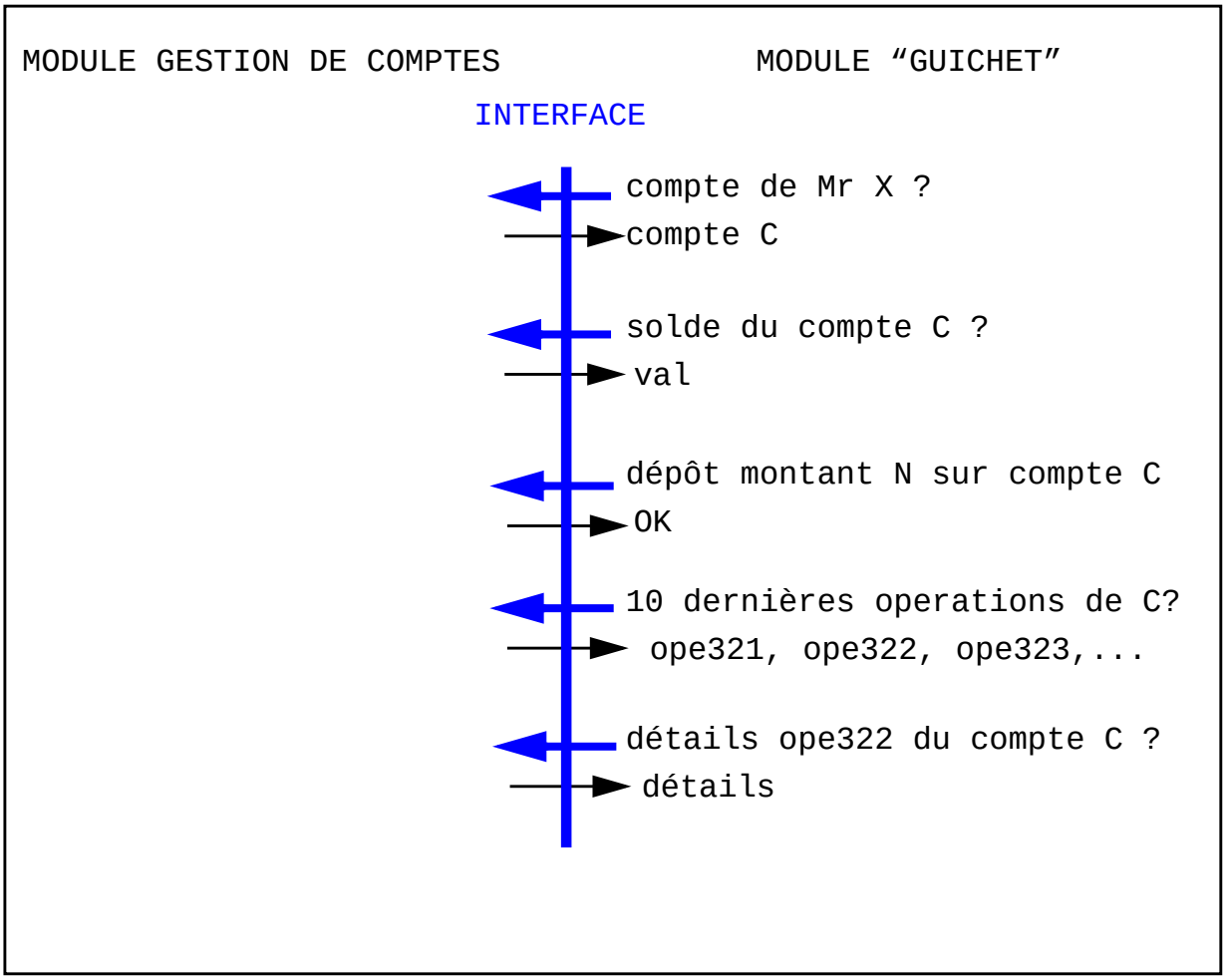
on a mis en place une architecture modulaire.

Certains langages favorisent la création de modules qui regroupent des fonctions, des variables partagées par ces fonctions et qui n’exposent à l’extérieur qu’un petit nombre de points d’entrée.

On a ainsi :

- Une décomposition en une hiérarchie de modules, les modules étant découpés en fonctions .
- Une encapsulation : les données partagées sont internes aux modules et on ne peut pas les modifier de “l’extérieur” d’une manière incontrolée.
- Une minimisation des accès: les points d’entrée sont peu nombreux. Beaucoup de fonctions sont reservées à l’usage exclusif du module.
- Des contrôles *a priori* sur les types des données, et sur les services des modules: si il appelle “calcul(x)” le programmeur doit préciser le module choisi pour lui rendre ce service (si ce module ne connait pas de fonction “calcul(x)” le compilateur refuse le code).

limites de la décomposition fonctionnelle

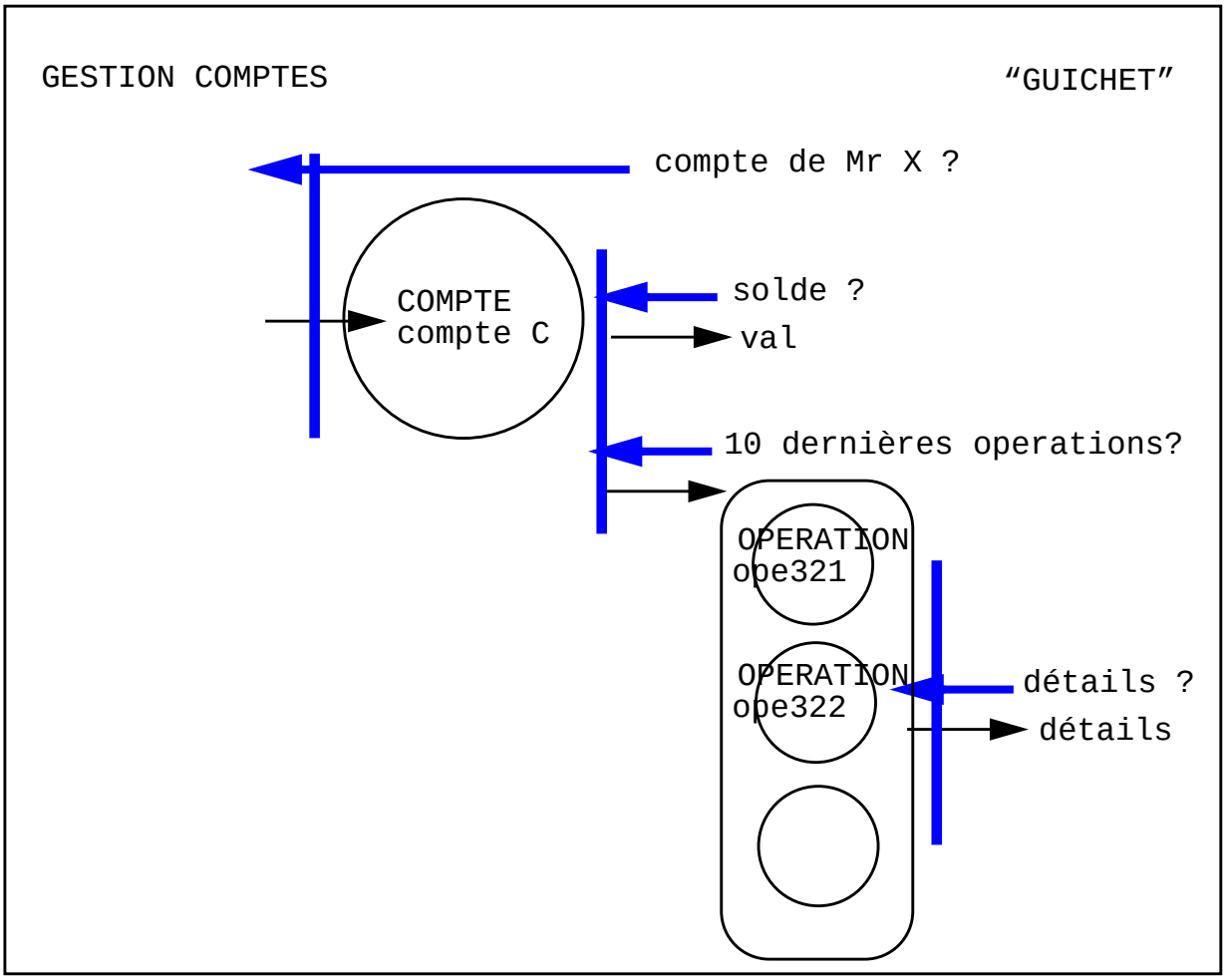


La décomposition descendante conduit à une multiplication des points d’entrée dans les modules. Ceci est contraire à l’exigence de minimisation des interfaces.

Par ailleurs l’arbre des dépendances fait en sorte que lorsque l’on définit un service, celui est défini dans le contexte étroit d’un sous-sytème particulier: il devient difficile de le réutiliser dans un autre contexte.

L’approche par “objets” va permettre de déléguer les interfaces d’accès à un sous-système, de concilier encapsulation et minimisation des interfaces. La réutilisation va être facilitée car ce sous-système acquiert une certaine autonomie.

L'approche "objet" : délégation des interfaces d'accès



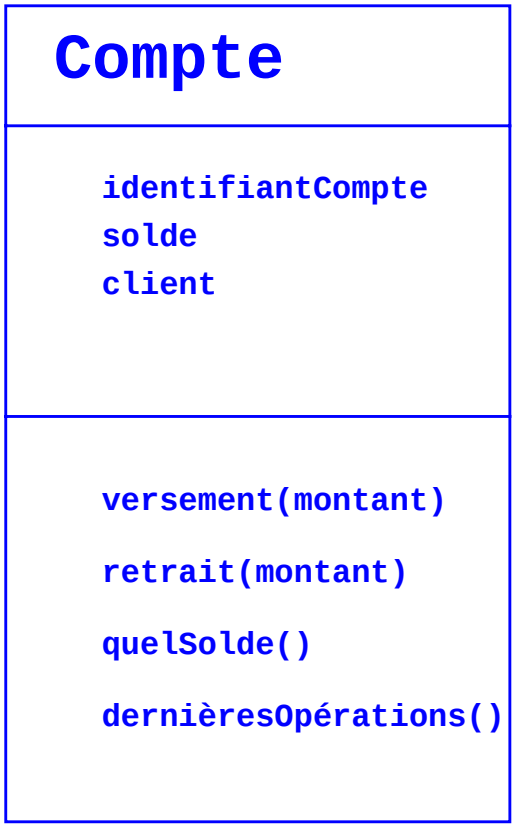
- Sur une demande de compte le code gestionnaire de compte rend un *objet* **instance** de la **classe** "Compte":
  c'est le "compte de Mr X" il contient des informations spécifiques à ce compte et il est structuré selon le modèle décrit par la définition commune à tous les comptes (définition de classe).

- Le "guichet" interroge directement CE "compte" particulier.
  Le "guichet" et le "gestionnaire de compte" n'ont pas les même droits sur cette instance.

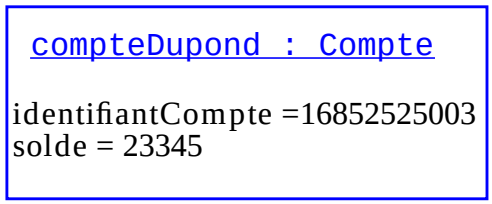
- Le point de vue de programmation est différent: on demande des services à des instances (on parle d'**envoi de messages**). On ne définit plus un service en terme de fonction - `ouvrir(porte)` - mais en terme de message -`sesame.ouvretoi()`-.

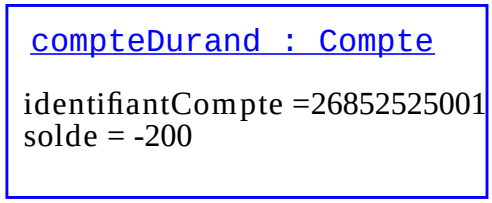
L'approche "objet" : classes , instances, attributs, méthodes

CLASSE



INSTANCES





compteDupond.identifiantCompte → 16852525003
compteDupond.solde → 23345

compteDurand.identifiantCompte → 26852525001
compteDurand.solde → - 200

compteDurand.versement(200)

compteDurand.solde → 0

L'approche "objet" : classes , instances, attributs, méthodes

Définir une classe c'est définir une matrice à partir de laquelle seront fabriqués différents "objets" programmatiques (les *instances* de la classe).

La définition d'une classe consiste à décrire un ensemble de services. La réalisation de ces services (les *méthodes* de la classe) s'appuie accessoirement sur des données (les *attributs* de la classe)

Si une classe "Compte" dispose d'attributs "identifiantCompte" et "solde", la création d'un Compte (comme `compteDurand`) consistera à générer une combinaison particulière de ces valeurs (l'attribut `identifantCompte` vaut `26852525001`, et l'attribut `solde` vaut `-200`)

L'appel de la méthode `versement` sur l'instance `compteDurand` modifiera son état interne (la valeur de `solde` devient 0).

Pour simplifier on pourrait dire que réaliser un programme dans un langage à objets c'est:

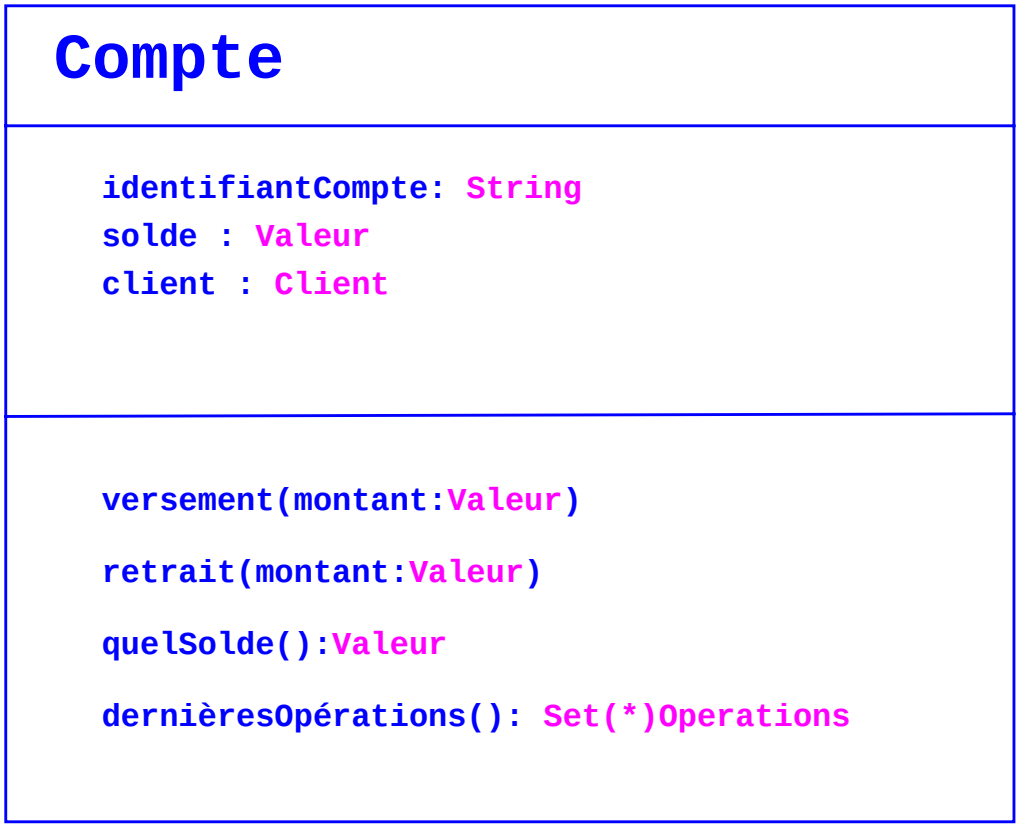
- Définir des modèles (les classes) qui regroupent des données et des comportements

- Créer des instances conformes à ces modèles et leur demander des services (envoi de message <=> appel de méthode sur une instance).

  Si on demande à un "gestionnaire de compte" de nous donner le "compte de MrX" il nous donne l'accès à une instance que l'on pourra interroger (pour demander le solde par ex.). C'est le jeu des références entre instances qui permettra de construire une application complexe.

L'approche "objet" : typage

CLASSE




```
// écritures refusées par le compilateur Java
compteDurand.numero // l'attribut "numero" n'existe pas
compteDurand.retrait() // argument manquant

// plus difficiles
compteDurand.client = "Durand" // "Durand" est une chaîne
                               // pas un objet de type Client
compteDurand.retrait(200) // 200 est un entier
                               // pas un objet de type Valeur
```

L'approche "objet" : typage

Lorsque l'on programme un envoi de message à un objet particulier il est important de vérifier si cet objet est effectivement capable de rendre ce service. Les variables qui désignent des objets sont *typées*.

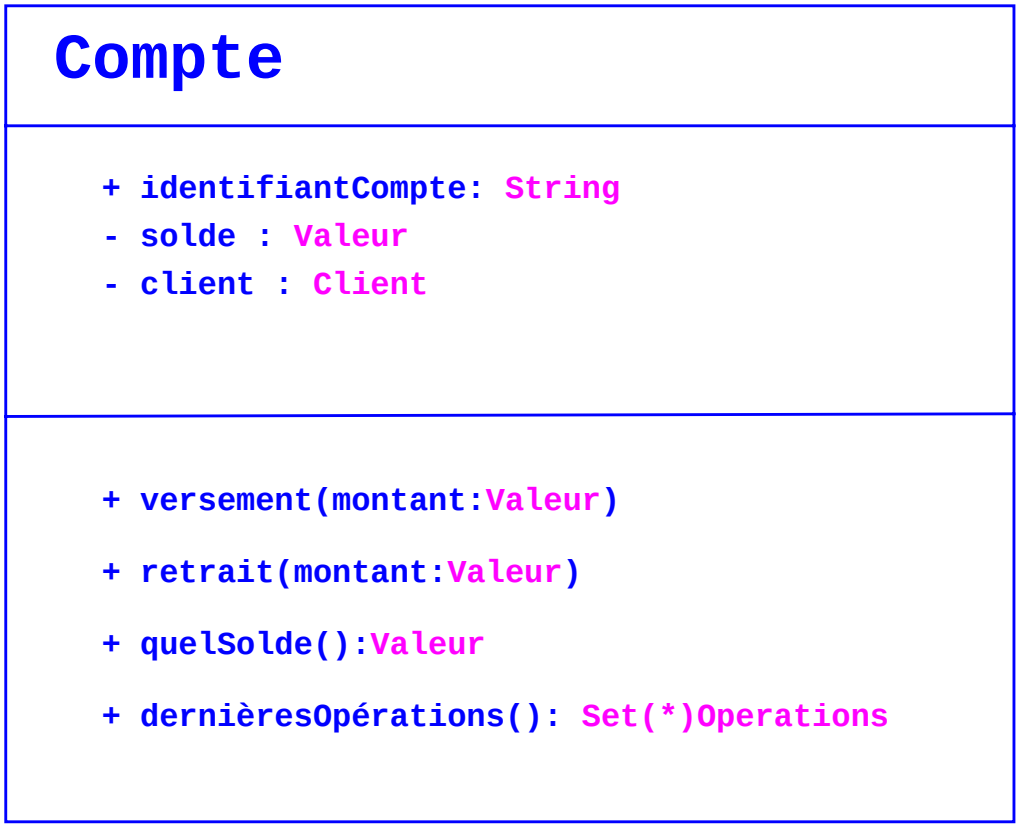
- Des contrôles de pertinence seront effectués à la compilation et / ou à l'exécution (En Java -langage compilé- on effectue des vérifications de type au moment où on transforme le programme source en exécutable. D'autres vérifications sont effectuées aussi au moment de l'exécution).

  Ainsi il sera possible de vérifier que, par exemple , pour une variable qui désigne une instance, les méthodes appelées ont une "signature" correcte (nom, type des paramètres, type des résultats, etc.) et , éventuellement, que les attributs désignés sont corrects (nom, type des attributs).

- Selon les langages la notion de type ne recouvre pas tout à fait la même réalité : dans certains langages, par exemple, un type est aussi lié à des caractéristiques physiques de la zone mémoire qui représente l'objet (taille, position des données élémentaires qui constituent les attributs, etc.). Ces connotations n'existent pas en Java pour les types "objet".

L'approche "objet" : encapsulation et visibilité

CLASSE




```
//refusé par le compilateur dans un autre code que celui de Compte
compteDurand.solde = nouveauSolde // pas droit d'accès
```

L'approche "objet" : encapsulation et visibilité

Application d'une contrainte de génie logiciel : pour contrôler ce qui se passe dans une instance il faut éviter que les objets relevant d'une autre partie du code soient amenés à connaître (et modifier de manière indue) les données que l'instance utilise en propre.

Certains attributs d'un objet sont destinés à être "privés" : ils ne sont accessibles que par les méthodes liées à l'instance. On pourrait également définir des méthodes "privées" qui ne seraient appelées que par d'autres méthodes de l'instance.

Dans l'exemple de la classe "Compte" si l'attribut "solde" était directement accessible on courrait le risque:

- de laisser réaliser des programmes qui laissent l'instance dans un état incohérent: on modifie "solde" sans mettre à jour la liste des opérations...

- d'avoir des difficultés pour faire évoluer l'application: si on décidait, par la suite, de faire en sorte que chaque fois que l'on réalise un opération de modification du solde il faille réaliser une écriture dans un fichier annexe, on serait obligé de rechercher dans le code tous les endroits où cette modification est opérée. Si la modification est centralisée, l'évolution est réalisée immédiatement.

L'approche "objet" : constructeurs

CLASSE

Compte

<<attributs>>

+ identifiantCompte: String
- solde : Valeur
- client : Client

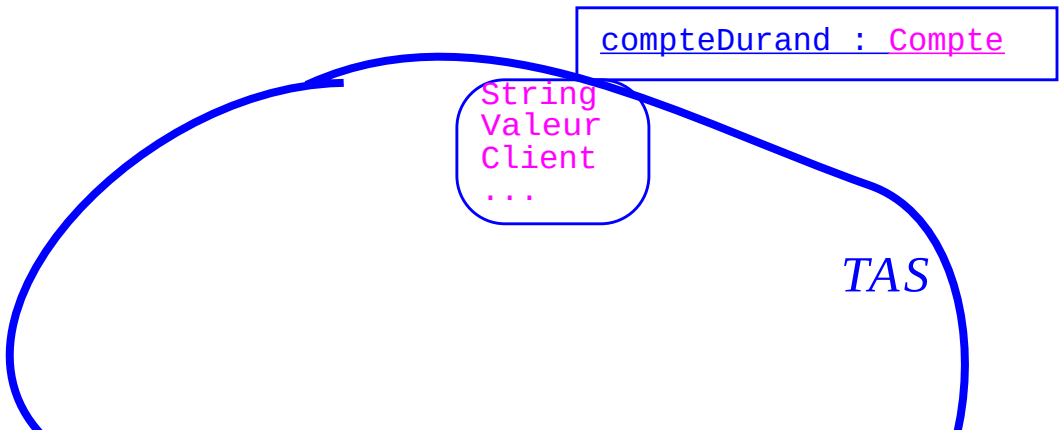
<<constructeurs>>

+ Compte(id:String , client: Client, depot:Valeur)

<<méthodes>>

+ versement(montant:Valeur)
+ retrait(montant:Valeur)
+ quelSolde():Valeur
+ dernièresOpérations(): Set(*)Operations

```
compteDurand = new Compte(numero, durand, depotInitial) ;
```



L'approche "objet" : constructeurs

Une opération particulière permet de créer des instances sur le modèle d'une classe.

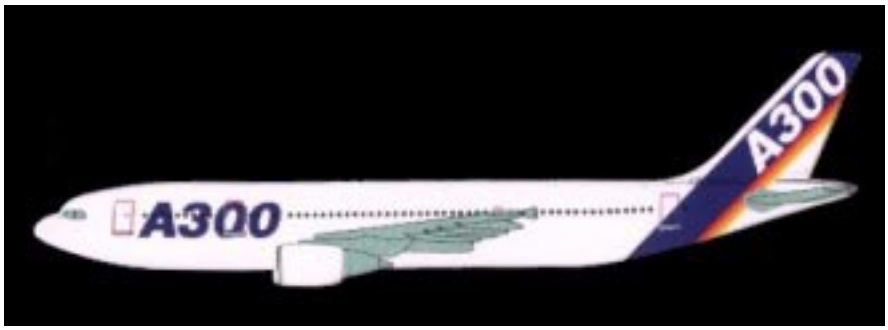
Cette opération permet d'allouer la mémoire qui va représenter l'instance et éventuellement d'initialiser certains attributs.

On définit un **constructeur** qui va permettre de réaliser ces opérations de création /initialisation de l'instance. Le code du constructeur permet d'exploiter des paramètres qui agiront sur l'initialisation d'attributs (et en particulier d'attributs "privés"). Noter qu'il peut y avoir plusieurs constructeurs (dans l'exemple on pourrait avoir un constructeur qui prévoit un dépôt initial, et un qui n'en prévoit pas -le solde est alors à zéro-).

Dans le cadre d'une première approche on peut dire que la définition d'une classe consiste en :

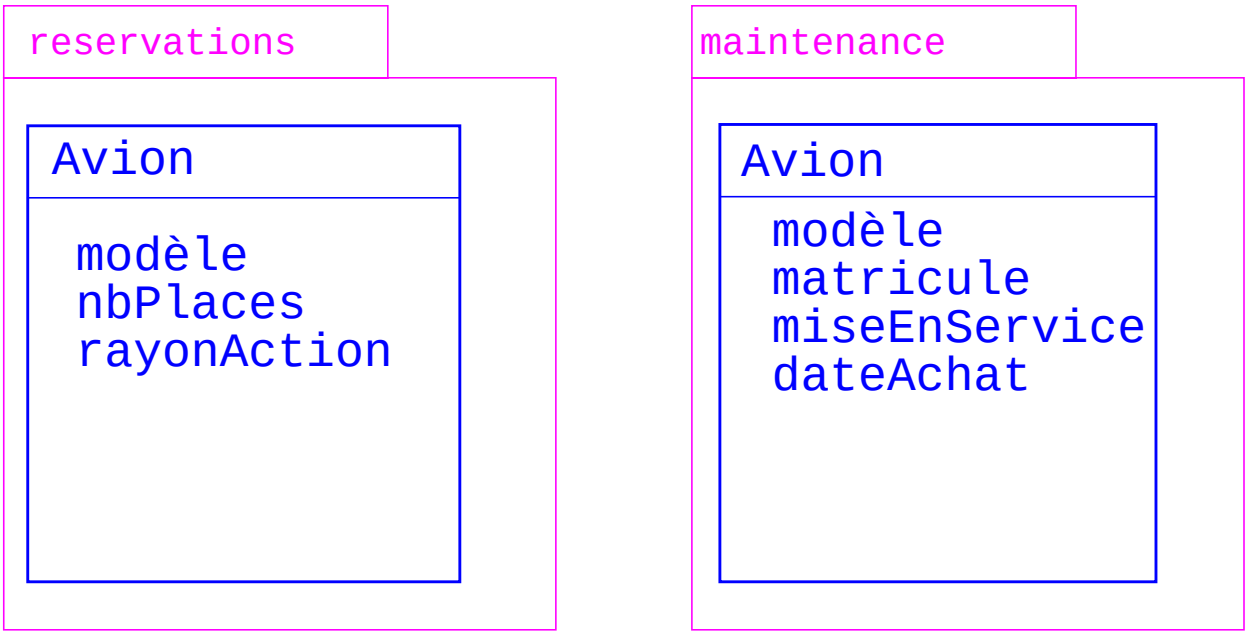
- la définition des attributs des instances (avec leur type, leur droit d'accès,..)
- la définition des constructeurs qui permettent de créer les instances
- la définition des méthodes qui vont pouvoir opérer sur ces instances.

L'approche "objet"



AVION "réel" (des centaines de milliers de composants....)

et modèles informatiques ....



L'approche "objet"

La définition de classes, la création d'instances et l'invocation de messages entre ces instances permettent de construire des applications basées sur la collaboration entre entités autonomes en termes de données et de traitements.

Par plusieurs aspects cette démarche conduit à des points de vue particuliers sur la programmation. Dans le cadre d'un contexte applicatif on cherche un modèle "utile". Ce modèle doit être à la fois ciblé et, si possible, réutilisable.

Il y a là une démarche (dont la discussion sort du cadre de ce cours). On cherchera à définir des composants relativement généraux qui pourront être réutilisés dans différentes circonstances liées au "métier". Ces composants s'affineront au fur et à mesure de l'avancement des projets: les modèles d'objets informatiques pourront s'inspirer de propriétés d'objets du "monde réel" et les filtrer par abstractions successives. Dans certains cas on arrive à concevoir / utiliser des dispositifs purement abstraits: des "modèles structurels" d'organisation (*patterns*).

Analyse et conception: qu'est ce qu'UML?

Dans les phases d'analyse et de conception d'un projet on peut définir des objets conceptuels qui permettront de fournir une base d'approche pour la définition des "objets" programmatiques.

- Analyse orientée objet

  Le but de l'analyse objet est de définir précisément ce qui doit être réalisé. Cette phase ne donne pas de solution sur la réalisation. C'est une phase qui n'est pas purement informatique et les documents associés doivent être compris par les informaticiens effectuant l'analyse et les experts du domaine qui fournissent les informations sur le métier.

  Lors de cette phase, il faut trouver les objets potentiels et les caractériser. Le but est de construire un modèle abstrait en conservant une correspondance entre ce modèle et les spécifications.

- Conception orientée objet

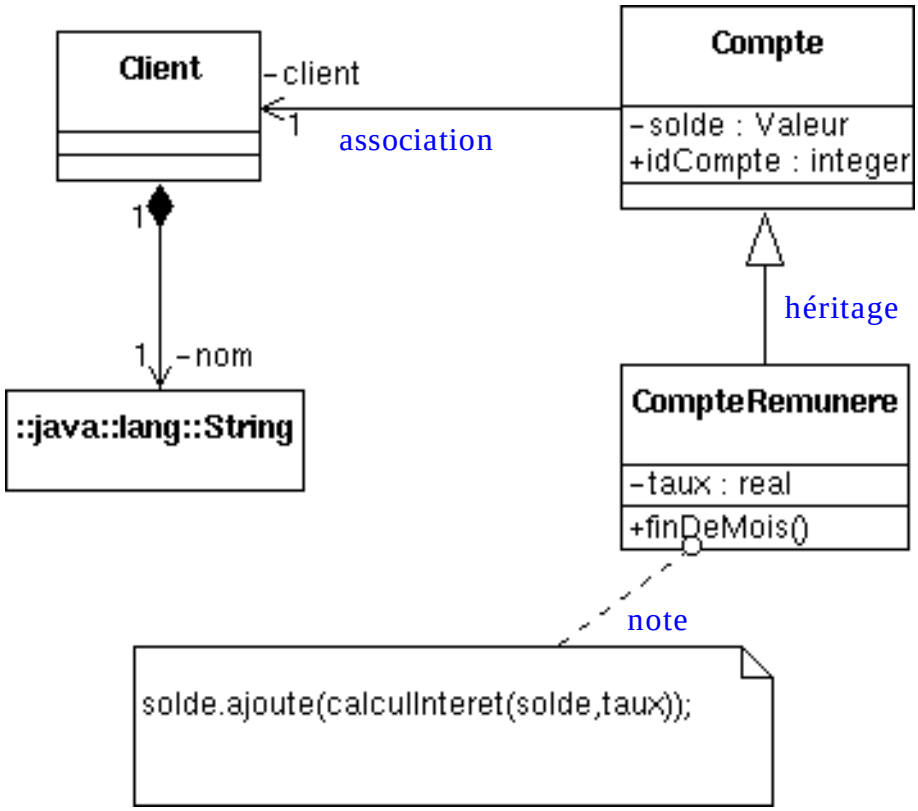
  Cette phase se base sur les documents fournis par la phase d'analyse. Elle sert à affiner les informations en fournissant des solutions informatiques aux problèmes posés. Il est possible de distinguer deux parties : la recherche de la structure du système (conception système / architecture logicielle) puis du logiciel (conception logicielle). A la fin de cette phase, il doit être possible de coder le système sans ambiguité à partir des documents de conception.

Les méthodes objets qui sont apparues dans les années 80 partagent un ensemble de concepts communs ou proches qu'il est souhaitable de décrire avec un même formalisme. Utiliser un langage de description commun permet aussi de partager et de capitaliser les expériences dans le domaine de l'objet.

U.M.L (Unified Modeling Language) est un langage de représentation de modèles qui peut être utilisé dans des cadres méthodologiques différents.

Analyse et conception: qu'est ce qu'UML?

Un exemple de diagramme de classe (Ce diagramme met en oeuvre des concepts qui seront expliqués ultérieurement il est mis ici à titre d'illustration)



Les diagrammes d'UML

- Un total de 9 diagrammes différents composent UML pour représenter le système sous un angle statique et dynamique
- Pas d'ordre d'utilisation imposé : ni pour l'analyse ni pour la conception
- Modéliser la vue statique
  - Diagramme de cas d'utilisation
  - Diagramme d'objets
  - **Diagramme de classes**
  - Diagramme de composants
  - Diagramme de déploiement
- Modéliser la vue dynamique
  - Diagramme de collaboration
  - Diagramme de séquence
  - Diagramme d'états-transitions
  - Diagramme d'activités

Les diagrammes d'UML

- UML se compose de 9 diagrammes différents dont la syntaxe est précisément définie. Ces diagrammes permettent de représenter différentes vues du système à réaliser : statique (classes, composants, cas d'utilisation) et dynamique (séquencement des messages entre objets).

- UML ne définit pas d'ordre d'utilisation. Le processus unifié indique de commencer avec un diagramme de cas d'utilisation qui permet définir les besoins du client. Ensuite, chaque diagramme peut être utilisé, retravaillé, affiné pour permettre de définir le système à réaliser.

- Tous ces diagrammes aident les concepteurs à préciser leurs pensées, à dialoguer avec les autres membres de l'équipe et à vérifier la cohérence du modèle.
  Un point très important dans toute démarche de conception et le cycle auteur/lecteur. Il faut toujours effectuer des revues des différents diagrammes avec plusieurs personnes (l'utilisation d'un langage de modélisation standard est donc très utile). Cette méthode de travail est la plus fiable pour identifier les éventuels problèmes de conception. Cette façon de faire oblige aussi à avoir une analyse simple et consistante ainsi qu'une documentation complète. La simplicité et la documentation garantissent une qualité accrue du logiciel et une capitalisation des connaissances

Adopter une approche par objets c'est mener aussi une reflexion sur l'architecture.

Exercices :

Notre premier exercice est un exercice d'analyse. Il se pratique en groupe et l'animateur aura pour tache de susciter et canaliser les propositions des stagiaires.

La notation à adopter sera la plus simple possible : pour chaque classe on définira quelques attributs importants sans préciser leur type, et, éventuellement, on nommera quelques méthodes essentielles (on n'abordera pas ici les constructeurs).

Le projet :

> La société `oubangui.com` vend des livres au travers d'internet.
>
> L'utilisateur qui se connecte consulte les descriptions des articles en vente, met tout article choisi dans un "Panier" et à la fin se fait connaître pour payer son Panier.
>
> La discussion portera sur l'analyse des classes principales.

Présentation de Java 2

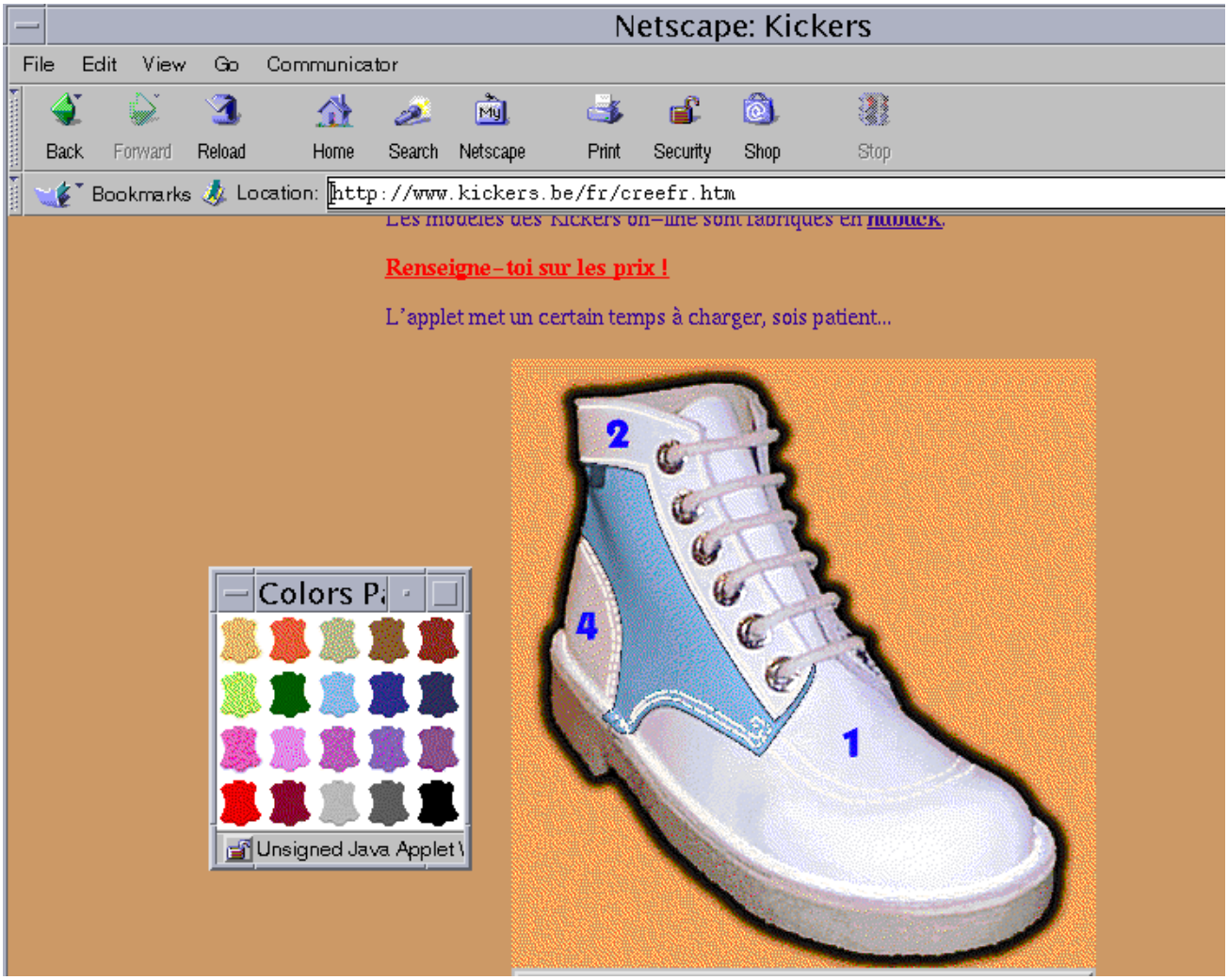
Points essentiels

Une introduction aux mécanismes fondamentaux de la technologie Java:

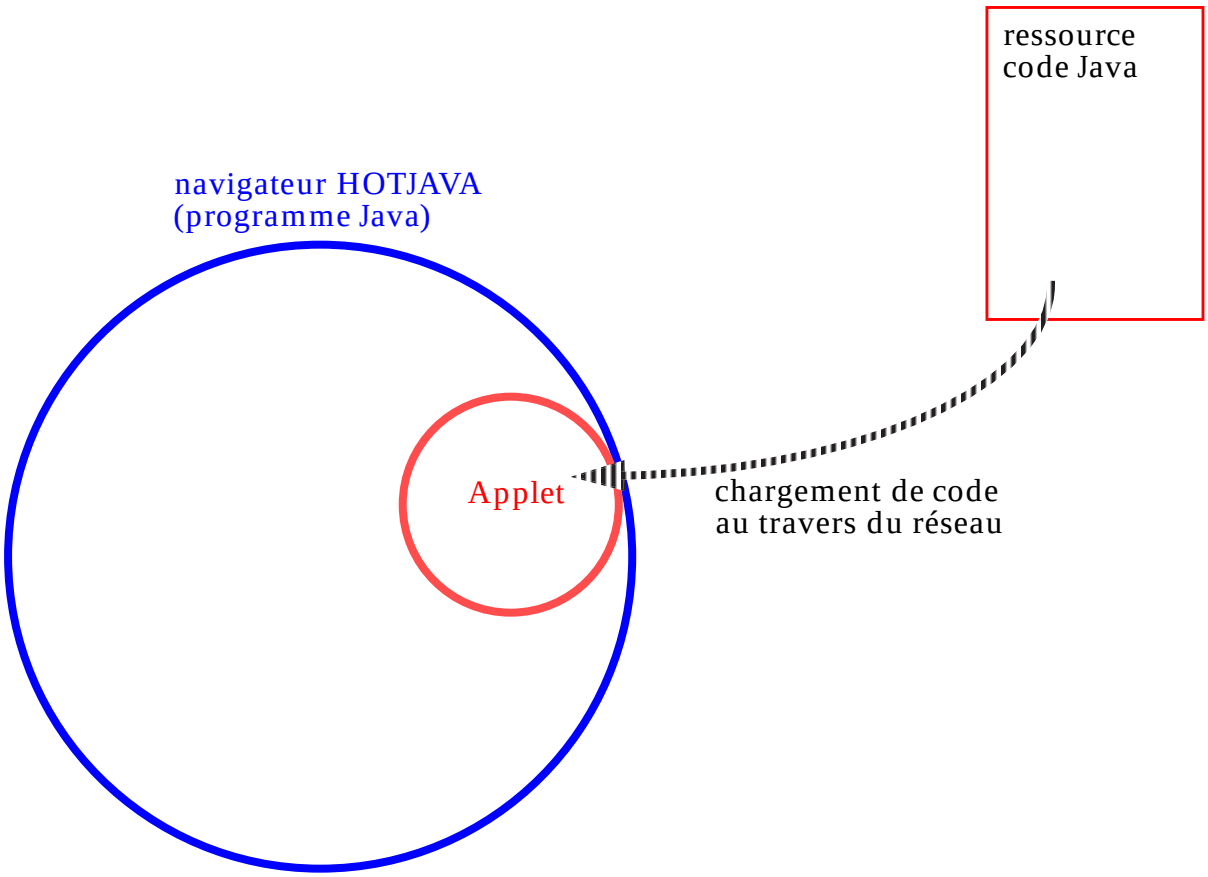
- Caractéristiques principales de la technologie Java.
- Code source, code exécutable
- La machine virtuelle
- Mécanismes d'exécution, mécanismes de contrôle
- Un premier programme Java.

Applications, Applets, ...: à la découverte de Java

Une Applet est un programme qui s'exécute au sein d'une page HTML (une "page" téléchargée par un navigateur Web). Un tel programme est chargé dynamiquement par le navigateur qui trouve sa référence dans le code HTML et qui demande le chargement de ce code depuis le serveur HTTP (le "serveur Web").



Applications, Applets,...: à la découverte de Java



Considérons la situation suivante:

- Prenons un navigateur qui soit un programme autonome réalisé entièrement à l'aide du langage de programmation Java (c'était le cas du navigateur *HotJava*)

- Au travers d'une page du Web ce navigateur télécharge un "code" Java (une Applet) depuis un serveur et exécute ce code

Quelles caractéristiques importantes du langage Java dévoile ce comportement?

A la découverte de Java

L'exemple précédent révèle quelques caractéristiques essentielles de Java:

- Le serveur HTTP envoie le même code à tous ses clients, (il ne sait pas s'il s'agit d'un PC, d'un macintosh, d'une station unix, etc.)
  Le code exécutable Java est un **code portable** : il est capable de s'exécuter sur des machines ayant des architectures et des systèmes d'exploitation différents.

- Le code est exécuté dynamiquement: l'application Java (le navigateur) est capable, dans le cadre d'une autre tâche, de découvrir une instruction l'enjoignant de charger un autre code Java (a priori inconnu!) et de déclencher son exécution.
  Cette **exécution dynamique** a un certain nombre de corollaires:

  - Il faut que le code importé ne puisse pas réaliser des opérations non souhaitées sur le site local. Un **mécanisme de sécurité** est intégré au langage.

  - La coopération dynamique entre différents codes suppose une modularité du code très bien étudiée. Java est un **langage à Objets** particulièrement modulaire (ex. pas de variables globales, pas de fonctions isolées, etc...).

  - L'exécution de l'Applet se déroule pendant que le programme principal continue à s'exécuter (y compris pour lancer d'autres Applets). Java permet d'accéder à des mécanismes d'exécution en parallèle (**processus légers** `-threads-`).

  - ...

Le langage de programmation Java

Conçu à l'origine comme environnement de développement pour des applications portables destinées à de l'électronique grand-public, Java a connu une forte expansion avec l'explosion du Web.

La communauté des développeurs a rapidement adopté ce langage pour sa clarté, sa puissance d'expression, son organisation de langage à objets, sa portabilité,...

Lorsqu'on parle du langage Java on fait référence à :

- Un langage de programmation (pour une définition plus formelle voir JLS : Java Langage Specification)
- Un environnement de développement
- Un environnement d'exécution
- Un environnement de déploiement

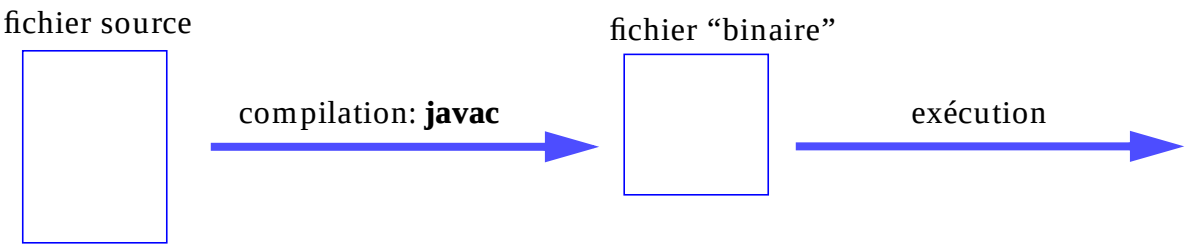
En fait les caractéristiques de Java ont permis l'éclosion de nouvelles formes d'organisation des systèmes d'information et de nombreuses technologies apparentées. Au sens strict il faut donc distinguer les différents concepts qui utilisent le vocable "Java" ("technologie Java", "langage de programmation Java").

Dans la suite de ce document chaque fois que nous utiliserons le mot "Java" sans préciser le contexte on devra comprendre "langage de programmation Java".

Dans les pages qui suivent nous allons introduire les mécanismes de fonctionnement de Java.

Code source, code exécutable

génération de code exécutable :



fichier source : **Calimero.java**

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

fichier binaire : **Calimero.class** (468 octets)

```
cafe babe 0003 002d 0020 0800 1507 0010
0700 1a07 001b 0700 1c0a 0004 0009 0900
0500 0a0a 0003 000b 0c00 0f00 0c0c 001e
0017 0c00 1f00 0d01 0003 2829 5601 0015
....
```

décompilation du fichier binaire :

```
...
Method void main(java.lang.String[])
   0 getstatic #7 <Field java.io.PrintStream out>
   3 ldc #1 <String "Hello World!">
   5 invokevirtual #8 <Method void println(java.lang.String)>
   8 return
...
```

Code Source, Code exécutable

Ce qui est exécuté par une application autonome (ou par un navigateur dans le cas d'une Applet) c'est un code binaire obtenu par compilation d'un programme source.

Un programme source est un texte contenant des instructions en langage JAVA. En général on s'attend à ce qu'un fichier `XXX.java` contienne la description d'un élément du langage appelé "classe" de nom `XXX`.

Le compilateur (`javac`) permet de générer un fichier exécutable de nom `XXX.class`.

C'est un fichier de ce type qui est (télé)chargé par un navigateur pour déclencher l'exécution d'une Applet .

Pour démarrer une application autonome on a besoin d'un fichier ".class" disposant d'un point d'entrée (`main`) et d'un exécuteur Java.

Point intermédiaire : *une application Java simple*

Après connexion sur votre compte utilisateur rédiger le fichier source **Calimero.java**

```
//
// l'application "Hello World" en version française
//
public class Calimero {
   public static void main (String[] args) {
      System.out.println("Bonjour Monde cruel!") ;
   }
}
```

Ces quelques lignes représentent un programme autonome minimum pour imprimer un message à l'écran. Quelques commentaires sur ce source

- Les trois première lignes (commençant par "//") constituent des commentaires.

- Le bloc suivant (`public class Calimero {....}`) constitue une définition de classe. A ce niveau du cours nous dirons qu'une classe désigne un module fonctionnel. Quelques remarques :

  - Dans le langage Java il n'y a pas de code en dehors des classes. En corollaire il n'existe pas de variable ou de fonction "isolée" (c'est à dire qui ne soit pas rattachée à une classe) :
    ligne 3 du programme `println()` est rattachée à une "variable" de classe nommée `out` elle même rattachée à la classe standard `System` . Les raisons de ces différents rattachements seront expliquées ultérieurement.

  - Pour toute classe déclarée dans un fichier le compilateur créera un fichier ".class" de même nom. Il y a, au plus, une seule classe marquée `public` dans un fichier source et le nom du fichier doit correspondre exactement au nom de la classe (une classe publique "Xyz" doit se trouver dans le fichier Xyz.java).

- Pour commencer l'exécution d'une application autonome à partir d'une classe donnée l'interprête recherche dans cette classe le point d'entrée standard désigné exactement par `public static void main (String[]` *`nomargument`*`)`

Compilation et exécution de Calimero.java

- A partir du fichier source Calimero.java la compilation se fait par ;

```
javac Calimero.java
```

- Si tout se passe bien (si le compilateur de renvoie pas d'erreur) on trouvera dans le répertoire courant un fichier exécutable "Calimero.class".

- Pour exécuter l'application faire :

```
java Calimero
```

Outre les éventuels problèmes d'accès aux commandes javac et java (problèmes de PATH) les problèmes les plus courants concernent :

- A la compilation: non correspondance entre le nom de la classe publique et le nom du fichier (la correspondance doit être exacte - y compris au niveau des minuscules / majuscules).

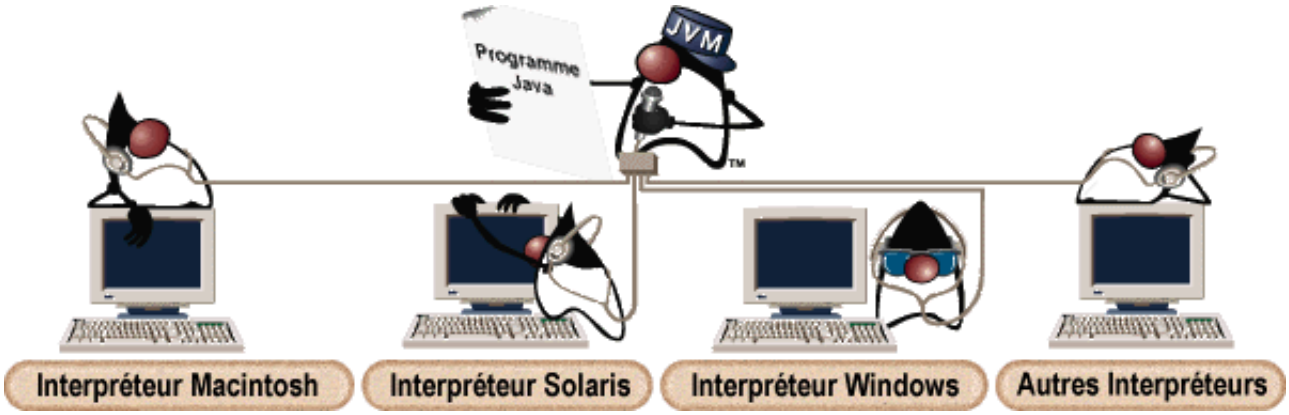
- A la compilation: erreurs de saisie dans le texte -erreurs de syntaxe, erreurs sur les noms, etc.-

- A l'exécution: l'exécuteur ne trouve pas la classe, soit parceque le nom de classe est mal saisie (erreur courante : "java Calimero.class"), soit parceque l'exécuteur ne trouve pas la ressource correspondante. En effet la commande java recherche la classe demandée dans un ensemble de répertoires comprenant des répertoires standard d'installation et des répertoires définis par la variable d'environnement CLASSPATH. Si cette variable est mal définie l'exécuteur risque de ne pas trouver le bon fichier.

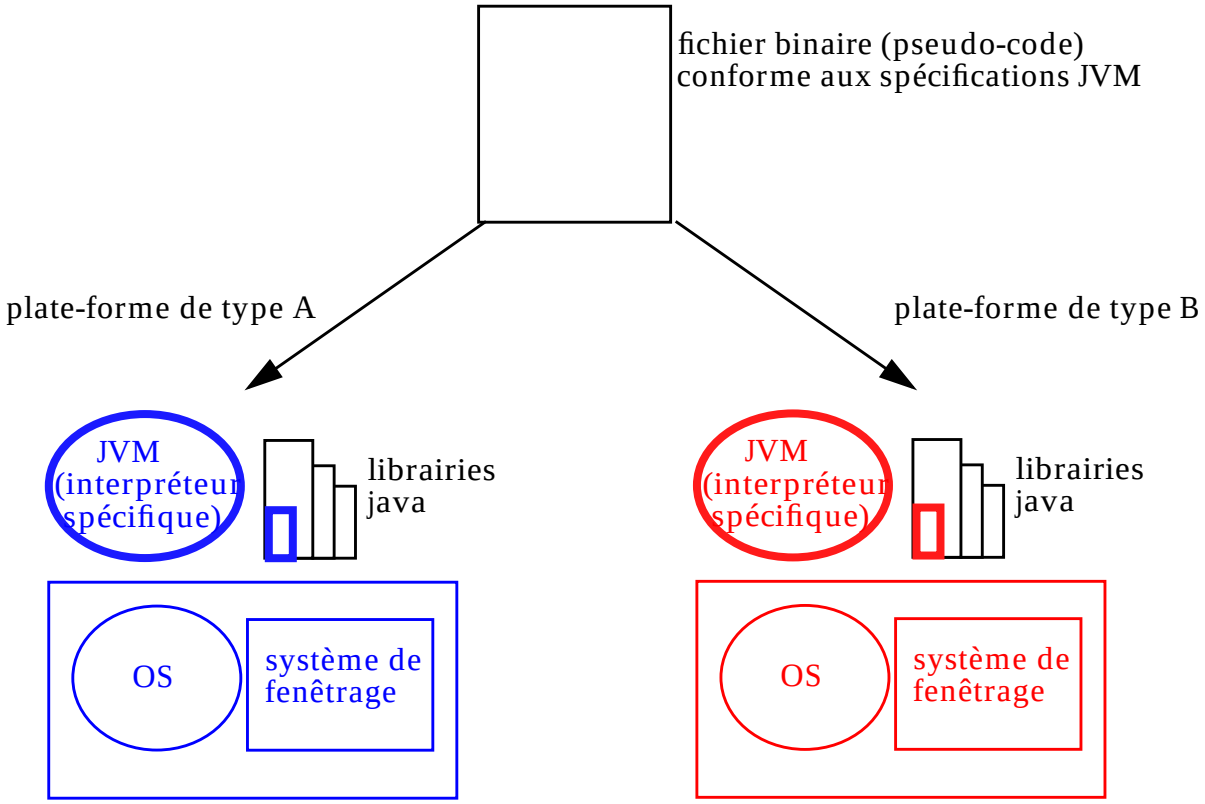
- Une autre source potentielle de déboires est une mauvaise définition du "main" qui serait absente ou qui ne correspondrait pas exactement à la signature
  `public static void main (String[] )`

Portabilité : la machine virtuelle

L e même “code” est exécuté par des plate-formes différentes :



Les JVM locales



La machine virtuelle

Que ce soit pour un navigateur ou pour une application autonome il y a un exécuteur de code Java.

Ce programme interprète le code binaire Java comme un code d'une **machine virtuelle**. Les spécifications du langage Java définissent très précisément la description et le comportement de cette machine virtuelle (c'est une "machine" à pile).

Le code "binaire" Java est en fait un pseudo-code (*bytecode*) portable: seules les implantations de la JVM (Java Virtual Machine) diffèrent selon les plate-formes.

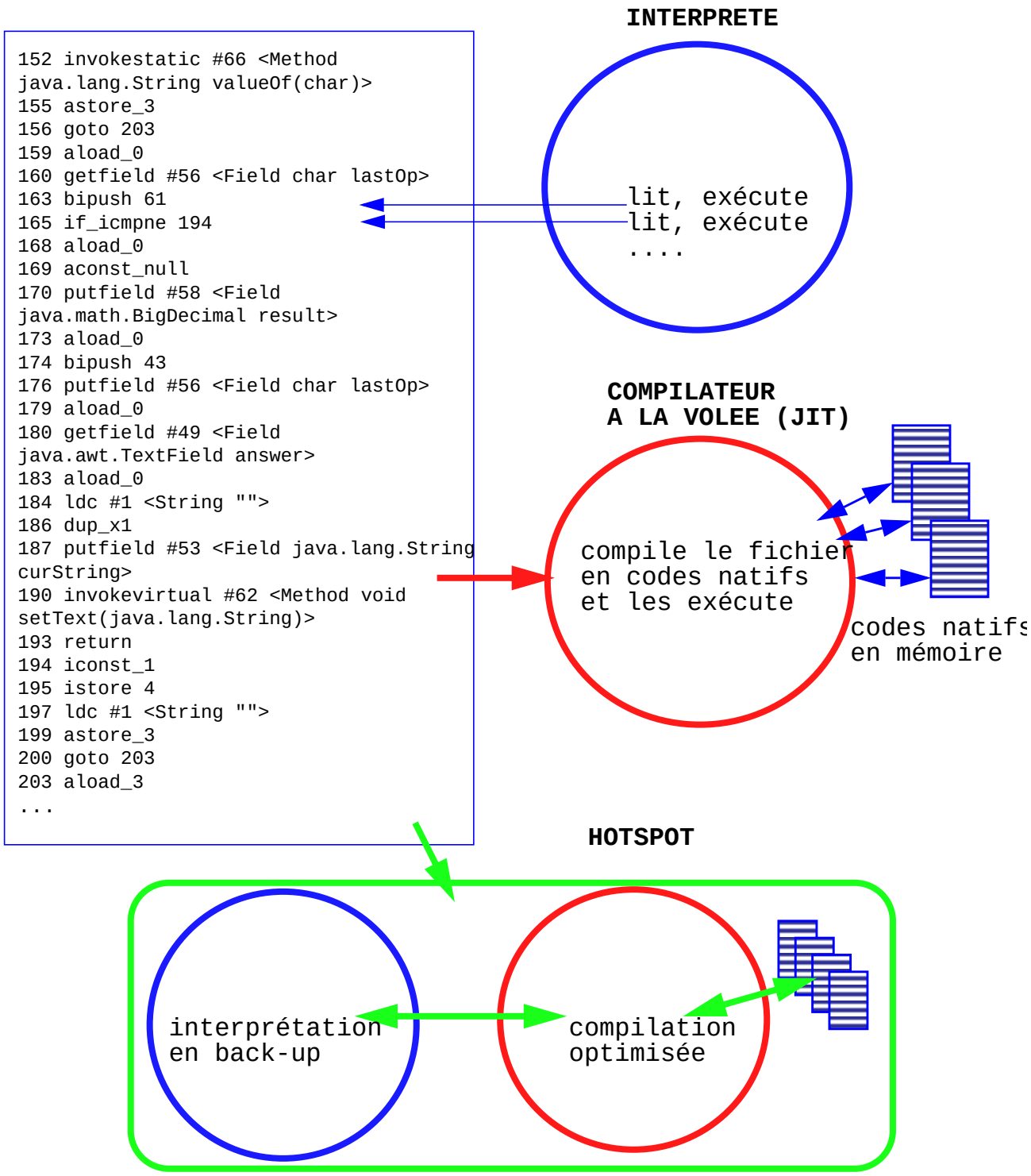
Pour exécuter un code Java sur un Système et une architecture donnée on doit donc trouver localement:

- Une implantation de la JVM (au sein d'un navigateur, ou pour lancer des applications autonomes)
- Les librairies constituant le noyau des classes standard Java. La plus grande partie est constituée de code Java (une partie est du code binaire spécifique à la plate-forme locale).

La *Java Virtual Machine Specification* définit entre autres:

- Le jeu d'instruction du pseudo-code
- La structure des registres et l'organisation de pile
- L'organisation du tas et de la mémoire (recupération automatique par *garbage-collector*)
- Le format des fichiers .class

Différents modes d'exécution



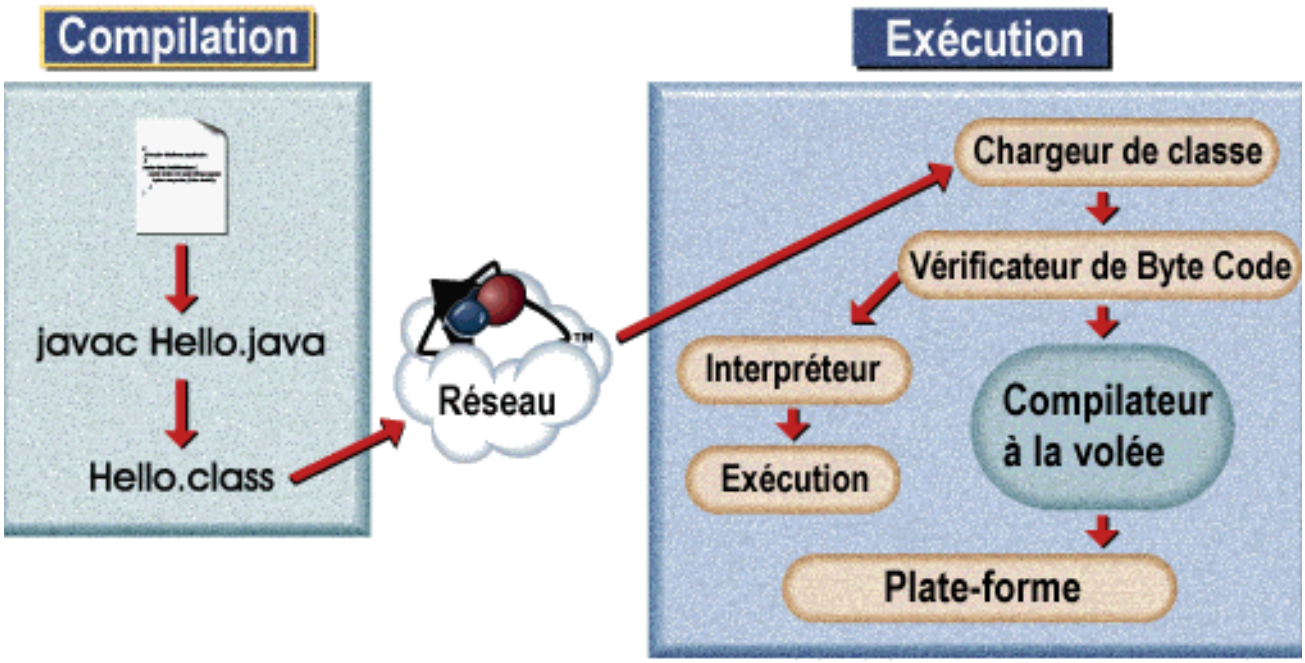
Les différents modes d'exécution

- Interprétation: Le programme qui implante le comportement de la JVM, lit le fichier de *bytecode* et exécute les opérations au fur et à mesure. C'est un interprête d'un code de bas niveau.
  Les interprêtes font des progrès constants en performance depuis les premières versions de Java.

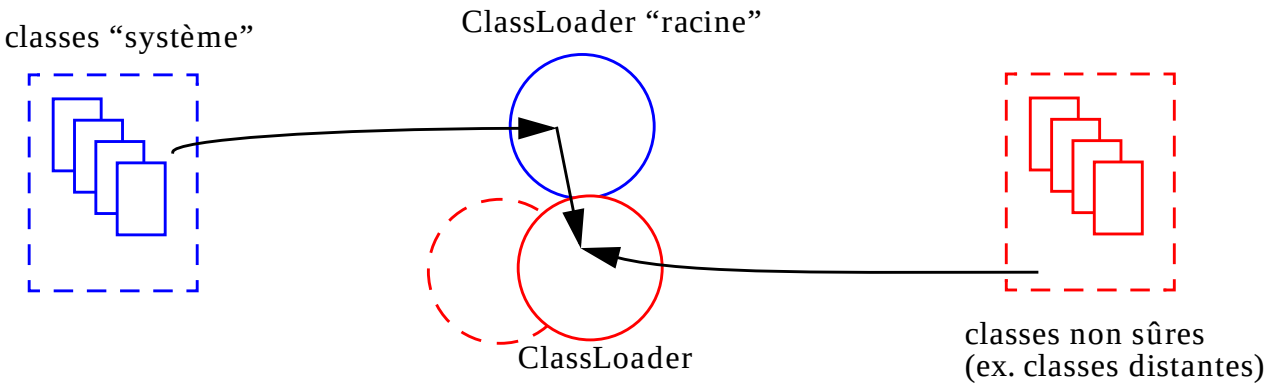
- JIT (Just-In-Time, compilation à la volée): Le programme chargé d'exécuter le code Java, compile directement ce pseudo-code en code natif de la plateforme locale, puis l'exécute. On fait le pari que le temps perdu par la compilation sera rattrapé du fait de la rapidité d'exécution du code natif.

- HotSpot : stratégies d'optimisation très complexes. L'optimiseur décide, au moment de l'exécution, s'il y a lieu de compiler ou d'interpréter (ex. boucle exécutée *N* fois : la situation n'est pas la même si *N* vaut 2 ou vaut des milliers). Par ailleurs ne cherche pas à compiler des cas rares et difficiles à compiler, fait de l'expansion de code (*inlining*), etc.

Sécurité : chargeur , vérificateur, gestionnaire de sécurité



Le chargement des classes



Sécurité : chargeur , vérificateur, gestionnaire de sécurité

Le chargement de code au travers du réseau pose un certain nombre de problêmes de sécurité :

- Quelle est le code Java que j'exécute? Est-il possible de m'adresser au travers du réseau un code (inamical?) qui remplace un de mes codes Java locaux?
  Le chargement de code au travers du réseau est de la responsabilité d'un **ClassLoader** (une classe Java).
  Un ClassLoader cherchera toujours à charger une classe dans les ressources système avant de s'adresser à d'autres ressources.

- Le code compilé téléchargé peut-il être corrompu et donc faire faire à la JVM locale des opérations illégales?
  Le **vérificateur de ByteCode** contrôle la conformité du code chargé (validité des opérations, validité des types, contrôles d'accès,...).

- Le code importé peut-il réaliser localement des opérations que je ne souhaite pas (obtention d'une information confidentielle, écriture dans le système de fichiers local, etc..)?
  Un **SecurityManager** fait respecter une **politique de sécurité**.
  Un objet **AccessControler** (une classe Java) est chargé de contrôler tous les accès critiques et de faire respecter une **politique de sécurité**
  Ainsi, par défaut, le code d'une Applet ne pourra pas connaître des informations critiques sur le site local (système de fichiers login, etc.), ne pourra pas écrire dans le fichiers locaux, déclencher des applications locales ou atteindre des sites sur le réseau autre que le site qui a distribué l'Applet elle-même.
  Des ressources locales permettent de définir des droits particuliers accordés à des intervenants dûment accrédités:

```
grant codebase "http://www.montaigne.fr/", signedBy "laBoetie"{
  permission java.io.FilePermission "/tmp/*", "read,write";
};
```

Exemple de fichier de configuration de sécurité

Robustesse: contrôles à la compilation et au run-time

La fiabilité de java s'appuie aussi sur de nombreux contrôles exercés soit par la machine d'exécution (*runtime*) soit par le compilateur (*compile-time*).

Le langage est très peu laxiste et de nombreux contrôles peuvent se faire à la compilation :

- vérifiation des "contrats de type" : affectation entre types compatibles, peu ou pas de transtypage implicite, obligation pour certains objets de remplir des obligations (*interface*),...
- vérification des obligations des méthodes (fonctions) : compatibilité des paramètres, obligation de retourner un résultat compatible avec le type déclaré en retour, ...
- vérification des droits d'accès entre objets : certains détails sont cachés et doivent le rester, interdictions éventuelles de redéfinition,..
- autres types de vérifications: si une variable locale risque d'être utilisée avant d'avoir été initialisée, si certaines variables (*blank final*) ne sont bien initialisées qu'une et une seule fois,...
- signale les obsolescences (évolutions de Java).

A l'exécution la JVM vérifie dynamiquement la validité de certaines opérations :

- accès hors limites dans des tableaux
- incompatibilités entre objets découvertes au *runtime*
- opérations illégales, etc.

Un mécanisme particulier de propagation et de traitement des erreurs permet d'éviter un effondrement intempestif et incontrolé des applications.

Robustesse : récupération des erreurs, gestion de la mémoire,

Les exceptions :

Dans le cas où le système d'exécution détecte une erreur au runtime, il génère un objet particulier appelé **exception**. Cet objet contient des informations sur l'incident et le système remonte la pile d'exécution à la recherche d'un code spécifique chargé de traiter l'incident. Si un tel code n'existe pas la machine virtuelle recherche une action par défaut (comme l'affichage d'un message) avant de terminer la tâche concernée

Les classes Java sont susceptibles de génerer toute sortes d'exceptions pour rendre compte des incidents spécifiques à telle ou telle action. Les programmeurs peuvent définir leurs propres exceptions, les déclencher et écrire du code pour les récupérer et prendre des mesures appropriées.

La gestion de mémoire :

Dans de nombreux langages comme C/C++ la gestion dynamique des objets en mémoire est une source constante de bugs très subtils et très coûteux en maintenance. Les "fuites mémoire" peuvent passer au travers des tests de validation et nécessitent des outils spécifiques de recherche et de validation.

Java entre dans la catégorie des langages disposant d'un "glaneur de mémoire" (ou "ramasse-miettes" -*garbage collector*-). On a un processus léger, de priorité basse, qui est réactivé de temps en temps et qui récupère automatiquement la mémoire occupée par des objets qui ne sont plus référencés. De plus ce processus compacte la mémoire occupée et gère de manière optimum la place disponible.

Cette gestion est facilitée par le fait que le programmeur n'a aucun moyen d'accéder directement à la mémoire (pas de pointeurs).

Utiliser la documentation

Utiliser la documentation

L'installation d'un SDK se fait en deux parties :

- installation de la JVM , des utilitaires et des librairies propre à votre architecture matérielle et logicielle. Voir `http://java.sun.com`

- installation de la documentation (identique pour toutes les plateformes).
  Consulter le fichier index.html dans le sous-répertoiree docs/api du répertoire d'installation de Java (bien entendu cette documentation peut-être installée pour être consultée en réseau).

La documentation de reférence a une structure d'hypertexte consultable par un navigateur HTML. Il est essentiel d'apprendre à rechercher des informations par ce biais.

La documentation des APIs Java a été directement extraite des sources de Java lui-même au moyen de l'utilitaire *javadoc*. Java dispose d'un formalisme standard qui permet d'inclure de la documentation directement dans le code. De cette manière vous pourrez vous-même documenter vos propres APIs.

Point intermédiaire : utiliser la documentation

- Lancer un navigateur
- Charger le document “racine” de la documentation (répertoire d’installation de Java)`/docs/index.html`
- Garder un “signet” (*bookmark*) sur ce document qui vous permettra d’y revenir pour parcourir tous les éléments de la documentation en ligne.
- Cliquer sur le lien “Java 2 Platform API Specification” (éventuellement sur le lien associé (NO FRAMES) qui permet d’avoir une présentation plus simple)
- Garder de nouveau un signet sur ce document qui est la racine de la documentation des classes standard Java.
- Cette documentation liste des “packages” (regroupement de classes). Cliquer sur le lien vers le package java.awt (les classes d’interactions graphiques)
- Dans la partie “Class Summary” cliquer sur la documentation de la classe “Button”.
- Comparer cette documentation avec le fichier source “Button.java” situé dans (répertoire d’installation de Java)`/src/java/awt/Button.java`)- Si ce répertoire “src” n’existe pas dans le répertoire racine de Java le créer par la commande qui décompresse la fichier “src.jar” (`jar -xvf src.jar`)

Exercices :

*Exercice * :*

Reprendre le programme Calimero.java, le saisir avec un éditeur, le compiler et l'exécuter.

*Exercice * :*

Déompiler le fichier Calimero.class en utilisant la commande `javap` (avec option `-c`).

Syntaxe, identificateurs, mots-clefs, types 3

Points essentiels

- Principes généraux de la syntaxe
- Identificateurs, mots-clefs
- Types primitifs
- Types “objets” : une introduction
- Types tableaux
- Allocation des différents types
- Conventions de codage

Syntaxe : généralités

Comme dans de nombreux langages informatiques un source Java est composé de “mots” qui sont délimités par des caractères (ou des ensembles de caractères) particuliers.

Ces “mots” peuvent être :

- des identificateurs : `nomDeVariable`, ...
- des mots-clefs réservés : `if` ,...
- des littéraux : `3.14`, `6.02e-9` , `“une chaîne”`, ...

Les délimitations :

- séparateurs : blancs, commentaires
- séparateurs de structure : `{` , `[` , `;` , ...
- opérateurs : `+` , `*` , `&&` , `>>=` ,...

```
if (variable < 0) {
        variable=0;// un commentaire est-il nécessaire?
}
```

Un texte source Java est en “format libre”, il n’y a pas de colonnes de format (comme en COBOL). Les seules contraintes sont le respect de la syntaxe (sinon le compilateur ne comprend plus rien) et le “bon usage” d’une présentation claire et aérée (sinon c’est le programmeur qui ne comprend plus rien!).

Commentaires

Trois notations sont possibles pour l'insertion de commentaires :

```
// commentaire sur une ligne

/* commentaires sur une ou plusieurs lignes */

/** insertion de documentation */
```

Les commentaires pour la documentation sont placés juste avant une déclaration JAVA et indiquent que son contenu doit être inclus automatiquement dans la documentation générée par l'outil *javadoc*.

Cette documentation est au format HTML.

Les règles de description et de formatage de cette documentation, ainsi que l'utilisation de l'outil *javadoc* se trouvent dans le document `./docs/tooldocs/javadoc/index.html` (sous répertoire d'installation du SDK JAVA 2)

Séparateurs

Une instruction se termine par un point virgule (;).

Elle peut tenir une ou plusieurs lignes.

```
total = ix + iy + iz + ij + ik+ il ; // une instruction
```

est la même chose que :

```
total = ix + iy + iz
          + ij + ik + il ;// deux lignes, une instruction
```

Un bloc regroupe un ensemble d'instructions, il est délimité par des accolades.

```
{ // un bloc
          ix = iy + 1 ;
          iz = 25 ;
}
```

Les blocs peuvent s'emboîter les uns dans les autres :

```
 {
          int ix = iy + 1 ;
          while ( ik < MIN ) { // bloc emboîté
                    ik = ik * ix;
          }
}
```

Les espacements (caractère espace, tabulation, retour chariot) peuvent se répéter à l'intérieur du code source. On peut les utiliser pour améliorer la présentation.

```
if(ix<iy)ix=iy;//compact

if ( ix < iy ) {
          ix = iy ;
} // plus aéré
```

Identificateurs

Dans le langage Java un identificateur commence par une lettre, un trait de soulignement (_), ou un signe dollar ($). Les caractères suivants peuvent également comporter des chiffres.
Les identificateurs opèrent une distinction entre majuscules et minuscules et n'ont pas de limitation de longueur.

Identificateurs valides:

- `identificateur`
- `fenêtre`
- `nomUtilisateur`
- `MaClasse`
- `_var_sys`
- `$picsou`

Les identificateurs peuvent contenir des mots-clés, mais ne doivent pas eux même être des mots-clés: "`longueur`" est un identificateur valide mais "`long`" ne l'est pas.

Consulter en fin de chapitre les conventions usuelles de nommage. Eviter autant que possible les caractères "$" dans les noms

Mots-clés

`abstract`	`do`	`implements`	`private`	`this`
`boolean`	`double`	`import`	`protected`	`throw`
`break`	`else`	`instanceof`	`public`	`throws`
`byte`	`extends`	`int`	`return`	`transient`
`case`	*`false`*	`interface`	`short`	*`true`*
`catch`	`final`	`long`	`static`	`try`
`char`	`finally`	`native`	`strictfp`	`void`
`class`	`float`	`new`	`super`	`volatile`
`continue`	`for`	*`null`*	`switch`	`while`
`default`	`if`	`package`	`synchronized`	assert (1.4)

Les littéraux `true`, `false` et `null` sont en minuscules (et non en majuscules comme en C++). Au sens strict il ne s'agit pas de mots-clés.

`goto` et `const` sont des mots-clés réservés mais inutilisés.
`assert` est un mot-clef à partir de la version 1.4.
Il n'y a pas d'opérateur `sizeof` qui donnerait la taille en octets d'une variable (comme en C++)

Déclaration de variables

En Java toute utilisation de variable doit être précédée d'une déclaration de son **type** (c'est à dire une déclaration des contraintes qui seront liées aux utilisations du symbole -aux utilisations du "nom" de cette variable-).

```
{ // exemples dans un bloc
          int x ; // "x" représente un entier Java
          String nom; // "nom" est d'un type défini
                      // par "String"
}
```

Pour chaque définition de variable il est conseillé de controler comment sont remplies les 3 conditions suivantes :

1. Comment la variable est-elle déclarée? (association type-symbole)
2. Comment la variable est-elle allouée en mémoire? (voir ci-après)
3. Comment la variable est-elle initialisée?

Portée d'une variable

Le langage Java définit précisément la "portée" d'une variable : c'est à dire la partie de code dans laquelle un symbole déclaré (le "nom" d'une variable) peut être utilisé et a un sens.

Dans un bloc de code on ne "voit" (on ne peut utiliser) que les symboles définis dans le bloc courant , ou le bloc englobant.

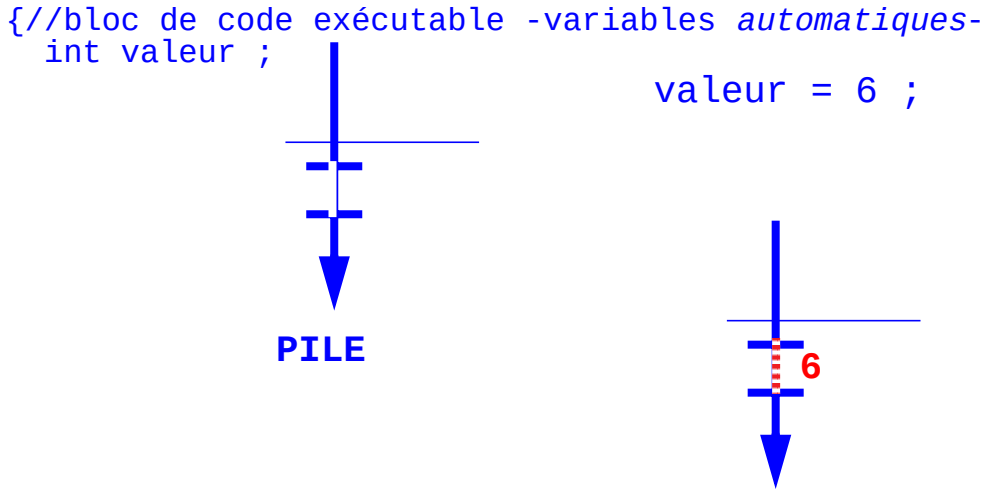
```
{// bloc englobant
        int nombre ;
        .....
        { // bloc interne
                int autreNombre ;
                ....
                nombre = 10 ; // correct
        } // fin bloc interne
        ...
        autreNombre = 7 ;    // ERREUR ! IMPOSSIBLE
        nombre = 20 ;       // possible
}
```

Certains dispositifs syntaxiques permettent aussi d'associer un symbole à un bloc :

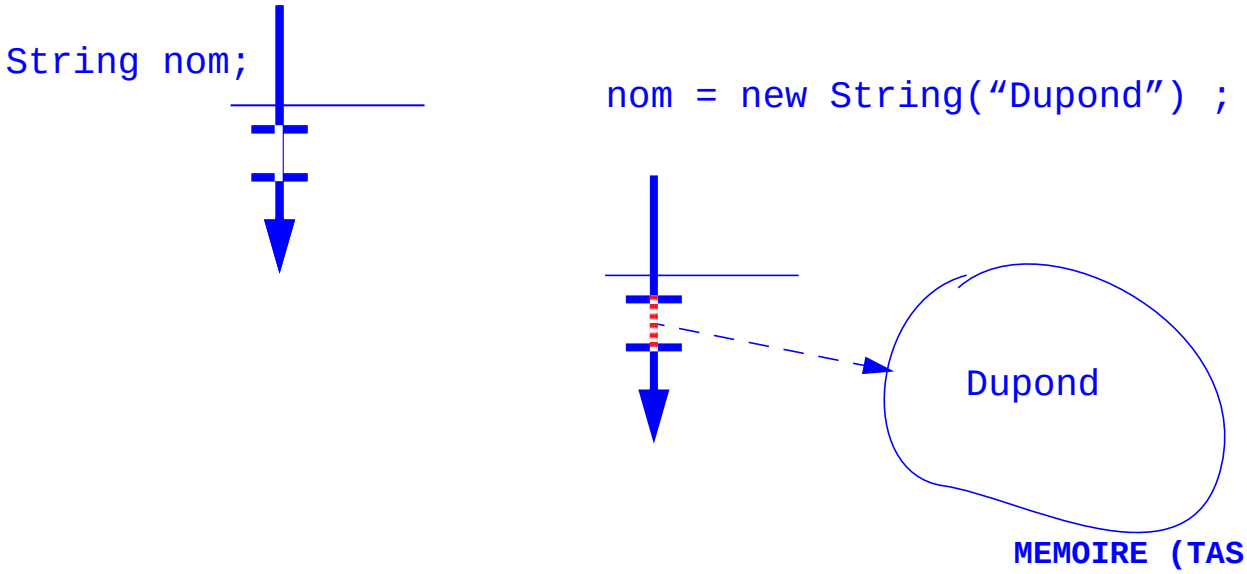
```
void service (int parm) {
        int x = parm ; // "parm" est défini dans ce bloc
        ...
} // fin "service"

void autreService( int autreParm) {
        int x = parm ; // ERREUR! IMPOSSIBLE
        ...
} // fin autreService
```

Types scalaires primitifs, types objets

Le langage Java distingue les types scalaires primitifs des autres types (objets). Dans un bloc de code la déclaration d'un type scalaire primitif, comme `boolean`, `int`, `float`, alloue un emplacement pour stocker une valeur du type considéré.



La déclaration d'une variable "objet" ne crée pas d'emplacement en mémoire pour stocker l'objet mais crée un emplacement pour stocker une *référence* vers un objet



Types primitifs : logiques

Les valeurs logiques ont deux états : vrai ou faux. En Java une telle valeur est représentée par une variable de type `boolean`. Deux littéraux seulement peuvent être employés pour représenter une valeur booléenne : `true` et `false`.

Voici une déclaration et une initialisation d'une valeur booléenne:

```
//déclare et initialise en même temps
boolean isOK = true ;
```

Attention! Il n'y pas d'équivalence entre les types entiers (comme `int`) et les booléens. Certains langages comme C ou C++ permettent d'interpréter une expression numérique comme représentant une valeur logique
Ceci est interdit en langage Java: lorsqu'une expression booléenne est requise, rien d'autre ne peut la remplacer.

Types primitifs: texte

Un caractère isolé est représenté à l'aide du type `char` .

Une valeur de type char représente un caractère UNICODE (UTF-16 : 16 bits, non signé) qui permet de représenter des caractères très divers (langues orientales, symboles).

Un littéral de type `char` doit figurer entre deux apostrophes :

```
'a'                 // le caractère a
'\t', '\n', '\r'// une tabulation, ligne, retour chariot
'\f', '\b', '\'','\\'// form-feed , backspace , apostrophe, \
'\u0240'            // un caractère UNICODE
        // code de la forme \uXXXX(4 chiffres hexadécimaux)
'\177'  // code caractère en octal (limité à 8 bits)
```

Une chaîne de caractères peut être représentée au moyen du type `String`.

Le type `String` n'est pas un type scalaire primitif mais un type prédéfini représenté par une *classe*. Cette classe a la particularité d'avoir une notation pour les constantes littérales :

```
String titre = "MENU DU JOUR \"ROYAL\" " ;
String platRecommandé = "Rôti de b\u0153uf\t 10 \u20AC";
          // 10 Euros pour un Rôti de boeuf !
          // chaînes dans le pool de constantes

String autreDupond = new String("Dupond") ;
```

Attention: contrairement à C/C++ il n'y a pas de terminateur \0 en fin de chaîne.

Caractères UNICODE : `http://www.unicode.org/charts`

Les chaînes `String` sont *immuables*: on ne peut pas, par exemple, modifier un caractère sans passer par la création d'une autre chaîne.

Types primitifs: numériques entiers

Il existe quatre types entiers dans le langage Java : `byte`, `short`, `int`, et `long`.

Tous ces types entiers sont des nombres signés.

La spécification du langage définit une implantation portable indépendante de la plate-forme (ordre des octets, représentation en complément à deux).

Taille implantation	**Type**	**Plage valeurs**
8 bits	**`byte`**	-2^{7} ... $2^{7}-1$
16 bits	**`short`**	-2^{15} ... $2^{15}-1$
32 bits	**`int`**	-2^{31} ... $2^{31}-1$
64 bits	**`long`**	-2^{63} ... $2^{63}-1$

Annexe

Ordre des octets dans un “int” portable et complément à deux

poids fort poids faible

0000 0000 0000 0000 0000 0000 0000 0001	1
0000 0000 0000 0000 0000 0000 0000 0010	2
0000 0000 0000 0000 0000 0000 1111 1111	255
1111 1111 1111 1111 1111 1111 1111 1111	-1
1111 1111 1111 1111 1111 1111 1111 1110	-2
1111 1111 1111 1111 1111 1111 0000 0001	-255

(complément à deux pour les nombres négatifs = complément à 1 + 1)

Constantes littérales entières

Les constantes littérales correspondants aux types entiers peuvent se représenter sous forme décimale, octale ou hexadécimale :

```
98          // la valeur décimale 98

077         // le zero en debut indique une notation octale
            // valeur : 63

0xBEBE      // 0x indique une notation hexadécimale
            // valeur : 48830
```

Les expressions concernant des nombres entiers sont de type `int` sauf si on précise explicitement une valeur `long` au moyen d'un suffixe "L". Les "L" minuscules et majuscules sont acceptés, mais il est préférable de ne pas utiliser la minuscule qui peut être confondue avec le chiffre 1.

```
98L         // valeur décimale de type long

077L        // 63 sur un long

0xBEBEL     // 48830 sur un long
```

Annexe

notation octale: base 8, les nombres sont représentés par les chiffres 0-7. La représentation est pratique pour visualiser des blocs de 3 bits (2^3). Ex.: 07070 -> 111000111000

notation hexadecimale: base 16, les nombres sont représentés par les chiffres 0-9 et les lettres A,B,C,D,E,F. La représentation est pratique pour visualiser des blocs de 4 bits (2^4). ex.: 0xF0F0 -> 1111000011110000

Types primitifs: numériques flottants

Il existe deux types numériques flottants dans le langage Java : `float` et `double` .

La spécification du langage définit une implantation portable indépendante de la plate-forme (I.E.E.E 754)

Taille implantation	Type
32 bits	**`float`**
64 bits	**`double`**

Une expression littérale numérique est un flottant si elle contient un point, une partie exponentielle (lettre E ou *e*) ou si elle est suivie de la lettre F ou D (*f* ou *d*).

```
3.14      //notation simple, valeur double -par défaut-

3.        // encore un double (valeur 3.00)

.3        // double (valeur 0.3)

3D        // double (risque confusion avec notation hexa)

6.02E23   // double en notation scientifique

123.4E-300D // D redondant

2.718F    // un float

2F        // danger : confusion avec notation hexa

2.F       // 2.00 float
```

(Pour les spécialistes) Java reconnait valeurs infinies, zeros signés et *NaN* :

```
System.out.println(-1d/0d);
 -Infinity
```

`Point intermédiaire`: *les types primitifs scalaires*

Les types primitifs scalaires sont “cablés” dans le langage (on ne peut pas en définir de nouveaux).

Les valeurs de ces types ont des représentations standard portables.

- type logique : `boolean`
- types entiers : `byte, short, int, long`
- type caractère : `char` (UNICODE 16 bits)
- types flottants : `float, double`

Mini-exercice dans un `main` d’une classe “TestPrimitifs” :

- déclarer une variable de type `int` x; une de type `short` y.
- initialiser y avec la valeur `0x7FF`, mettre la valeur de y dans x (instruction: `x = y;`)
  tracer ces valeurs par l’appel de `System.out.println`
- Créer une variable de type `char` de nom “car” et l’initialiser; affecter “car” à “x”; tracer “car” et “x” par `System.out.println`;
- Compiler et exécuter le programme.
- Modifier le programme en affectant “car” à “y”. Compiler, que constatez vous?

Objets: agrégats, création explicite par new

De nombreux langages de programmation disposent de moyen de créer de nouveaux types par agrégation d'un ensemble de valeurs individuelles (ayant éventuellement des types différents). Ces regroupements sont appelés types structurés (*struct* du langage C, *structure* ou *zone groupe* de COBOL) ou types enregistrements (*Record* de Pascal).

Une définition analogue en Java se ferait au moyen d'une classe:

```
class Individu {
          String nom ;
          int age ;
          float salaire ;
}
```

Une telle approche fournit un moyen d'associer des parties qui constituent le modèle d'un "individu" en les traitant non pas comme des éléments isolés mais comme des composants d'un ensemble plus vaste. Un seul nom est requis pour désigner une variable qui représente un "individu".

```
Individu quidam ;
```

Comme toute déclaration de type non-primitif , cette déclaration n'alloue pas la mémoire nécessaire pour stocker ses composants et les gérer, il faut explicitement demander cet espace ;

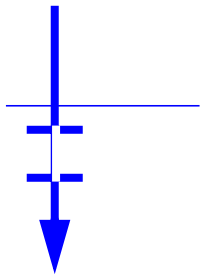
```
quidam = new Individu() ;
```

Après cette déclaration Java permet l'accès aux membres de *l'objet* ainsi créé. Ceci se fait à l'aide de l'opérateur "." :

```
quidam.salaire = 10000.f ;
quidam.age = 24 ;
quidam.nom = ”Dupond” ;
```

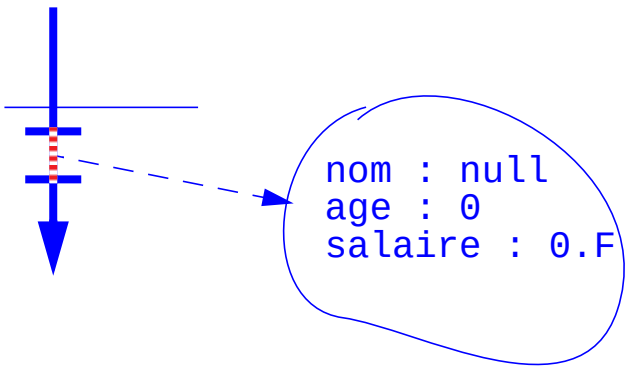
Objets: allocation

```
Individu quidam;
```



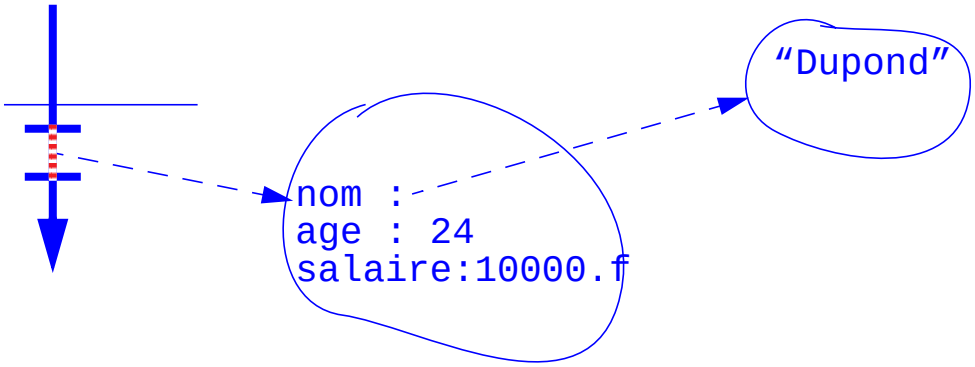
La variable “quidam” existe mais ne référence personne

```
quidam = new Individu();
```



La mémoire a été allouée, la référence désigne un Individu
les membres de l’objet ont des valeurs nulles

```
quidam.nom = “Dupond”;
quidam.age = 24 ;
quidam.salaire = 10000.f ;
```



Objets: introduction à la terminologie

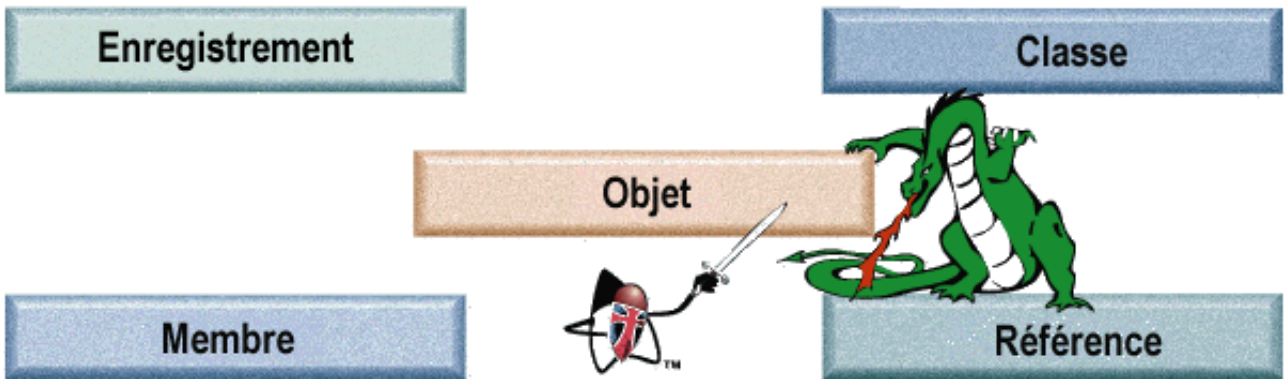
Premier rappel sur la terminologie “Objet” :

- *Classe* - cadre permettant de définir de nouveaux types dans le langage de programmation Java. La déclaration de classe décrit, entre autres, les composants associés à toute variable de ce nouveau type.
  De nombreux autres aspects seront vus ultérieurement.

- *Objet* - *instance* d’une classe. Une variable de type classe doit référencer une zone mémoire créée par `new` et obéissant aux contraintes imposées par la définition de classe. La classe est un modèle, l’objet est ce que vous fabriquez à partir de cette matrice.

- *Variable membre* - un membre est l’un des éléments qui constituent un objet.
  Les termes : *variable membre, variable d’instance, attribut* ou *champ* sont également utilisés. Un objet peut compter d’autres membres que des champs (ceci sera vu ultérieurement).

- *Référence* - en Java une variable définie comme ayant un type d’une classe donnée sert à désigner un objet de ce type. En fait c’est une référence vers une instance.



Point intermédiaire : les types objets

Les variables de type “objet” sont des références vers des zones mémoire dans le tas qui peuvent “contenir” plusieurs autres valeurs.

Ces types peuvent être définis par le programmeur .

Mini-exercice :

La plupart de nos mini-exercices se situent dans le monde (virtuel) des enfants sages dans lequel le seul commerce digne de ce nom est celui des confiseries. Dans ce qui suit une “confiserie” est , par exemple, un bonbon et non le magasin dans lequel on les vend.

- créer une classe de nom “Confiserie” qui contienne un champ “nom” de type `String` et un champ “prix” de type double. Compiler ce fichier source.

- créer une classe “TestObjet” avec un `main`, créer une instance de Confiserie, initialiser les valeurs des champs. Faire `System.out.println` de cette instance et `System.out.println` de l’un de ses champs. Compiler, exécuter

Tableaux : déclaration

Un tableau regroupe un ensemble de données d'un même type. C'est un objet (non-scalaire) avec un définition simplifiée par rapport à celle d'une classe.

```
char[] tChars ; // déclaration d'un tableau de "char"
                // tableau de scalaires

String[] tMots ; // déclaration d'un tableau de String
                 // tableau d'objets
```

Les déclarations : `char tChars[];` ou `String tMots[];` sont possibles, on notera toutefois que le type ne précise pas le nombre d'éléments du tableau.

Comme pour les autres objets ces déclarations ne créent pas d'instance, il faut donc allouer ces objets (en précisant leur dimension):

```
tChars = new char[20] ; // le tableau existe
          // avec 20 caractères '\0'

tMots = new String[3] ; // tableau de 3 références String
          // toutes nulles!
```

Il faut ensuite initialiser les éléments du tableau:

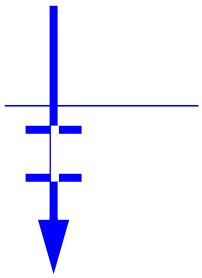
```
tChars[0] = 'A'; tChars[1] = 'B' ; ....

tMots[0] = "Ga" ; tNoms[1] = "Bu" ;....
```

On notera que les index commencent à zéro.

Il existe aussi des notations littérales de tableaux qui permettent de simplifier les déclarations et les initialisations.

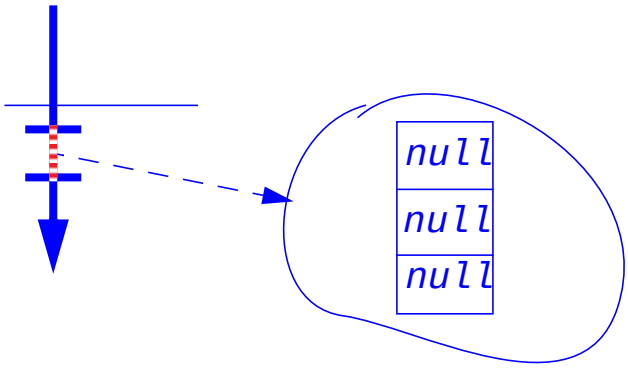
Tableaux: allocation

```
Individu[] personnages;
```



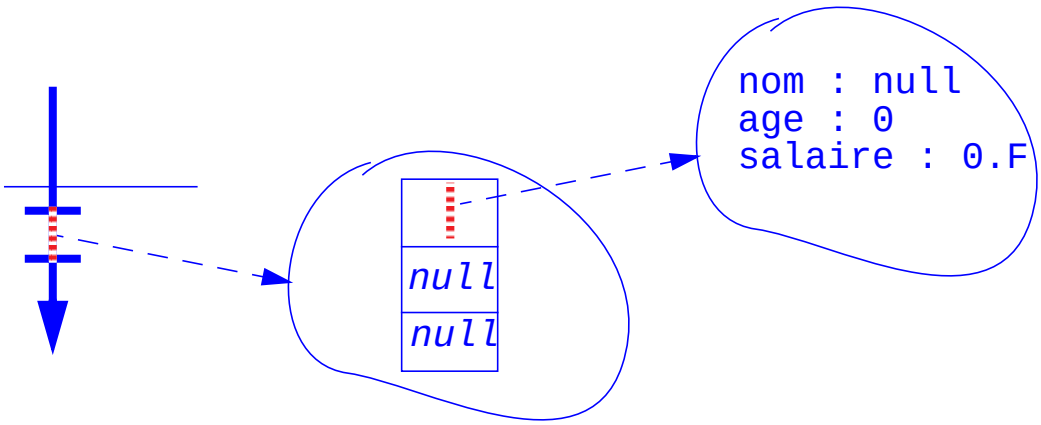
La variable “personnages” existe (est typée) mais ne référence rien elle ne peut être utilisée

```
personnages = new Individu[3];
```



La mémoire a été allouée, la référence désigne un emplacement mémoire de trois références (toutes à *null*) vers des instances de `Individu`

```
personnages[0] = new Individu();
```



Tableaux: initialisations

Cette notation permet d'allouer et d'initialiser un tableau :

```
tMots = new String[] { ”Ga”, ”Bu”, ”Zo”, ”Meu”} ;
```

Ici l'allocation ne précise pas le nombre d'éléments qui est fixé par la description littérale qui suit.

De manière plus courante on utilise une notation raccourcie qui regroupe déclaration, allocation et initialisation :

```
String[] shadock = {
          ”Ga” ,
          ”Bu” ,
          ”Zo” ,
          ”Meu”
} ;
```

Cette notation est autorisée pour tout type d'élément :

```
char[] monnaies = {
          ’\u20AC’, // Euro
          ’\u00A3’ ,// Livre sterling
          ‘\u20A4’ ,// Livre turque
          ’$’ ,
          ’\u00A5’ // Yen
};

Individu[] candidats = {
          new Individu() ,
          new Individu(),
          new Individu() ,
};

double[] vals = { Math.PI, Math.E } ;// constantes

Color[] palette = {
          Color.blue ,
          Color.red ,
          Color.white,
} ; // ces objets couleurs sont des constantes
```

Voir aussi: *Tableaux multidimensionnels*, page 99.

Limites de tableau

Dans le langage Java, les indices des tableaux sont des entiers commençant à zero (pas d'index négatif!).

Le nombre d'éléments dans le tableau est mémorisé dans le tableau lui même à l'aide du champ `length`. Cette valeur est utilisée lors de tout accès au tableau pour détecter les dépassements d'indices. La machine virtuelle Java génère une erreur d'éxécution en cas d'accès illégal.

```
int[] dim1;
....
          // +allocations + initialisations

System.out.println(dim1.length) ;
```

On pourra ainsi utiliser ce champ `length` comme limite de boucle pour parcourir un tableau. Attention: `length` ne peut pas être modifié!
Une fois créé un tableau ne peut pas être redimensionné.
Cependant il est possible d'utiliser la même variable référence pour désigner un nouveau tableau :

```
int monTableau[] = new int[6] ; ....
monTableau = new int[10]; ...
```

Dans ce cas le premier tableau alloué est effectivement perdu (sauf s'il existe une autre référence active qui le désigne).

Pour utiliser des objets qui permettent de stocker des collections d'autres objets sans se soucier autre mesure des limites utiliser les classes de *collection* du package `java.util` (comme `ArrayList` ou `HashMap`).

Voir aussi: *Copie efficace des éléments d'un tableau*, page 100.

Accès aux éléments d'un tableau

Soit le tableau:

```
Individu[] personnages;
..... // initialisation diverses
```

par la suite on pourra consulter les éléments du tableau ;

```
Individu individuCourant = personnages[0] ;
```

mais on pourra aussi consulter un champ d'une référence anonyme (c'est à dire qu'on ne crée pas une référence de type "Individu" comme dans l'exemple ci-dessus):

```
System.out.println(personnages[0].nom) ; // le nom du premier
                                         // Individu du tableau
```

Point intermédiaire : *les types tableau*

Les tableaux sont des objets qui contiennent une séquence d'éléments du même type.

Chaque "type" tableau est caractérisé par le type de ses éléments, un tel type peut désigner des instances de tailles différentes (la taille ne fait pas partie du type). Une fois allouée une instance de tableau ne peut pas changer de taille.

Mini-exercice : Dans un fichier source de nom "TestTableaux" doté d'un `main`:

- Créer et initialiser un tableaux d'entiers et un tableau d'objets de type "Confiserie". Les deux tableaux doivent avoir une taille supérieure à 2.
- Faire `System.out.println` sur le deuxième élément de chaque tableau.
- Compiler , exécuter

récapitulation: déclarations de variables

```
// la classe “Individu” est supposée connue par cette classe

public class Affectations {
        public static void main (String[] tArgs) {
                //SCALAIRES PRIMITIFS
                int ix;
                int iy ;
                float fx, fy; //deux déclarations
                ix = -1 ;
                iy = 0xFF ;
                fx = 3.414f ;
                fy = 3e-12f ;
                double dx = 3.1416 ;//double par défaut
                boolean estVrai = true ;
                char euro = ’\u20AC’ ;

                //OBJETS
                String nom ;
                nom = ”roméo” ;
                Individu roméo ; //déclaration
                roméo = new Individu(); // allocation
                roméo.nom = nom ; //initialisation
                System.out.println(roméo.age) ; // 0 !
                Individu juliette = new Individu() ;

                Individu[] quidams; //déclaration
                quidams = new Individu[3] ; //allocation
                quidams[0] = new Individu();// initialisation
                System.out.println(quidams[1]); // null!
                Individu[] acte2scene2 = { roméo, juliette };
        }
}
```

Quelques affectations illégales :

```
iy = 3.1416 ; //le litteral est un double pas un int
fx = 3.1416 ; // idem: double pas float
dx = 3,1416 ; // pas de “,” comme séparateur décimal
estVrai = 1 ; // piège pour programmeurs C/C++
```

Conventions de codage

Il est très vivement conseillé de respecter les conventions de codage suivantes ;

- *Classes* - les noms des classes sont composés d'un ou plusieurs mots accolés, chaque mot commençant obligatoirement par une majuscule:
  ```
  class LivreDeCompte
  class Compte
  ```
  La même convention s'applique aux *interfaces* (un autre type que nous verrons ultérieurement).
  Ne pas utiliser : caractères accentués, '_', '$'

- *Variables, variables membres* - les noms de variables commencent obligatoirement par un caractère minuscule.
  ```
  int nombreClients ;
  ```
  La même convention s'applique aux *méthodes* (aux "fonctions" dans le langage Java).
  Eviter : les caractères '_' et '$' , les noms sur une seule lettre

- *Constantes* - Noms complètement en majuscules avec le caractère "_" (souligné) pour séparer les mots.
  ```
  TAILLE_MAX
  PI
  ```

- *Packages* - Noms complètement en minuscules
  fr.asso.gourousjava.util

Placez une instruction par ligne et utilisez une indentation pour que le code soit plus lisible.

voir `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`. Pour les commentaires de documentation lire par exemple :"The Design of Distributed Hyperlinked Programming Documentation." `http://www.javasoft.com/docs/javadoc-paper.html`

Compléments

Les compléments techniques suivants constituent une annexe de référence

Tableaux multidimensionnels

Le langage Java a une manière particulière de permettre la création de tableaux multi-dimensionnels. Dans la mesure où on peut déclarer des tableaux de n'importe quel type, on peut déclarer des tableaux de tableaux (ou des tableaux de tableaux de tableaux, etc.)

Voici, par exemple un tableau à deux dimensions :

```
int[][] dim2 = new int[3][] ;
dim2[0] = new int[5] ;
dim2[1] = new int[5] ;....
```

Le premier appel crée un tableau de 3 éléments. Chacun des éléments est une référence nulle qui devra désigner un tableau d'entiers, chacun de ces éléments est ensuite initialisé pour référencer un tel tableau (qui doit être ensuite rempli.)

Une déclaration comme : `new int[][3];` ne serait pas légale, par contre une notation synthétique comme : `new int[3][5];`est possible.

Du fait de ce mode de construction il est possible de réaliser des tableaux de tableaux qui ne soient pas rectangulaires. Ainsi l'initialisation d'un des éléments du tableau précédent pourrait être :

```
dim2[2] = new int[2] ;
```

Et en utilisant une notation qui regroupe déclaration, allocation et initialisation:

```
int[][] dim2 = {
          { 0 , 1, 2, 3, 4 } ,
          { 6 , 7, 8, 9, 5 } ,
          { 10, 20 }
} ;
```

Copie efficace des éléments d'un tableau

Une copie efficace de tableau s'effectue au moyen d'une fonction de service définie dans la classe `System` :

```
int[] origine = { 11, 22, 33 } ;
int[] cible = { 0, 1, 2, 3 } ;
System.arraycopy(origine, 0, cible, 1, origine.length);
// copie origine dans cible a partir de l'index 1
// resultat : { 0 , 11, 22, 33 }
```

System.arraycopy() copie des références pas des objets lorsqu'on opère sur des tableaux d'objets. Les objets eux-même ne changent pas.

L'allocation des `String`

Les constantes de type chaîne de caractères sont allouées dans un *pool* de constantes, alors que les chaînes allouées par `new` sont allouées dans le tas.

```
String nom = "toto" ;
String autreNom = "toto" ;
String encoreUnNom = new String("toto") ;
```

Les références "nom" et "autreNom" sont les mêmes (elles désignent la même chose en mémoire), par contre "encoreUnNom" est une reférence différente qui désigne une zone mémoire allouée par `new`.

Corrigés des mini-exercices

TestPrimitifs.java :

```
public class TestPrimitifs {
   public static void main (String[] args) {
      int x ;
      short y = 0x7FF  ;
      x = y ;
      System.out.println(x) ;
      System.out.println(y) ;
      char car = 'A' ;
      x = car ; // un char est affectable à un entier
      System.out.println(car) ; // affiche A
      System.out.println(x) ;   // affiche 65
      //y =car ; NE MARCHE PAS
      // un char n'est pas affectable à un short
      // meme taille mais l'un est signé (short)
      // l'autre pas
   }
}
```

Confiserie.java:

```
public class Confiserie {
   String nom ;
   double prix ;
}
```

TestObjet.java:

```
public class TestObjet {
   public static void main (String[] tb) {
      Confiserie unBonbec;
      unBonbec = new Confiserie() ;
      unBonbec.nom = "bétise"  ;
      unBonbec.prix =  0.1 ; // nos prix sont en Euros!
      System.out.println(unBonbec) ;
       // donne un truc bizarre comme : Confiserie@2542fbc9
      System.out.println(unBonbec.prix) ;
   }
}
```

TestTableaux.java :

```
public class TestTableaux {
   public static void main (String[] tbArgs) {
      int[]  tbInt = {-1, 0, 0xFFFFFFFF } ;
      Confiserie[] tbObj = {
         new Confiserie(),
         new Confiserie(),
         new Confiserie()
      } ;
      System.out.println(tbInt[1]) ; //0
      System.out.println(tbObj[1]) ;// un truc "bizarre"
   }
}
```

Exercices :

Dans les exercices suivant il est demandé de “tracer” des instances: vous pouvez utiliser `System.out.println` ou faire appel aux services d’une classe qui sera mise à votre disposition par l’animateur.

*exercice * : une classe simple*

- Soit une classe “Point” avec deux variables membres de type entier “x” et “y”.
- Définir un “main” dans cette classe pour la tester : créer deux instances de Point de nom pt1 et pt2 , initialiser ces instances avec des valeurs et les tracer par “System.out.println”.

*exercice ** (suite du précédent): manipulation de références*

- Créer une nouvelle variable de type Point et de nom “copie”, affecter à “copie” la variable “pt2” (rappel : faire copie = pt2;), tracer pt2, copie
- Modifier les champs de “copie”, tracer pt2, copie
- Créer un tableau de “Point” contenant pt1, pt2, copie
- Modifier les champs du deuxième élément du tableau, tracer le contenu des éléments du tableau.

Syntaxe: expressions et structures de contrôle 4

Points essentiels

- Notion de méthode
- Opérateurs
- Conversions
- Structures de contrôle

Notions de méthode: principes, paramètres, résultats

Dans de nombreux langages de programmation la réalisation d'un service est confiée à une fonction. Ces fonctions sont généralement de la forme:

```
resultat = fonction (x, y, z) ;
```

La forme générale d'une déclaration de fonction est :

type de résulat nom de la fonction (liste ordonnée de paramètres)

```
double annuité ( double montantPrêt,
                 int durée ,
                 double taux ) {
        // code avec un ou plusieurs return valeur double
}
```

En Java un dispositif analogue se trouve dans la définition et l'usage de **méthodes** .

Une définition de méthode comprend:

- un nom qui la désigne (voir règles de nommage)
- une déclaration de résultat :
  - si aucun resultat n'est prévu le type déclaré est `void`
  - si un résultat d'un type donné est déclaré il doit y avoir dans le code un (ou plusieurs) `return` d'une expression du type demandé.
- une liste de paramètre (éventuellement vide):
  chaque paramètre constitue une déclaration de variable (type, nom) dans la portée du bloc de définition de la méthode.

```
// définitions
void procédure(int x) {// définition
}

int nombreAuHasard() { // définition
}
```

Méthodes d'instance

Dans les langages à objets les "services" sont, en général, demandés à une instance donnée. Exemple:

```
public class Position {
        int posX ;
        int posY;

        // DEFINITION DE LA METHODE
        boolean estDansPremierQuadrant() {
                return ...//expression permettant de dire
                // si la combinaison des champs posX, posY
                // donne une position dans premier quadrant
                // (posX ET posY >0)
        }
}
```

et son utilisation par un objet de type "Position" :

```
Position maPosition ;
... // allocation + initialisations des champs
// UTILISATION DE LA METHODE
if(maPosition.estDansPremierQuadrant()) {...}
```

On a ici demandé un service à une instance particulière de la classe `Position`. Cette requête concerne un propriété rendue par la combinaison des valeurs contenues dans l'objet: on parle de méthode d'instance. De manière imagée on qualifie aussi cette "question" adressée à une instance d'"envoi de message".

Au contraire de langages dans lesquels l'appel de fonction se fait dans un contexte général :

```
y = f(x) ; // en langage C par exemple
```

En Java l'appel d'une méthode se fait forcément soit dans le contexte d'une instance, soit, nous allons le voir ensuite, dans le contexte d'une classe:

On notera que les appels des services ont nécessité l'emploi de l'opérateur de qualification "**.**": les méthodes sont des membres (comme les champs). Dans l'exemple ci-dessus "`estdansPremierQuadrant`" est membre d'une instance de la classe "`Position`" .

Contexte des méthodes d'instance

Les méthodes d'instance ont la possibilité d'accéder aux variables membres de l'instance courante (posX et posY dans l'exemple).

La notation générale pour l'accès aux membres est :

```
référence_instance.membre
```

Si l'instance désignée est l'instance courante on la note **this** (mot reservé), d'où la rédaction:

```
class Position {
        int posX ;
        int posY;

        // DEFINITION DE LA METHODE
        boolean estDansPremierQuadrant() {
                return (this.posX > 0) && (this.posY > 0);
                // sens de l'expression : posX positif
                //        ET posY positif
        }
}
```

Les méthodes d'instance peuvent donc manipuler des variables d'instance et des variables locales "automatiques" (paramètres et variable déclarées dans les blocs de la méthode)-Nous verrons ultérieurement d'autres variables qui sont les variables partagées- . Bien entendu le code des méthodes d'instance peut aussi faire appel à d'autres méthodes de la même instance.

Méthodes de classe :

Certains services n'ont pas besoin de manipuler l'état d'une instance, ils opèrent pratiquement en dehors du contexte d'un objet particulier.

Exemple: pour calculer un sinus en Java on pourra faire appel à

```
double x = 3.14 ;
double res = Math.sin(x)
```

Cette notation permet de rappeler qu'il n'existe pas de fonction "libre" en Java: on demande le service de calcul du sinus à une classe (`java.lang.Math` en l'occurence). L'intérêt de cette approche apparaît quand on peut écrire :

```
double autreRes = StrictMath.sin(x) ;//Java 1.3
```

qui propose un autre mode de calcul.

La consultation de la documentation de la classe java.lang.Math décrit la méthode "sin" avec l'en-tête suivant :

```
// DESCRIPTION DE l'EN TETE de DEFINITION
static double sin (double a) ...
          // à ne pas confondre avec l'utilisation!
```

Le mot-clef "static" indiquant qu'il s'agit d'une méthode correspondant à un service proposé par une classe.

```
// DEFINITION DU MAIN STANDARD
public static void main (String[] tbArgs) {..// procédure
```

On notera donc que la notation "." permet de rattacher un membre soit à une instance soit à une classe :

```
maPosition.estDansPremierQuadrant() ;//méthode d'instance
System.arraycopy(tbOrig, 0, tbCible, 0, tbOrig.length) ;
          // méthode de classe

int ix = 20 ;       // primitif scalaire
Integer myInteger ; // objet de type "Integer"
... ; // on initialise "myInteger"
String str = myInteger.toString() // meth. d'instance
String str2 = Integer.toString(ix)// meth. de classe
```

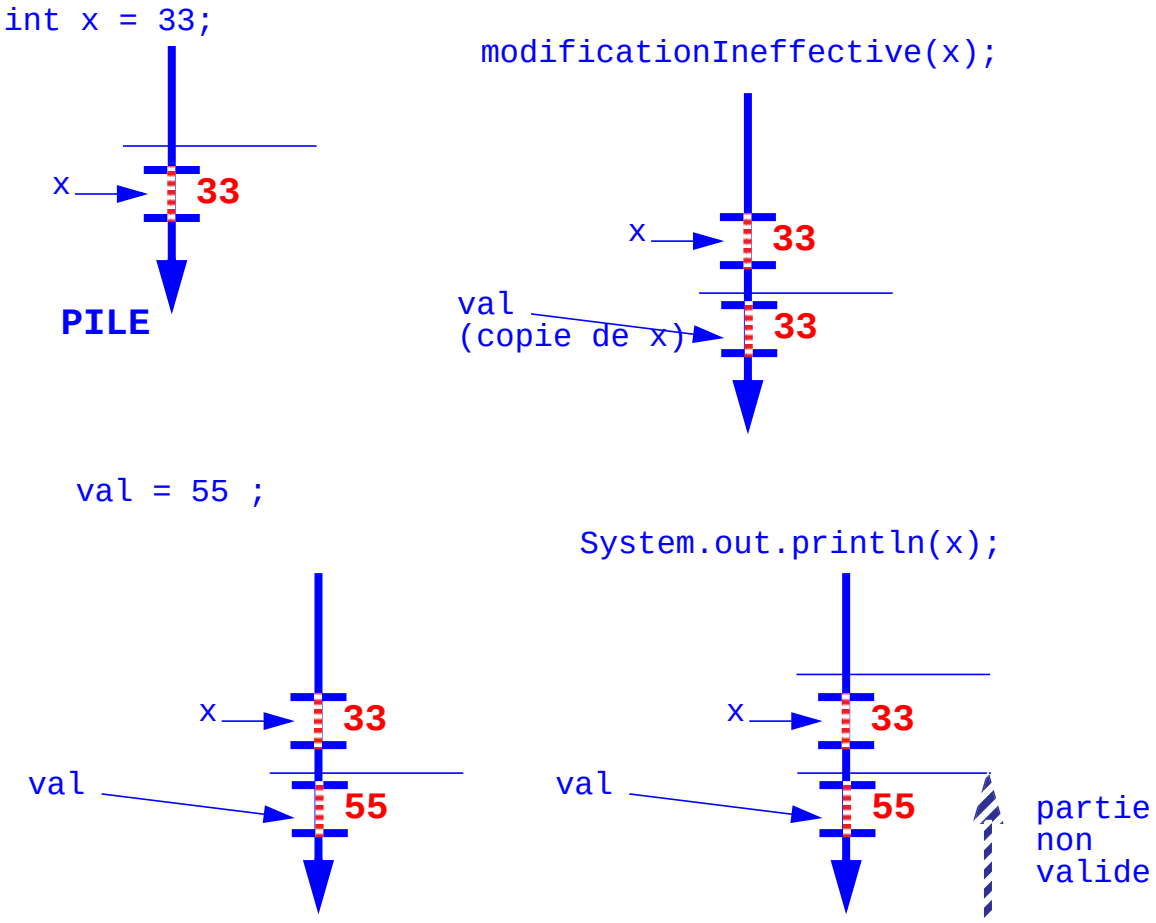
Passage de paramètres

En Java les paramètres des méthodes sont toujours passés **par valeur**.

```
public void modificationIneffective (int val) {
        val = 55;
}
```

"val" étant une copie d'une valeur dans le programme appelant cette méthode n'aura aucun effet sur la valeur d'origine.

```
int x = 33;
instanceCourante.modificationIneffective(x) ;
System.out.println(x) ; // donne 33!
```



Voir aussi: *Effets de bord*, page 136.

Point intermédiaire : les méthodes

Les méthodes de Java rendent des services fonctionnels: elles prennent éventuellement des paramètres en entrée et rendent un résultat (qui peut être de type `void`).

Les méthodes sont définies à l'intérieur du code des classes.

Les méthodes de classe rendent un service général qui n'est pas lié à l'état d'une instance particulière. Les méthodes d'instance rendent, en général, un service qui est lié à la combinaison des valeurs contenues dans l'instance.

L'appel d'une méthode est qualifié (notation "."): on demande un service soit à une classe, soit à une instance.

Mini-exercice:

- Reprendre le code de la classe "Confiserie" vu précédement. La doter d'une méthode d'instance de *signature* :

```
public double prixArrondi()
```

- Faire en sorte que cette méthode renvoie un double égal à la valeur entière la plus proche du "prix" de l'instance
  - c'est à dire 1.0 si le prix est de 0.6, 1.0 s'il est de 1.2, etc...- (voir méthode de classe `rint(double arg)` de la classe Math).

- Tester l'appel de cette méthode, en créant et initialisant une instance de "Confiserie" et en affichant le résulat de l'appel de "prixArrondi" sur cette instance.

Opérateurs

A quelques nuances près les opérateurs Java sont proches de ceux de C/C++.

- *Opérations arithmétiques* : **+**, **-** (y compris formes unaires), ***** (multiplication), **/** (division), **%** (reste de la division)

- *Opérations bit-à-bit sur entiers* : **&** (et) , **|** (ou), **~** (non), **^** (ou exclusif), **>>** (décalage à droite - x >> 4 décale de 4-), **<<** (décalage à gauche).

- *Opérations logiques, tests*: **&&** (et), **||** (ou), **!** (négation), **==** (égalité), **!=** (différence), **?:** (test `return ( x > y) ? x : y ;`)

- *Opérations logiques sur valeurs numériques* : **<**, **>** (inférieur, supérieur), **<=**, **>=** (inférieur ou égal, supérieur ou égal)

- *Incrémentation/décrémentation* : **++** (pré ou post incrémentation), **---** (pré ou post décrémentation)

```
nb = 0 ;
ix = nb++ ; // ix = 0 , nb = 1
iy = ++nb ; // iy = 2 , nb = 2
```

- *Affectation* : **=** (l'affectation rend un resultat : `iy = (iz = it);`) pour les opérations arithmétiques et bit à bit, opérations de "réaffectation" (`x+=2` correspond à `x = x + 2`) :

Java dispose également de quelques opérateurs spécifiques .

Il est vivement conseillé de parenthéser les expressions complexes plutôt que de s'appuyer sur l'ordre de préséance donné dans le tableau "priorité des Opérateurs", page 129.

Opérations arithmétiques

- Addition : `somme = num1 + num2 ;`
- Soustraction: `diff = num1 - num2 ;`
- Multiplication: `produit= num1 * num2 ;`
- Division: `quotient = num1/num2 ;`
- teste de la division: `reste = num1%num2 ;`

Les opérations sur les types numériques peuvent fournir des résultats erronés (troncation, perte de précision, etc. pour les entiers il n'y pas de contrôle de dépassement des limites -la somme de deux entiers positifs suffisament grands peut donner un entier négatif!-).

La division entre deux entiers fournit un entier (division entière)

L'opérateur reste de la division **%** opère également entre nombres flottants:

```
5%3 -> 2
5%(-3) -> 2
(-5)%3 -> -2
(-5)%(-3) -> -2
5.0 % 3.0 -> 2.0
(-5.25) % 3.0 -> -2.25
```

Concaténation

L'opérateur **+** assure la concaténation d'objets de type `String` en produisant un nouvel objet `String` :

```
String titre = ”Dr. ” ;
String nom = ”Alcofibras ” + ”Nasier” ;
String invite = titre + nom ;
```

Le résultat de la dernière ligne donnera

```
Dr. Alcofibras Nasier
```

A partir du moment où un des opérandes de l'opérateur + est un objet de type String l'autre argument est alors converti en String. Toutes les valeurs d'un type primitif scalaire peuvent être converties en String.

```
System.out.println( ”montant TVA = ” + (montant * tauxTVA)) ;
```

Si l'argument est un objet c'est la méthode `toString()` de l'instance qui est appelée (le résultat peut être surprenant et nous étudierons ultérieurement comment utiliser au mieux cette propriété).

Opérations logiques “bit-à-bit”

Le langage Java dispose d’opérateurs logiques “bit-à-bit” sur tous les types assimilables à des entiers. Ces opérateurs agissent au niveau de la représentation bit-à-bit de la valeur :

- opérateur ET bit-à-bit :
```
  00101101
& 01001111
  00001101
```

- opérateur OU bit-à-bit:
```
  00101101
| 01001111
  01101111
```

- opérateur OU EXCLUSIF (XOR) bit-à-bit:
```
  00101101
^ 01001111
  01100010
```

- opérateur NEGATION bit-à-bit:
```
~ 01001111
  1011000
```

- opérateur DECALAGE A GAUCHE:
  `45<<1` donne 45 x 2^1: 90
  `00101101 +` décalage de 1 `: 01011010`
  `45<<2` donne 45 x 2^2 : 180
  `00101101 +` décalage de 2 `: 10110100`
  (attention: cette valeur sur un *byte* est de -76 et non de 180!)

```
// couleur modele RGB dans un entier (voir java.awt.Color)
int couleur(int r, int g, int b, int a) {
       return ((a & 0xFF) << 24) | // transparence alpha
                 ((r & 0xFF) << 16) | // rouge
                 ((g & 0xFF) << 8)  | // vert
                 ((b & 0xFF) << 0);     // bleu
}
```

Voir aussi: *Opérations “bit-à-bit” : décalage à droite avec >> et >>>*, page 130.

Opérations logiques sur booléens

Les opérateurs logiques **&&** (ET), **||** (OU), **!** (Négation) ne sont utilisées que sur des variables de type `boolean` (ou sur des expressions rendant un `boolean`)

&&

	true	*false*
true	*true*	*false*
false	*false*	*false*

||

	true	*false*
true	*true*	*true*
false	*true*	*false*

!

true	*false*
false	*true*

Voir aussi: *Opérations logiques “bit-à-bit” sur booléens*, page 131.

Voir aussi: *Evaluation des opérations logiques*, page 131.

opérations de test

- comparaisons **>**, **>=**, **<**, **<=**

  Ces opérateurs ne concernent que des valeurs numériques (assimilables à des entiers ou flottantes). Remarquer la notation pour “inférieur ou égal” (**<=**) et pour “supérieur ou égal” (**>=**)

- comparaisons **==** , !=

  “égalité” (**==**) et “différence” (**!=**) peuvent s’appliquer à des types scalaires (numériques assimilables à des entiers -dont les caractères-, numériques flottants, booléens) mais aussi à des références d’instance. Dans ce dernier cas la signification est particulière puisqu’on indique simplement si les variables référencent ou non la même instance (ce problème sera exposé ultérieurement)

- opérateur de test: ***condition* ? *résultat*: *sinon***

  Il s’agit d’une expression fonctionnelle d’un choix. Exemple:

```
return res!=null ? res: resParDéfaut ;
```

  Le terme *condition* doit fournir un résultat booléen, les expressions *résultat* et *sinon* doivent avoir des types compatibles (nota: contrairement au C/C++ ces deux expressions résultat ne peuvent être de type `void`)

affectations optimisées

Pour comprendre la forme particulière de certaines affectations il faut revenir à la manière dont se compile une opération comme :

```
x = y + 2;
```

Les valeurs x et y sont successivement chargées dans un registre, mais si on fait :

```
x = x+ 2 ;
```

on recharge inutilement une deuxième fois “x”. Pour permettre une optimisation de très bas niveau on peut écrire une expression plus performante:

```
x += 2 ;
```

C’est pour cette raison que lorsqu’il y a re-affectation d’une valeur modifiée par un opérateur on peut utiliser une forme optimisée des opérateurs: `+= -= *= /= %= &= |= ^= <<= >>= >>>=`

```
masque &= 0xFF ;
```

Un cas encore plus poussé d’optimisation concerne les expresssions d’incrémentation / décrémentation:

```
compteur++ ; // pour compteur=compteur + 1 ;
```

Ces expressions existent sous deux formes: pre-incrémentation ou post-incrémentation (respectivement pre- ou post- decrémentation).

```
int compteur = 5 ;
int y = compteur-- ; // y vaut 5, compteur vaut 4
int z = --compteur ; // z et compteur valent 3
```

Cette facilité permet d’écrire de manière élégante (et parfois illisible!) des expressions complexes. On notera qu’à la base de ce comportement il y a le fait que l’affectation est en Java une opération fonctionnelle qui rend un résultat:

```
int compte = (compteur = 20) ; // est correct
instance.methode(compteur = 30) ; //correct(mais pas très lisible)
```

Conversions, promotions et forçages de type

Le fait d'assigner une valeur d'un type donné à une variable d'un autre type suppose une conversion. Dans certains cas, si les deux types sont compatibles, Java effectuera la conversion automatiquement : par exemple une valeur de type `int` peut toujours être affectée à une variable de type `long`. Dans ce dernier cas on a une *promotion* : le type `long` étant plus "grand" que le type `int` il n'y a pas de risque de perte d'information.

Inversement s'il y a un risque de perte d'information le compilateur exige que vous specifiiez explicitement votre demande de conversion à l'aide d'une opération de transtypage:

```
long patouf = 99L ;
int puce = (int) patouf ;
```

Dans le cas d'expressions complexes il est vivement conseillé de mettre l'ensemble de l'expression à convertir entre parenthèses pour éviter des problèmes dus à la précédence de l'opérateur de transtypage.
Le type `char` demande également des précautions : bien qu'il soit assimilable à un type numérique sa plage de valeurs va de 0 à 2^{16} -1 alors que la plage des valeurs de `short` va de -2^{15} à 2^{15} -1 une conversion explicite est donc toujours nécessaire.

Les conversions de références seront étudiées ultérieurement, elles suivent également des règles de promotion (mais dans ce cas il faudra définir ce que "promotion" veut dire). Par ailleurs un tel transtypage n'a jamais pour effet de transformer effectivement une instance.

Conversions, promotions et forçages de type

Les règles de conversion et de promotion s'appliquent également aux paramètres des méthodes :

```
class Gibi {
          void traiter(int param) { ....
          }
          ...
}

Gibi instance = new Gibi() ;
...
short petit ; .....
instance.traiter(petit) ; // conversion implicite
long gros ;
instance.traiter((int) gros) ; //conversion explicite
```

Il faut également prendre en compte d'autres aspects :

- Les méthodes Java peuvent être *surchargées* c'est à dire que la même méthode peut accepter plusieurs définitions. Ces définitions diffèrent par le type des paramètres qui, de plus, ne sont pas forcément incompatibles entre eux. Ainsi la classe `java.io.PrintStream` (voir `System.out`) connait les méthodes :

  ```
  void println(String x)
  void println(char x)
  void println(int x)
  void println(long x) ....
   // quelle est la méthode réellement appelée
   // par: println(petit) ?
  ```

- Voir aussi: *Types des calculs*, page 132.

Point intermédiaire : les opérateurs

Mini-exercice :

- Cet exemple est un calcul d'expression dans un bloc de code (en l'occurence un `main`) il n'utilise pas d'instance d'objet.
  Dans un `main` associé à la classe "TestExpr", écrire une expression qui calcule un nombre résultat (de type `double`) obtenu à partir d'une variable "montant" de type `double` (initialiser à 22.5) et d'une autre variable "taux" de type `int` (initialiser à 42)
  la formule de calcul est la suivante :

$$resultat = montant \div (1 + ((50 - taux) \div 1000))$$

  Raisonnablement le calcul doit rendre quelque chose proche de 22.32

- Exercice complémentaire (pour les très rapides ou ceux qui révisent): arrondir le calcul de façon à ce que le résultat n'ait qu'au maximum deux chiffres après la virgule -voir par ex. l'utilisation de `Math.rint()` -

Structures de contrôle: branchements

Instructions conditionnelles: if-else

La syntaxe de base pour les branchements conditionnels est :

```
if (expression_booléenne)
        instruction ou bloc
 else
        instruction ou bloc
```

Exemple :

```
double salaire = employé.getSalaire();
if (salaire < PLAFOND ) {
        salaire *= tauxAugmentation ;
} else {
        salaire += petitePrime ;
}
        // préférer les blocs dans tous les cas
```

Le `if` de Java diffère de celui du C/C++ car il utilise une expression `booléenne` et non une valeur numérique.

```
if (x) // x est un int
```

est illégal. Par contre on peut écrire :

```
if (x != 0) { //
```

La partie `else` est optionnelle et peut être omise s'il n'y a pas d'action à effectuer lorsque la condition est fausse.

Structures de contrôle: branchements

Instructions de branchement multiple : switch

La syntaxe de base pour l'aiguillage multiple est :

```
switch (expression_choix) {
          case constante1 : //entier non long ou char
                    instructions ;
                    break ; //optionnel
          case constante2 :
                    instructions ;
                    break ; //optionnel
          ....
          default :
                    instructions ;
                    break ; // optionnel
}
```

Le résultat de *expression_choix* doit pouvoir être affecté sans restriction à un entier `int`. Les types `byte`, `short` ou `char` sont possibles (ainsi que `int`!), par contre les types flottants, `long` ou les références d'objets ne sont pas possibles.

```
switch (codecouleur) {
          case 0: // permet d'avoir fond bleu et tracés rouges
                    fenêtre.setBackground(Color.blue) ;
          case 1: // tracés rouges
                    fenêtre.setForeground(Color.red);
                    break;
          case 2: // tracés verts seulement
                    fenêtre.setForeground(Color.green);
                    break;
          default: // quoiqu'il arrive
                    fenêtre.setForeground(Color.black);
}
```

L'étiquette `default` est optionnelle mais permet d'aiguiller sur un choix par défaut. L'omission de `break` permet des définitions complexes.

Structures de contrôle: boucles

Boucle avec test haut : while

```
while(test_booléen)
          instruction ou bloc
```

Bien entendu pour rentrer dans la boucle le test doit être vrai, par ailleurs la valeur testée doit évoluer dans le corps de la boucle.

```
while (maPosition.estDansPremierQuadrant()) {
          maPosition.posX -= 10 ;
          maPosition.posY -= 10 ;
}
```

Boucle avec test bas : do ... while

```
do
          instruction ou bloc
while(test_booléen) ;
```

L'instruction ou le bloc après le do est toujours executé au moins une fois.

```
do {
          maposition.posX += 10 ;
          maposition.posY += 10 ;
} while (! maPosition.estDansPremierQuadrant());
```

Structures de contrôle: boucles

Itération avec boucle for

```
for (initialisations ; test_haut ; operations_itération)
          instruction ou bloc
```

Exemple :

```
for (int incr= 1 ;
          !maPosition.estDansPremierQuadrant() ;
                    incr *= 2) {
          maPosition.posX += incr ;
          maPosition.posY += incr ;
}
```

expression courante : parcours d'un tableau

```
for(int ix = 0 ; ix <tableau.length; ix++) {
          System.out.println(tableau[ix]);
}
```

* Chacun des 3 arguments du `for` peut être vide (un test vide s'évalue à true).
* Dans l'initialisation on peut déclarer des variables dont la portée est celle de la boucle
* On peut utiliser le séparateur "," dans l'initialisation ou dans les opérations d'itération qui sont exécutées en bas de boucle.

```
int[] tb = new int[6] ;
// et que fait cette boucle?
for (int ix = 0, iy = tb.length ; ix < tb.length ; ix++, iy--)  {
      tb[ix] = iy ;
}
```

Structures de contrôle: débranchements

Déroutements dans une boucle

Le déroulement normal d'une boucle peut être dérouté par une des directives :

- **`break`** : sortie prématurée d'une boucle

```
for (int ix = 0; ix <tb.length; ix ++) {
          int lu = System.in.read() ; //
          if ( (-1 == lu) || ('\n' == lu)
                    || ('\r' == lu)) {break;}
          tb[ix] = lu ;
}
/* noter ici la promotion automatique en entier
des opérandes du "==" "'\n' == lu"
*/
```

- **`continue`** : renvoi direct en fin de boucle (test pour les boucles `do..while` et `while`, opérations d'itération pour boucle `for`).

```
for (int ix = 0; ix <tb.length; ix ++) {
          int lu = System.in.read() ; //
          if ( (-1 == lu) || ('\n' == lu)
                    || ('\r' == lu)) {
                    tb[ix] = '|' ;
                    continue;
          }
          tb[ix] = lu ;
}
```

Il existe d'autres possibilités de sortie d'une boucle et, en particulier, le `return` d'une méthode.

Voir aussi: *Structures de contrôle: Déroutements étiquetés*, page 133.

Point intermédiaire : les structures de contrôle

Mini-exercice :

- Reprendre la classe “TestTableaux” et tracer tous les éléments de chaque tableau.

Compléments

Les compléments techniques suivants constituent une annexe de référence

priorité des Opérateurs

La table suivante liste les opérateurs de Java par ordre de préséance . Les indications “G a D” et “D a G” donne le sens de l’associativité (Gauche à Droite et Droite à Gauche)

Séparateur	`.  []  ()  ;  ,`

D a G	`++  --  +` unaire `-` unaire `~  !` `(`transtypage`)`
G a D	`*  /  %`
G a D	`+  -`
G a D	`<<  >>  >>>`
G a D	`<  >  <=  >= instanceof`
G a D	`==  !=`
G a D	`&`
G a D	`^`
G a D	`|`
G a D	`&&`
G a D	`||`
D a G	`?:`
D a G	`=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=`

Il est vivement conseillé de parenthéser les expressions complexes plutôt que de s’appuyer sur l’ordre de préséance donné dans ce tableau.

Opérations “bit-à-bit” : décalage à droite avec >> et >>>

Le langage Java dispose de deux opérateurs de décalage à droite.

L’opérateur `>>` assure un décalage à droite *arithmétique* avec conservation du signe. Le résultat de ce décalage est que le premier opérande est divisé par deux à la puissance indiquée par l’opérande de droite. Ainsi :

```
128 >> 1 donne 128/2¹ = 64
256 >> 4 donne 256/2⁴ = 16
-256 >> 4 donne -256/2⁴ = -16
```

En fait les bits sont décalés à droite et le bit de signe est conservé.

Le décalage à droite logique `>>>` (non signé) travaille purement sur la représentation des bits, des zéros sont automatiquement insérés sur les bits les plus significatifs.

```
1010 .... >> 2 donne 111010 ...
1010 ... >>> 2 donne 001010 ...
```

Les opérandes droits sont évalués modulo 32 pour les opérandes gauches de type `int` et modulo 64 pour ceux de type `long`. Ainsi :
`int x; ... x >>>32 ;` donnera une valeur inchangée pour x (et non 0 comme on pourrait s’y attendre).
L’opérateur >>> n’est effectif que sur des types `int` et `long`.

Opérations logiques “bit-à-bit” sur booléens

En Java le type `boolean` n’est pas assimilable à un type entier.

Pourtant certaines opérations logiques bit à bit s’appliquent entre booléens et retournent un booléen .

- & correspond au ET logique (AND)
- | correspond au OU logique (OR)
- ^ correspond au OU EXCLUSIF logique (XOR)

Toutefois l’évaluation des formes & et | diffère des opérations && et ||

Evaluation des opérations logiques

Les opérateurs && et || assurent des expressions logiques avec “court-circuit” d’évaluation. Considérons cet exemple :

```
UneDate unJour ;
... ; // déroulement du programme
if( (unJour != null) && (unJour.numeroDansMois > 20)) {
          // on fait la paye!
}
```

L’expression logique testée par le “if” est légale et, surtout, protégée contre les incidents. En effet si la première partie de l’expression (`unJour != null`) est fausse, la seconde partie (`unJour.numeroDansMois ..`) n’est pas évaluée -ce qui est une bonne chose étant donné que l’expression déclencherait alors une erreur d’évaluation de la JVM (recherche de contenu à partir d’une référence nulle)-.

De la même manière l’évaluation complète d’une expression || (OU) est abondonnée dès qu’un membre est vrai.

Types des calculs

Les opérations arithmétiques entières se font automatiquement au minimum avec un `int` et donc les opérandes sont automatiquement promus :

```
short a, b;
....// initialisations de a et b
short res1 = a + b ; // ERREUR COMPILATION
short res2 = (short) (a+b) ;
          // OK mais risque perte précision
```

Dans certains cas (utilisation de constantes littérales) le compilateur est capable toutefois de déduire le type effectif du résultat rendu par le calcul.

En fait on a ici un cas particulier de la règle générale de la promotion des opérateurs numériques :

- si un des opérateurs est un `double` alors l'autre est converti en `double`
- sinon si un des opérateurs est un `float`, l'autre est converti en `float`
- sinon si un des opérateurs est un `long`, l'autre est converti en `long`
- sinon tous les opérandes sont convertis en `int`

Structures de contrôle: Déroutements étiquetés

A l'intérieur d'un bloc de programme il est possible d'étiqueter une boucle (par un *label*) pour servir de cible à une directive `break` ou `continue` enfouie dans une des sous-structures -ceci permet d'éviter l'emploi d'une directive *goto* qui n'existe pas en Java (bien que ce soit un mot réservé)-

```
bye : while (true) {
        for (int ix = 0; ix <tb.length; ix ++) {
                int lu = System.in.read() ; //
                switch (lu){
                 case -1 :
                          break bye ; //saut en dehors du while
                 case '\n' : case '\r' :
                          tb[ix] = '|';
                          continue ;
                 case '\t' :
                          tb[ix] = ' ';
                          break ;
                 default :
                          tb[ix] = lu
                }
                /* nota: toute etiquette dans un switch doit
                 * être "affectable" au type du sélecteur
                 */
                ...; // du code
        }
        ....; // encore du code !
}
```

Portée des variables, cycle de vie

Nous avons vu des définitions de variables en trois endroits : comme membre d'une classe, comme paramètre d'une méthode ou à l'intérieur d'un bloc d'exécution (dans le cadre d'une définition de méthode par exemple). Dans ces deux derniers cas on dit que la variable est *locale* (ou *automatique).*

variables locales

Une variable automatique est allouée sur la pile; elle est créée quand l'exécution entre dans le bloc et sa valeur est abandonée quand on sort du bloc. La portée de la variable (sa "visibilité") est celle du bloc -et des sous-blocs-

- Dans un bloc on ne peut pas définir une variable locale dont le nom rentre en conflit avec une autre variable locale d'un bloc englobant.

```
int ix = 0 ; int iz = 0 ;
....
for (int ix = 0;ix<tb.length;ix++) { // ERREUR sur ix
          float iz ; // ERREUR conflit avec iz
}
```

- Une variable locale **doit** être explicitement initialisée avant d'être utilisée. Cette situation est détectée par le compilateur qui provoque alors une erreur.

```
public void petitCalcul() {
          int ix = (int)(Math.random() * 100);
          int iy,iz;
          if (ix > 50) {
                    iy = 9;
          }
          iz = iy + ix;// ERREUR
                    // iy peut être non-initialisée
          ...
```

variables d'instance, variables de classe

Il est possible de définir des variables en dehors des blocs d'exécution (donc au plus "haut niveau" dans la définition de classe. Ces variables peuvent être :

- Des variables membres d'une instance. Ces variables sont créées et intialisées au moment de la création de l'instance par l'appel de new ClasseXyz(). Ces variables existent tant que l'objet est referencé et n'est pas recupéré par le glaneur de mémoire.

```
public class Position {
        int posX ; //initialisation implicite
        int posY = -1; // initialisation explicite
        ....
```

(rappel) utilisation d'une variable d'instance :

```
Position maPosition = new Position() ;
System.out.println(maPosition.posX); // donne 0 !
System.out.println(maPosition.posY); // donne -1
```

- Des variables partagées (variables de classe) définies en utilisant le mot-clef `static`. Ces variables sont créées et initialisées au moment du chargement de la classe par le `ClassLoader`. Ces variables existent tant que la classe existe (Ces variables seront expliquées dans un chapitre ultérieur).

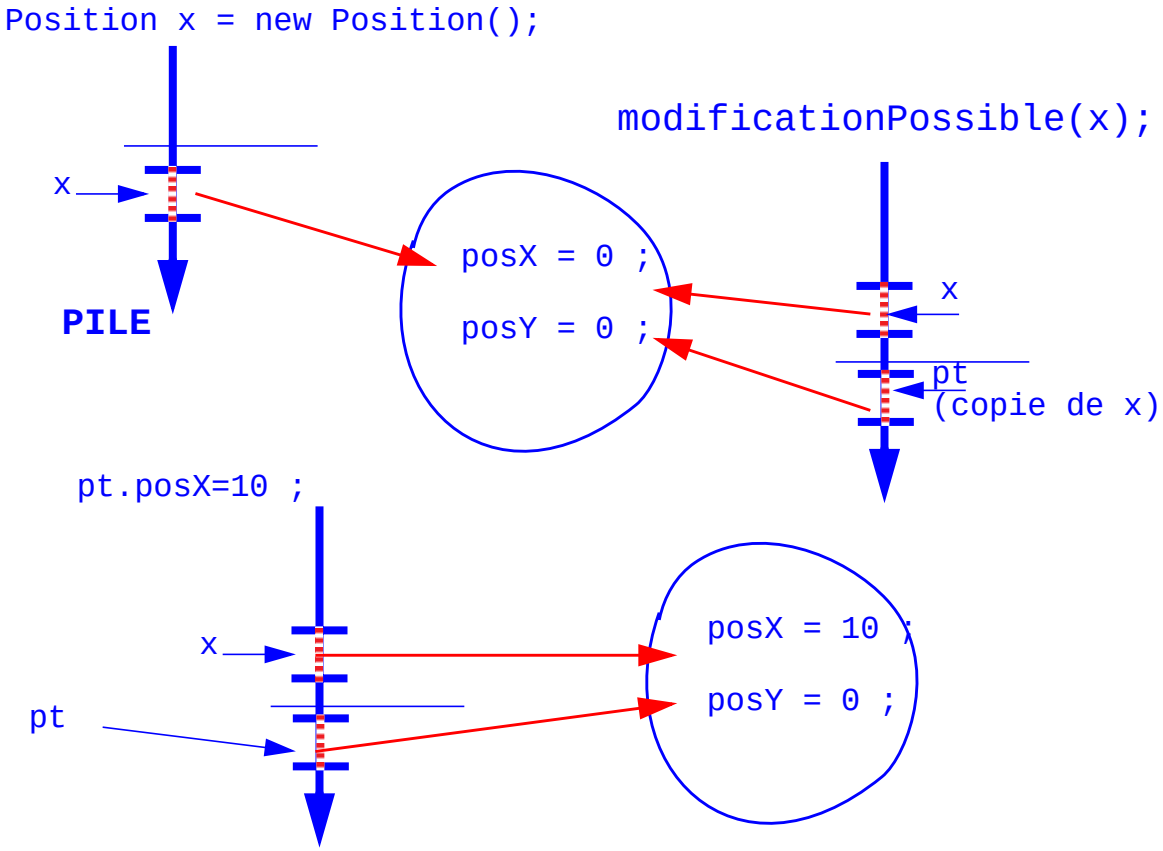
Initialisations implicites des variables membres

byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000' (NULL)
boolean	false
toutes références	null

Effets de bord

Le passage de paramètre par valeur d'un objet revient à une copie de sa référence

```
public void modificationSansEffet (Position pt) {
        pt = new Position() ;// ne modifiera pas la ref originale
}

public void modificationPossible (Position pt) {
        pt.posX = 10 ; // modifie le contenu de la ref.
}
```



Dans les deux cas le paramètre est une copie d'une référence, la modification de la référence n'aura aucun effet sur le programme appelant, par contre une modification du contenu de la référence est possible.

Un conseil: éviter de spécifier des méthodes qui ont ainsi un *effet de bord* (un des paramètres sert à rendre un résultat)

Corrigés des mini-exercices

Confiserie.java (avec arrondi):

```
public class Confiserie {
   String nom ;
   double prix ;

   double prixArrondi() {
      return Math.rint(this.prix) ;
   }

   public static void main(String[] args) {
      Confiserie monNougat = new Confiserie() ;
      monNougat.nom= "montelimar" ;
      monNougat.prix = 0.6 ;
      System.out.println(monNougat.prixArrondi()) ;
      monNougat.prix = 1.2 ;
      System.out.println(monNougat.prixArrondi()) ;
   }
}
```

TestExpr.java :

```
public class TestExpr {
   public static void main (String[] tb) {
      double montant = 22.5;
      int taux = 42 ;
         // piège : savoir calculer avec 1000.0 et non 1000
      double resultat = montant / (1 + ((50 -taux)/1000.0)) ;
      System.out.println(resultat) ;
         // calcul arrondi
      System.out.println( Math.rint(resultat * 100) /100) ;
   }
}
```

boucles dans TestTableaux2.java :

```
public class TestTableaux2 {
   public static void main (String[] tb) {
      int[]  tbInt = {-1, 0, 0xFFFFFFFF } ;
      Confiserie[] tbObj= {
         new Confiserie(),
         new Confiserie(),
         new Confiserie()
      } ;
      for (int ix = 0 ; ix < tbInt.length; ix++) {
         System.out.println(tbInt[ix]) ;
      }
      for (int ix = 0 ; ix < tbObj.length; ix++) {
         System.out.println(tbObj[ix]) ;
      }
   }
}
```

Exercices :

Exercice * (utilisation de la documentation) :

- Rechercher dans la documentation de la classe `java.lang.Integer` une méthode qui tente d'analyser une chaîne pour la transformer en entier. C'est à dire une méthode telle que `f("3")` rende `3`.
  Cette méthode est une méthode de classe qui prend un argument de type String et rend un résultat de type entier primitif.

Exercice * (suite):

- Lorsque l'on utilise la méthode de classe `main`, le paramètre est un tableau de `String`. Ce tableau représente les arguments de lancement du programme.
  Ecrire une classe `Add` qui additionne les arguments qui lui sont passés en paramètres :

```
java Add 3 -3 9
9
java Add 10 11 12 13
46
```

Objets et classes 5

Points essentiels

Introduction à la programmation Objet en Java:

- Classes et instances
- Méthodes d'instance
- Encapsulation
- Constructeurs
- Référence `this`

Classes et objets

Comme nous l'avons vu une *classe* peut permettre de définir un modèle de regroupement d'un ensemble de données.

A partir de ce modèle on pourra créer des *objets* qui associeront des valeurs aux membres de la classe.

```
public class UneDate {
          public int jour, mois, année ;
          ...
}
```

La combinaison des *états* à l'intérieur d'une instance est initialisée après la création de l'objet .

```
UneDate ceJour = new UneDate() ;
ceJour.jour = 1;
ceJour.mois = 1;
ceJour.année = 2000 ;
```

Cette combinaison d'états est susceptible d'évoluer pendant la durée de vie de l'objet : c'est le code associé à la classe qui définira la manière dont ces évolutions sont possibles.

Méthodes d'instance

Dans de nombreux langages de programmation lorsqu'on définit un type enregistrement (un *agrégat*), on définit des fonctions qui opèrent sur ce type, mais il n'y a pas d'association particulière entre le code de définition de ces fonctions et celui de définition du type.

Le langage Java permet une association plus étroite, ces "fonctions" sont décrites dans le corps de la classe. De plus les *méthodes d'instance* sont décrites de manière à opérer sur l'instance sur laquelle on les invoque.

```
public class UneDate {
          public int jour, mois, année ;
          ...// autres champs

          public boolean estFérié() {
                    ...//code calcul utilisant
                     // this.jour, this.mois, this.année
          }//End estFérié()

}
```

La manière d'exprimer les instructions du langage est fortement marquée par ce point de vue : on demande un service à une instance, on lui "envoie un message".

```
UneDate ceJour = new UneDate() ;
...//initialisations diverses
if (ceJour.estFérié()) {
          ....
}
```

Méthodes d'instance

Les méthodes d'instance peuvent rendre une grande variété de service comme renseigner sur des propriétés de l'instance, modifier l'instance ou même créer un autre objet en fonction de l'état de l'instance courante.

```
public class UneDate {
          public int jour, mois, année ;
          ...// autres champs
          public boolean estFérié() {...}
          ...// autres méthodes

          public UneDate leLendemain() {
                    UneDate res = new UneDate() ;
                    ....// calculs, affectations champs
                    return res ;
          }//End leLendemain()
}
```

Utilisation:

```
UneDate ceJour = new UneDate() ;
...// initialisations
UneDate lendemain = ceJour.leLendemain() ;
```

Méthodes d'instance (suite)

Une méthode d'instance peut servir à modifier de manière cohérente et coordonnée les états internes de l'instance.

```
public class UneDate {
          ....
          public void passeAuLendemain() {
                    // modifie le(s) champ(s) jour
                    // et éventuellement mois, an
          }
}
```

Utilisation:

```
UneDate jourCourant .....
.....
jourCourant.passeAuLendemain() ;
          //"jourCourant" est modifié
          //l'instance ne contient plus les mêmes valeurs
```

Une telle méthode est intéressante dans la mesure où elle concentre la logique d'une modification de l'instance, mais encore faut-il faire en sorte qu'une modification incohérente de l'instance ne soit pas possible.

Par ailleurs on peut aussi décider que les objets d'une classe donnée sont *immuables*, c'est à dire qu'on ne peut pas modifier leur état interne. C'est une décision de conception qui sera controlée par toutes les méthodes d'instance. Ainsi les objets de type `String` sont immuables (pour avoir des chaînes de caractères modifiables on utilisera les objets de type `StringBuffer`).

Point intermédiaire : objets et méthodes d'instance

Par l'appel de méthode d'instance on "envoie un message" à l'instance : on lui demande un service . Ce service est , en général, lié à la combinaison des valeurs contenues dans l'instance et gérées par elle.

Mini-exercice :

- Modifier la classe "Confiserie" et la doter d'une méthode d'instance
  `double prixClient(int nb)`
  qui calcule le prix de vente de "nb" confiseries en fonction:
  - du "prix" associé à l'instance
  - de l'application d'une taxe sur la prévention de la carie dentaire qui est de 5% du montant

Encapsulation

Il est possible de restreindre l'accès aux variables membres d'une instance en utilisant un modificateur d'accès comme `private`.

```
public class UneDate {
          private int jour;
          private int mois;
          private int année ;

          public int getJour() {
                    return jour ;
          }
          public int getMois() { ...}
          ....
          public String toString() {
           ... // date sous forme texte
          }
          public void passeAuLendemain() {
                    // sait modifier les champs privés
          }
}
```

L'utilisation du modificateur `private` dans la déclaration des champs jour, mois, année rend impossible l'accès à ces valeurs par d'autres codes que ceux contenus dans la définition de classe. De cette manière on force un accès indirect qui oblige à passer par des méthodes qui définissent les conditions précises de ces accès et maintiennent la cohérence de l'instance.

Ainsi dans la classe "UneDate" on va pouvoir rendre impossible une modification sans contrôles du champ "jour" (on risquerait d'avoir un 30 février) et on va centraliser la logique (complexe) qui régit l'évolution d'une date.

Le fait de cacher les états internes de l'objet est qualifié d'*encapsulation*. L'encapsulation permet de découpler nettement :

- d'une part la "vision" qu'une classe offre à l'extérieur au travers de son API.
- d'autre part la réalisation concrète des services offerts par la classe.

Encapsulation

Le fait de forcer l'utilisateur d'une classe à passer par une vision abstraite d'une classe permet de découpler l'utilisation et la réalisation et permet même d'anticiper sur des évolutions possibles.

```
public class Compte {
        ... //champs divers
        private double solde ;
        public double getSolde () {
                return this.solde ;
        }
        public void setSolde(double valeur) {
                // operations diverses
                this.solde = valeur ;
        }
```

Si dans la classe "Compte" on avait défini le champ "solde" comme librement accessible on aurait laissé développer des applications qui affectent librement cette valeur. Si, par la suite, les règles de gestion évoluent et il apparaît qu'à toute modification du solde on doit réaliser une opération complémentaire, on est obligé de rechercher dans tous les codes les utilisations de ce champ pour pouvoir rajouter ces nouvelles instructions. Si, par contre, on a centralisé l'accès à "solde" dans une méthode de modification "setSolde", l'ajout de ce nouveau code est centralisé et immédiat.

On notera que la convention de nommage couplée pour les accesseurs / mutateurs -"typeXXX **get**XXX()" et "void **set**XXX(typeXXX)"- est standard en Java.

Initialisations : constructeur

Un autre aspect de l'encapsulation est de prévoir un mécanisme pour remplacer et contrôler les initialisations explicites comme:

```
UneDate ceJour = new UneDate();
ceJour.jour = 1;
ceJour.mois = 1;
ceJour.année = 2000 ;
```

Dans ce cas on centralise la création / initialisation de l'instance dans un ou plusieurs codes de *constructeur.*

```
public class UneDate {
          private int jour, mois, année ;
          .... // accesseurs
          .....
          public UneDate(int leJour,int leMois,int lAn){
                    this.jour = leJour ;
                    this.mois = leMois ;
                    this.année = lAn ;
                    // au fait :
                    // les paramètres sont-ils valides?
          }
}
```

Utilisation:

```
UneDate ceJour = new UneDate(1,1,2000) ;
```

Un constructeur doit avoir exactement le même nom que sa classe.
Attention: ce n'est pas une méthode (pas de résultat par exemple) et, de plus, on ne le considère pas comme un *membre* de la classe (comme le sont les champs et les méthodes)

Initialisations : constructeur

Dans l'exemple de la classe "UneDate" on pourrait aussi définir un constructeur qui initialise automatiquement l'instance avec la date du jour:

```
import java.util.GregorianCalendar ;// base des "dates" en Java

public class UneDate {
        private int jour, mois, année;
        // accesseurs
        public int getJour() {return jour;}
        ...
        public UneDate(int leJour,int leMois,int lAn){
                ...
        }

        public UneDate() {
                GregorianCalendar cl = new GregorianCalendar();
                // contructeur met à la date du jour
                this.jour=
                 cl.get(GregorianCalendar.DAY_OF_MONTH) ;
                .... // autres initialisations
        }
}
```

Exemple:

```
System.out.println("jour=" + new UneDate().getJour()) ;
```

Il est intéressant de remarquer que la classe "UneDate" pourrait ainsi avoir plusieurs constructeurs avec des "signatures" différentes ::

```
UneDate dateDuJour = new UneDate() ;
UneDate dateCritique = new UneDate(1,1,2000) ;
```

On a ainsi une *surcharge* des constructeurs analogue à la surcharge possible pour des méthodes (voir, par ex. les différentes versions de "`println`" dans la classe `java.io.PrintStream`).

Voir aussi: *Le constructeur par défaut*, page 156.

Voir aussi: *Opérations d'initialisations*, page 156.

Instances : la référence this

Normalement un membre doit être rattaché à une instance à l'aide d'une notation "." (`ceJour.toString()`, `ceCompte.clef`). Toutefois dans une définition de classe il est possible de désigner un membre de la classe courante sans faire appel à cette notation: la reférence à l'instance courante **`this`** peut être implicite dans de nombreux cas.

L'usage de "this" s'impose dans les cas suivants:

- Pour lever une ambiguité de nommage,

```
public class UneDate {
        private int jour, mois, année ;
        public UneDate(int jour,int mois,int année){
                this.jour = jour ;
                this.mois = mois ;
                this.année = année ;
        ....
```

- Pour passer une référence à l'instance courante dans un appel de méthode,

```
        public void trace() {
                System.out.println(this) ;
        }
```

- Pour invoquer un autre constructeur de la classe dans la défintion d'un constructeur. Dans ce cas cette invocation doit impérativement être la première instruction du code du constructeur.

```
public class Compte {
        ...
        public Compte(Client client,float dépôt,float découvert){
                ....
        }
        public Compte(Client client, float dépôt) {
                this(client,dépôt,DECOUVERT_STANDARD);
        }
        public Compte(Client client) {
                this(client, 0F) ;
        }
```

Point intermédiaire : encapsulation, constructeurs

L'encapsulation des champs d'une instance permet de n'exposer que les services et rend ainsi ces services plus indépendants de la réalisation et de ses évolutions.

Les constructeurs permettent de réaliser une opération synthétique d'allocation-initialisation et de définir les façons d'opérer les initialisations "légales" de l'instance.

Mini-exercice :

- Reprendre la classe Confiserie, la modifier et , en particulier, la doter d'un constructeur

Récapitulation: architecture d'une déclaration de classe

Bien qu'encore incomplète voici la description du schéma général d'un fichier source java.

```
....// déclaration de package (sera vue ultérieurement)
....// liste des directives "import"
....// comme : import java.util.BigDecimal ;

// déclarations de classe (une seule peut être "public")

... class XXX {
        // l'ordre des déclarations est sans importance
        // (sauf pour l'ordre relatif des initialisations)

        // MEMBRES d'INSTANCE
        champs (avec initialisations éventuelles)
        méthodes
        ...
        // MEMBRES de CLASSE (static)

        méthodes
         // public static void main(String[] x): cas particulier
        ...
        // CONSTRUCTEURS
        constructeur(s)
}
```

Compléments

Les compléments techniques suivants constituent une annexe de référence

Le constructeur par défaut

Chaque classe est dotée d'un constructeur par défaut: il s'agit du constructeur sans paramètres que l'on invoque par `new Xxxx()`. Toutefois il faut prendre garde au fait qu'à partir du moment où on définit un constructeur avec des paramètres pour une classe qui n'avait pas de constructeur explicite ce constructeur par défaut n'existe plus (le compilateur refusera son emploi). Il est bien sûr alors possible de définir explicitement un constructeur sans paramètres.

Opérations d'initialisations

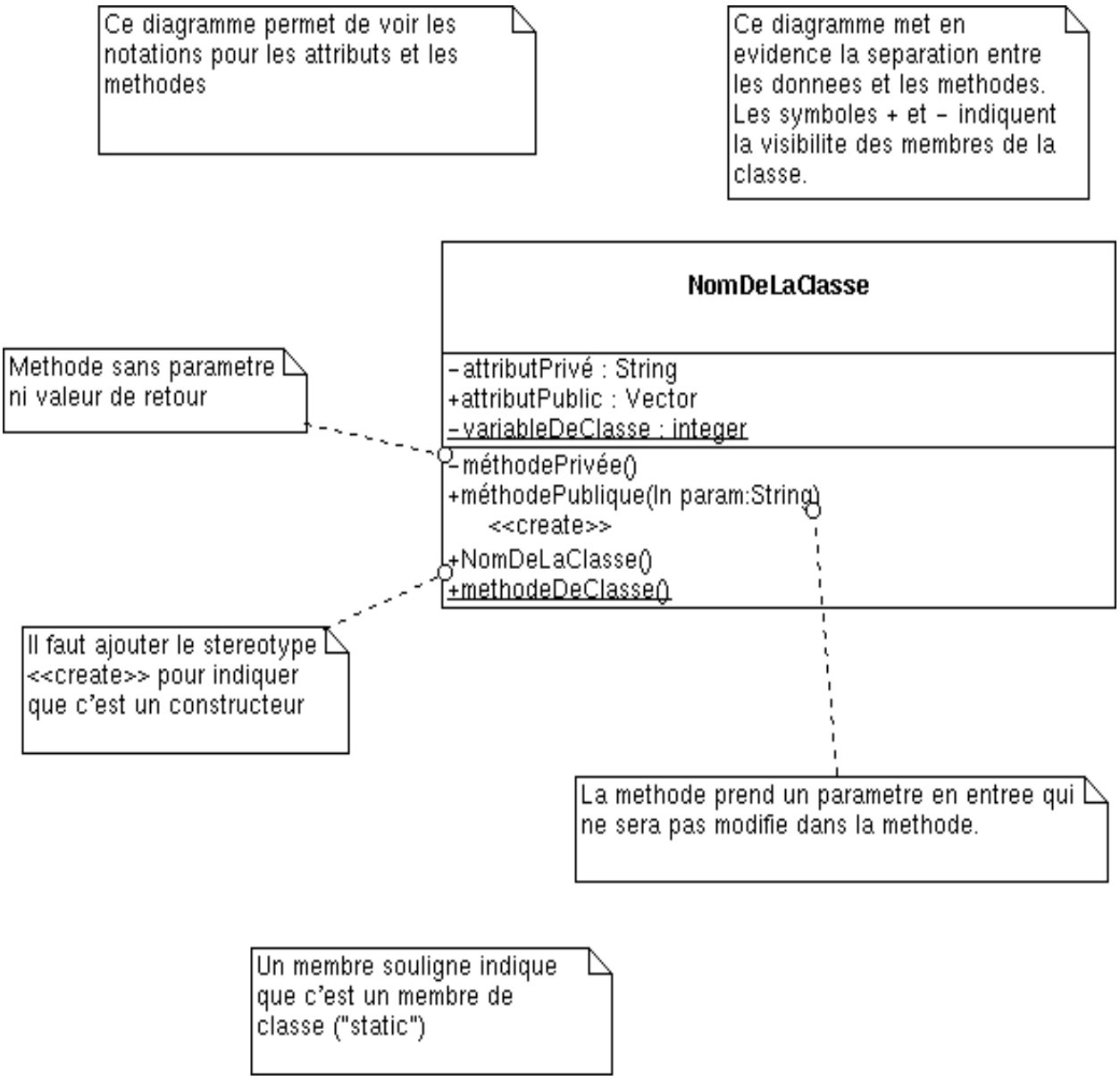
Il est possible de faire en sorte que certaines variables membres soient initialisées au moment de la création de l'instance :

```
import java.math.BigDecimal ;
import java.util.ArrayList ;

public class Compte {
        private BigDecimal solde = new BigDecimal(.0) ;
        private long heureCréation = System.currentTimeMillis() ;
        private ArrayList opérations = new ArrayList() ;
        ...
}
```

Lors de la création d'une instance la machine virtuelle alloue l'espace nécessaire, affecte les valeurs par défaut aux variables membres et invoque le constructeur. Quand on "rentre" dans le code du constructeur:

- Les initialisations explicites sont exécutées (dans l'ordre spécifié dans le code de la classe).
- Le reste du code du constructeur est exécuté.

Utilisation notation UML



Corrigés des mini-exercices

Confiserie.java avec méthode d'instance :

```
public class Confiserie {
    String nom ;
    double prix ;

   public double prixClient(int nb) {
      // 1.05 à  mettre ensuite en Constante
      return nb * this.prix * 1.05 ;
   }
}
```

Confiserie.java (encapsulation, constructeurs, etc.):

```
public class Confiserie {
    private String nom ;
    private double prix ;

   public Confiserie( String nom, double prix) {
      this.nom = nom ;
      this.prix = prix ;
   }

   public String getNom() { return this.nom;}
   public double getPrix() { return this.prix;}

   public double prixClient(int nb) {
      return nb * this.prix * 1.05 ;
   }
}
```

Exercices :

Exercice ** :

Nous allons commencer une réalisation simple du projet “`oubangui.com`”.

Ecrire et tester une première version des classes `Livre`, `Panier` (on attendra le prochain exercice pour définir la classe `Client`).

- classe Panier : pour simplifier mettre les livres achetés dans un tableau de 20. Pour l’instant la méthode permettant de mettre un `Livre` dans le Panier ne fait pas de contrôle sur le nombre maximum de livres.

- doter toutes les classes d’une méthode `String toString()` qui permettra de donner une chaîne descriptive de l’instance (qui pourra être utilisée pour tracer). En effet lorsqu’on appelle `System.out.println` en mettant une instance quelconque en argument, `println` appelle implicitement la méthode `toString()` sur cette instance.

- écrire un petit test qui crée unPanier, lui attribue deux ou trois livres et affiche ensuite le contenu de cette instance.

Composition et association d'objets, héritage 6

Points essentiels

Principes fondamentaux de la programmation Objet en Java (suite):

- Définition d'objets à partir d'autres objets : composition, association, délégation.
- Héritage
- Spécialisation de méthodes
- Polymorphisme
- Invocation du constructeur de la classe mère.

Imbrications d'instances : composition, association

Comme nous l'avons vu une *classe* permet le regroupement d'un ensemble de données. Ces données ne sont pas forcément des valeurs primitives.
On peut construire une instance à partir d'autres instances.

Ainsi en supposant que l'on ait défini une classe "Adresse" on pourrait définir:

```
public class Salarie{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        ...
        public String toString() {
                  // permet d'afficher les informations contenues
        }
        // ACCESSEURS
        public String getID() { return id;}
        ...
        // Autres méthodes
        public double coûtHoraire() {
        ....
        }
}
```

Notons toutefois que d'un point de vue conceptuel on a ici deux situations différentes :

- On peut considérer qu'une instance de "Salarié" *contient* effectivement une instance de "Adresse" (relation de *composition*). Les deux instances ont un cycle de vie étroitement lié (même si l'adresse peut changer).

- Le champ "supérieurHiérarchique" est une référence vers une autre instance relativement indépendante (relation d'*association*)

Imbrications d'instances : délégation

Bien entendu on peut définir des méthodes d'instance qui utilisent des méthodes des instances "contenues".
Par exemple, dans l'exemple précédent, la définition de la méthode `toString`() pourrait faire appel à une méthode `toString()` du champ "adresse".

Considérons maintenant l'exemple suivant:

```
public class Executant{
        private Salarie employé ;
        // données spécifiques
        private Tache[] tâchesDuMois;
                  ....
        // méthodes spécifiques
        public double getCoût() {
                  ...
        }
        ...
        //Constructeur
        public Exécutant(Salarie personne) {
                  employé = personne ;
        }
        // méthodes déléguées
        public int getID() {return employé.getID()}
        public double getCoûtHoraire() {
                  ...// on demande à l'instance "employé"
        }
        ....
}
```

Ici plutôt que d'élargir la définition de la classe "Salarié" pour l'utiliser dans une autre perspective (la gestion des tâches) on a préféré définir une nouvelle classe qui utilise les services de la classe précédente. Comme certaines des méthodes de la classe "Salarié" sont pertinentes pour la classe "Exécutant" on *délègue* leur réalisation à une instance associée qui fait partie de la définition de l'Executant. Il aurait été mal avisé de dupliquer les informations gérées par la classe "Salarié" dans la classe "Exécutant".

Relation “est un”

Supposons que pour la gestion du personnel on ait besoin de créer une classe particulière pour les “Managers”.
Si on définissait cette classe en partant de zéro on aurait quelque chose comme :

```
public class Manager{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        private Salarie[] subordonnés ;
        ...
```

Ici on constate que l’on est en train de reprendre les éléments principaux de la classe “Salarié” pour les recopier dans cette nouvelle classe.

Or d’un point de vue conceptuel un “Manager” **est un** Salarié (avec, certes, quelques caractéristiques supplémentaires). Il serait donc pertinent de factoriser le code qui concerne le concept de Salarié pour ne définir que ce qui concerne le cas particulier du “Manager”. L’écriture et la maintenance de ces classes en serait facilitée.

Comme tout langage à objets Java propose un mécanisme d’**héritage** pour gérer cette situation.

Héritage: mot-clef extends

```
public class Salarie{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        ...
}

// fichier différent pour la classe Manager

public class Manager extends Salarie {
        private Salarie[] subordonnés;
        ...
```

La classe “Manager” est définie maintenant pour disposer des champs et méthodes qui sont *héritées* de la définition de la *classe parente.*

La classe “fille” (Manager) disposera en propre :

- de champs d’instance spécifiques
- de méthodes d’instance spécifiques
- de méthodes d’instance qui constituent des redéfinitions de méthodes héritées (*spécialisation*).
- de constructeurs (on n’hérite d’aucun constructeur)

Dans la documentation de l’API du SDK la description d’une classe ne comprend que ces définitions spécifiques. Lorsqu’on recherche un certain membre d’une classe il peut être nécessaire de rechercher sa description parmi les ancêtres de la classe courante.

Spécialisation des méthodes

Reprenons l'exemple de la classe "Salarié":

```
public class Salarie{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        ...
        public String toString() {
                // permet d'afficher les informations contenues
        }
        // Accesseurs
        public String getID() { return id;}
        ...
        // Autres méthodes
        public double coûtHoraire() {
        ....
        }
}
```

La classe "Manager" s'écrirait par exemple:

```
public class Manager extends Salarie {
        // champs spécifiques
        private Salarie[] subordonnés;
        ...
        // méthodes spécifiques
        public Salarie[] getSubordonnés() { return subordonnés;}
        ....
        // méthodes redéfinies
        public double coûtHoraire() {
                ...//calculé différemment pour les managers
        }

        public String toString() {
                return super.toString() +
                        " nombre de subordonnés=" +
                                subordonnés.length +
                        ..... // autre champs
        }
}
```

Spécialisation de méthodes (method overriding)

Dans l'exemple de la classe "Manager" les méthodes `coûtHoraire()` et `toString()` existent dans la classe mère et sont redéfinies dans la classe fille pour être spécialisées.

La nouvelle définition de ces méthodes peut s'appuyer sur un code complètement autonome ou sur un code faisant usage de la méthode définie dans la classe mère. Dans ce dernier cas la méthode originelle est référencée par le mot-clef **`super`** (`super.toString()` dans l'exemple).

Quelques remarques complémentaires:

- Java ne connaît que l'*héritage simple*. Au contraire d'autres langages à objet qui connaissent l'héritage multiple une classe Java ne peut dériver (mot-clef `extends`) que d'une seule classe mère. Cette caractéristique du langage est conforme à un souci de génie logiciel: il vaut mieux que le programmeur rende explicite ses intentions. Que se passerait-il si la classe "Manager" héritait à la fois de "Salarié" et de "Actionnaire", et si les deux classes mère avaient une méthode `toString()`? Plutôt que de définir des règles complexes Java préfère laisser le programmeur décrire précisément le comportement de la classe.

- Un type Java est un véritable "contrat" passé entre ceux qui définissent la classe et ceux qui l'utilisent. Le fait de créer un nouveau type par héritage d'un type existant ne permet pas d'aggraver les termes du "contrat" initial: on ne peut pas spécialiser un méthode en aggravant ses conditions d'utilisations, le compilateur Java veille donc à ce qu'une méthode redéfinie ne soit pas plus "privée" que la méthode originelle .

```
public class Mere {
        public int methode(int arg) { ....

public class Fille extends Mere {
        private int methode (int arg)// ERREUR COMPILATION
         // le type retourné doit aussi être le même
         // par contre: methode(String arg) est possible
```

Ceci nous amène à préciser la notion de "type" d'une référence vers un objet: une référence typée "Salarié" peut-elle désigner un objet dont le type effectif est "Manager"?

Point intermédiaire : héritage

L'héritage a un double aspect : conceptuel (relation "est un") et opérationnel (mutualisation de code).

Mini-exercice :

- Suite à une nouvelle disposition réglementaire l'application de la taxe sur la carie dentaire est modifiée : les produits contenant moins de 50% de sucre seront moins taxés (rappelons que nous sommes ici dans un monde virtuel...)

- Créer une nouvelle classe "ConfiserieLight" dérivée de "Confiserie" et ayant un champ supplémentaire "tauxSucre" de type `int` (qui sera ultérieurement utilisé pour calculer le montant effectif de la taxe).
  Définir un constructeur pour cette classe
  Normallement le code ne doit pas se compiler: analyser pourquoi...

- Exercice complémentaire (pour les très rapides ou ceux qui révisent): re-définir:

```
double prixClient(int nb)
```

Le nouveau calcul opère de la manière suivante:

$$resultat = (prixClient\text{avant}) \div (1 + ((50 - tauxSucre) \div 1000))$$

(voir un calcul équivalent dans la classe "TestExpr")

Polymorphisme

Conceptuellement un “Manager” **est un** “Salarié”, par ailleurs du fait de l’héritage des classes on doit s’attendre à trouver associés à une instance de “Manager” tous les membres (champs et méthodes) associés à une instance de “Salarié”. De ce fait il est légal d’écrire en Java:

```
Salarie salarié = new Manager() ;
Salarie[] équipe1 = {
                    new Salarie(),
                    new Salarie(),
                    new Manager(),
          ....
          } ;
```

Les objets ont un *type effectif*, mais les variables qui les désignent sont *polymorphes*: c’est à dire qu’elles peuvent être utilisées pour des objets sous différentes formes (ayant éventuellement des types effectifs différents). Il s’ensuit que le typage des variables a des conséquences différenciées au *run-time* et au *compile-time*:

Evaluation dynamique:

Soit le programme :

```
for(int ix = 0; ix < équipe1.length; ix++){
          System.out.println(équipe1[ix].toString());
}
```

Quelle est la méthode toString() qui va être appelée quand le membre du tableau sera effectivement un “Manager”: celle de “Salarié” ou celle de “Manager”?

En fait l’exécuteur Java appelle bien la méthode liée au type effectif de l’objet, c’est à dire que, dynamiquement, au moment de l’exécution il “choisit” la bonne méthode. Cette caractéristique importante du polymorphisme est appelée *liaison dynamique (virtual method invocation).*

Héritage: on hérite des membres pas des constructeurs

Le fait que l'on ait:

```
public class Salarie{
        ....
        public Salarie (String id, String nom, Adresse adresse){
                ....
```

N'autorise pas automatiquement à invoquer un constructeur analogue sur une classe dérivée :

```
Manager monManager = new Manager(sonId, sonNom, sonAdresse);
```

Il faut ici que la classe "Manager" ait explicitement défini le constructeur correspondant! Les constructeurs n'étant pas considérés comme des membres de la classe on n'en hérite pas dans les classes dérivées.

Voir aussi: *Perte du constructeur par défaut*, page 179.

Mot-clef super pour l'invocation d'un constructeur

Dans l'exemple précédent nous avons noté que:

- Il faut définir explicitement un constructeur pour la classe dérivée "Manager"

- Lors de la construction de l'instance de Manager, il faudra "construire" une instance de la classe mère. Comme celle-ci est dépourvu de constructeur implicite (sans argument) on est donc obligé d'appeler un constructeur explicite et ceci doit se faire obligatoirement dans la première instruction du constructeur courant:

```
public class Manager extends Salarie {
        .....
        public Manager(String id,String nom,Adresse adr,int nbS){
                  super(id,nom,adr);
                  ...// autres instructions si nécessaire
        }
```

- Dans cet exemple les champs principaux de `Salarie` sont marqués `private`. Le code de la classe `Manager` ne peut y accéder directement: pour les initialiser il est obligé de passer uniquement par le code prévu par le rédacteur de la classe `Salarie` (donc par l'appel de `super`) et pour les consulter il doit passer par les accesseurs dont il hérite. D'autres solutions sont possibles pour relacher un peu ces contraintes d'encapsulation (voir chapitre suivant)

Voir aussi: *Opérations d'initialisation des instances*, page 179.

Point intermédiaire : *héritage, constructeur*

Le polymorphisme permet d'améliorer la maintenabilité du code: le code demandeur d'un service ne sait pas comment il est rendu (découplage). La création d'un nouveau type d'instance ne fait pas modifier des codes existant qui s'intéressent à des propriétés générales des objets.

Le contructeur d'une classe doit s'acquitter des contraintes d'initialisations éventuellement définies pour sa super-classe (emploi explicite de `super`).

Mini-exercice :

- compléter le constructeur de "ConfiserieLight" pour que le code se compile

Spécialisations courantes: toString, equals, clone

Toute classe est implicitement dérivée de la classe java.lang.Object. Chaque fois que l'on écrit :

```
public class MaClasse {
        ...
```

on a en fait:

```
public class MaClasse extends Object {
        ...
```

On peut donc utiliser la classe Object chaque fois que l'on veut gérer des collections c'est à dire des ensembles d'objets que l'on veut stocker, accéder, etc. :

```
Object[] tableauDeNimporteQuoi;//utilisé aussi par les collections
        // du package java.util (par ex. ArrayList)
```

Par ailleurs toute classe hérite des méthodes de la classe Object (voir documentation) et en particulier de:

- `toString()`: donne une chaîne descriptive de l'instance courante. Ainsi la méthode `println(Object obj)` de la classe `java.io.PrintStream` appelle implicitement `obj.toString()` (il en est de même de la concaténation entre chaînes et objets).

- `equals()`: est redéfinie dans de nombreuses classes pour exprimer que le *contenu* de deux instances est le même (par défaut `equals` rend le même résultat que == c'est à dire le fait que les deux instances sont en fait le même objet en mémoire). La redéfinition de `equals` dans une classe suppose souvent une redéfinition de la méthode `hashCode()` dont la description sort du périmètre de ce cours.

- `clone()`: permet de redéfinir la manière dont on crée une nouvelle instance qui est une copie de l'instance courante.

Forçage de type pour des références

Considerons l'exemple suivant:

```
import java.util.ArrayList ;

public class ListeDeSalarie {
   private ArrayList liste = new ArrayList() ;

   public void add(Salarie sl) {
      liste.add(sl) ;
   }

   public Salarie get(int index) {
      return (Salarie) liste.get(index) ;
   }
}
```

On a voulu définir une collection pour laquelle il y ait un contrôle de type fort. La classe `java.util.ArrayList` permet de gérer un tableau "extensible" d'instances de la classe `Object`. Nous voulons un dispositif analogue qui ne contienne que des instances de type `Salarie`.

- La méthode `add(Salarie)` s'appuie sur la méthode `add(Object)` de `ArrayList`. Dans la mesure où un `Salarie` dérive de `Object` la promotion de la référence "sl" est implicite.

- L'inverse n'est pas vrai: on ne peut "convertir" implicitement le résultat de la méthode `get` de `Arraylist` qui rend un `Object` en une référence de `Salarie`. On est obligé d'utiliser l'opérateur de *projection de type*(*cast*) qui force le compilateur à accepter l'opération.

De manière générale cet opérateur indique au compilateur qu'il doit prendre en compte le fait que la référence désignée doit accepter les obligations découlant du type proposé (et donc doit accepter de référencer des appels de méthodes spécifiques au type demandé)

Notons que :

- Si au moment de l'exécution le type effectif de l'objet n'est pas compatible avec le type de la référence "forcée" une erreur d'exécution surviendra (`ClassCastException` : voir chapitre sur les exceptions).

- Contrairement à ce qui se passe avec des conversions de types primitifs scalaires il n'y a pas de transformation d'objet. Le fait d'écrire `(Manager)unSalarie` ne crée pas une nouvelle instance de type effectif "Manager".

Nota: pour traduire le terme anglais *cast* nous employons ici des terminologies non standard : "conversion" lorsque l'opération s'applique sur un scalaire, "projection de type" sur une référence et "transtypage" pour réunir les deux concepts.

Noter ici l'utilisation de la délégation: si "Liste deSalarie" avait hérité de ArrayList on aurait aussi hérité des méthodes `add(Object)` et `Object get(int)` dont on cherche précisément à se débarasser! Par ailleurs le fait de cacher la réalisation par `ArrayList` permet éventuellement d'implanter la liste d'une manière différente et de faire évoluer cette réalisation sans impacter les autres codes.

Opérateur instanceof

Reprenons l'exemple précédent:

```
Salarie[] équipe1 = {
                    new Salarie(),
                    new Salarie(),
                    new Manager(),
          ....
          } ;
```

Peut-on parcourir ce tableau en sélectionnant une instance spécifiquement parcequ'elle répond au contrat de `Manager`?

L'opérateur `instanceof` permet de répondre précisément à cette question (l'opérateur permet de tester si une instance répond au contrat du type passé en paramètre).

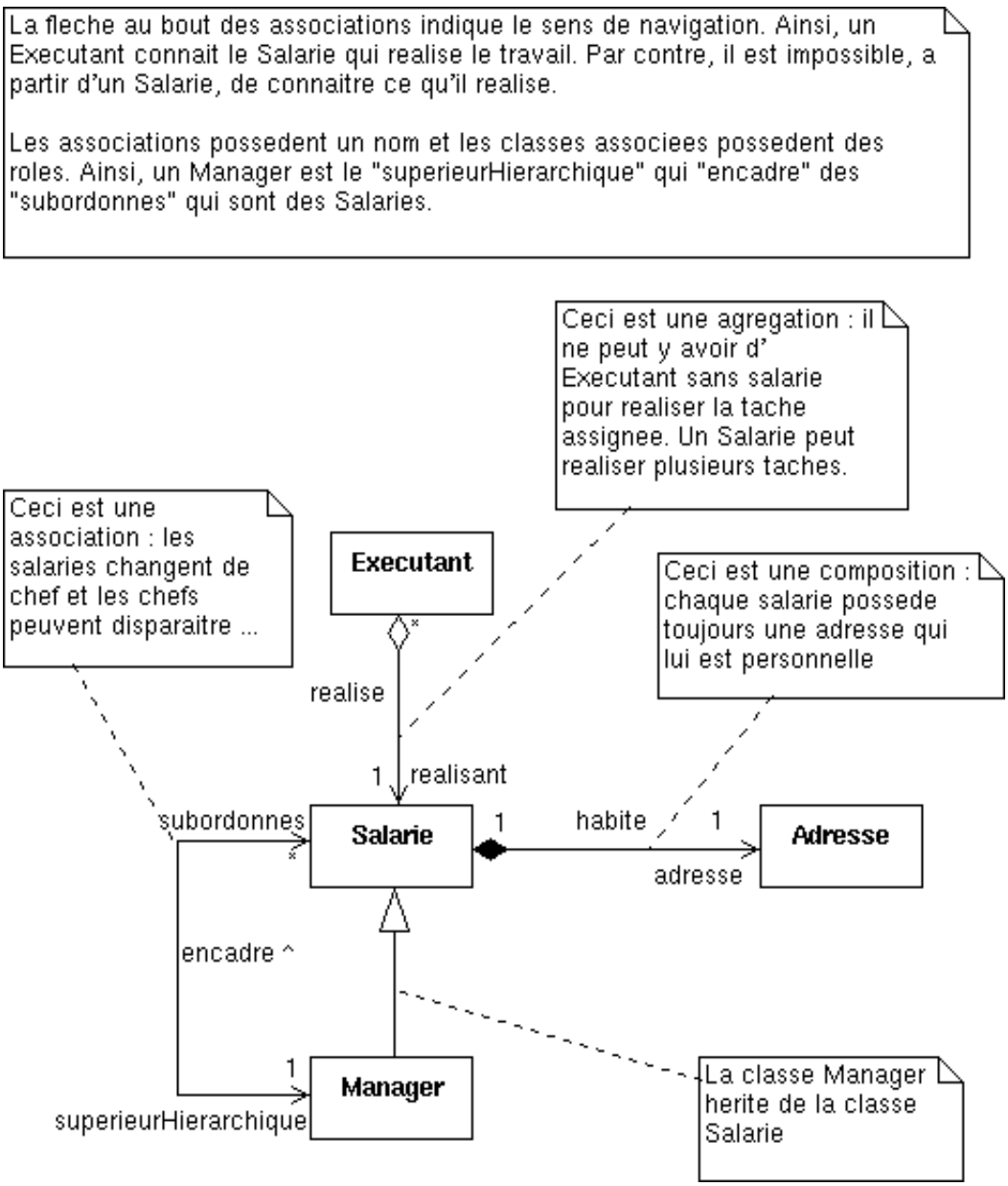
```
ListeDeSalarie invités ;
...
for (int ix = 0 ; ix < équipe1.length; ix++) {
          if ( équipe1[ix] instanceof Manager) {
                    invités.add(équipe1[ix]);
          }
}
...
```

ATTENTION: le test ne donne pas forcément le type effectif de l'objet! Ainsi dans l'exemple précédent l'expression "`instanceof Salarie`" appliquée à un objet de type effectif Manager rendrait un résultat positif: un Manager est un Salarié!

Compléments

Les compléments techniques suivants constituent une annexe de référence

Utilisation notation UML



Perte du constructeur par défaut

ATTENTION: le cas du constructeur par défaut (sans argument) conduit à une situation particulière:
On sait que le fait de définir un constructeur avec arguments fait disparaître ce constructeur implicite (sauf si on donne une définition explicite du constructeur sans argument).
Or, lors de l'opération de construction d'une instance, les opérations de construction des instances des classes mères doivent être successivement appelées.
Si la classe mère a "perdu" son constructeur par défaut, le compilateur ne nous permettra pas de disposer d'un constructeur par défaut pour la classe courante.
Dans l'exemple précédent il devient impossible d'invoquer:

```
new Manager(); // ERREUR COMPILATION
```

car on ne peut faire :

```
new Salarie(); // constructeur inexistant
```

Opérations d'initialisation des instances

Ordre des évaluations au moment de la construction de l'instance (précise les informations données au chapitre précédent):

1. allocation de l'instance
2. initialisations des super-classes (au travers de super())
3. évaluations des initialisations explicites de variables membres (l'ordre est important)
4. exécution du reste du code du constructeur.

Rupture du lien statique

L'évaluation dynamique ne s'applique qu'aux méthodes d'instance susceptibles d'être spécialisées. Tout autre type de redéfinition dans une sous-classe est en général à éviter car les effets en sont très difficiles à controler.

- Il est possible de redéfinir une méthode de classe dans une sous-classe; comme pour les méthodes d'instance on ne doit pas aggraver le "contrat" de la méthode redéfinie. L'appel de la méthode correcte est décidé au compile-time: cette rupture du lien statique (*hiding*) n'est viable que si les classes utilisatrices ont été correctement recompilées. Par ailleurs il est interdit de redéfinir une méthode d'instance par une méthode statique (et réciproquement).

- Bien que cela soit possible il vaut mieux également éviter des redéfinition de variables. La nouvelle définition "cache" l'ancienne (que l'on peut toujours accéder par `super`) et du coup une instance peut, par exemple, avoir deux variables de même nom dont les conditions d'accès (et éventuellement les types) diffèrent!

- Bien que ce conseil de prudence dans les redéfinitions soit général il y a des cas où on peut les justifier dans des dispositifs avancés très particuliers (*introspection*). Ceci est utilisé dans des cas qui sortent du périmètre de ce cours (voir, par ex. l'utilisation et, éventuellement la redéfinition, des méthodes privées `readObject`, `writeObject` des objets `Serializable`)

Surcharges

Il y a souvent une confusion entre surcharge (*overloading*) et spécialisation (*overriding*). Dans le langage Java les deux termes ont des sens précis qui diffèrent considérablement.

Si deux méthodes d'une classe (déclarées localement ou héritées) ont le même nom mais des signatures différentes alors il y a surcharge.

On peut avoir des méthodes avec le même nom, une liste de paramètres (et éventuellement un type de retour) différents.

```
public int meth(int x) { ....
public int meth(int x, String s) { ....// surcharge OK
public int meth(int float) { .... // surcharge OK
public float meth(String x) { .... // surcharge OK
public float meth(int x) { ...// ERREUR la différence ne peut
                  // porter uniquement sur le type de retour
```

Les spécialisations de méthodes surchargées se font au coup par coup (chacune avec sa signature): il est ainsi possible d'hériter de deux méthodes publiques surchargées, d'en spécialiser une et de toujours hériter de l'autre (Java diffère de C++ sur ce point).

Les classes d'encapsulation (wrapper classes)

Nous avons vu que les collections opéraient sur des objets (leur contenu est défini comme utilisant des instances de Object). Ceci interdit de mettre des primitifs scalaires dans des collections.

Dans des situations analogues on peut utiliser des instances de classes d'encapsulation. Ces classes (comme `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`) permettent de créer des instances d'objet qui contiennent une valeur scalaire immuable. Diverses méthodes permettent d'obtenir cette valeur sous différentes formes; leur méthode `equals` teste l´égalité pour le scalaire encapsulé.

Corrigés des mini-exercices

ConfiserieLight.java (ne peut se compiler) :

```
public class ConfiserieLight extends Confiserie {
    private int tauxSucre ;

   public ConfiserieLight( String nom, double prix, int taux) {
      // ERREUR! on ne sait pas initialiser la super-classe
      this.nom = nom ;// impossible! nom est privé
      this.prix = prix ; //impossible! prix est privé
      tauxSucre = taux ;
   }

   public int getTaux() { return tauxSucre;}
}
```

ConfiserieLight.java (se compile) :

```
public class ConfiserieLight extends Confiserie {
    private int tauxSucre ;

   public ConfiserieLight( String nom, double prix, int taux) {
      super(nom, prix) ;
      tauxSucre = taux ;
   }

   public int getTaux() { return tauxSucre;}

   // pourrait se calculer d'une autre manière !
   public double prixClient(int nb) {
      double resultat = super.prixClient(nb) /
                  (1 + ((50 -tauxSucre)/1000.0)) ;
      return Math.rint(resultat * 100) /100 ;
   }

   public static void main (String[] args) {
      Confiserie[] tb = {
         new Confiserie("roudoudou", .10) ,
         new ConfiserieLight("roudoudouLight", .10, 25) ,
         new ConfiserieLight("pseudoLight", .10, 50) ,
         new ConfiserieLight("ultraLight", .10, 0) ,
      } ;
      for(int ix = 0 ; ix < tb.length ; ix ++ ) {
         System.out.println(tb[ix].prixClient(100) );
      }
   }
}
```

Exercices :

*Exercice ** :*

Toujours le projet “oubangui.com”.

Définir une classe `Client` et une classe `Revendeur`. Un `Revendeur` est un `Client` mais, en particulier, il dispose d’un taux de réduction spécial lorsqu’on lui vend des livres.

Modifier la classe `Panier` de manière à permettre de calculer le prix total du panier. Le calcul de ce prix tient compte du taux de réduction applicable au client.

Montrer le fonctionnement de cette classe en définissant deux paniers avec les mêmes livres mais attribués à des clients différents (un `Client` et un `Revendeur`).

Modularité, modificateurs 7

Points essentiels

La modularité en Java:

- Les paquetages
- Contrôles d'accès entre objets
- Modificateur `final`
- Modificateur `static`

packages

Dès que l'on développe un code qui va au delà d'un petit exemple provisoire, les classes Java doivent être regroupées dans des *packages*.

Un package regroupe un ensemble de classes autour d'un thème commun. Ces classes sont supposées se "connaître" entre elles: c'est à dire qu´étant, en pratique, écrites et maintenues ensembles, elles ont la possibilité de constituer des modules logiques dans lesquels le degré d'encapsulation n'est pas le même que pour les classes "extérieures" au package.

De manière standard le package constitue l'élément de granularité des versions de logiciel Java. Un programmeur utilisant les services d'un package peut avoir besoin de connaître dynamiquement sa version (voir classe `java.lang.Package`)

Les packages sont eux-mêmes organisés en hiérarchies et il est de coutume de préfixer leur nom à partir d'un identifiant internet. Soit, par exemple, deux sociétés connues comme `acme.com` et `gibis.fr` , des packages fournis par ces sociétés pourraient être:

```
com.acme.finance.utils.swap
fr.gibis.utils
```

Rappel: à partir du moment où une classe fait partie d'un package le "vrai" nom de la classe est précédé du nom du package :

```
package fr.gibis.utils ;
public class Chapeau { ....
```

Le "vrai" nom de cette classe est: `fr.gibis.utils.Chapeau` . C'est ce nom qui sera passé au lancement de la machine virtuelle si cette classe est munie d'un point d'entrée :

```
java fr.gibis.utils.Chapeau
```

Packages

Java Platform Packages	
java.applet	Provides the classes necessary to create an applet and t
java.awt	Contains all of the classes for creating user interfaces an
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data betv
java.awt.dnd	Provides interfaces and classes for supporting drag-and
java.awt.event	Provides interfaces and classes for dealing with different
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing
java.awt.im	Provides classes and an interface for the input method fr
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-
java.awt.print	Provides classes and interfaces for a general printing AP
java.beans	Contains classes related to Java Beans development.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data strear
java.lang	Provides classes that are fundamental to the design of th
java.lang.ref	Provides reference-object classes, which support a limit
java.lang.reflect	Provides classes and interfaces for obtaining reflective in
java.math	Provides classes for performing arbitrary-precision integ
java.net	Provides the classes for implementing networking applic
java.rmi	Provides the RMI package.
java.rmi.activation	Provides support for RMI Object Activation.
java.rmi.dgc	Provides classes and interface for RMI distributed garba
java.rmi.registry	Provides a class and two interfaces for the RMI registry.
java.rmi.server	Provides classes and interfaces for supporting the server
java.security	Provides the classes and interfaces for the security frame
java.security.acl	The classes and interfaces in this package have been sup
java.security.cert	Provides classes and interfaces for parsing and managing
java.security.interfaces	Provides interfaces for generating RSA (Rivest, Shamir a DSA (Digital Signature Algorithm) keys as defined in N
java.security.spec	Provides classes and interfaces for key specifications an
java.sql	Provides the JDBC package.
java.text	Provides classes and interfaces for handling text, dates,
java.util	Contains the collections framework, legacy collection cla tokenizer, a random-number generator, and a bit array).

Organisation pratique des répertoires

Lors d'une exécution locale les fichiers .class à exécuter par la J.V.M. doivent se trouver dans une ressource. Cette ressource peut être un répertoire ou une archive java (fichier .jar). Dans les deux cas le fichier .class se trouve dans une hiérarchie correspondant à la hiérarchie des packages.

Ainsi le fichier `Chapeau.class` contenant la classe de nom `fr.gibis.utils.Chapeau` se trouvera dans un répertoire:

racine_classes`/fr/gibis/utils (UNIX)`
racine_classes`\fr\gibis\utils (WINDOWS)`

Pour toute exécution *racine_classes* (ou archive jar) doit se trouver mentionné dans une liste décrite dans le contenu de la variable d'environnement CLASSPATH (ex. UNIX `/home/me/jclasses:.`)

Voir également le mécanisme d'extension dans la documentation `docs/guide/extensions/extensions.html`

Pour le développement :

- Tenir compte du fait que le compilateur a besoin d'accéder aux fichiers *.class* de toutes les classes utilisées dans le code (pour vérifier leur interface). On peut aussi passer des chemins d'accès en paramètre (option -classpath).

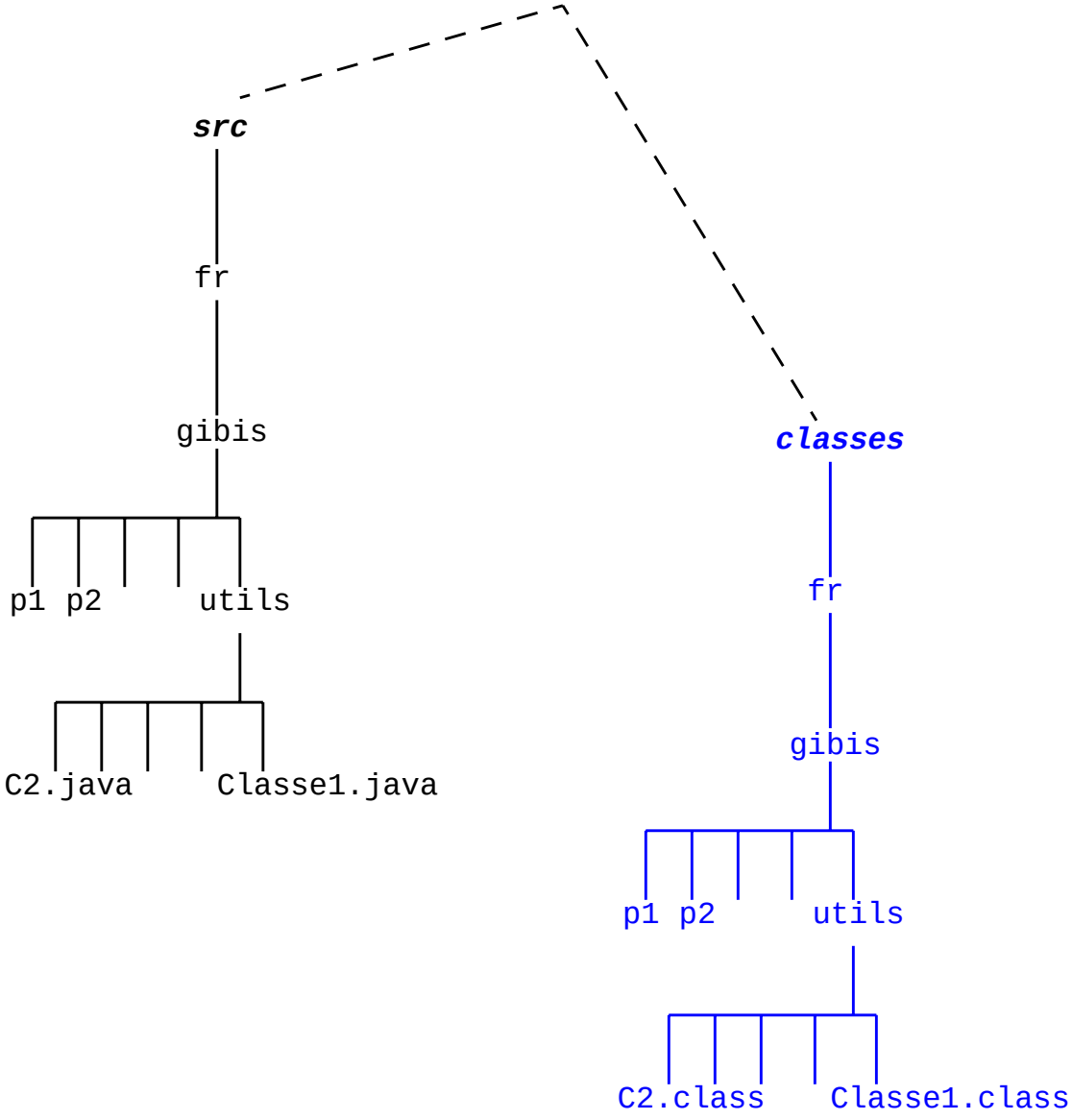
- Générer les .class dans le répertoire racine approprié :

`javac` **`-d`** *`racine_classes fichier_java`*

- Il vaut mieux organiser les sources dans une hiérarchie de répertoires correspondant à la hiérarchie des packages et permettre au compilateur de vérifier l'antériorité des sources par rapport au .class correspondant.

`javac -d` *`racine_classes`* `-sourcepath` *`racine_sources fichier_java`*

Organisation pratique des répertoires



javac -classpath *classes* -d *classes* -sourcepath *src* Classe1.java

Déclaration de visibilité (import)

Pour utliser les services d'une classe dans un code on peut donner son "vrai" nom:

```
new fr.gibis.utils.Chapeau(java.awt.Color.blue) ;
```

On peut également demander au compilateur de rechercher lui même le "vrai" nom de la classe dans une liste de packages:

```
import fr.gibis.utils.* ; // pour Chapeau
import java.awt.* ; // pour Color
......
         new Chapeau(Color.blue) ;
```

La directive `import` (qui doit se situer avant toute déclaration de classe) permet au compilateur de lever toute ambiguité sur un nom de classe. Noter que :

- L'on peut "importer" spécifiquement une classe sans "importer" toutes les classes de son package. .

```
import java.awt.Color ;
```

Il faut préférer ce type de déclaration si on veut obtenir un comportement efficace du compilateur (recompilation automatique).

- La notation * permet de voir toutes les classes du package mais pas les sous-packages

```
import java.awt.*; // on ne voit pas java.awt.event.KeyEvent
```

- Il est inutile d'importer les autres classes situées dans le package courant.

- `import` est une directive destinée au compilateur elle n'affecte en rien les performances du *run-time*.

- La vérification des "signatures" des classes par le compilateur se fait en utilisant les fichiers binaires (il n'y a pas d'inclusion de déclarations comme en C/C++ par ex.). Le compilateur est capable d'obtenir directement les informations d'API par "*introspection*" des classes.

Contrôles d'accès

Les membres et constructeurs d'une classe peuvent avoir quatre types de droits d'accès: `public`, `protected`, accès par défaut et `private`. Au premier niveau d'un fichier source les classes peuvent être `public` ou avoir un niveau d'accès par défaut

Résumé des règles d'accès:

Modificateur	**Même Classe**	**Même Package**	**Sous-classe (hors pack.)**	**Ailleurs**
`public`	Oui	Oui	Oui	Oui
`protected`	Oui	Oui	oui *	
par défaut	Oui	Oui		
`private`	Oui			

L'accès `protected` permet en règle générale aux sous-classes situées dans un autre package d'accéder au contenu de la classe courante. Toutefois dans certains cas très particuliers cet accès n'est pas autorisé et est refusé par le compilateur (Voir *Compléments sur les modificateurs d'accès*, page 202.).

Le choix des modificateurs d'accès ne doit pas être traité à la légére et demande un soin tout particulier au niveau de la conception d'une application.

Point intermédiaire : packages, visibilité

Mini-exercice:

- mettre “Confiserie” et “ConfiserieLight” dans un package “org.sucres”. Modifier les déclarations, compiler avec des options appropriées.

- Exercice complémentaire (pour les très rapides ou ceux qui révisent):
  jouer avec les modificateurs des déclarations des membres (protected par ex.)

Modificateurs `final`

Classes marquées `final`

Une classe définie comme `final` ne peut plus avoir de classe dérivée (on ne peut pas hériter d'une classe `final`).

Pour des raisons de sécurité de nombreux types de base (comme String, StringBuffer, les types objets encapsulant des types primitifs -Integer, Double, etc.-) sont déclarés final.

Méthodes marquées `final`

Une méthode définie comme `final` ne peut plus être redéfinie dans une sous-classe.

Exemple: dans la classe `java.awt.Window` la méthode `final String getWarningString()` permet de savoir que cette fenêtre ne vient pas d'une application locale mais d'une application supposée "peu fiable" comme une Applet. Pour d'évidentes raisons de sécurité on doit empécher de redéfinir cette méthode.

Remarque: le fait de marquer une méthode comme `final` permet au compilateur de faire de l'optimisation dans certains codes d'appel de cette méthode: puisqu'il n'y a pas de redéfinition possible l'évaluation dynamique en fonction du type effectif de l'objet n'a plus cours. Cette optimisation est plus générale avec les méthodes marquées `private` (voir JLS 13.4.21)

Modificateurs `final`

Variables marquées `final`

- Variables membres initialisées: on ne peut modifier la valeur d'une variable membre initialisée et marquée "final"
  `public static final int TOTAL_HORAIRE_MAX = 35 ;`
  On a en général une constante (Attention: si cette variable est une instance on peut éventuellement modifier un de ses propres membres). Pour examiner la raison du modificateur `static` Voir *variables partagées*, page 197.

- Variables membres non initialisées (*blank final*) : une variable membre marquée "final" et non initialisée est obligatoirement initialisée au moment de la construction de l'instance. La valeur ne peut plus être ensuite modifiée:

```
public class Compte {
        public final String clef ;
        ...
        public Compte(String id, Client client,...) {
                ...
                this.clef = id ;
        }
        ...
}
```

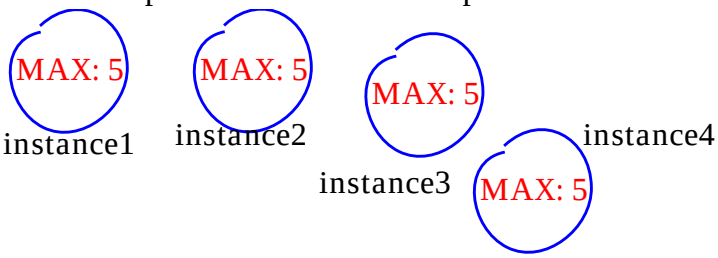
Modificateur static (rappel)

variables partagées

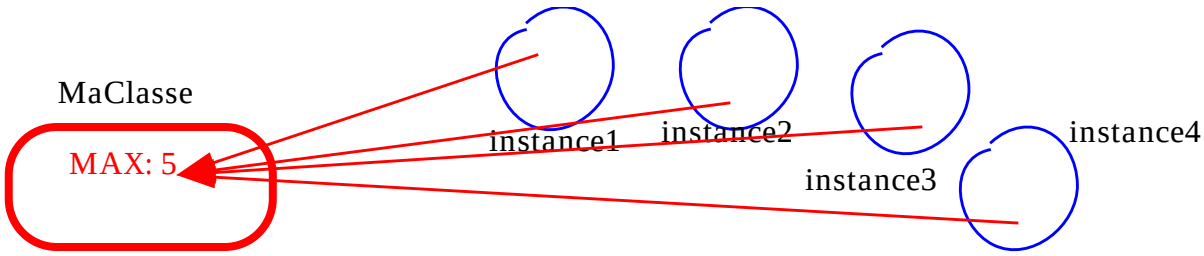
Reprenons l'exemple précédent sur les constantes et regardons ce qu'aurait produit une déclaration comme :

```
public final int MAX = 5 ;
```

Dans ce cas on aurait eu une copie de cette constante par instance :



Comment avoir une seule copie de MAX qui soit partagée par toutes les instances? En installant cette variable en mémoire dans la représentation de la classe elle-même::



On a ainsi une variable partagée.

Une variable partagée n'est pas nécessairement une constante.

Les variables liées à une classe (et non à une instance) sont marquées `static`.

```
public class Contexte {
        public static boolean surDoze =
         System.getProperty("os.name").startsWith("Win");
        ...
}
```

Ici la variable est initialisée au moment du chargement de la classe par un `ClassLoader`. Pour consulter sa valeur :
`if (Contexte.surDoze) {..`

```
// autre exemple
public class Client {
        private static int compteur ; // variable de classe
        public final int id = compteur++; // var d'instance
        ...
}
```

Ici lors de la création d'une instance on utilise la variable de classe et on l'incrémente.

Autre exemple de variable partagée :

`System.out`

méthodes de classe

On a parfois besoin de créer du code qui n'est pas lié à l'état d'une instance particulière. Dans ce cas on définit une méthode de classe (marquée `static`). Exemples de méthodes de classe:

```
Math.sin(x) ...
Integer.parseInt(s) ...
System.getProperty(s)...
```

```
/ une méthode de classe
public class Contexte {
        ...
        public static int getValFromDB(String dbname, String nm){
                ....// va chercher une valeur entière
                ....// sauvegardée dans une ressource
        }
}
```

- On peut invoquer une méthode statique en la qualifiant par le nom de la classe (ou, éventuellement, par un nom d'instance de cette classe -mais cecla obscurcit le code-). Par contre à l'intérieur d'une méthode statique il n'y a pas d'instance courante (pas de `this`) et on ne peut accéder implicitement à des méthodes ou variables d'instance.

- `main` est `static` parcequ'il faut pouvoir l'invoquer avant que toute instanciation de classe se soit produite. Bien entendu c'est dans ce `main` qu'il faudra créer des instances et c'est sur ces instances que l'on pourra désigner des membres.
  Encore une fois on ne peut accéder dans un main à des membres d'instance de la classe dans laquelle est defini le main si on n'a pas créé d'instance!

```
public class Test {
        int membre ;
        public static void main (String[] args) {
                membre = Integer.parseInt(args[0]); //ERREUR!
        }
}
```

Voir aussi:*bloc d'initialisation de classe*, page 162..

Récapitulation: architecture d'une déclaration de classe

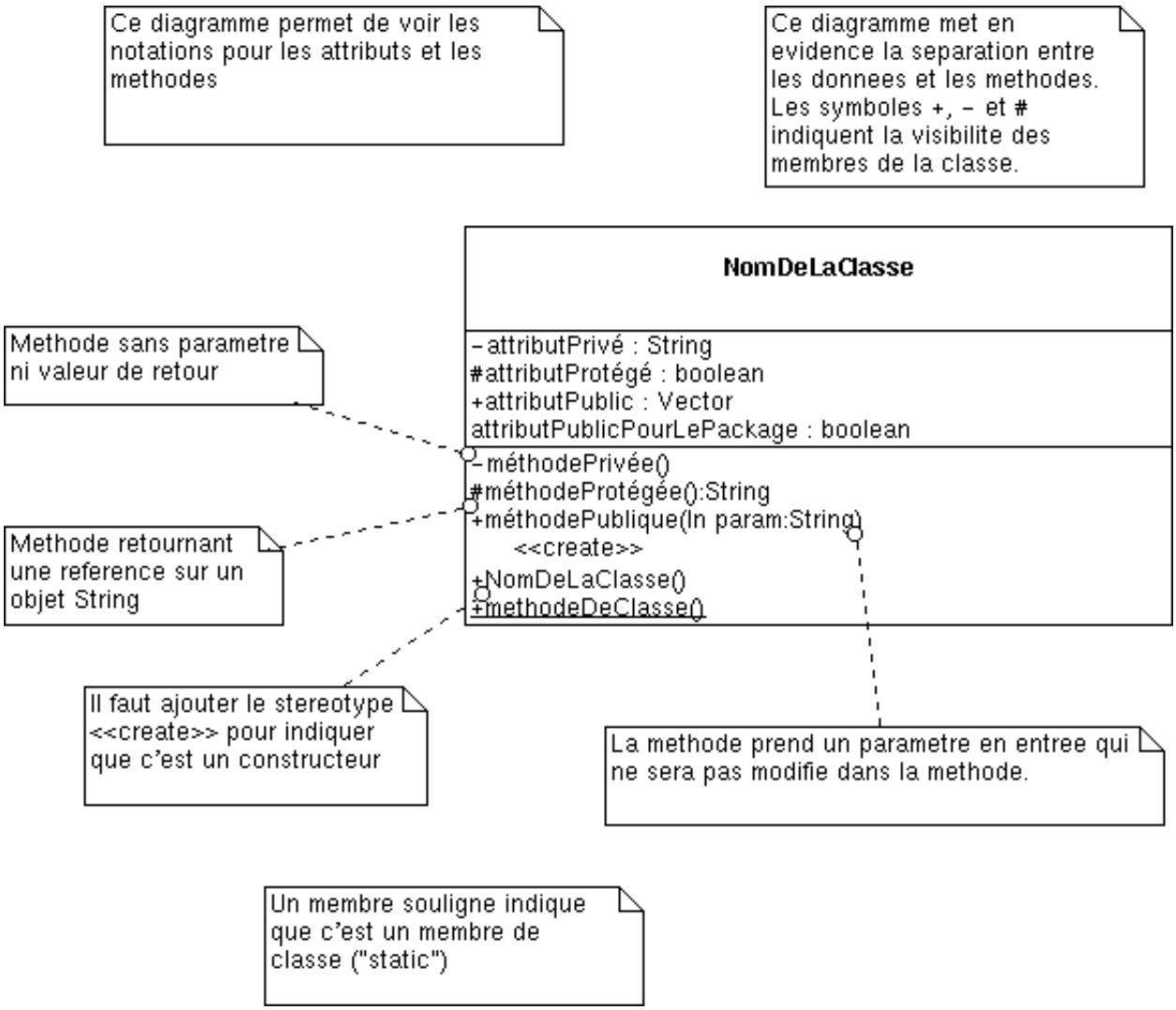
Plan général des déclarations de classe premier niveau:

```
package yyy;
import z.UneCLasse ; // A préférer
import w.* ;

public class XX { // public ou "visibilité package"
                  // une classe publique par fichier
         // ordre indifférent sauf pour les expressions
         // d'initialisation de premier niveau

         //MEMBRES STATIQUES
         variables statiques (voir "variables de classe",  page 386)
                   var. statiques initialisées
                   var. statiques "final"/constantes de classe
         méthodes statiques(voir "Méthodes de classe",  page 393)

         //MEMBRES D'INSTANCE
         variables d'instance(voir "variables d'instance",  page 384)
                   var.d'instance initialisées
                   var d'instance "blank final"
         méthodes d'instance(voir "Méthodes d'instance",  page 391)

         // CONSTRUCTEURS
         constructeurs(voir "Constructeurs",  page 394)
}
```

Compléments

Les compléments techniques suivants constituent une annexe de référence

Utilisation notation UML



Compléments sur les modificateurs d'accès

Sous certains aspects on peut dire que la hiérarchie classe / package définit en Java des zones de responsabilité du programmeur et que les modificateurs d'accès (public,private, protected) reflètent ce point de vue.

Quelques exemples :

- Le modificateur `private` donne un accès au programmeur uniquement dans la définition de la classe courante. Ainsi:

```
public class X {
        private int val ;
        public void meth(X autreInstance) {
                  int iz = autreInstance.val ;
                  autreInstance.val = ++iz ;
        }
}
```

est correct. L'objectif n'est pas d'empécher l'accès au membre privé "val" d'une autre instance mais de dire que cette opération rentre dans la responsabilité du programmeur de la classe locale.

- Le cas de `protected` est plus subtil :

```
package t1 ;
public class X {
        protected int val ;

        public void meth(X autreInstance) {
                int iz = autreInstance.val ;
                autreInstance.val = ++iz ;
        }
}
-----------------------------------------------------
package t2 ;
import t1.X ;

public class Z extends X {

        public void meth2(X autreInstance) {
                val = 10 ;
                /* ne marchent pas!!
                int iz = autreInstance.val ;
                autreInstance.val = ++iz ;
                */
        }

        public void meth3(Z autreInstance) {
                int iz = autreInstance.val ;
                autreInstance.val = ++iz ;
        }

        public static void main(String[] args) {
                X unX = new X() ;
                unX.meth(new Z()) ;
        }
}
```

pour plus de détails voir le chapitre 6.6.2 du document de spécification du langage (JLS).

La notion de zones hiérarchisées de responsabilité de programmation permet également de comprendre pourquoi un membre `protected` est accessible depuis les autres codes du même package (et pas seulement des sous-classes).

bloc d'initialisation de classe

Un bloc de code marqué `static` est évalué au moment du chargement de la classe par un `ClassLoader`. Bien entendu, en ce qui concerne la classe courante, ce code ne peut faire appel qu'à des membres statiques .

```
public class Client {
private static int compteur ;
static {
          if (Boolean.getBoolean("monAppli.confdb")) {
                   compteur = Contexte.getValFromDB(
                             "client","compteur");
          } else {
                   compteur = Integer.getInteger(
                             "monAppli.client.compteur").
                                       getValue() ;
          }
}
....
```

On peut évaluer ainsi successivement plusieurs blocs `static` dans l'ordre de leur définition.

Appel de méthodes dans un constructeur

La phase d'initialisation d'une instance est, nous l'avons vu, critique: l'ordre des initialisations est important et il ne faut pas que l'on se retrouve dans une situation où l'on soit amené à utiliser un champ non correctement initialisé.

Considérons cet exemple:

```
public class Pere {
   public Pere() {
      init() ;
   }

   public void init() {
      System.err.println("Je suis un Père") ;
   }

   public static void main(String[] args) {
         new Fils() ;
   }
}
```

```
class Fils extends Pere {
   String[] tb = new String[1] ;
   public void init() {
      System.err.println("Je suis un Fils") ;
      tb[0] = "Fils!" ;
   }
}
```

Le résultat de l'exécution est surprenant:

```
Je suis un Fils
Exception in thread "main" java.lang.NullPointerException
        at Fils.init(Pere.java:20)
        at Pere.<init>(Pere.java:3)
        at Fils.<init>(Pere.java:16)
        at Pere.main(Pere.java:12)
```

Explication: le constructeur du `Fils` appelle le constructeur du `Pere`. Celui-ci appele la méthode "init": du fait de l'évaluation dynamique c'est la méthode "init" redéfinie dans `Fils` qui est appelée. Or, à ce moment, l'initialisation d'instance de `Fils` est incomplète et le tableau "tb" n'est pas alloué!

Cet effet est certes très particulier mais doit nous amener à réflechir à quelques principes:

- Le code d'un constructeur doit être concis et ne servir qu'à initialiser l'instance.
- Si on appelle une méthode de l'instance courante dans un constructeur il peut être pertinent de la déclarer `private` ou `final`.

Exercices :

Installer les classes Livre, Panier, Client, Revendeur dans un package de la société oubangui.com.

Modifier éventuellement les classes de manière à revoir les modes d'accès aux membres, étudier la possibilité d'avoir des membres *final* (et/ou des membres *static*).

Les exceptions 8

Points essentiels

Le traitement des erreurs en Java:

- Le mécanisme des exceptions
- Définir une nouvelle exception
- Déclencher une exception
- Capturer et traiter une exception
- Propager ou traiter?

Le traitement des erreurs

Dans un certain nombre de langage le traitement des erreurs n'est pas pleinement intégré dans les mécanismes du langage, ainsi :

- Si les fonctions sont uniquement de la forme: `resultat = f(arguments)`, on a donc l'obligation de tester les résultats qui peuvent être d'une nature anormale (la fonction retourne donc deux types de résultats: les résultats normaux et les erreurs).

```
/* exemples en langage C */
        if (( fd = open(filename, ...)) < 0 ) {
                  /* tester une variable d'environnement
                   * qui indique l'erreur */
        }
        ....
        if( (fp = fopen(filename,...)) == NULL) {
                  /* idem tests environnement*/
        }
        ....
```

- Lorsque l'exécution se trouve dans un état anormal, le programme peut s'interrompre brutalement en laissant l'environnement dans un état incohérent (et en laissant l'utilisateur perplexe...)

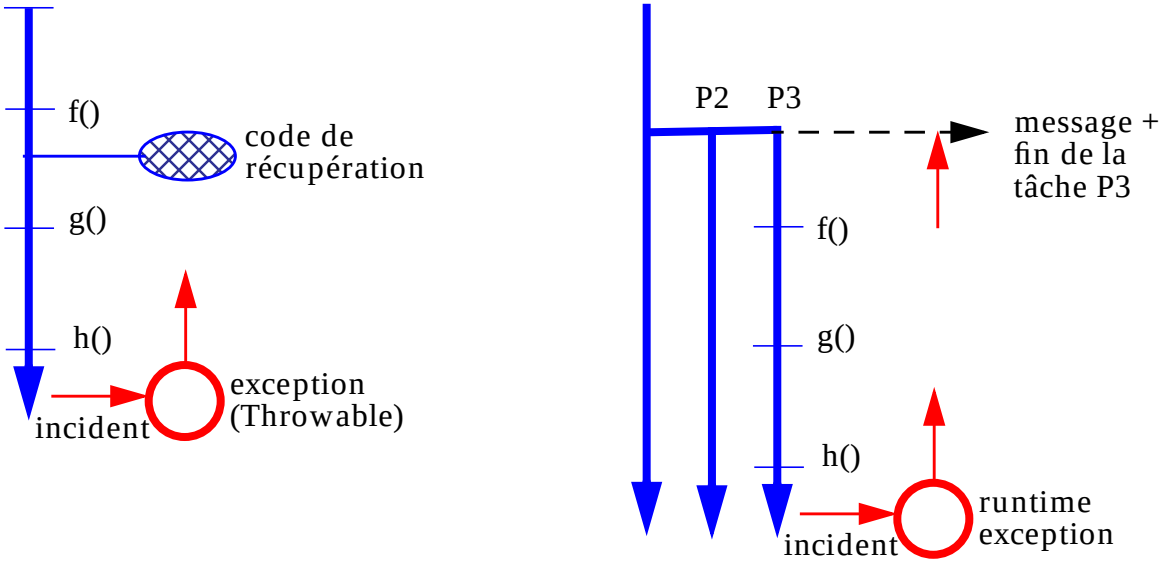
Dans des sytèmes modernes il est d'usage de distinguer la "sortie" normale des résultats d'une sortie distincte pour les erreurs. Ainsi, par exemple, un programme Java pourra afficher des résultats sur `System.out` et afficher les erreurs sur `System.err` .

Le même principe s'applique aux méthodes Java: il y a deux manières de "sortir" de l'éxécution, l'une est la voie normale (avec un éventuel retour de résultat) et l'autre est la voie exceptionnelle qui est gérée par un mécanisme spécifique: le mécanisme des exceptions.

Ce mécanisme permet de définir par programme des conditions d'erreurs. Il est aussi pleinement intégré à la machine virtuelle Java pour permettre de récupérer des erreurs d'exécution comme des erreurs d'index dans un tableau, des divisions par zéro, des déréférencements de variables de valeur `null`, etc.

Le mécanisme des exceptions Java

Lorsqu'une condition d'erreur est détectée (soit par le programme, soit par le système d'exécution) on génère un objet particulier (une exception) qui, éventuellement, contiendra toutes les informations nécessaires au diagnostic. Cet objet est ensuite "jeté" dans la pile (il dérive d'ailleurs de la classe `Throwable`), c'est à dire que l'exécution sort de son déroulement normal et remonte la pile jusqu'à ce qu'elle rencontre un code particulier chargé de récupérer l'objet diagnostic.



Les classes Java sont susceptibles de génerer toute sortes d'exceptions pour rendre compte des incidents spécifiques à telle ou telle action. Les programmeurs peuvent définir leurs propres exceptions, les déclencher et les récupérer.

Dans la plupart des cas quand un programmeur écrit un code qui déclenche une exception, le compilateur s'assure qu'il y a bien un code pour récupérer l'incident. Dans le cas contraire on remonte jusqu'au sommet de la pile (liée au processus courant) et l'exception est traitée par un mécanisme par défaut (qui écrit un diagnostic sur la sortie d'erreur standard).

Exemple de récupération d'exception

Soit le programme:

```
public class Addition{
        public static void main (String[] args) {
                int somme = 0 ;
                for (int ix = 0 ; ix < args.length; ix++) {
                        somme += Integer.parseInt(args[ix]) ;
                }
                System.out.println("somme=" +somme);
        }// main
}
```

et son exécution avec des arguments erronnés:

```
java Addition 1 2 douze 1OO
Exception in thread "main" java.lang.NumberFormatException: douze
        at java.lang.Integer.parseInt(Integer.java:409)
        at java.lang.Integer.parseInt(Integer.java:458)
        at Addition.main(Addition.java:7)
```

Ici la machine virtuelle a été arrétée au premier argument erroné. Si l'on voulait récupérer les erreurs pour les traiter d'une autre manière il faudrait mettre en place un mécanisme adapté.

En standard sur le SDK java 2 les exécutions se font avec un compilateur à la volée. Selon les exécuteurs il faut parfois prendre des dispositions particulières pour obtenir des informations sur les numéros de ligne des méthodes dans la pile par exemple en version 1.2 exécuter java avec une variable d'environnement ;
`JAVA_COMPILER=NONE`

Exemple de récupération d'exception

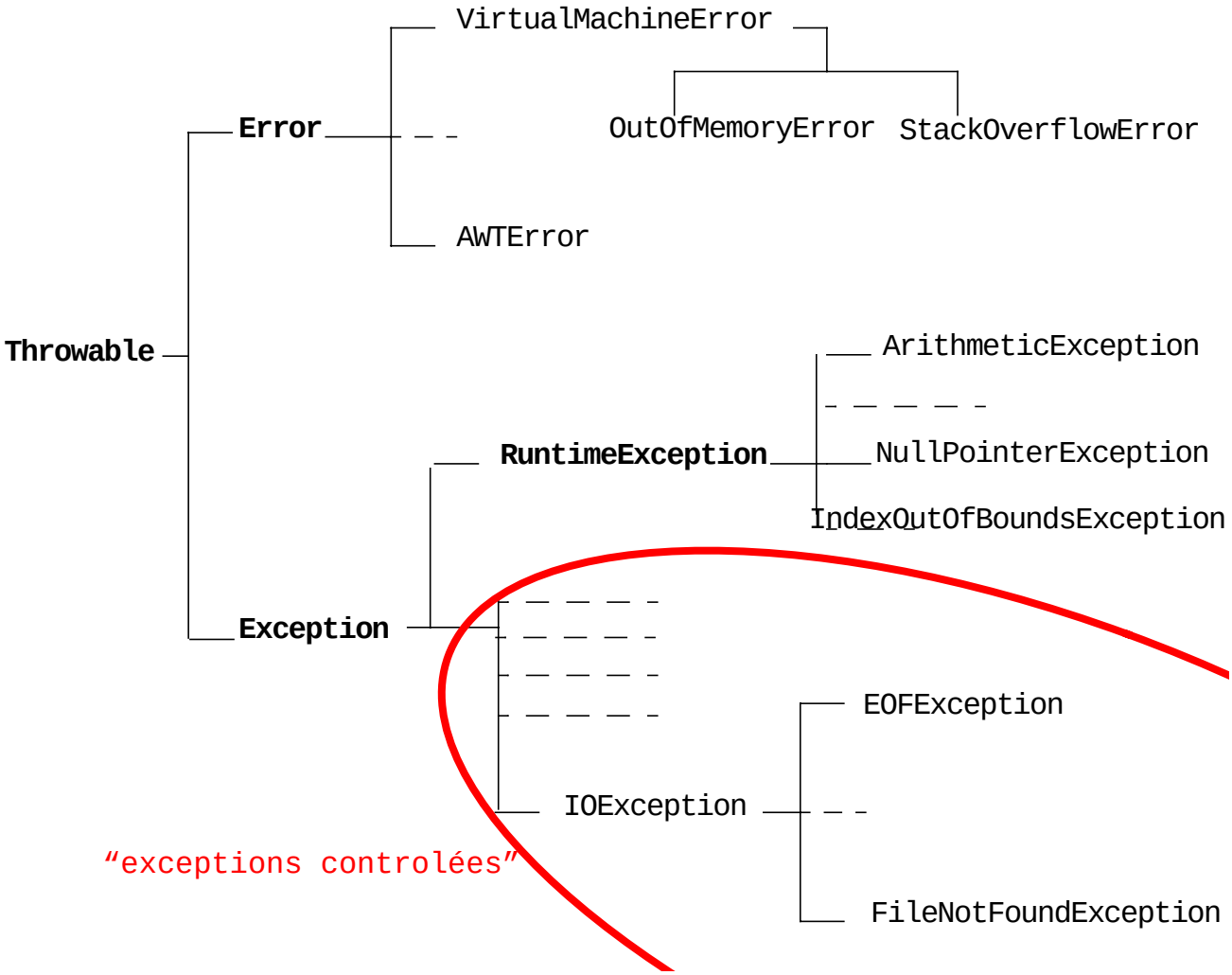
La recupération d'exception passe par la mise en place de blocs `try-catch`;

```
public class Addition{
        public static void main (String[] args) {
                int somme = 0 ;
                boolean ok = true ;
                for (int ix = 0 ; ix < args.length; ix++) {
                        try {
                                somme += Integer.parseInt(
                                        args[ix]) ;
                        } catch (NumberFormatException exc){
                                ok = false ;
                                System.err.println(exc);
                        }
                }
                if (ok) {
                        System.out.println("somme=" +somme);
                }
        }// main
}
```

Le résultat de l'exécution (ici 100 est écrit avec deux lettres 'o' majuscules):

```
java Addition 1 2 douze 1OO
java.lang.NumberFormatException: douze
java.lang.NumberFormatException: 1OO
```

Hiérarchie, exceptions courantes



La classe `Throwable` ne doit pas être utilisée:

- `Error` indique un problème sévère de JVM dont la récupération peut s'avérer fort difficile voire impossible.
- `RuntimeException` indique un problème d'exécution souvent imprévisible (et souvent détecté par l'exécuteur lui-même). Sauf à faire un code bavard et sur-protégé l'implantation du code de récupération de ces exceptions doit être soigneusement justifiée.
- Les autres exceptions doivent impérativement être récupérées et le compilateur lui-même va le contrôler.

Hiérarchie, exemples d'exceptions standard

- **`RuntimeException`** :
  - `ArithmeticException` : ex. division entière par zéro.
  - `NullPointerException` : tentative de déréférencement sur une variable objet dont la valeur est `null`.
  - `ClassCastException` : transtypage impossible au *run-time.*
  - `ArrayIndexOutOfBoundsException` : dans un tableau tentative d'accès à un index qui n'existe pas.
  - `SecurityException` : lors d'une exécution sous le contrôle d'un `SecurityManager` (par ex. dans une Applet), l'`AccessControler` est susceptible de refuser l'accès à une ressource locale.
- **exceptions contrôlées**:
  - `ClassNotFoundException`: au *run-time* la demande de chargement dynamique d'une classe échoue.
  - `InterruptedException`: interruption d'un Thread
  - `java.io.FileNotFoundException`: fichier non trouvé
  - `java.net.MalformedURLException`: erreur dans la syntaxe de description d'une U.R.L.
  - `java.beans.PropertyVetoException`: la modification d'une propriété est refusée par une instance chargée de donner son avis.
  - ...

Définir une nouvelle exception

On peut créer une nouvelle exception par dérivation de la classe Exception.

```
public class ExceptionAffectationHoraire extends Exception {
        public Executant exécutant;
        public Tache tâche;

        public ExceptionAffectationHoraire(String motif,
                            Executant exec,
                            Tache tâche) {
                super(motif);
                this.exécutant = exec;
                this.tâche = tâche ;
        }

        // utiliser getMessage() de la classse Exception
        // pour récupérer le motif de l'erreur

        // par contre : s'agissant d'un objet compte-rendu
        // la mise en place d'accesseurs pour les champs
        // est généralement superfétatoire
}
```

Déclencher une exception

Pour écrire du code qui génère une exception il faut :

- Une instance d'un objet dérivé de Exception
- Utiliser la directive **`throw`** pour provoquer la remontée de cette instance dans la pile des appels.

Exemple :

```
if (totalHoraire > TOTAL_HORAIRE_MAX) {
        throw new ExceptionAffectationHoraire(
                "dépassement maximum légal " ,
                exécutantCourant, tâcheCourante) ;
}
```

D'un point de vue de l'organisation du code, on notera que, si le déclenchement d'un exception constitue un déroutement de l'exécution normale, on a par contre un effet de modularité bien meilleur que celui d'autres mécanismes. Au contraire du déroutement par `goto`, tel qu'il existe dans d'autres langages, on n'a pas besoin de savoir au moment du développement du code **où** et **comment** l'erreur sera récupérée.

Chaînage d'exceptions

Une exception peut être définie comme contenant une autre exception:

```
public class ExceptionSauvegarde extends java.io.IOException {

        public Exception detail;// pas nécessaire en 1.4

        public ExceptionSauvegarde(String s, Exception ex) {
                super(s); // en 1.4: super(s,ex)
                detail = ex;// pas nécessaire en 1.4
        }

        public String getMessage() {
                if (detail == null){return super.getMessage();}
                else {
                 return super.getMessage() + "; " + detail;
                }
        }
        // en 1.4 voir getCause() ;
}
```

Exemple de déclenchement:

```
import java.io.* ;
....
try {
        ....
} catch (IOException exc) {
        .... // code éventuel
        throw new ExceptionSauvegarde(
                opérationCourante + " non réalisée" , exc) ;
}
```

Blocs try-catch

Pour mettre en place du code protégé :

- Mettre le code susceptible de propager une ou plusieurs exceptions dans un bloc **`try`**
- Adjoindre au bloc try un ou plusieurs blocs **`catch`**. Chaque bloc catch gérant une exception particulière ou un ensemble d'exceptions représentées par un ancêtre commun (l'ordre des blocs est important en cas de recouvrement des types d'exceptions).

```
try {
          ....
          // code susceptible de propager des exceptions
          ...
} catch (ExceptionSauvegarde exc) {
          ...
          // code spécifique à cette exception
          ...
} catch (IOException exc) {
          ...
          // code pour les autres erreurs entrées/sorties
          ...
} catch (Exception exc) {
          ...
          // code pour les autres cas
          ...
}
```

Point intermédiaire: exceptions -déclenchement récupération-

Une exception est un objet de diagnostic qui est créé et “jeté” dans la pile par un mécanisme de déclenchement d’erreur.

Ce mécanisme est lié aux vérifications de l’exécuteur ou à un code explicitement écrit par un programmeur:

```
if( conditionErronée) {
        throw new ObjetDiagnosticException(param) ;
}
```

D’autre parties du code sont chargées de la récupération : blocs “catch” associés au bloc “try” qui regroupe les instructions sous contrôle.

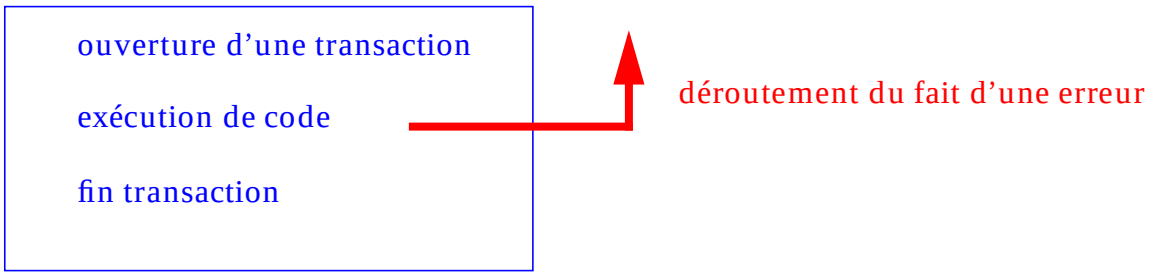
Mini-exercice:

- Dans un code “TestURL” faire un main qui analyse son (ou ses) paramètre(s) pour tester si c’est une chaîne de caractère désignant une URL valide (comme “http://java.sun.com”).
  Utiliser le constructeur de java.net.URL prenant une chaine en paramètre : ce constructeur est susceptible de déclencher une Exception de type MalformedURLException.

- Exercice complémentaire (pour les très rapides ou ceux qui révisent): utiliser aussi la méthode “openStream()” pour tester si l’URL est accessible en lecture.

Bloc finally

Du fait du déroulement particulier des exceptions dans un bloc try certaines instructions du bloc peuvent ne pas être exécutées (au détriment du bon fonctionnement du programme) :



Dans ce cas la "fin de transaction" ne sera jamais exécutée et l'environnement peut se trouver dans un état incohérent.

Un bloc **`finally`** accolé à un bloc `try` englobe du code qui sera toujours exécuté quelque soit la façon dont on sort du bloc try : exception, return, déroutement étiqueté, ... (sauf si on invoque `System.exit(...)`).

```
try {
        ... //ouverture transaction
        ...// exécution de code
        ...
} finally {
        /* fermeture transaction (code compatible avec un
         * échec de l'ouverture !)
         */

}
```

Le plus souvent on aura :

```
try {
        ...
} catch ( ...) {
        ...
} finally {
        .. // code exécuté dans tous les cas
        ...// même si le catch refait un "throw"
}
```

Règle : déclarer ou traiter

Pour encourager l'écriture de code robuste le langage Java exige que pour tout emploi de code susceptible de lever une Exception qui ne soit pas une erreur d'exécution liée au *run-time* (classes `Error` et classes `RuntimeException`), il soit garanti qu'il y ait du code de traitement récupérant ces exceptions.

Si un programmeur utilise un code susceptible de déclencher une exception le compilateur vérifie que :

- Dans la méthode courante il y a un bloc try-catch qui englobe le code critique et qui sait agir sur les exceptions à traiter:

```
public void methodeF() {
          ....
          try {
                    ... // code susceptible de provoquer
                    // Une exception de type UneException
          } catch (UneException exc) {
                    ....
          }
}
```

- Si un tel bloc n'existe pas alors la méthode courante déclare propager ces exceptions :

```
public void methodeF() throws UneException {
          // code susceptible de provoquer une Exception
}
```

De cette manière tout code utlisant la méthode saura qu'il doit à son tour traiter ou propager les exceptions impliquées.

A la suite du mot-clef `throws` vient une liste de toutes les exceptions susceptibles d'être propagées par la méthode :

```
public Taches[] tachesDuMois()
          throws ExceptionAffectationHoraire,IOException{
```

Cette liste fait partie des caractéristiques significatives de la méthode et apparaît dans sa documentation.

Récapitulation: modèle des méthodes

Il nous faut maintenant modifier et préciser notre précédente définition des caractéristiques significatives d'une méthode :

modif.+accès type_résultat **nom_méthode** (liste_paramètres) liste_exceptions

```
public double annuité ( double montantPrêt,
                        int durée ,
                        double taux )
                                  throws Exception1, Exception2 {
          // code avec un ou plusieurs return valeur double
}
```

Une méthode comporte donc:

- Un nom : plusieurs méthodes peuvent avoir le même nom et des paramètres différents (méthodes *surchargées*)
- Des paramètres éventuels: caractérisent la *signature* de la méthode qui permet de distinguer les méthodes surchargées entre elles.
- Un type de résultat (ou void) : lorsqu'une méthode d'instance est redéfinie dans une sous-classe le type du résultat doit être le même.
- Un modificateur d'accès . Lors d'une redéfinition de méthode d'instance dans une sous-classe le contrat de type ne peut pas être aggravé: la nouvelle méthode ne peut pas être plus "privée" que la méthode initiale. D'autres modificateurs complémentaires (`final`, `native`, `abstract`, etc.) sont possibles.
- Une liste éventuelle d'exceptions propagées : lors d'une redéfinition de méthode d'instance dans une sous-classe le contrat de type ne peut pas être aggravé: la nouvelle méthode ne peut pas propager "plus" d'exceptions controlées que la méthode initiale. Tout au plus peut-elle déclarer propager des sous-classes de ces exceptions controlées.

Compléments

Les compléments techniques suivants constituent une annexe de référence

Les compte-rendus d'erreur

Que faire dans un bloc catch? Si on doit faire remonter un message à l'utilisateur il faut soigneusement choisir où placer cette génération de message et la nature même de ce message. A titre d'exemple faire un
`System.err.println(exceptionCourante) ;`
dans un composant ne sera pas d'une grande utilité quand ce composant sera réutilisé dans une interface graphique!

De la même manière les méthodes `printStackTrace` (qui impriment l'état de la pile courante) sont utiles pour le debug, mais ne doivent pas être utilisées de manière indiscriminée au sein d'une application destinée à un utilisateur non-informaticien (voir logging API dans java 1.4)

Initialisations d'instance

Imaginons une classe qui ait une déclaration de champ d'instance avec initialisation explicite :

```
import java.net.* ;
public class NeSeCompilePas {
        URL java = new URL(“http://java.sun.com”) ;
```

Ce code ne se compile pas car le constructeur de `URL` propage une exception qui doit être controlée.

Il est possible de controler cette exception soit dans une initialisation située dans un constructeur soit dans un bloc d'initialisation d'instance:

```
public class SeCompile {
        URL java ;
        {try {//debut bloc d'instance -analogue bloc static-
         java = new URL("http://java.sun.com") ;
                  } catch(MalformedURLException exc) {}
        }// fin bloc (plusieurs blocs possibles)
```

Ce dispositif, rare, n'est pratiquement utilisé que dans les classes anonymes (voir chapitre sur les classes locales)

Corrigés des mini-exercices

TestURL.java :

```
// devrait etre dans un package!
import java.net.* ;
import java.io.* ;

public class TestURL {
   public static void main(String[] tb) {

      for(int ix = 0 ; ix <tb.length ;ix++) {
         try {
            URL url = new URL(tb[ix]) ;
            url.openStream() ;
         } catch(IOException exc) {
            System.err.println(exc) ;
            continue ;
         }
         System.out.println("Ok : " + tb[ix]) ;
      }
   }
}
```

Exercices :

Exercice [**] :

Reprendre les classes du projet “oubangui.com”, modifier les méthodes de manières à ce que leur appel dans des conditions erronées provoque une exception

Bien entendu seules quelques méthodes importantes sont concernées, pour la gestion du `Panier` on modifiera la réalisation de la liste des livres au lieu d’un tableau de `Livre` on utilisera un vecteur (type `java.util.ArrayList`) : l’ajout d’un livre se fera par la méthode `add(Object)` et la consultation se fera par `Object get(index)`.

Si vous avez du temps vous modifierez également les valeurs pour utiliser la classe `java.math.BigDecimal`.

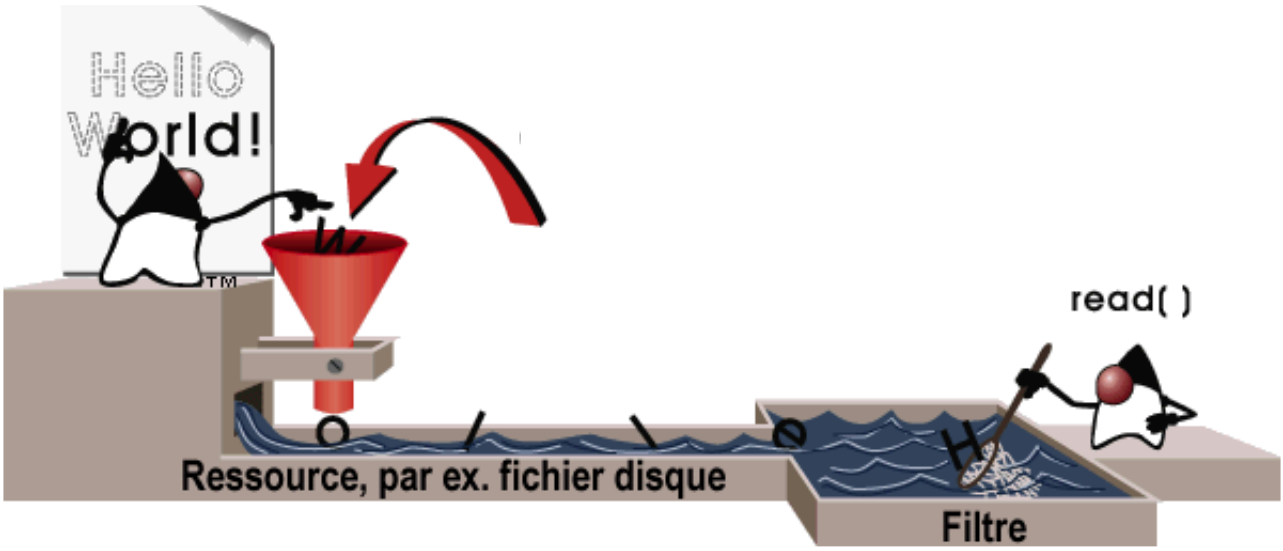
Introduction aux entrées/sorties 9

Points essentiels

Java dispose d'un modèle d'entrée/sortie portable entre plate-formes:

- Les *flots* (*streams*) sont le concept de base des E/S
- Certains types de flots sont associés à des *ressources* (fichier sur disque, buffer en mémoire, ...)
- Certains autres sont des *filtres* c'est à dire des flots qui transforment la manière dont opère l'entrée/sortie.
- Les E/S s'opèrent en combinant divers filtres sur une ressource.

Entrées/Sorties portables, notion de flot (stream)



Notion de flot (stream)

Selon les systèmes d'exploitation les notions fondamentales d'entrées/sorties varient considérablement. Java s'est trouvé dans l'obligation de proposer un concept unificateur qui soit adaptable à des situations très différentes.

Au travers du concept de *flot* Java présente les échanges d'E/S comme une succession continue de données. Un producteur qui écrit *dans* un flot écrit des données les unes à la suite des autres. Un consommateur qui lit ces données *depuis* un flot, les lit successivement les unes à la suite des autres.

Dans cette idée unificatrice on peut englober des modalités de réalisation très différentes:

- des flots qui diffèrent par l'origine effective des données. On pourra, par exemple, "lire" des données *depuis* des fichiers, une ligne de communication, un buffer en mémoire,...
  C'est la nature de la *ressource* qui différencie ces flots.

- des flots qui diffèrent par la manière dont ils opèrent ces lectures/écritures. On pourra, par exemple, avoir des lectures qui opèrent avec bufferisation, compression, cryptage, traduction de types de données, etc..
  Ces flots sont appelés des *filtres (filter)* et peuvent être combinés entre eux pour lire/écrire de/vers des *ressources (node streams)*.

Flots d'octets (InputStream, OutputStream)

La définition fondamentale des flots est de décrire un dispositif pour lire / écrire des octets (`byte`).

Ces classes sont donc munies de méthodes de bas niveau comme :

```
java.io.InputStream
         int read() throws IOException
          // lit en fait un byte dans un int
          // + d'autres read(...) de bas niveau

java.io.OutputStream
         void write(int octet) throws IOException
          // écriture d'un byte passé comme int
          // + autres write(...) de bas niveau
```

Les deux classes sont munies d'une méthode de fermeture

```
         void close() throws IOException
```

Flots de caractères (Reader, Writer)

Dans la mesure où Java utilise de manière interne des caractères UNICODE codés sur 16 bits. on a été amené à définir d'autres types fondamentaux de flots adaptés aux caractères. Ce sont les **`Reader`**s et les **`Writer`**s.

```
java.io.Reader
         int read() throws IOException
          // lit en fait un char dans un int
          // + d'autres read(...) de bas niveau

java.io.Writer
         void write(int carac) throws IOException
          // écriture d'un caractère passé comme int
          // + autres write(...) de bas niveau
          // dont write(string s)
```

Bien entendu ces classes disposent également de `close()`.

Typologie par “ressources”

En prenant l’exemple des seuls flots d’entrées comparons les `InputStream`s et les `Reader`s

catégorie	**`InputStream`**	**`Reader`**
lecture dans tableau mémoire	`ByteArrayInputStream`	`CharArrayReader` `StringReader`
mécanisme producteur/consommateur entre Threads	PipedInputStream	PipedReader
fichiers	FileInputStream	FileReader **InputStreamReader**
flots fabriqués par d’autres classes	java.lang.Process getInputStream() java.net.Socket getInputStream() java.net.URL openStream() ...	

Une classe permet de passer du domaine des `InputStream`s à celui des `Reader`s : **`InputStreamReader`**

Conversions octets-caractères

On utilise les caractères UNICODE à “l’intérieur” du monde Java, mais de nombreuses ressources texte n’utilisent pas ce mode de codage. On a ainsi des fichiers “texte” qui peuvent avoir été fabriqués par des systèmes divers qui emploient des jeux de caractères différents (le plus souvent avec des caractères codés sur 8 bits).

Lorsqu’on échange du texte entre le “monde” Java et le monde extérieur il est essentiel de savoir quel est le mode de codage de texte adapté à la plateforme cible.

Deux classes `InputStreamReader` et `OutputStreamWriter` permettent de passer d’un flot d’octets à un flot de caractères on opérant les conversions appropriées.

```
try {
      FileInputStream fis = new FileInputStream("fichier.txt") ;
      Reader ir = new InputStreamReader(fis,"ISO-8859-1") ;
         .....
} catch (UnsupportedEncodingException exc) {....
} catch (FileNotFoundException exc) {....
}
```

Le second argument du constructeur indique le mode de codage (“ISO-8859-1” ou “ISO-8859-15” dans la plupart des pays européens, “Cp1252” sous Windows). La liste des codages acceptés se trouve dans la documentation sous `docs/guide/internat/encoding.doc.html` - voir aussi l’outil `native2ascii` ou la documentation 1.4 de `java.nio.charset`- On notera qu’il existe un codage nommé UTF8 qui permet de fabriquer un flot de caractères UNICODE codés sur 8 bits.

Il existe également des constructeurs pour ces deux classes dans lesquels on ne précise pas le mode de codage: c’est le mode de codage de la plate-forme locale qui est pris par défaut.

Filtres

Les *filtres* sont des dispositifs qui transforment la manière dont un flot opère: bufferisation, conversion des octets en données Java, compressions, etc.

- Les filtres sont des flots dont les constructeurs utilisent toujours un autre flot:

```
// exemple de déclarations des constructeurs
public BufferedInputStream(InputStream in) ...
public DataInputStream(InputStream in)...
```

- On utilise les filtres en les associant avec un flot passé en paramètre du constructeur. Ce flot peut lui même être un autre filtre, on combine alors les comportements:

```
BufferedInputStream bis = new BufferedInputStream(
                  new FileInputStream (nomFichier)) ;

DataOutputStream dos = new DataOuputStream(
                  new BufferedOutputStream (
                            new FileOutputStream (nomFic))) ;
```

- Chaque filtre dispose de méthodes spécifiques, certaines méthodes ont la propriété de remonter la chaîne des flots :

```
dos.writeInt(235);
dos.writeDouble(Math.PI);
dos.writeUTF(”une chaîne accentuée”) ; // chaîne écrite en UTF8
dos.flush() ; //transmis au BufferedOutputStream
...
dos.close() ; // ferme toute la chaîne
```

Filtres courants

InputStream/OutputStream

- `Buffered*` : la bufferisation permet de réaliser des E/S plus efficaces (en particulier sur le réseau).
- `Data*` : permet de lire/écrire des données primitives Java
- `Object*` : permet de lire/écrire des instances d'objets. Ce type de flot est très important (mais sort du périmètre de ce cours).
- `PushBackInputStream`: permet de "remettre" dans le flot des données déjà lues.
- `SequenceInputStream`: permet de considérer une liste ordonnée de flots comme un seul flot.

Reader/Writer

- `Buffered*` : bufferisation pour des caractères. Permet de lire un texte ligne à ligne.
- LineNumberReader : BufferedReader avec numérotation des lignes.
- `PrintWriter`: pour disposer des méthodes print/println (voir aussi `PrintStream`)
- `StreamTokenizer` : permet de faire de l'analyse syntaxique de texte.

Dans d'autre packages on trouvera des filtres spécialisés : java.util.zip.ZipInputStream, java.security.DigestInputStream,...

Il existe également un dispositif d'entrée/sortie qui ne s'inscrit pas dans la logique des Streams : `RandomAccessFile` (fichier à accès direct)

Compléments

Les compléments techniques suivants constituent une annexe de référence

RandomAccessFile

Exemple d'ouverture :

```
RandomAcessFile raf ;
try {
        raf = new RandomAccessFile("stock.raf","rw") ;
catch(Exception exc) {
        ....
}
```

Le premier argument donne le nom du fichier impliqué, le second indique le mode d'accès ("r" pour lecture seule, "rw" pour lecture/écriture -dans ce cas si le fichier n'existe pas il est créé-).

Les exceptions susceptibles d'être levées concernent le mode (chaîne incorrecte), la désignation ou les droits d'accès au fichier .

La lecture et écriture de données peut s'opérer sur des données primitives scalaires (int, byte, double, char,...) et sur des chaînes de caractères (capacités analogues aux `DataStream`) -toutefois dans ce dernier cas il devient difficile de controler correctement les accès directs en fonction d'une taille prédéterminée-.

Pour gérer l'accès direct :

```
void seek (long position) ; //positionner dans le fichier
                    // commence en origine zero

long getFilePointer() ; // position actuelle

long length() ; // position fin de fichier
```

Exercices :

*Exercice **:

Ecrire une classe qui permette de lire un fichier texte ligne à ligne. On passera au constructeur un nom de fichier, le test affichera d'abord une ligne, puis ensuite toute les lignes du fichier.

*Exercice *** (suite du précédent):

Pour tester des portions d'une application il est souvent très utile de mettre en place un système de fichier testeur. Par exemple: vous avez besoin pour tester votre application d'un certain nombre de données; au lieu de les saisir avec une interface graphique, vous écrivez un fichier texte simple qui décrit ces données et vous les lisez.

Vous allez mettre en place un tel système de test pour initialiser une liste de `Livre`. Comme il s'agit d'un exercice nous allons adopter un formalisme très simple :

- chaque ligne du fichier texte contient une information. Pour séparer les différents livres on laisse une ligne vide.
- dans cet exemple on fera en sorte que seul l'ordre respectif des informations soit important: par ex. la première ligne permettra de lire le numero du livre, la seconde le titre, la troisième l'auteur, etc.

Exemple de fichier de test :

```
0-7821-2700-2
the java2 certification study guide
Simon Roberts
450
10
un livre sur la certification Java
nécessaire pour la préparation

1-56592-262-X
Java in a nutshell
David Flanagan
288
25
un livre d'introduction à java
```

Annexe: Les entrées/sorties

Le chapître sur les entrées/sorties est un introduction technique. Si vous souhaitez maitriser cet aspect de la programmation Java il y a des investissements à faire:

- maitrise des différents filtres et de leur combinaison
- apprentissage des entrées-sorties objet
- utilisation des entrées/sorties sur le réseau et dans des archives de ressources.

Ces points sont développés dans nos cours avancés.

I.H.M. portables : AWT 10

Points essentiels

- AWT permet des mises en page indépendantes de la plate-forme
- Les composants d'interaction graphiques sont disposés à l'intérieur de composants particuliers : les Containers.
- Cette disposition se fait automatiquement et est contrôlée par un gestionnaire de disposition associé au Container.
- Etude de quelques gestionnaires de disposition standard: FlowLayout, BorderLayout, GridLayout.
- Introduction au système graphique de bas niveau (partie facultatifve

Le package AWT

Le package AWT fournit les composants fondamentaux pour réaliser des interactions graphiques (I.H.M. *interactions Homme-Machine*). Les logiciels écrits avec AWT sont portables: le même code s'exécutera sans modifications sur des machines différentes et des systèmes de fenêtrages différents.

La manière dont cette portabilité est assurée doit être mentionnée: pour un composant graphique comme `Button`, il existe dans la librairie locale de la JVM une classe de correspondance (*peer class*) qui s'appuie sur les caractéristiques du système de fenêtrage local. Ainsi quand on utilisera un `Button` Java on obtiendra un bouton Motif sous Motif, un bouton Windows sous Win*, un bouton Mac sur Apple, etc. Une implantation locale de la classe `Toolkit` assure la liaison entre les composants Java et les composants "natifs".

Une autre librairie graphique, `javax.swing`, s'appuie essentiellement sur des composants pur Java et offre donc une plus grande variété puisqu'on n'a pas besoin de disposer d'un composant correspondant existant sur tous les sytèmes de fenêtrage.

Parmi les classes du package AWT on trouve:

- des composants standard (`Window, Panel, Button, Label`,..) et des classes associées (`Menu, CheckboxGroup`,..)
- des gestionnaires de disposition (`LayoutManager,..`) qui permettent d'agencer des composants.
- des classes liées au graphique de bas niveau : `Graphics, Graphics2D, Color, Font, Image`,…
- des événements et gestionnaires de tâches graphiques: `AWTEvent, MediaTracker, PrintJob`,…
- des "structures de données" : `Point, Rectangle, Dimension`,…

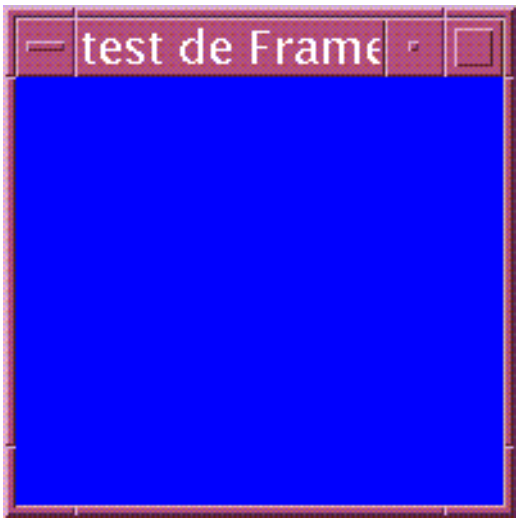
Composants et Containers

Les interfaces graphiques sont bâties à partir de composants comme des boutons (`Button`), des champs de saisie (`TextField`), des étiquettes (`Label`), etc. Ces composants dérivent de la classe `Component`.

Certains `Component`s sont des `Container`s c'est à dire des composants qui "contiennent" d'autres composants (y compris, éventuellement, d'autres `Container`s). Ainsi à la "racine" d'une application autonome on utilisera une instance dérivée de la classe `Window` comme `Frame` et on disposera "dans" cette fenêtre des composants Label, TextField ainsi que des `Container`s comme `Panel` (panneau) qui contiendront à leur tour d'autres composants.

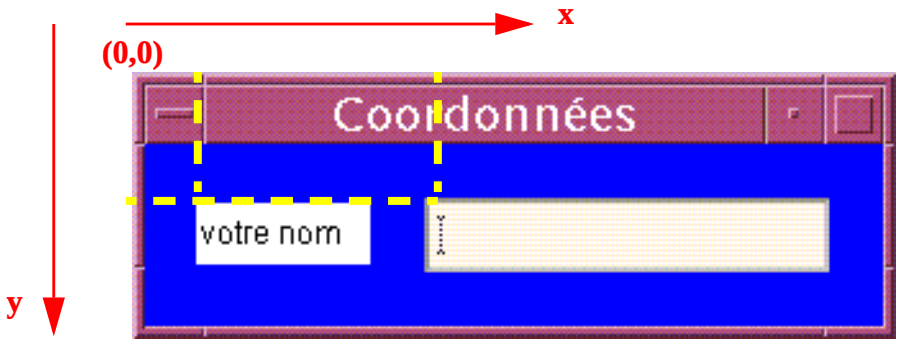
Exemple simple de mise en place de `Frame` sans contenu:

```
import java.awt.* ;

public class TestFrame{

   public static void main(String[] args) {
      Frame fenêtre = new Frame("test de Frame");
      fenêtre.setSize(200,200); // taille en points
      fenêtre.setBackground(Color.blue) ;
      fenêtre.show() ;
   }
}
```



Taille et position des composants: les gestionnaires de disposition

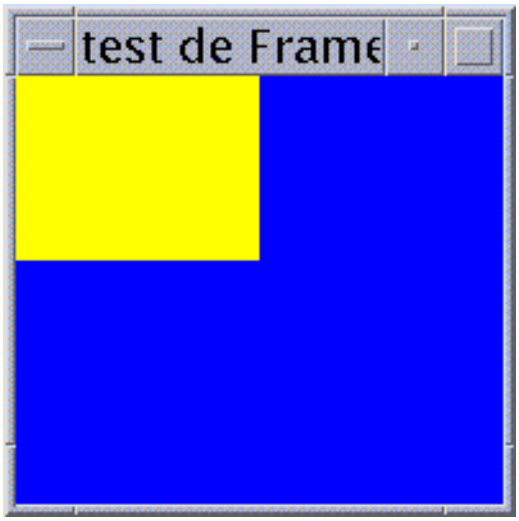
Quand on est “dans” un Container particulier on a affaire à un système de coordonnées qui part du coin supérieur gauche du container. Les composants peuvent se repérer par rapport à ce système de coordonnées exprimées en points.



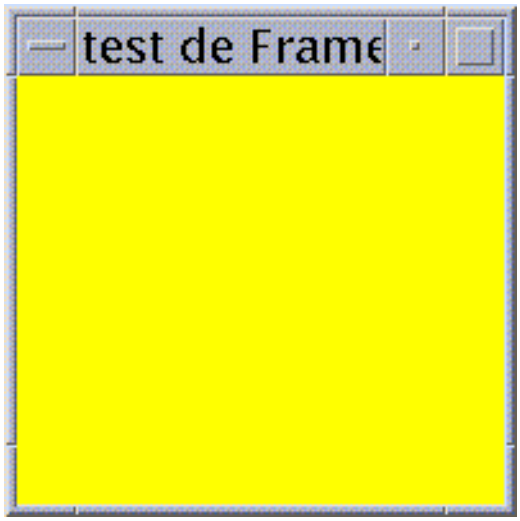
Bien qu’il soit possible de positionner les composants à l’intérieur d’un container en utilisant le système de coordonnées c’est une opération compliquée et surtout non portable. En Java on laisse généralement un gestionnaire de disposition (LayoutManager), associé au container, prendre en charge automatiquement le dimensionnement et la mise en place des composants.

Examinons le programme suivant:

```
import java.awt.* ;

public class TestAvecPanel{

   public static void main(String[] args) {
      Frame fenêtre = new Frame("test de Frame");
      fenêtre.setSize(200,200);
      fenêtre.setBackground(Color.blue) ;
      if (args.length != 0) { fenêtre.setLayout(null) ;}
      /* on rajoute un panneau */
      Panel panneau = new Panel();
      panneau.setBackground(Color.yellow);
      panneau.setSize(100,100);//ne sert pas si controlé par
Manager
      fenêtre.add(panneau);
      fenêtre.show() ;
   }
}
```



sans LayoutManager



avec LayoutManager

On a ici disposé un Panel à l'intérieur d'un Frame (`fenêtre.add(panneau)`). Ce Frame dispose par défaut d'un gestionnaire de disposition que l'on peut enlever (`fenêtre.setLayout(null)`). Dans ce cas on retrouve un panneau jaune de dimension 100,100 en coordonnées 0,0.

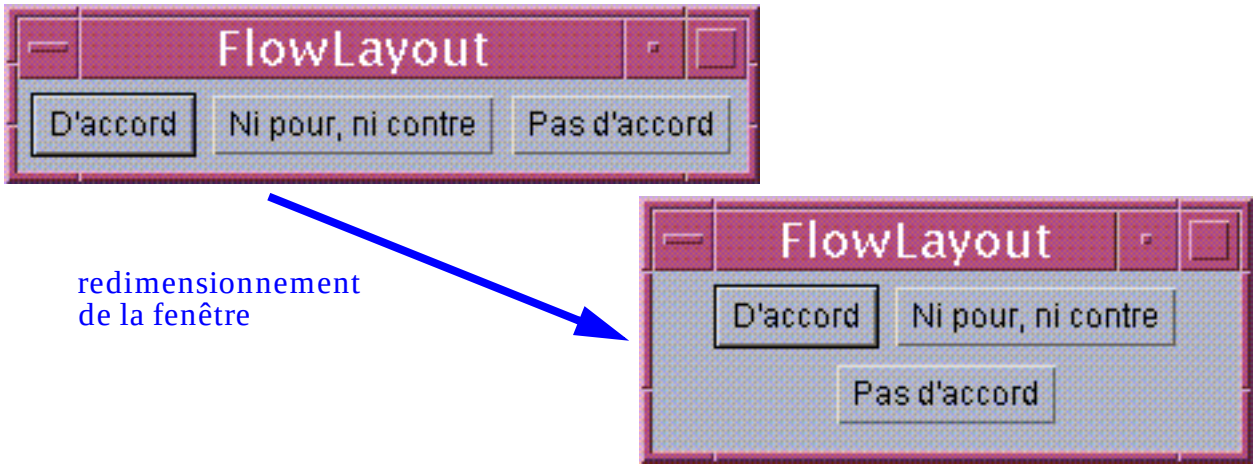
Si le gestionnaire de disposition du `Frame` reste actif, l'effet obtenu n'est pas celui qu'on pense car la dimension du panneau n'est plus sous contrôle direct et l'instruction `panneau.setSize(100,100)` est remise en cause par ce gestionnaire.

En fait cette prise de contrôle est avantageuse: en fonction d'une logique définie à l'avance le LayoutManager va disposer les composants et, éventuellement, changer leurs dimensions. Il suivra toutes les déformations du Container (demandées par la plate-forme, par l'utilisateur, par le Container englobant,...) et conservera la logique de la disposition.

Chaque type fondamental de Container a un LayoutManager par défaut qui peut être changé (par ex. par la méthode `setLayout(LayoutManager)`)

FlowLayout

Le gestionnaire de disposition FlowLayout positionne les composants “les uns à la suite des autres”. Au besoin il crée une nouvelle ligne pour positionner les composants qui ne tiennent pas dans la ligne courante.



A la différence de beaucoup d’autres gestionnaires de présentation une instance de FlowLayout ne modifie pas la taille des composants (tous les composants comportent une méthode `getPreferredSize()` qui est appelée par les gestionnaires de disposition pour demander quelle taille le composant voudrait avoir).

Exemple d’utilisation:

```
import java.awt.* ;

public class FlowFrame extends Frame{

   protected Component[] tb ;

   public FlowFrame(Component[] contenu) {
      super("FlowLayout") ;
      setLayout(new FlowLayout()) ;
      tb = contenu ;
      for ( int ix = 0 ; ix < tb.length; ix++) {
         add(tb[ix]);
      }
   }
```

```
    public void start() {//pas standard (juste une analogie)
       pack();
       show() ;
    }

    public static void main(String[] args) {
       Button[] boutons = { new Button ("D'accord"),
          new Button ("Ni pour, ni contre"),
          new Button ("Pas d'accord"),
       } ;
       FlowFrame fenêtre = new FlowFrame(boutons) ;
       fenêtre.start() ;
    }
}
```

- La fenêtre (Frame) ne disposant pas de gestionnaire FlowLayout par défaut on la dote d'une instance de FlowLayout.
  Notons que le `FlowLayout` est le gestionnaire par défaut associé aux containers de type Panel.

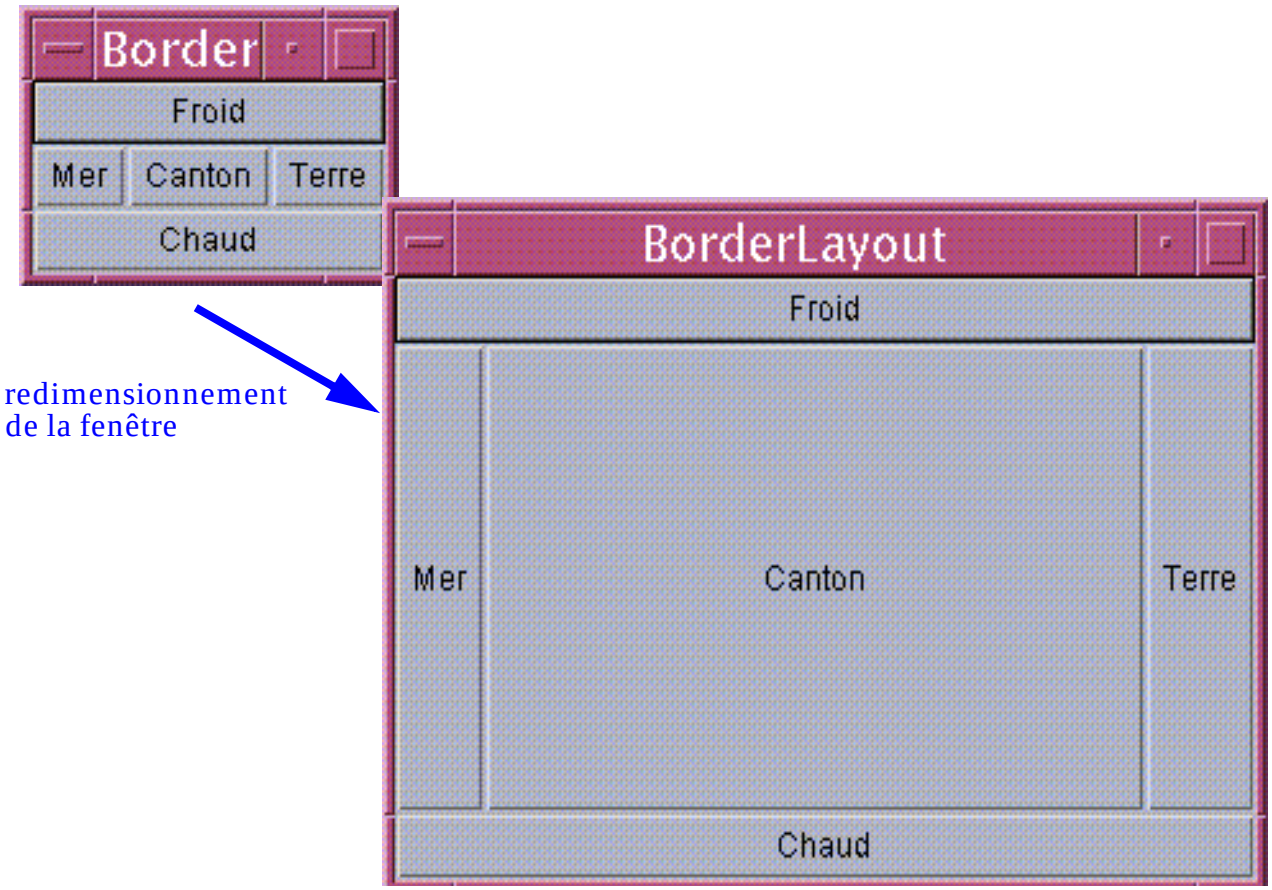
- Il existe plusieurs manières de construire un FLowLayout qui permet de faire varier l'alignement (à gauche, à droite, centré,..) ou de faire varier la taille des gouttières séparant les composants.

- Les composants sont ajoutés au Container en employant la méthode `add(Component)` , bien entendu l'ordre des `add` est important puiqu'il détermine l'ordre des composants dans la disposition.

Notons également dans l'exemple l'emploi de la méthode `pack()` sur la fenêtre: plutôt que de fixer une taille arbitraire on demande à la fenêtre de prendre la plus petite taille possible en fonction des composants qu'elle contient.

BorderLayout

`BorderLayout` divise son espace de travail en cinq zone nommées et les composants sont ajoutés explicitement à une zone donnée. On ne peut placer qu'un composant par zone, mais toutes les zones ne sont pas nécessairement occupées.



Exemple d'utilisation:

```
import java.awt.* ;

public class BorderTest {
   private Frame fen;
   private Button nord, sud, est, ouest, centre ;
```

```
public BorderTest() {
      // initialisations
      fen = new Frame("BorderLayout") ;
      nord = new Button("Froid") ;
      sud = new Button("Chaud") ;
      est = new Button("Terre") ;
      ouest = new Button("Mer") ;
      centre = new Button("Canton") ;

      // dispositions
      fen.add(nord, BorderLayout.NORTH) ;
      fen.add(sud, BorderLayout.SOUTH) ;
      fen.add(est, BorderLayout.EAST) ;
      fen.add(ouest, BorderLayout.WEST) ;
      fen.add(centre, BorderLayout.CENTER) ;
   }

   public void start() {// methode non-standard
      fen.pack() ;
      fen.show() ;
   }

   public static void main(String[] args) {
      BorderTest bt = new BorderTest() ;
      bt.start() ;
   }
}
```

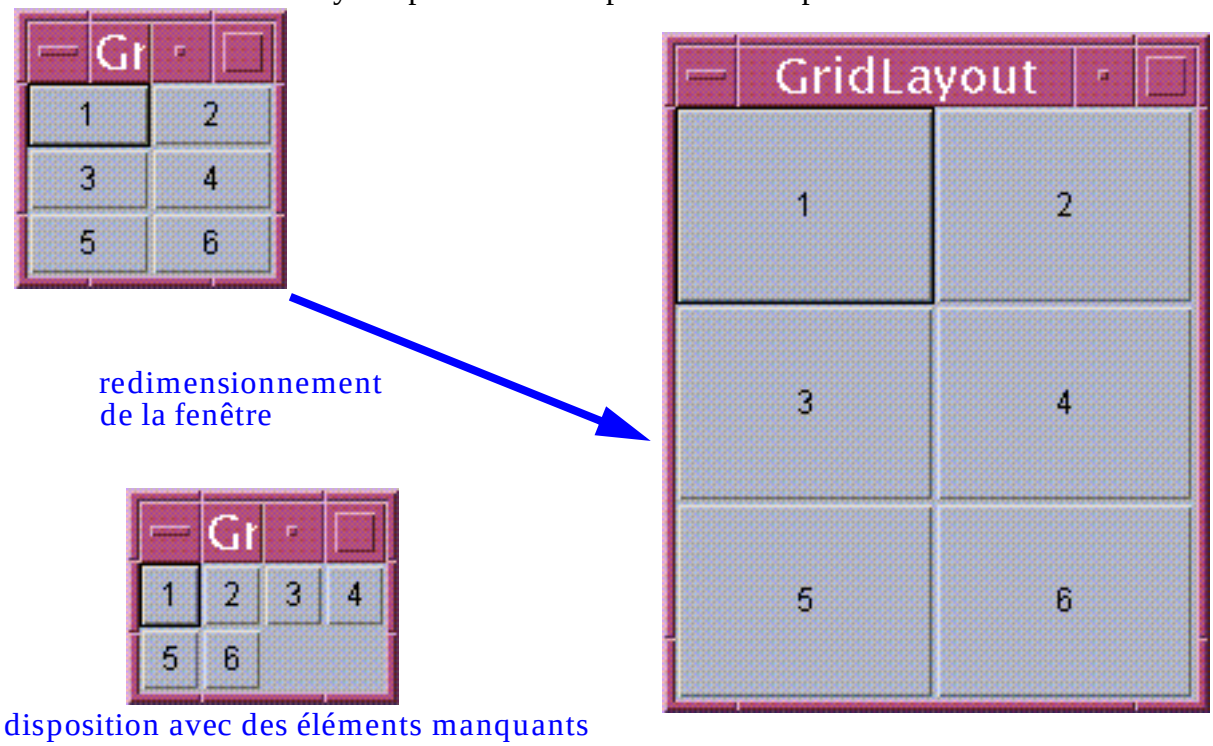
- Dans l'exemple `setLayout` n'a pas été utilisé car `BorderLayout` est le gestionnaire par défaut des containers de type `Window`.

- Les `add` sont qualifiés: il faut préciser dans quelle zone on met le composant (par défaut un `add` simple met le composant dans la zone centrale).

- En cas d'élargissement du container le gestionnaire respecte les hauteurs "préférées" des composants en NORTH et SOUTH, et les largeurs "préférées" des composants en EAST et WEST. La zone centrale tend à occuper toute la place restante dans les deux directions.

GridLayout

GridLayout permet de disposer les composants dans un tableau.



On peut créer un GridLayout en précisant les nombre de lignes et de colonnes (par exemple `new GridLayout(3,2)`).Toutes les lignes et les colonnes prennent la même dimension de façon à ce que les composants soient alignés.

Exemple d'utilisation:

```
import java.awt.* ;

public class ButtonPanel extends Panel{

   public ButtonPanel( int cols, String[] noms) {
      setLayout(new GridLayout(0, cols)) ;
      for (int ix = 0 ; ix < noms.length; ix ++) {
            add(new Button(noms[ix])) ;
      }
   }
}
-----------------------------------------------
import java.awt.* ;

public class TestButtonPanel {
   public static void main(String[] args) {
      Panel clavier = new ButtonPanel (
            Integer.parseInt(args[0]),
            new String[] { "1", "2", "3", "4", "5", "6", });
     Frame fenêtre = new Frame("GridLayout") ;
     fenêtre.add(clavier) ;
     fenêtre.pack() ;
     fenêtre.show() ;
   }
}
```

- Le constructeur de GridLayout permet de ne pas préciser une dimension (en la fixant à zéro) le gestionnaire calcule alors le nombre nécessaire de lignes ou de colonnes.
  Il existe également un constructeur permettant de fixer la taille des gouttières entre cellules.

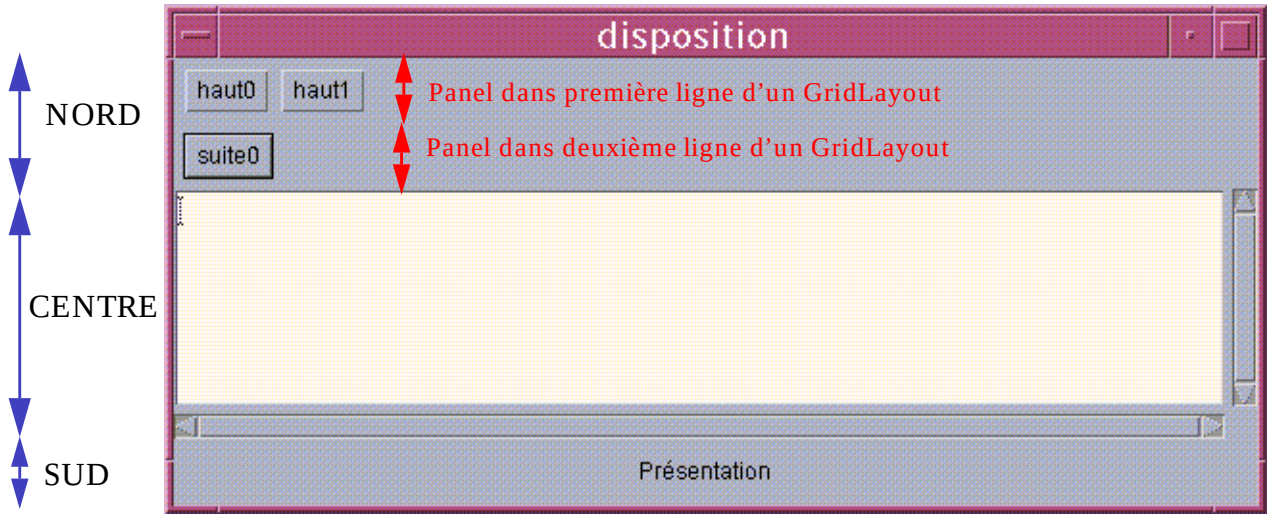
- On utilise ici un `add` non qualifié: l'ordre des appels est important pour déterminer la position dans la grille.

- Le gestionnaire GridLayout ne respecte pas les tailles "préférées" des composants et tend à occuper tout l'espace nécessaire.

On notera, par ailleurs, que, dans l'exemple, la disposition s'est faite dans un `Panel` qui a été lui même ajouté à un `Frame`. Comme le gestionnaire de `Frame` est un `BorderLayout` et qu'on a utilisé un `add` non qualifié, le `Panel` occupe toute la zone centrale et fait office de "fond" sur la fenêtre principale.

Combinaisons complexes

Les présentations sont, en général, obtenues en utilisant des `Panels` intermédiaires d'agencement. Ce type d'imbrication est essentiel pour des présentation complexes.



Exemples de codes correspondant:

```
import java.awt.* ; // pour tous: cité pour mémoire
-------------------------------------------------
public class Lineaire extends Panel {
   // serait mieux avec GridBagLayout
   public Lineaire(LayoutManager lm) {
      super(lm) ;
   }

   public void add(Component[] tb) {
         for (int ix = 0 ; ix < tb.length; ix ++ ) {
            add(tb[ix]) ;
         }
   }
}
-------------------------------------------------
public class Colonne extends Lineaire {
   public Colonne() {
      super(new GridLayout(0,1)) ;
   }
}
```

```
public class Barre extends Lineaire {
   public Barre() {
      super(new FlowLayout(FlowLayout.LEADING)) ;
   }
}
-------------------------------------------------------
public class Presentation extends Panel {
   public Presentation(Component[] enHaut,
                              Component centre,Component enBas){
      super(new BorderLayout());

      Colonne haut = new Colonne() ;
      haut.add(enHaut);

      add(haut, BorderLayout.NORTH) ;
      add(centre, BorderLayout.CENTER) ;
      add(enBas, BorderLayout.SOUTH) ;
   }
}
-------------------------------------------------------
public class TestPresentation  {
   public static void main(String[] args) {
      Barre barre1 = new Barre() ;
      Barre barre2 = new Barre() ;
      barre1.add( new Button[]
                  {new Button("haut0"), new Button("haut1")});
      barre2.add( new Button[]
                  {new Button("suite0")});

      Presentation pres = new Presentation(
                  new Barre[] {barre1, barre2},
                  new TextArea(7,77) ,
                  new Label("Présentation", Label.CENTER)) ;

      Frame frame = new Frame("disposition") ;
      frame.add(pres) ;
      frame.pack() ;
      frame.show() ;
   }
}
```

Autres gestionnaires de disposition

Dans le package AWT on trouve deux autres gestionnaires de disposition standard:

- `CardLayout` : permet de disposer des composants en pile les uns au dessus des autres. Un seul composant est visible et, par programme, on peut remettre un des composant sur le “dessus” de la pile.

- `GridBagLayout` : est un gestionnaire très utile, très souple et très complexe. Il permet de disposer des composants dans une grille. Les lignes et les colonnes ne sont pas toutes de même taille et l’occupation des cellules peut être finement paramétrée (respect des tailles “préférées”, position dans la cellule, occupation de plusieurs cellules, etc.).

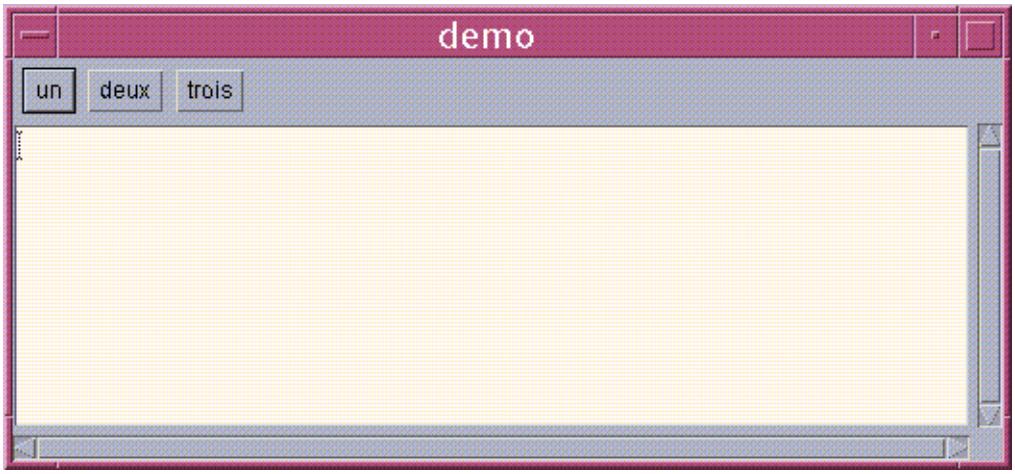
L’étude de ces gestionnaires sort du périmètre de ce cours.

Point intermédiaire : les gestionnaires de disposition

Les gestionnaires de disposition (LayoutManager) permettent de concevoir des mises en forme indépendantes de la plate-forme et du système de fenetrage. Ils gèrent la logique de disposition de composants à l'intérieur d'un Container.

Mini-exercice :

- Mettre en place un `Frame` contenant un `TextArea` de 10 lignes par 60 colonnes (`new Textarea(10,60)`) et, en bandeau, un `Panel` contenant 3 boutons. L e Panel est doté d'un FlowLayout cadrant les composants à gauche
(`new FlowLayout(FlowLayout.LEFT)`)



Accès au système graphique de bas niveau

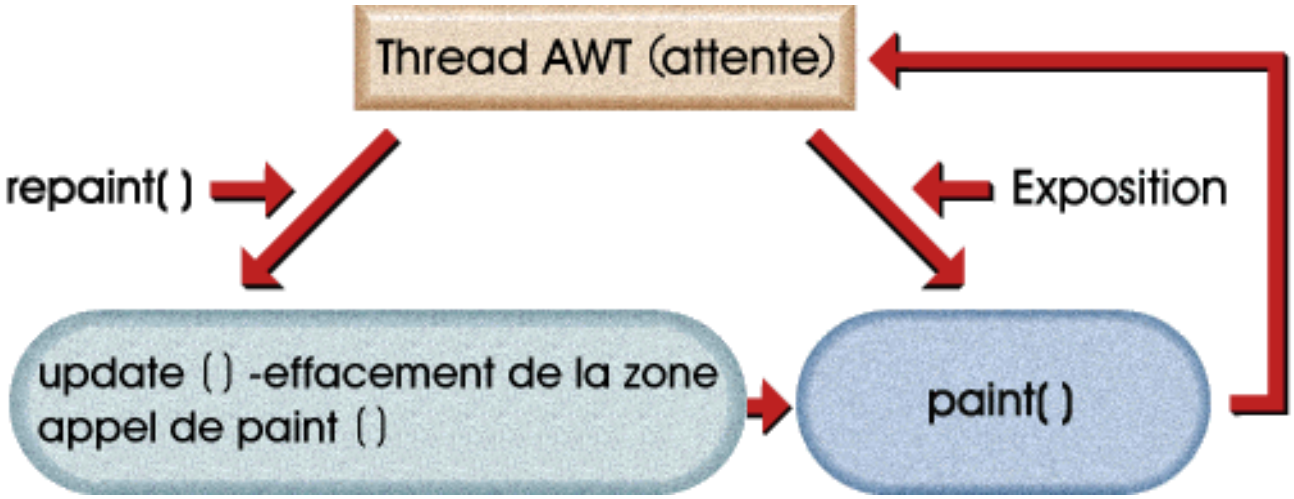
Normallement le “dessin” des composants sur l’écran est complètement pris en charge par AWT, mais il est possible de faire de la programmation plus purement graphique .

Un certain nombre de composants disposent d’un “fond” sur lequel il peut être intéressant de dessiner: le composant `Canvas` est spécifiquement prévu pour cela mais on peut aussi utiliser un `Panel` et ses dérivés ou un `Frame` (en dehors de la zone de la “décoration” de fenêtre). En pratique on peut directement sous-classer `Component` ou `Container` pour obtenir un fond transparent sur lequel on dessine: cette technique est utilisée pour définir des composants graphiques “pur java” comme Swing.

Tous les composants sont dotés de méthodes qui gèrent l’affichage graphique et sa mise à jour lors d’évènements divers (retour de la fenêtre courante au premier plan, etc.). C’est en redéfinissant une ou plusieurs de ces méthodes que l’on obtient l’effet désiré.

Accès au système graphique de bas niveau

Les mises à jour du système graphique de bas niveau :



- **`repaint()`** : demande de rafraichissement de la zone graphique. Cette demande est asynchrone et appelle update(Graphics). L'instance de `java.awt.Graphics` donne un contexte graphique qui permet aux méthodes de dessin d'opérer.

- **`update(Graphics`**) : par défaut efface la zone et appelle paint(Graphics). Cette méthode peut-être redéfinie pour éviter des papillotements de l'affichage (séries d'effacement / dessin).

- **`paint(Graphics)`** : la redéfinition de cette méthode permet de décrire des procédures logiques de (re)construction des dessins élémentaires qui constituent la présentation graphique.

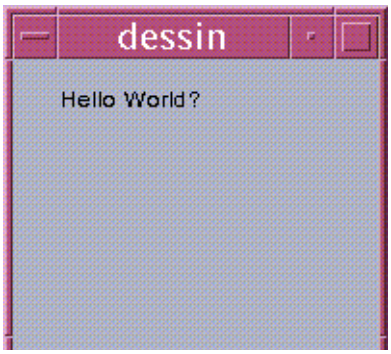
La méthode `paint()` prend un argument qui est une instance de la classe `java.awt.Graphics`. Cette classe fournit les primitives graphiques fondamentales.

Accès au système graphique de bas niveau

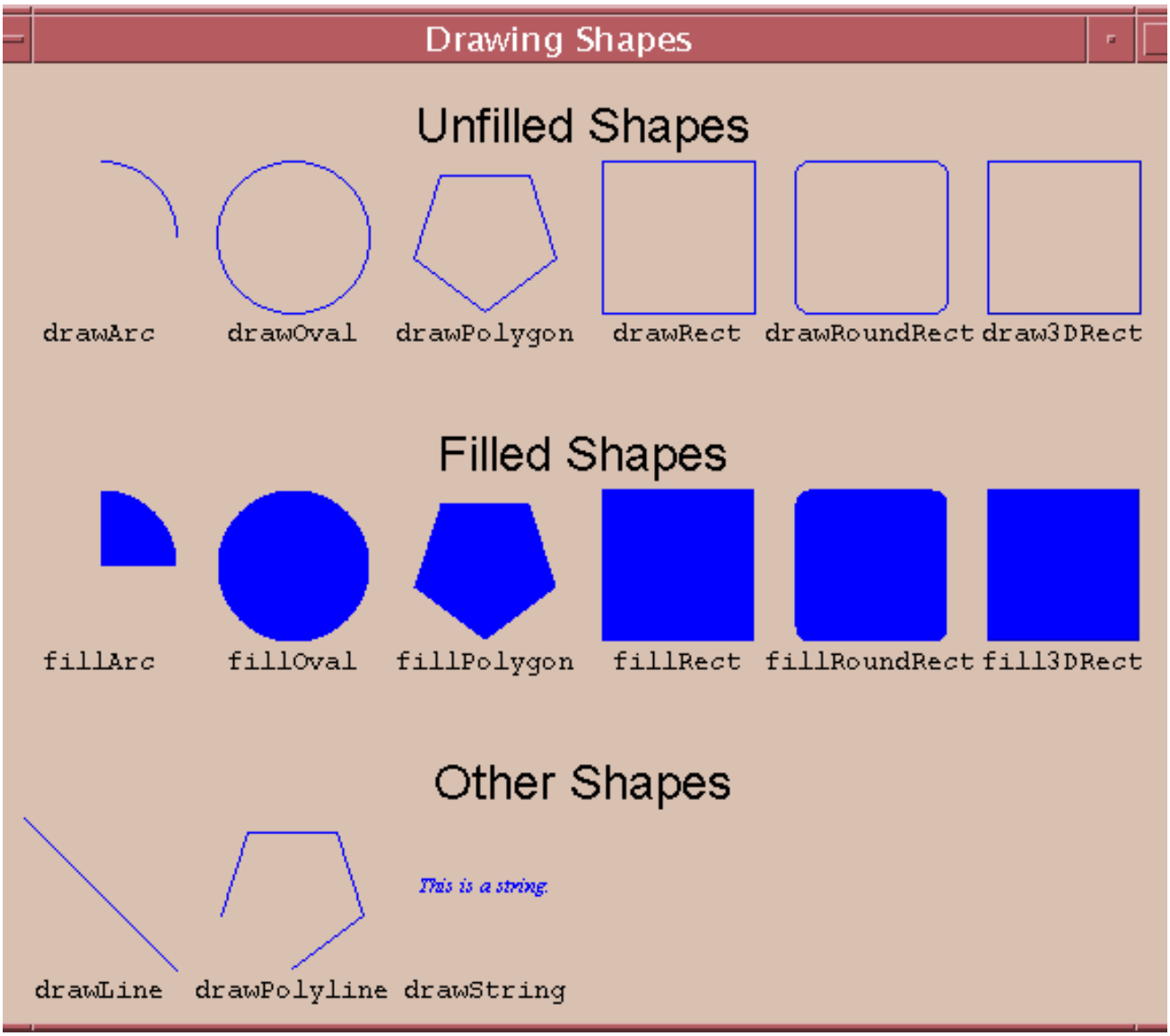
Voici un exemple minimal de composant sur lequel on écrit du texte de manière graphique:

```
package fr.gibis.graf ;
import java.awt.* ;


public class HelloWorld extends Component {

        public void paint(Graphics gr) {
                gr.drawString(“Hello World?”, 25 ,25) ;
        }

        // ce composant n’a pas de taille par lui même
        // on le met dans un frame dont on fixe la taille
        // ceci est juste pour une démonstration
        public static void main (String[] args) {
                Frame fen = new Frame(“dessin”) ;
                fen.add(new HelloWorld()) ;
                fen.setSize(200,200) ;
                fen.show() ;
        }
}
```

Les arguments numériques de la méthode drawString() sont les coordonnées x et y du début du texte. Ces coordonnées font référence à la “ligne de base” de la police. Mettre la coordonnée y à zero aurait fait disparaître la majeure partie du texte en haut de l’affichage (à l’exception des parties descendantes des lettres comme “p”, “q” etc.)



Quelques services de la classe Graphics :



Point intermédiaire : le graphique de bas niveau

Mini-exercice:

- Voir ci-après un fichier d'exemple d'une Applet.
  Une Applet est, à la base un Panel graphique, parmi les manipulations possibles il y a les appels au système graphique de bas niveau.

```
package demo ;
import java.applet.* ;
import java.awt.* ;
public class MonApplet extends Applet {
         public void paint(Graphics grf) {
                   grf.drawString("L'Applet vous salue bien!",
                             30, 30) ;
                   // autres appels graphiques
         }
}
```

Rajouter quelques appels graphiques de la classe Graphics en utilisant certaines des méthodes citées dans le schéma précédent.

Saisir un fichier HTML de lancement "demo.html" contenant :

```
<APPLET code="demo.MonApplet.class" width=300 height=100>
</APPLET>
```

Visualiser le fichier "demo.html" soit depuis un navigateur soit depuis le testeur d'applet "appletviewer" en lançant la commande:
`appletviewer demo.html`

Compléments

Les compléments techniques suivants constituent une annexe de référence

Autres exemples de programmation graphique

Un Panel transparent :

```
import java.awt.* ;
/** Panel tranparent.
   * par défaut dispose d'un Flowlayout
*/
public class TransparentPanel extends Container{

   public TransparentPanel() {
      super() ;
      setLayout(new FlowLayout()) ;
   }

   public TransparentPanel(LayoutManager lay) {
      super() ;
      setLayout(lay) ;
   }
}
```

Un composant affichant une image

Ceci est une version simplifiée qui illustre l'utilisation d'un `Mediatracker` chargé de contrôler le chargement asynchrone des images:

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

public class ImageComponent extends Component{

   protected Image   image;
   protected boolean imOK;
   protected Dimension fixedDims ;
   protected Dimension minDims ;

   public ImageComponent( String nom){
      this( nom, null) ;
   }
```

```
    public ImageComponent(String nom, Dimension dims){
       fixedDims = dims ;
       // getImage() de Toolkit prend en argument une  URL
       image = Toolkit.getDefaultToolkit().getImage(nom) ;
       MediaTracker tracker = new MediaTracker(this);
       tracker.addImage(image, 0);
       try {
          tracker.waitForID(0);
       }
       catch (InterruptedException e){}

       if (tracker.statusID(0, false) == MediaTracker.COMPLETE) {
          imOK = true;
       } else{
             // A NE PAS FAIRE ceci est un exemple
             System.err.println(" image non trouvée");
     }
       minDims = new Dimension(image.getWidth(this),
                               image.getHeight(this)) ;
    }

    public void update (Graphics g){
       // override update to *not* erase the
       // background before painting
       paint(g);
    }

    public void paint(Graphics og) {
       Dimension dim =  getSize() ;
       if (imOK) {
             og.drawImage(image, 0,0,dim.width, dim.height, this) ;
       }
    }

    public Dimension getPreferredSize() {
       return (fixedDims != null)? fixedDims : minDims ;
    }
    public Dimension getMinimumSize() {//on se répéte!
       return (fixedDims != null)? fixedDims : minDims ;
    }
    public Dimension getMaximumSize() {
       return (fixedDims != null)? fixedDims : minDims ;
    }
}
```

Corrigés des mini-exercices

DemoFrame.java :

```
package demo;
import java.awt.* ;

public class DemoFrame extends Frame {
   public DemoFrame() {
      super("demo") ;
      Panel nord = new Panel(new FlowLayout(FlowLayout.LEFT)) ;
      nord.add(new Button("un"));
      nord.add(new Button("deux"));
      nord.add(new Button("trois"));
      add(nord, BorderLayout.NORTH) ;
      add( new TextArea(10, 60), BorderLayout.CENTER) ;
   }
   public static void main(String[] args) {
      Frame demo = new DemoFrame() ;
      demo.pack() ;
      demo.show() ;
   }
}
```

MonApplet.java :

```
package demo ;
import java.applet.* ;
import java.awt.* ;
public class MonApplet extends Applet {

   public void paint(Graphics grf) {
      grf.setColor(Color.yellow) ;
      grf.fill3DRect(5,5,200,100,false) ;
      grf.setColor(Color.black) ;
      grf.drawString("L'Applet vous salue bien!", 30, 30) ;
   }
}
```

Exercices :

Exercice * :

Mettre en place une fenêtre d'interaction qui permette de constituer un Panier d'achat dans votre application `oubangui.com`.

NOTE IMPORTANTE : à ce stade vous ne pouvez pas "sortir" de votre application en fermant la fenêtre (option "close"). Générer une interruption au clavier (^C) pour mettre fin à l'exécution du programme.

Le modèle d'événements AWT 11

Points essentiels

Le traitement des événements permet d'associer des comportements à des présentations AWT :

- Il faut associer un gestionnaire d'événement à un composant sur lequel on veut surveiller un type donné d'événement.
- A chaque type d'événement correspond un contrat de réalisation décrit par un dispositif Java particluier : l'interface.
- Lorsque la réalisation de ce contrat d'interface conduit à un code inutilement bavard on peut faire dériver le gestionnaire d'événement d'une classe "Adapter".
- L'organisation relative des différents composants chargés de gérer la présentation, les événements, la logique applicative pose des problèmes architecturaux. Les "classes internes" offrent des solutions élégantes.

Les événements

Lorsque l'utilisateur effectue une action au niveau de l'interface utilisateur, un *événement* est émis. Les événements sont des objets qui décrivent ce qui s'est produit. Il existe différents types de classes d'événements pour décrire des catégories différentes d'actions utilisateur.

Sources d'événements

Un événement (au niveau de l'interface utilisateur) est le résultat d'une action utilisateur sur un composant AWT source. A titre d'exemple, un clic de la souris sur un composant bouton génère un ActionEvent. L'ActionEvent est un objet contenant des informations sur le statut de l'événement par exemple:

- `getActionCommand()` : renvoie le nom de commande associé à l'action.
- `getModifiers()` : renvoie la combinaison des "modificateurs", c'est à dire la combinaison des touches que l'on a maintenues pressées pendant le click (touche Shift par exemple).
- `getSource()` : rend l'objet qui a généré l'événement.

Gestionnaire d'événements

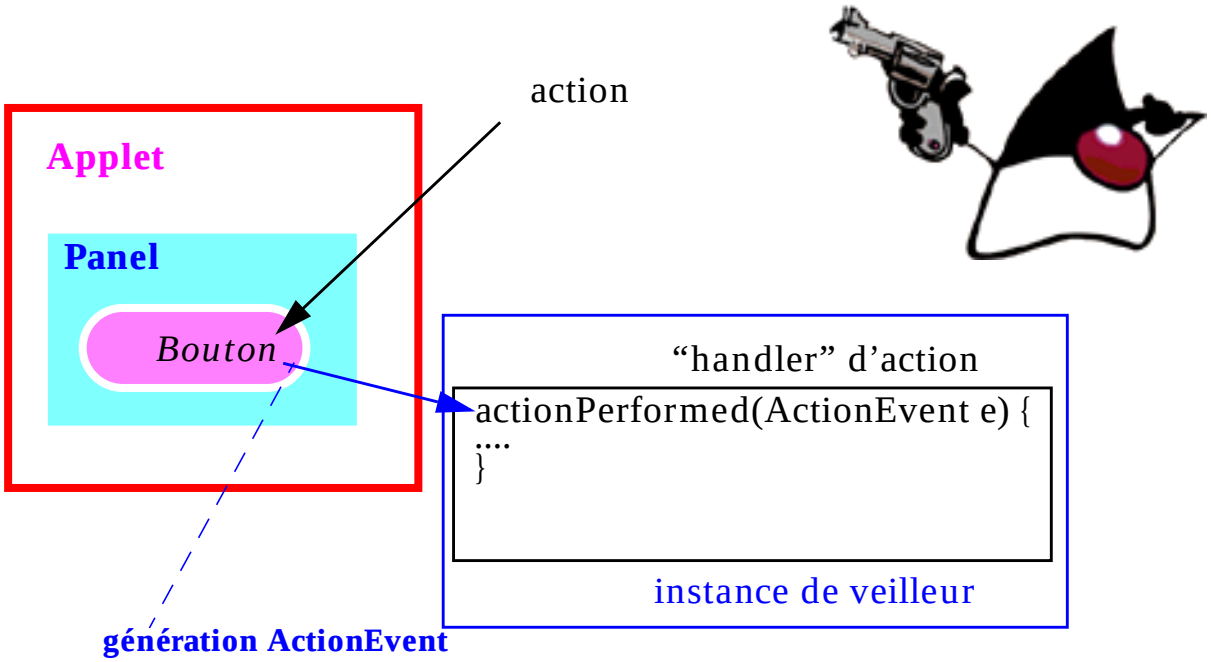
Un "gestionnaire" (*handler*) d'événement est une méthode qui reçoit un objet `Event` , l'analyse et traite les conséquences de l'interaction utilisateur.

Pour certains types d'événements il peut y avoir plusieurs types de gestionnaires qui seront respectivement appelés en fonction de l'action qui s'est produite au niveau du composant source. Ainsi pour un "événement souris" on pourra distinguer un gestionnaire correspondant à l'action du "clic" ou un gestionnaire correspondant au fait que le curseur soit passé sur le composant concerné.

Modèle d'événements

Modèle par délégation

Le JDK 1.1 a introduit dans Java un modèle d'événement appelé modèle d'événement par délégation. Dans un modèle d'événement par délégation, les événements sont (indirectement) générés par un composant graphique (qualifié de "*source*" de l'événement); on doit associer à ce composant un objet de traitement d'événement (appelé veilleur: `Listener`) pour recevoir l'événement et le traiter. De cette façon, le traitement d'événement peut figurer dans une classe distincte du composant impliqué..



Un "veilleur" est susceptible de traiter plusieurs actions liées à un type d'événement et donc de disposer de plusieurs gestionnaires : par exemple un gestionnaire chargé de veiller sur les "clics" souris et un chargé de veiller sur le passage de la souris sur le composant.

modèle d'événements

La nature du "veilleur"

Les composants susceptibles de générer des événements sont dotés de méthodes d'enregistrement des veilleurs spécialisés. On peut avoir plusieurs méthodes d'enregistrement par exemple pour un bouton `Button`:

```
addActionListener(...) ; addMouseListener(...) ;...
```

Quelle est le type de l'argument qui permet de désigner la nature du veilleur?

- Si ce type était celui d'une classe particulière on serait obligé au niveau de la réalisation de créer une sous-classe pour implanter le code de traitement.
  Bien que possible cette solution serait plus rigide: on pourrait avoir besoin d'insérer ces codes de traitement dans des instances qui dérivent d'autres classes (et, pourquoi pas, des instances qui sont elles mêmes des dérivés de composants graphiques).
  On souhaiterait avoir ici à la fois un controle de type strict (l'objet passé en paramètre doit savoir traiter l'événement considéré) et une grande souplesse au niveau du type effectif de l'objet qui rend le service.

- Le type du veilleur est défini par une interface Java

modèle d'événements

Les interfaces

Une interface est en Java une déclaration de type qui permet de décrire une capacité.

```
public interface ActionListener {
        public void actionPerformed(ActionEvent evt) ;
}
-------------------// autre exemple dans un autre fichier ".java"
public interface FocusListener {
        public void focusGained(FocusEvent evt) ;
        public void focusLost(FocusEvent evt) ;
}
```

Les méthodes décrites par ces déclarations n'ont pas de corps (juste un en-tête caractéristique).

Ces déclarations se font comme pour les classes dans des fichiers ".java" et leur compilation créé un fichier ".class".

A partir de ce moment une classe pourra déclarer adhérer au contrat ainsi défini . Elle sera obligée d'implanter le code correspondant aux méthodes déclarées mais cela lui permettra d'être "vue" comme conforme au contrat de type exigé.

```
public class Veilleur extends X implements ActionListener {
        ... // codes divers
        public void actionPerformed(ActionEvent act) {
                  // code spécifique
        }
}
```

On pourra alors écrire :

```
monBouton.addActionListener(monVeilleur) ;
```

puisqu'un `Veilleur` "est un" `ActionListener`.

Nous reverrons les interfaces dans le chapitre "Interfaces et classes abstraites", page 286

Exemple de mise en place de traitement d'événement

Les événements sont des objets qui ne sont envoyés qu'aux veilleurs enregistrés. A chaque type d'événement est associé une interface d'écoute correspondante.

A titre d'exemple, voici une fenêtre simple comportant un seul bouton :

```
import java.awt.*;
public class TestButton {
   public static void main (String args[]){
      Frame fr = new Frame ("Test");
      Button bt = new Button("Appuyer!");
      bt.addActionListener(new VeilleurBouton());
      fr.add(bt,BorderLayout.CENTER);
      fr.pack();
      fr.setVisible(true);
   }
}
```

La classe `VeilleurBouton` définit une instance de traitement de l'événement .

```
import java.awt.event.*;
public class VeilleurBouton implements
                              ActionListener{
   public void actionPerformed(ActionEvent evt) {
      System.err.println("AÍe!");
   }
}
```

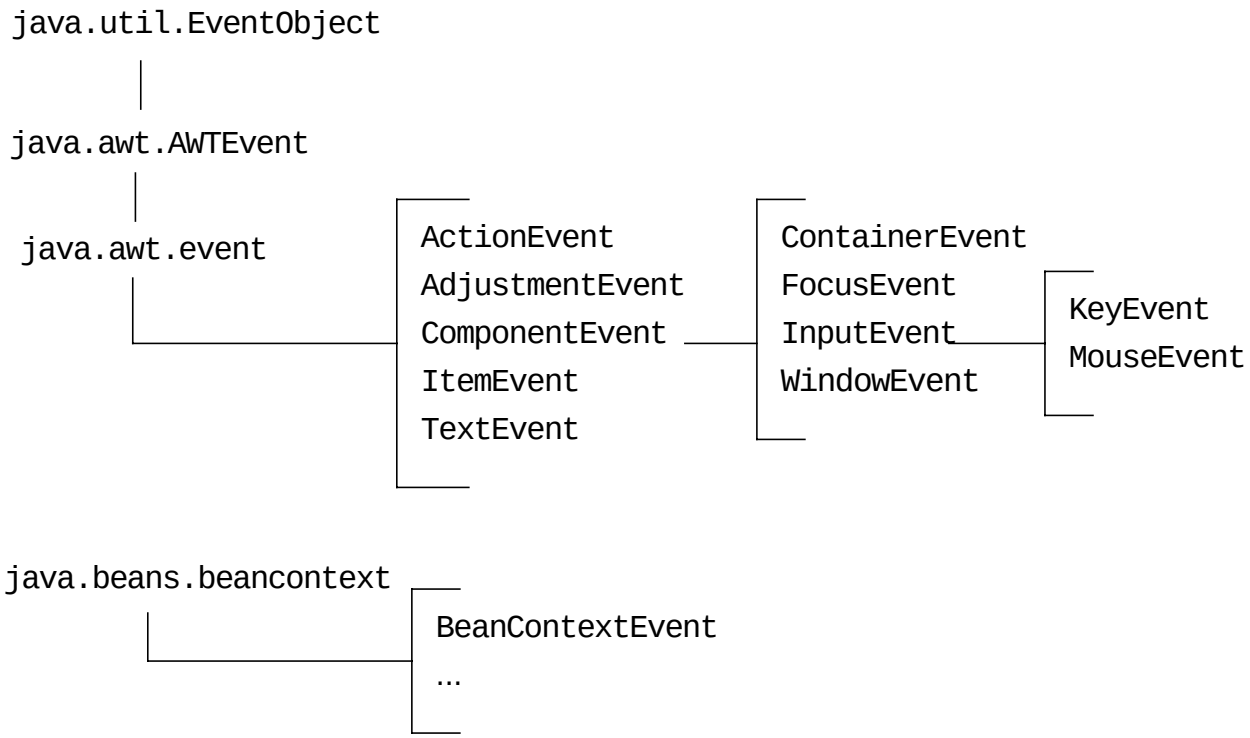
Exemple de mise en place de traitement d'événement

- La classe `Button` comporte une méthode `addActionListener(ActionListener)`.
- L'interface `ActionListener` définit une seule méthode , `actionPerformed` qui recevra un `ActionEvent`.
- Lorsqu'un objet de la classe `Button` est créé, l'objet peut enregistrer un veilleur pour les `ActionEvent` par l'intermédiaire de la méthode `addActionListener`, on passe en paramètre un objet d'une classe qui "implante" (*implements*) l'interface `ActionListener`.
- Lorsque l'on clique sur l'objet Bouton avec la souris, un `ActionEvent` est envoyé à chaque `ActionListener` enregistré et la méthode `actionPerformed (ActionEvent)est invoquée`.

Catégories d'événements

Plusieurs événements sont définis dans le package `java.awt.event`, et d'autres événements sont définis ailleurs dans l'API standard..

Pour chaque catégorie d'événements, il existe une interface qui doit être "implantée" par toute classe souhaitant recevoir ces événements. Cette interface exige qu'une ou plusieurs méthodes soient définies dans les classes de réalisation. Ces méthodes sont appelées lorsque des événements particuliers surviennent. Le tableau de la page suivante liste les catégories et indique le nom de l'interface correspondante ainsi que les méthodes associées. Les noms de méthodes sont des mnémoniques indiquant les conditions générant l'appel de la méthode



On remarquera qu'il existe des événements de bas niveau (une touche est pressée, on clique la souris) et des événements abstraits de haut niveau (Action = sur un bouton on a cliqué, sur un TextField on a fait un <retour chariot>, ...)

Tableau des interfaces de veille

Catégorie	Interface	Methodes
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
Mouse Motion	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Mouse	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustement	AdjustmentListener	adjustementValueChanged(AdjustementEvt)
Component	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResize(ComponentEvent) componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)
InputMethod	InputMethodListener	caretPositionChanged(InputMethodEvent) inputMethodTextChanged(InputMethodEvent)

+ JDK1.3 : HierarchyBoundsListener, HierarchyListener
+ JDK1.4 : MouseWheelListener, WindowFocusListener, WindowStateListener

Evénements générés par les composants AWT

Composant AWT	*Action*	adjust.	compnt	cont	focs	item	key	mse	mtn	text	win
Button	●		●		●		●	●	●		
Canvas			●		●		●	●	●		
Checkbox			●		●	●	●	●	●		
CheckboxMenuItem						●					
Choice			●		●	●	●	●	●		
Component			●		●		●	●	●		
Container			●	●	●		●	●	●		
Dialog			●	●	●		●	●	●		●
Frame			●	●	●		●	●	●		●
Label			●		●		●	●	●		
List	●		●		●	●	●	●	●		
MenuItem	●										
Panel			●	●	●		●	●	●		
Scrollbar		●	●		●		●	●	●		
ScrollPane			●	●	●		●	●	●		
TextArea			●		●		●	●	●	●	
TextComponent			●		●		●	●	●	●	
TextField	●		●		●		●	●	●	●	
Window			●	●	●		●	●	●		●

Détails sur les mécanismes

Récepteurs multiples

La structure d'écoute des événements AWT permet actuellement d'associer plusieurs veilleurs au même composant. En général, si on veut écrire un programme qui effectue plusieurs actions basées sur un même événement, il est préférable de coder ce comportement dans la méthode de traitement.

Cependant, la conception d'un programme exige parfois que plusieurs parties non liées du même programme réagissent au même événement. Cette situation peut se produire si, par exemple, un système d'aide contextuel est ajouté à un programme existant.

Le mécanisme d'écoute permet d'appeler une méthode `add*Listener` aussi souvent que nécessaire en spécifiant autant de veilleurs différents que la conception l'exige. Les méthodes de traitement de tous les veilleurs enregistrés sont appelées lorsque l'événement survient.

L'ordre d'appel des méthodes de traitement n'est pas défini. En général, si cet ordre a une importance, les méthodes de traitement ne sont pas liées et on ne doit pas utiliser cette manière de les appeler. Au lieu de cela, il faut enregistrer simplement le premier veilleur et faire en sorte qu'il appelle directement les autres. C'est ce qu'on appelle un multiplexeur d'événements

Adaptateurs d'événements

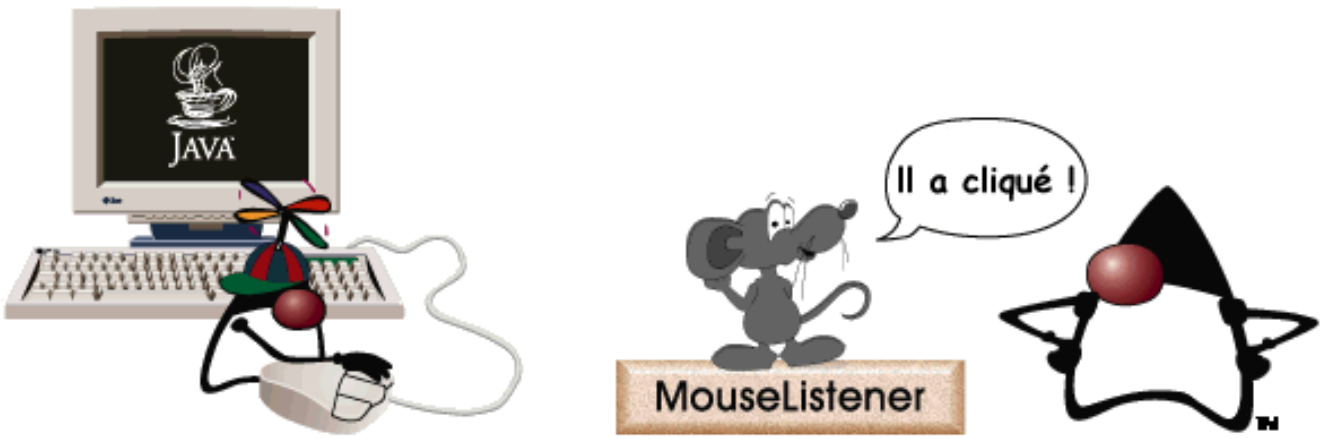
Il est évident que la nécessité d'implanter toutes les méthodes de chaque interface de veille représente beaucoup de travail, en particulier pour des interfaces comme MouseListener, ComponentListener....

A titre d'exemple, l'interface MouseListener définit les méthodes suivantes :

- `mouseClicked (MouseEvent)`
- `mouseEntered (MouseEvent)`
- `mouseExited  (MouseEvent)`
- `mousePressed (MouseEvent)`
- `mouseReleased(MouseEvent)`

Pour des raisons pratiques, Java fournit une classe d'*adaptateurs* pour pratiquement chaque interface de veille, cette classe implante l'interface appropriée, mais ne définit pas les actions associées à chaque méthode.

De cette façon, la routine d'écoute que l'on définit peut hériter de la classe d'adaptateurs et ne surcharger que des méthodes choisies.



Adaptateurs d´événements

Par exemple :

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

public class ChasseurDeClics extends MouseAdapter {

  //Nous avons seulement besoin du traitement mouseClick,
  //nous utilisons donc l'adaptateur pour ne pas avoir à
  //écrire toutes les méthodes de traitement d'événement

  public void mouseClicked (MouseEvent evt) {
    //Faire quelque chose avec le clic de la souris . . .
  }
}
```

Attention à la spécialisation de méthodes: une déclaration comme
`public void mouseclicked(MouseEvent evt)`
est légale et crée simplement une nouvelle méthode par contre le veilleur construit ne réagira pas au moment de la mise en oeuvre.

Exercices :

Exercice * destruction d'une fenêtre:

On reprendra l' exercice précédent de disposition graphique (interaction de saisie d'un `Panier`,...)

On captera sur le `Frame` l'événement de fermeture de fenêtre (`windowClosing`) et on arrétera (directement) l'application par

```
System.exit(0)
```

Considérations architecturales

La mise en place d'une interface utilisateur est particulièrement intéressante par les problèmes d'organisation qui sont posés. Il faut faire collaborer des codes qui traitent la logique applicative (par exemple les codes des classes `Livre` et `Panier` dans nos exercices) avec les codes chargés de la présentation (les codes qui permettent de présenter un Livre et de présenter l'interaction d'achat). De plus il faut impliquer les codes qui gèrent les événements et permettent la mise à jour des instances (le `Panier` est rempli, on met à jour la présentation graphique,etc..).

Dans le cadre de techniques d'organisation on met souvent en place un modèle architectural (*pattern*) nommé Modèle/Vue/Controleur .

Sans entrer dans ces techniques nous allons nous poser un problème d'organisation courant: imaginons un exemple très simple dans lequel un Frame contient un bouton `Button` et un `TextField` dans lequel nous affichons le nombre de clics sur le bouton.

Si nous écrivons un code tel que :

```
bouton.addActionListener(new ControleurAction())
```

Comment l'instance de contrôle (de type `ControleurAction`) peut-elle connaître le champ de type `TextField` pour lui demander d'afficher un message?

Considérations architecturales

Le gestionnaire d'événement dans une classe externe

Le fait que le controleur doive connaître un composant sur lequel il doit agir nous amène à réaliser une classe qui garde en mémoire une référence sur ce composant cible :

```
public class ControleurAction implements ActionListener {
        private int compteur ;
        private TextField message ;

        public ControleurAction (TextField cible) {
            message = cible ;
        }

        public void actionPerformed(ActionEvent evt) {
            message.setText("nombre de clics =" + (++compteur));
        }
}
```

et dans le code de disposition de composants graphiques :

```
bouton.addActionListener(new ControleurAction(leTextField));
```

Quelques remarques sur cette conception:

- On écrit un code de classe *ad hoc* : il n'est pratiquement pas réutilisable en dehors de ce besoin précis. De plus la maintenance n'est pas facilitée: dans le code de la classe d'interaction on n'a pas le code de contrôle d'événement "sous les yeux", une modification de la réalisation peut nécessiter une intervention sur deux fichiers. Que se passe-t-il si on décide de remplacer le champ `TextField` par un `Label`?
  Cette remarque devient d'autant plus pertinente qu'on peut être amené à ne créer qu'une seule instance de ControleurAction et à cibler de nombreux composants dans le code de gestion d'événement.

Considérations architecturales

Le gestionnaire d'événement intégré

Le gestionnaire d'événement peut être intégré à la classe de disposition graphique:

```
public class Disposition extends Frame implements ActionListener{
        ...
        private TextField leTextField ;
        ....
        private int compteur ;

        public void actionPerformed(ActionEvent evt) {
           message.setText(""nombre de clics =" + (++compteur));
        }

        ....
           bouton.addActionListener(this) ;
        ...
}
```

Ici on a tout "sous la main" mais::

- si on doit veiller sur plusieurs composants (par exemple faire de cette manière des `addActionListener` sur plusieurs boutons) le code du gestionnaire d'événement risque de devenir scabreux -il devra implanter des comportements différents en fonction du composant *source*: ce qui est criticable dans le cadre d'une programmation "à objets"-

- si, de plus, le code doit assurer simultanément la veille de Listeners complexes comme `MouseListener` et `WindowListener` on risque d'aboutir à un fouillis de gestionnaires d'événements.

Considérations architecturales

Le gestionnaire d'événement dans une classe interne

Une autre solution consiste à définir une classe de veille dans le contexte de la classe de disposition. Ceci est rendu possible en Java par la définition d'une **classe interne**.

```
public class Disposition extends Frame {
        ...
        private TextField leTextField ;
        ....
        private class ControleurAction implements ActionListener{
            int compteur ;
            public void actionPerformed(ActionEvent evt) {
                  leTextField.setText(
                          "nombre de clics =" + (++compteur));
            }
        }// fin classe interne

        ....
           bouton.addActionListener(new ControleurAction()) ;
        ...
}
```

On a ici une classe membre d'instance de la classe `Disposition` qui est située dans le contexte de la classe englobante. Du coup l'instance du veilleur a directement accès au champ "leTextField".

Pour plus de détails techniques voir également le chapitre en annexe: "Classes membres, classes locales", page 304

Considérations architecturales

Le gestionnaire d'événement dans une classe anonyme

Dans l'exemple précédent on a une classe interne qui est définie alors qu'il n'y a qu'une seule instance utilisable. Il est donc possible d'utiliser une écriture plus synthétique qui est celle d'une **classe anonyme** :

```
public class Disposition extends Frame {
        ...
        private TextField leTextField ;
        ....
           bouton.addActionListener(new ActionListener() {
              int compteur ;
              public void actionPerformed(ActionEvent evt) {
                  leTextField.setText(
                          "nombre de clics =" + (++compteur));
              }
            }) ;// fin classe anonyme
        ...
}
```

La syntaxe indique une définition à la volée d'une classe après l'appel du `new`. Cette classe (sans nom) fait implicitement `extends Object implements ActionListener` .

En voici un autre exemple (toujours dans le code de `Disposition`)

```
this.addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent evt) {
                dispose() ; //méthode de Frame
        }
}) ;
```

Ici la classe anonyme fait implicitement `extends WindowAdapter`

Les classes internes sont un dispositif architectural supplémentaire ceci ne veut pas dire qu'il faille systématiquement les utiliser en toutes circonstances.

Interfaces et classes abstraites 12

Points essentiels

Notions avancés de programmation Objet en Java: les types abstraits

- Interfaces
- Conception avec des Interfaces Java
- Les classes abstraites

Types abstraits

Nous avons vu qu'un type "objet" de Java pouvait être considéré comme un "contrat". Lorsqu'une référence est déclarée on s'intéresse à un ensemble de propriétés de l'objet désigné, mais pas nécessairement à toutes les propriétés effectives de l'objet. Ainsi il n'est pas choquant d'écrire:

```
Reader ir = new InputStreamReader(fis,"ISO8859-1") ;
```

Cette instruction est possible car la relation d'héritage permet d'assurer que le "contrat" de type est effectivement rempli par l'objet référencé.

On pourrait imaginer que le polymorphisme (cette capacité à ne s'intéresser qu'à certaines propriétés) puisse s'appliquer au travers de types abstraits purs. C'est à dire :

- des types définis par ce qu'il font et non par ce qu'il sont.
- des types tels que le "contrat" de type puisse être réalisé par des objets n'ayant aucune relation de parenté (c'est dire des objets qui appartiennent à des hiérarchies d'héritage différentes).

Imaginons, par exemple, une application bancaire dans laquelle on définisse une classe "Client" :

```
public class Client {
          .....
          Messager messager ;
          .....
```

L'idée du champ "messager" est d'avoir un dispositif qui permette de prévenir le client de la réalisation d'une opération.

Selon les clients ces dispositifs peuvent être: un simple courrier, un fax, ou un courrier électronique. On aura donc un service qui pourra être rendu par des objets de nature très différente. Et pourtant le "service" est le même partout, on a besoin d'un objet qui sache répondre au "contrat" décrit par :

```
public void envoiMessage(String message)
```

Types abstraits

En Java la défintition de ce “service” se ferait par:

```
public interface Messager {
        public void envoiMessage(String message);
}
```

Les classes qui seraient effectivement capables de rendre le service seraient définies par :

-

```
public class Fax extends Telephone implements Messager {
        public void envoiMessage(String message) {
        ...
```

-

```
public class EMail extends AgentReseau implements Messager {
        public void envoiMessage(String message) {
        ...
```

-

```
public class Courrier extends Imprime implements Messager {
        public void envoiMessage(String message) {
        ...
```

“Messager” va maintenant s’employer comme un véritable type:

```
        Client dupond, durand ;
        ....
        dupond.messager = new Fax(“0141331733”) ;
        durand.messager = new EMail(“durand@schtroumpf.fr”) ;
        ....
        for (int ct= 0 ; ct < tableauClient.length; ct++){
                tableauClient[ct].messager.envoiMessage(
                        “Tout va bien!”);
        }
```

Déclarations d'interface

Une interface est un type qui se déclare d'une manière analogue à une classe:

- Une interface publique par fichier source. Une interface fait partie d'un package.
- Le fichier source doit porter le même nom que l'interface
- La compilation génère un fichier ".class" (par ex. Messager.class)

La déclaration d'une interface ne contient que:

- des en-têtes de méthodes publiques. Il n'y a pas de "corps" de définition des méthodes (on dit qu'elles sont *abstraites*). La déclaration peut comporter des clauses `throws`.
- des déclarations de variables `static final`.

Exemple:

```
package fr.gibi.util ;

public interface Messager {
         public static final int NORMAL = 0 ;
         public static final int URGENT = 1;
         public static final int NON_URGENT = 2 ;

         public void envoiMessage(String message, int urgence)
                   throws ExceptionRoutage, ExceptionDelai ;
                   // pas de corps: méthode abstraite
}
```

Noter aussi la possibilité de définir des interfaces dérivées comme:

```
public interface MessagerFiable extends Messager, AgentFiable {
         //on réunit automatiquement les contrats des 2 interfaces
}
```

Réalisations d'une interface

Lorsqu'on déclare une classe qui réalise effectivement le contrat d'interface, il ne suffit pas d'implanter les méthodes du "contrat" il faut explicitement préciser "je connais la sémantique du contrat décrit par cette interface, et je m'engage à la respecter" et ceci se fait par la déclaration `implements`. Par ailleurs une même classe peut s'engager sur divers contrats d'interface :

```
public class Fax extends Telephone
        implements Messager,ConstantesGIBI,//interf. spécifiques
        Cloneable,Comparable, Serializable{ // standard
```

Il est possible de tester si un objet correspond à un contrat d'interface en faisant appel à l'opérateur `instanceof`. Certaines interfaces sont d'ailleurs juste des "marqueurs": il n'y a aucun corps dans leur déclaration et elles sont utilisées pour signaler une propriété d'un objet (c'est le cas de `Cloneable` et de `Serializable` dans l'exemple).

A partir du moment où un objet implante une interface il est possible de l'utiliser avec une référence du type de l'interface :

```
monClient.messager = new Fax(numero) ;
```

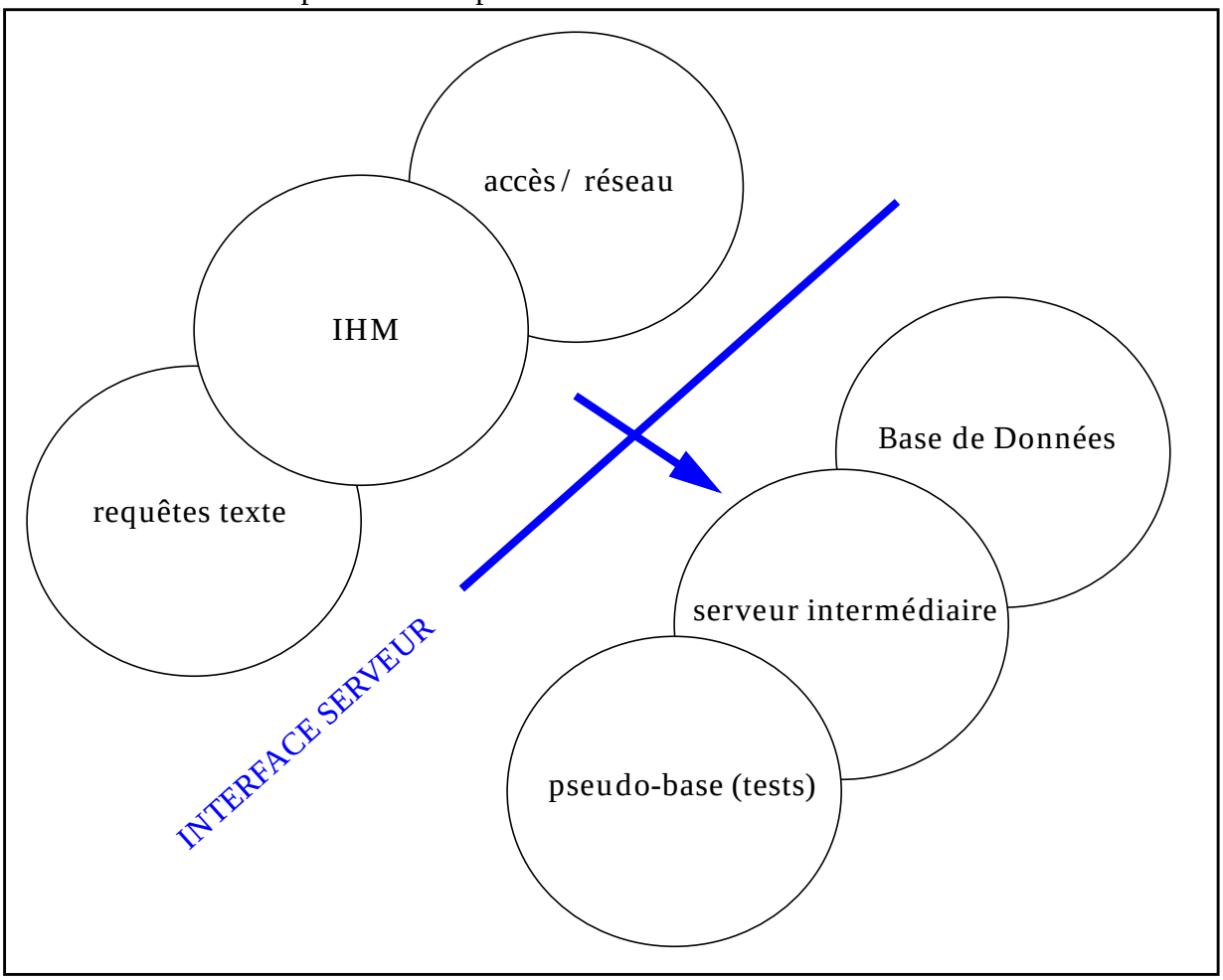
Autre Exemple :

```
public static void envoiEtArchivage(
                            String message,
                            Messager messager) {
        ....
}

...
Utils.envoiEtArchivage("Tout va très bien!", faxCourant) ;
```

Lorsqu'un objet réalise une interface ses descendants (les classes qui en dérivent) héritent de cette propriété.

Conception avec des interfaces

Les interfaces sont un mécanisme essentiel de la conception en Java. Elle permettent un **découplage** entre un service et sa réalisation. En définissant une architecture basée sur des interfaces on obtient une architecture plus modulaire dans laquelle un service donné peut être rendu par des composant différents



Dans cet exemple le service défini par “interface serveur” peut être rendu par un programme simulant le comportement d’une base de données (important pour les tests de la partie “cliente” du service) puis par une vraie base ou par un serveur intermédiaire (dans une architecture à trois parties).

Conception avec des interfaces

On trouvera de nombreuses utilisations des interfaces dans les *collections* du package `java.util`.

Une collection permet de regrouper un ensemble d'objets, le choix de la méthode de stockage se fait en fonction de la manière dont on veut accéder aux objets (objets ordonnés: `List`, objets dans un dictionnaire `Map`, etc.), ou en fonction d'autre propriétés (ensembles sans duplication: `Set`, etc.).

Supposons maintenant que l'on veuille obtenir tous les objets d'une collection sans s'intéresser à la réalisation effective de cette collection. On demandera alors à la collection de nous fournir un itérateur de type Iterator :

```
Iterator iter = maCollection.iterator() ;
while(iter.hasNext()){
        System.out.println(iter.next());
}
```

`Iterator` est une interface (tout comme `Collection` d'ailleurs).

Les classes abstraites

Une démarche courante en conception objet est d'abstraire: plusieurs concepts qui semblent superficiellement différents (et qui sont rendus par des classes différentes) peuvent être associés au travers d'une super-classe commune qui va factoriser les comportements.

Cette super-classe n'est pas nécessairement complète: certaines parties de son comportement sont définies "contractuellement" (comme dans une interface Java) mais ne connaissent pas de réalisation concrète. On a ainsi une super-classe qui dispose éventuellement de variables et de méthodes membres, de constructeurs, etc. mais pour laquelle la réalisation effective de certaines méthodes reste en suspens.

On a donc une classe (*classe abstraite*) pour laquelle l'instanciation est impossible et qui impose à ses classes dérivées de définir des comportements effectifs pour les méthodes *abstraites*.

Exemple: dans une application graphique on veut définir des objets géométriques `UnRectangle`, `UnOvale`, `UnLosange`, etc... ces différentes figures s'inscrivent dans un rectangle ayant des coordonnées, une largeur, une hauteur (la classe AWT `Rectangle` sert à décrire cela). Pour simplifier la gestion de ces différents objets on peut les faire dériver d'une classe abstraite Figure :

```
import java.awt.* ;
public abstract class Figure {
        protected Rectangle posDims ;
        protected Figure(Rectangle def) {
                posDims = def;
        }
        public abstract void paint(Graphics gr) ;
        public String toString() {
                return posDims.toString() ;
        }
        ... // autre méthodes communes
}
```

Chacune des figures concrètes qui deriveront de `Figure` devra savoir comment utiliser la méthode `paint(Graphics)` pour se dessiner.

Ainsi la classe `UnOvale` pourrait s'écrire :

```
public class UnOvale extends Figure {
        public UnOvale (Rectangle def) {
                super(def);
        }
        public UnOvale(int x, int y, int w, int h) {
                this(new Rectangle(x,y,w,h))
        }
        public void paint(Graphics gr) {
                gr.drawOval(posDims.x, posDims.y,
                        posDims.width, posDims.height);
        }
        ...
}
```

Bien qu'on ne puisse créer d'instance dont le type effectif soit une classe abstraite on peut utiliser une classe abstraite pour typer une référence:

```
public class MonCanvas extends Canvas {
        Figure[] tbFig = { new UnOvale(0,0, 12, 12),
                new UnLosange(20,40,33,33),
                ...
                } ;
        ....
        public void paint(Graphics gr) {
                ...
                for (int ix = 0 ; ix <tbFig.length ; ix++) {
                        tbFig[ix].paint(gr) ;
                }
        }
}
```

- A partir du moment où on déclare une méthode `abstract` la classe doit être marquée `abstract`.
- Si une sous-classe d'une classe abstraire ne fournit pas de réalisation pour une des méthodes abstraites dont elle hérite elle doit, à son tour, être marquée `abstract`.
- Bien entendu seules des méthodes d'instance peuvent être `abstract`.

Une suite aux mini-exercices?

Voici quelque chose que vous pourriez développer lorsque vous réviserez ce cours :

- Notre commerce de confiserie élargissant sa gamme de produits on trouvera au sommet de la hiérarchie une classe abstraite “Sucrerie” dont dériveront “Bonbon”, “Gateau”, “Soda” etc.

  On ne peut pas créer d’instance de “Sucrerie” mais son code permet de mutualiser les modalités de calcul de prix, de stock, etc.

- Suite à l’apparition des modalités de taxation modifiées pour les sucreries “allégées” on définira une interface “IsLight” (qui, par exemple, exigera une information sur le taux de sucre).

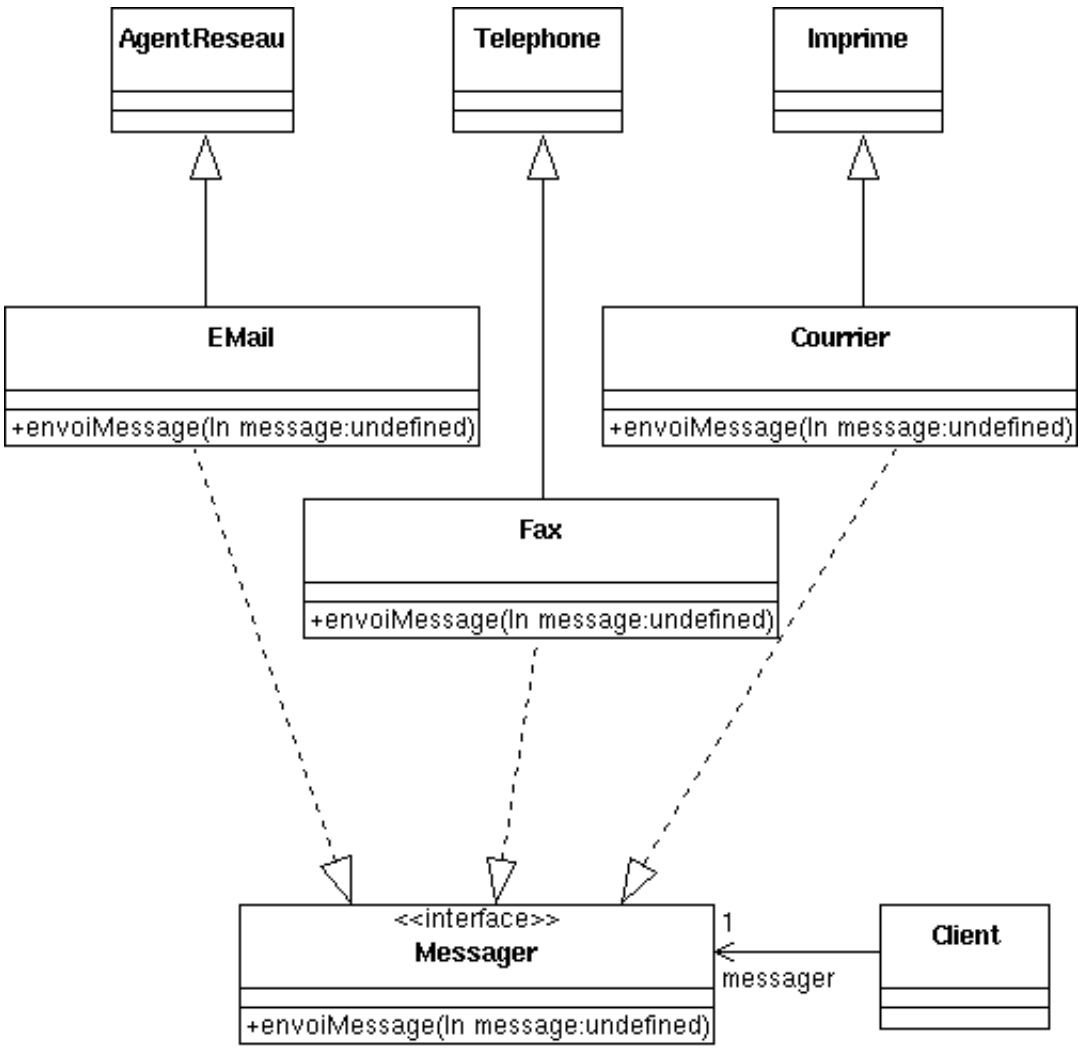
  On pourra ensuite définir “BonbonLight”, “SodaLight” etc.

Compléments

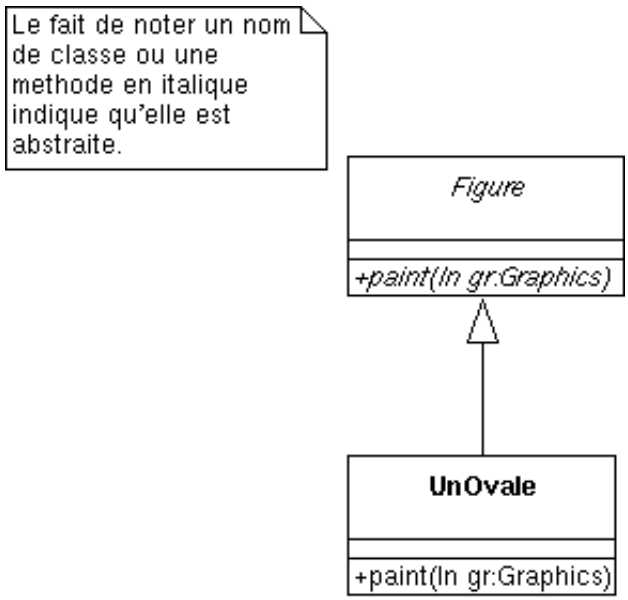
Les compléments techniques suivants constituent une annexe de référence

Utilisation notation UML

interfaces



Classes abstraites



Utilisation des interfaces pour l'import de constantes

Une utilisation annexe des interfaces est de mettre en place un dispositif syntaxique qui simplifie la désignation de variables partagées non modifiableless:

```
public interface TraceCst {
   public static final boolean TRACE = true;
}
```

permettra une utilisation simplifiée de constantes statiques (on suppose l'existence d'une classe `Traceur`):

```
public class Voir implements TraceCst {
   public static void main (String[] args0) {
      if(TRACE) {
         Traceur.trace("trace généralités") ;// code non généré
      }
   }
}
```

Ici la constante "TRACE" n'a pas eu besoin d'être qualifiée (en écrivant `TraceCst.TRACE`).

L'utilisation d'une constante "TRACE" évaluable au *compile-time* permet d'écrire un code qui ne sera pas généré par le compilateur. On a ici l'équivalent d'un dispositif de compilation conditionnelle.

Pour un système de trace voir en Java 1.4 assertions et "logging" API.

Exercices :

Exercice * :

Lire la documentation de la classe `java.util.Arrays` methode `sort(Object[])`.

Modifier votre classe `Livre` pour la rendre triable par cette méthode.

Exercice ** :

Le projet commercial de la jeune pousse `oubangui.com` évoluant rapidement il a été décidé de vendre aussi des disques, des CDs , des jeux video,etc.

Un Panier sera donc constitué avec des `Produits`.

Définir un `Produit`, modifier `Panier`, `Livre` et créer par exemple une classe `Disque`.

ANNEXES 13

Les chapitres suivants ne font pas, à proprement parler, partie du cours présenté par l'animateur.

Les sujets abordés le sont à titre de référence ou d'illustration

Classes membres, classes locales 14

Points essentiels

Aspects avancés de la programmation en Java: définition de classes “à l’intérieur” d’une autre classe.

Introduction: un problème d'organisation

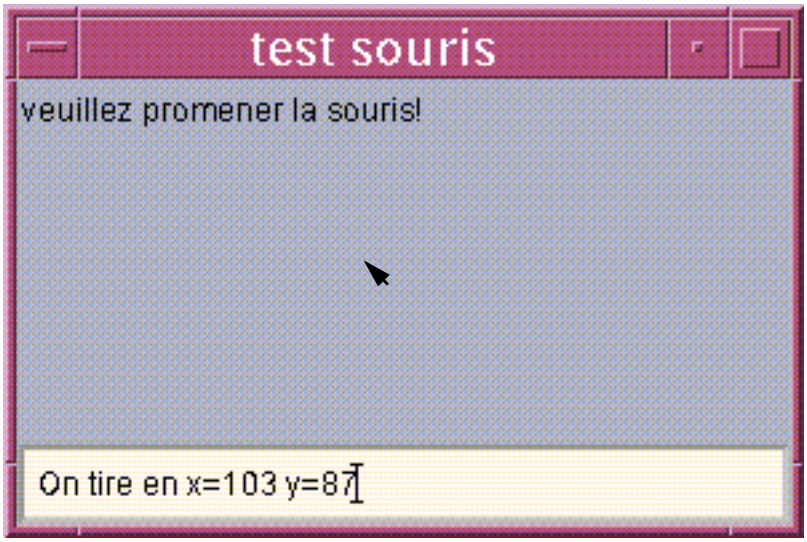
Soit le programme suivant :

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

public class IHMsouris {

   private Frame frame  = new Frame("test souris");
   private TextField txt = new TextField(30) ;
   private int initszH ;
   private int initszW ;

   public IHMsouris(int largeur, int hauteur) {
      initszH = hauteur ;
      initszW = largeur ;
      frame.add(new Label("veuillez promener la souris!"),
         BorderLayout.NORTH) ;

      frame.add(txt,BorderLayout.SOUTH) ;
      .....
   }// constructeur

   public void start() {
      frame.setSize(initszW, initszH) ;// préférer pack()
      frame.show() ;
   }

   public static void main (String[] args) {
      IHMsouris ihm = new IHMsouris(300,200) ;
      ihm.start() ;
   }
}
```

L'objectif est ici d'avoir une zone centrale (dans un BorderLayout) dans laquelle on puisse suivre le déplacement d'une souris qu'on "tire". Un message informatif doit être affiché dans un autre composant ("txt").

L'événement à suivre est `MouseMotion` et la méthode qui nous intéresse est `mouseDragged(MouseEvent)`

Il faut compléter le code de manière à obtenir à peu près l'aspect et le comportement suivant :



Envisageons diverses solutions:

- Ecrire une classe externe qui dérive de MouseMotionAdapter.
  Pour que le veilleur puisse manipuler le `TextField` correspondant il faudra passer ce composant en paramètre du constructeur:
  ```
  public MyAdapter(TextComponent tc) { ....
  ```
  puis faire dans le code :
  ```
  frame.addMouseMotionListener(new MyAdapter(txt));
  ```

- Faire en sorte que la classe "IHMsouris" implante l'interface `MouseMotionListener` et que la méthode réalisant `mouseDragged` accède au `TextField`.

Dans un cas on crée une classe générale *ad hoc* pour répondre à un besoin très particulier. Dans l'autre cas on a un code bavard qui peut, en plus, devenir extraordinairement compliqué s'il faut gérer différents suivis de souris en fonction des composants sur lesquels on "tire".

Il est possible d'écrire ce code autrement en utilisant des classes locales imbriquées.

Introduction: organisation avec une classe locale

Voici une manière d'écrire le code correspondant au problème précédent:

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

public class IHMsouris {

   private Frame frame  = new Frame("test souris");
   private TextField txt = new TextField(30) ;
   private int initszH ;
   private int initszW ;

   public IHMsouris(int largeur, int hauteur) {
      initszH = hauteur ;
      initszW = largeur ;
      frame.add(new Label("veuillez promener la souris!"),
         BorderLayout.NORTH) ;

      frame.add(txt,BorderLayout.SOUTH) ;
      frame.addMouseMotionListener( new MouseMotionAdapter() {
         public void mouseDragged(MouseEvent evt) {
            txt.setText("On tire en x=" + evt.getX()
                     + " y=" + evt.getY()) ;
         }
      });
   }// constructeur

   public void start() {
         // rappel : éviter setSize -portabilité-
      frame.setSize(initszW, initszH) ;
      frame.show() ;
   }

   public static void main (String[] args) {
      IHMsouris ihm = new IHMsouris(300,200) ;
      ihm.start() ;
   }
}
```

On a ici utilisé une classe locale anonyme pour répondre à des besoins très locaux.

Analysons le code :

- `new MouseMotionAdapter() {....}`
  permet d'invoquer un constructeur d'une classe que nous définnissons comme dérivant de MouseMotionAdapter. C'est, sous une forme condensée, l'écriture de
  ```
  class MouseMotionAdapter1
   extends MouseMotionAdapter { ...}
  ```
  et de
  ```
  new MouseMotionAdapter1() ...
  ```

- Cette nouvelle classe ne porte pas de nom (d'où "classe anonyme") et cette invocation / définition rappelle également les tableaux anonymes:
  ```
  method( new String[] {"un", "deux"}) ;
  ```
  Ceci dit il y a effectivement un nouveau fichier ".class" qui est généré et qui porte un nom lié à la classe hôte + un numéro (ici `IHMsouris$1.class`).Ce fichier ".class" devra toujours accompagner celui de la classe qui permet de le définir.

- Cette instance de nouvelle classe est "imbriquée" avec celle de la classe englobante: toutes les deux ont accès au champ "txt" bien que celui-ci soit privé dans la définition de la classe "IHMsouris".

Nous venons d'avoir ici un aperçu sur d'autres caractéristiques de Java:

- Il est possible de définir des classes à l'intérieur d'une classe. Ces classes peuvent être des classes membres (au même titre qu'un champ ou une méthode membre) ou des classes locales (définies dans un bloc).

- Les classes membres d'instance ou les classes locales peuvent, sous certaines conditions, avoir un accès privilégié aux membres de l'instance englobante (ou un accès à des variables locales).

Classes et interfaces membres statiques

On peut définir une classe ou une interface comme membre statique d'une classe.

Prenons par exemple une classe "Compte" et une classe "Client":

```
public class Client implements Comparable{// pour commencer
   public final String id ;
....
   public int compareTo(Object autre) {
      return this.id.compareTo(((Client)autre).id) ;
   }

   public boolean equals(Object autre) {
      try {
      return (0 == this.compareTo(autre)) ;
      } catch (ClassCastException exc) { return false;}
   }
...
```

Cette classe a un particularité : on a défini un "ordre naturel" qui s'applique à toutes les instances de la classe "Client". Parcequ'un Client est "Comparable" on pourra écrire :

```
if( 0 < client1.compareTo(client2)) {...}
....
Client[] tbClient = .... ;
....// les comparaisons permettent de trier ...
java.util.Arrays.sort(tbClient) ; //tri de la table
```

Cette situation n'est pas applicable aux instances de "Compte": on voudrait permettre au programmeur de trier un tableau de Compte soit par le champ "solde" soit par le champ "client" (voir code page suivante faisant appel à des classes statiques membres)

```
import java.util.* ;
import java.math.* ;

public class Compte {
  private Client client ;
  private BigDecimal solde ;
.....

  public static class CompareParClient implements Comparator{
      public int compare( Object lun, Object lautre) {
         return ((Compte)lun).client.compareTo(
            ((Compte)lautre).client);
      }
      //
   }// classe membre

  public static class CompareParSolde implements Comparator{
      public int compare( Object lun, Object lautre) {
         return ((Compte)lun).solde.compareTo(
               ((Compte)lautre).solde);
      }
   }// classe membre
......
}// classe englobante
```

Ici un programmeur pourra écrire :

```
Compte.CompareParSolde comparateur = new Compte.CompareParSolde();
if(0 < comparateur.compare(compte1, compte2)) {....}
....
Arrays.sort(tbCompte, comparateur) ;// autre forme de “sort”
```

On a ici défini deux classes membres statiques pour contrôler la manière dont les comparaisons (et donc les tris) peuvent opérer. Leur nom est associé à celui de la classe englobante (“Compte.CompareParSolde”) mais le fichier “.class” correspondant est “Compte$CompareParSolde.class”.

Bien entendu le fait d’imbriquer les définitions de classe doit correspondre à un lien de dépendance conceptuelle.Toutefois on a ici plus qu’une simple hiérarchie analogue à la hiérarchie des packages. En effet la classe ou l’interface interne bénéficie des droits d’accès à l’intérieur de la classe englobante.

La classe interne peut également être `private` ou `protected` (ce que ne peuvent pas être les autres classes).

Classes membres d'instance

Une classe peut être un membre d'instance.

```
public class MyFrame extends Frame{

        public class ButtonListener implements ActionListener{
                public void actionPerformed(ActionEvent evt) {
                        ....
                }
        }
}
```

Ici on a une classe interne liée à une instance. Depuis une autre classe la classe interne est de type `MyFrame.ButtonListener`. Par contre un constructeur de `ButtonListener` a besoin d'une instance de `MyFrame`. Ainsi , par exemple, depuis le "main"(statique) de `MyFrame`:

```
 MyFrame fen = new MyFrame() ;
MyFrame.ButtonListener bl = fen.new ButtonListener();
```

Une telle classe à la posssibilité d'accéder aux autres membres de l'instance englobante.

Ici aussi le fichier ".class" sera généré sour le nom "MyFrame$ButtonListener.class"

Classes membres d'instance

ex. réalisation d'une interface par une classe privée:

`java.util.Enumeration` est une interface publique permettant de référencer une instance d'objet qui permet de parcourir une collection. Un tel objet doit avoir deux méthodes: `boolean hasMoreElement()` et `Object nextElement();`

```
// la classe Pile gère une pile d'objets
public class Pile {
        private Object[] tableExtensible ;
        private int sommetDePile ;
        ...
        private class ParcoursPile
                  implements java.util.Enumeration {
                  int index = sommetDePile ;
                  public boolean hasMoreElements(){
                   return index >= 0;
                  }
                  public Object nextElement(){
                   return tableExtensible[index--];
                  }
        }// fin parcoursPile
        public java.util.Enumeration elements() {
         return new ParcoursPile();
        }
        ...
}
```

On notera une autre particularité des classes membres (d'instance ou de classe): la classe englobante a aussi un accès privilégié aux membres de la classe interne. Il est par exemple possible d'avoir une classe interne publique qui a un constructeur privé accessible uniquement par le code de la classe englobante: cela permet de construire une classe dont la construction est reservée à une autre classe.

Classes dans un bloc

Il est possible de définir une classe locale dans un bloc de méthode.

Une telle classe peut accéder soit à des variables d'instance, soit à des variables locales (ou des paramètres de la méthode).

```
public class MyFrame extends Frame {
        protected TextField messager = new TextField(30) ;
        ....
        public Button createButton(String nom,final String mess){
                Button res = new Button(nom) ;
                class BListener implements ActionListener{
                        public void
                         actionPerformed(ActionEvent evt) {
                         messager.setText(mess) ;
                        }
                }
                Blistener list = new BListener() ;
                ...
                res.addActionListener(list) ;
                return res;
        }
        ...
}
```

L'accès à des variables locales nécessite que celles-ci soient marquées `final`.

Le nom du fichier ".class" sera généré automatiquement en fonction de l'ensemble du code de la classe (par ex. : "MyFrame1BListener.class") et il faudra livrer ce code en même temps que le code principal "MyFrame.class".

Classes anonymes

L'utilisation de classes anonymes permet parfois une simplification de l'écriture de classes locales.

```
public class MyFrame extends Frame {
        protected TexField messager = new TextField(30) ;
        ....
        public Button createButton(String nom,final String mess){
                Button res = new Button(nom) ;
                res.addActionListener(new ActionListener(){
                        public void
                         actionPerformed(ActionEvent evt) {
                         messager.setText(mess) ;
                        }
                });
                return res;
        }
        ...
}
```

On remarquera dans l'exemple la notation :

```
new ActionListener(){...}
```

`ActionListener` étant une interface on a en fait une écriture synthétique qui marque la création d'un nouvel objet qui "implémente" l'interface considérée.

On ne peut pas définir un constructeur dans une classe anonyme, mais on peut l'invoquer au travers d'un constructeur de la classe mère:
`new ClasseMere(arg1, arg2) { ...` *redéfinitions* `...}`

Le nom du fichier ".class" sera généré automatiquement par le système (par exemple "Myframe$1.class").

Récapitulation: architecture d'une déclaration de classe

Plan général des déclarations de classe premier niveau:

```
package yyy;
import z.UneCLasse ; // A préférer
import w.* ;

public class XX { // public ou "visibilité package"
                  // une classe publique par fichier
         // si méthode "abstract" la classe doit etre "abstract"
         // ordre indifférent sauf pour les blocs et expressions
         // d'initialisation de premier niveau

         //MEMBRES STATIQUES
         variables statiques
                   var. statiques initialisées
                   var. statiques "final"/constantes de classe
         méthodes statiques
         classes ou interfaces statiques
                   classes statique + accès privilégié classe
                                       englobante

         //MEMBRES D'INSTANCE
         variables d'instance
                   var.d'instance initialisées
                   var d'instance "blank final"
         méthodes d'instance
                   "patrons" de méthodes d'instance (abstract,...)
         classes d'instance
                   classes avec accès instance englobante

         //BLOCS DE PREMIER NIVEAU
         blocs statiques
          /* évalués au moment du chargement de la classe */
         blocs d'instance
          /* évalués au moment de la création de l'instance */

         // CONSTRUCTEURS
         constructeurs

}
```

Compléments

Les compléments techniques suivants constituent une annexe de référence

Levée des ambiguités sur `this` et `super` dans une classe interne

Pour désigner des membres de l'instance courante l'usage de `this` est généralement implicite et le compilateur sait rattacher correctement un nom de membre isolé à l'instance courante.

Dans le cas de classes imbriquées cette résolution est plus complexe. Dans l'exemple:

```
public class Pile {
        private Object[] tableExtensible ;
        private int sommetDePile ;
        ...
        private class ParcoursPile
                  implements java.util.Enumeration {
                  int index = sommetDePile ;
                  public boolean hasMoreElements(){
                   return index >= 0;
                  }
                  public Object nextElement(){
                   return tableExtensible[index--];
                  }
        }// fin parcoursPile
        public java.util.Enumeration elements() {
         return new ParcoursPile();
        }
        ...
}
```

Implicitement le compilateur comprend :

```
return this.index >= 0;
return Pile.this.tableExtensible[index--];
```

Cette notation ***NomClasse*.this** peut être explicitement utilisée pour lever une ambiguité à l'intérieur de la classe interne. De même la notation ***NomClasse*.super** est utilisable si besoin est. Voir également "Constructeurs", page 394 et suivante.

Membres `static` des classes internes

Une classe interne ne peut pas avoir de membre déclaré `static`. Il existe une exception à cette règle qui concerne les champs `static final` intialisés qui sont des constantes de *compile-time*.

Exercices :

Exercice **: mise en place du fonctionnement de la saisie interactive du `Panier` client dans l'application `oubangui.com`.

Packages fondamentaux 15

Points essentiels

- Les “packages” standard constituent les librairies qui doivent accompagner toute JVM.
- Ils fournissent, de manière harmonisée, un cadre pour des opérations essentielles :
  - services communs
  - structures de données, fonctions de calcul
  - interactions graphiques portables
  - entrée / sorties,
  - accès réseau
  - sécurité
  - internationalisation
  - composants beans
  - APIs et classes pour accès SQL, pour l’invocation distante (RMI) et pour CORBA

java.lang

Ce package est automatiquement importé dans tout source Java.

On y trouve des objets et des services fondamentaux.

- La classe racine : `Object` et interfaces utilitaires: `Cloneable`, `Comparable`
- Introspection et exécution dynamique : `Class`, `Package`, package `java.lang.reflect`, …
- Exécution : `ClassLoader`, `SecurityManager`,
- Services dépendant du contexte local d'exécution : `System`, `Runtime`, `Process`,
- Processus légers : `Thread`, `ThreadGroup`, interface `Runnable`
- Manipulations avancées sur la gestion des variables: `ThreadLocal`, package `java.lang.ref`
- Racine des classes exceptions : `Throwable`
- Classes "d'encapsulation" des types scalaires : `Integer`, `Byte`, `Short`, `Float`, `Double`, (`+ Number`), `Character`, `Boolean`, (`+Void`)
- Chaînes de caractères : `String`, `StringBuffer`
- Fonctions mathématiques : `Math`

java.lang

Classes d'encapsulation

Dans certaines situations (collections d'objets, paramètres génériques,...) on ne peut pas utiliser une valeur primitive scalaire mais on peut utiliser une instance d'une classe d'encapsulation (`Integer`, `Float`, `Character`, etc.)

Chacune de ces instances encapsule une valeur scalaire immuable. Les constructeurs, méthodes, méthodes statiques fournissent de nombreux services associés :

```
Integer oInt = new Integer(500);
Integer oInt2 = new Integer("500");
short val = oInt.shortValue();
int valeur = Integer.parseInt("500") ;
String hexS = Integer.toHexString(500) ;
```

StringBuffer

Les objets de type String étant également immuables il peut être préférable, lorsqu'on doit faire des opérations de transformation de chaîne de caractères, d'utiliser des objets `StringBuffer`.

```
StringBuffer buf = new StringBuffer("Nacho");
buf.insert(2,"bu").append(new
StringBuffer("sonid").reverse()).append("aure ").append(6) ;
String res = buf.toString() ;
```

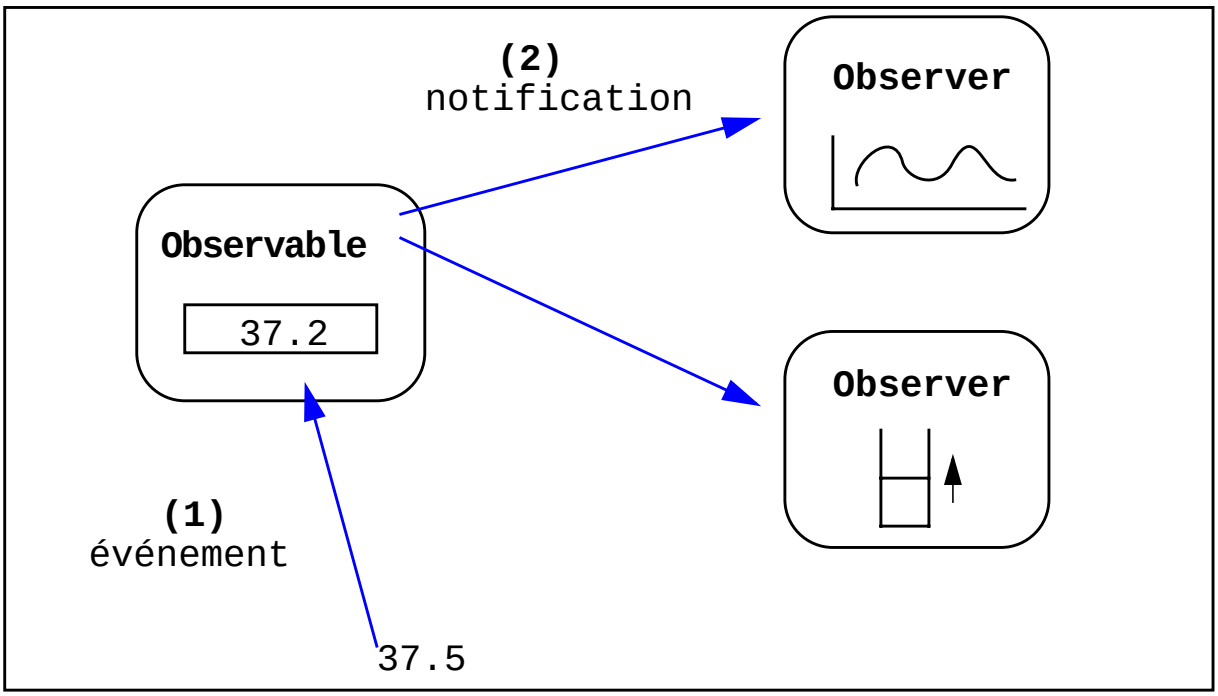
java.util

Utilitaires et structures de données pour le programmeur:

- Classes et interfaces pour les Collections :
  - Ensembles (`Set`): `HashSet`
  - Listes (`List`) : `LinkedList`, `ArrayList`, `Vector`, `Stack`,....
  - Dictionnaires (`Map`) : `HashMap`, `HashTable`, `Properties`....
  - arbres: `TreeMap, TreeSet`
  - Utilitaires pour collections: interfaces de parcours `Enumeration`, `Iterator` . Fonctions de recherche, de tris, etc. sur des collections (`Collections`), des tableaux (`Arrays`) en utilisant les interfaces `Comparator` ou `java.lang.Comparable`.
- Gestion du temps: `Date`, `TimeZone`, classe abstraite `Calendar` et le calendrier standard `GregorianCalendar`.
- “patterns” : `Observer`/`Observable`
- .....

java.util

Modèle Observer/Observable




```
public class CoursBourse extends Observable {
          private HashTable cours ;
          ....
          public void changeCote(String valeur, Money cote){
                    cours.put(valeur,cote) ;
                    notifyObservers(valeur) ;
          }
          ....
}
```

```
public class AfficheCours extends ... implements Observer {
          // A la création s'enregistre auprès de l'Observabl
          ....
          public void update(Observable cours, Object valeur)
                    // Met à jour l'affichage
          }
}
```

Internationalisation (i18n)

L'internationalisation des applications est favorisée par l'emploi du jeu de caractères UNICODE et par des classes et packages comme:

- `java.util.Locale` : permet de désigner des "aires culturelles" hiérarchisées. Par exemple l'aire du Français, sous-ensemble Canadien.

- Les classes dérivées de `java.util.ResourceBundle`: permettent d'organiser des hiérarchies de ressources. Si, par exemple, on ne trouve pas un mot dans le dictionnaire spécifique au Canadien-Français on va le chercher dans le dictionnaire du Français

- Le package `java.text` permet de traiter des problèmes d'adaptation de formats de dates, de formats des nombres, de ternir compte de l'ordre alphabétique local, de mettre en place des messages paramétrables, etc.

- Le package `java.awt` permet de traiter des dispositions dépendantes de l'aire culturelle (par ex. de droite à gauche si besoin est).

Numeriques divers

- `java.math.BigInteger`:
  permet de réaliser des opérations sur des entiers de taille illimitée

- `java.math.BigDecimal`:
  permet de réaliser des calculs avec des nombres décimaux. La taille des nombres n'est pas limitée et on peut spécifier la précision et la manière de réaliser des arrondis. Convient bien pour représenter des valeurs financières.

- `java.util.Random`:
  permet de générer des séries de nombres aléatoires.

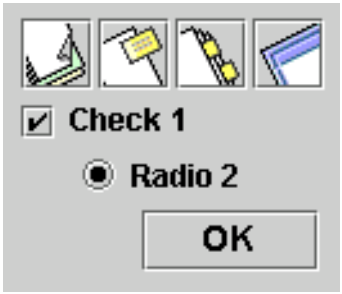
Interactions graphiques portables : AWT, SWING

Il existe deux bibliothèques d'interface graphique : AWT fournit un petit nombre de composants sur une base native, javax.swing fournit des composants pur java (composants "dessinés") plus variés mais plus complexes.

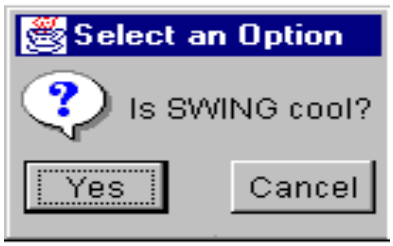
- AWT de base :
  - Bibliothèque de composants simples (disponibles sur toutes plateformes de fenêtrage),
  - Dessin graphique simple et 2D de base,
  - Technique de disposition des composants (auto-adaptation à la taille des fenêtres)
  - Technique de gestion des événements
- Sous-packages spécialisés : coupé/collé (`dnd`, `datatransfer`), 2D (`geom`), `image`,`...`
- "Extensions" livrées en standard : `accessibility`,Swing

Interactions graphiques portables : AWT, SWING

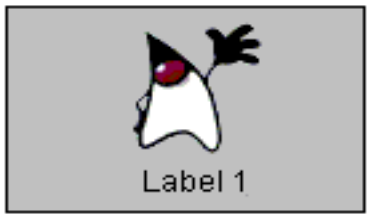
Exemples de composants Swing:



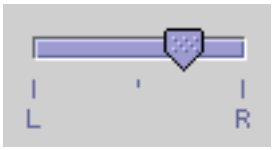
JButton



JOptionPane

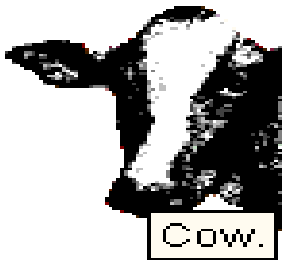


JLabel

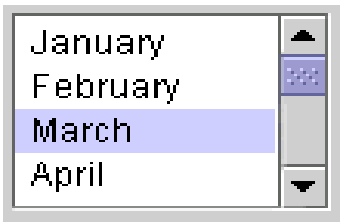


JSlider

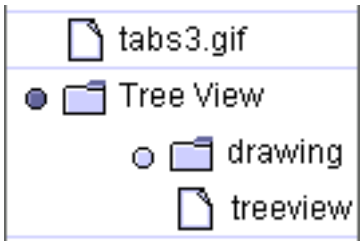
JScrollPane



JToolTip



JList



JTree

JTable

Entrées/sorties

Outre les filtres spécialisés qui permettent de manipuler des données primitives (`DataInput`, `DataOutput`) un aspect très important des E/S est la capacité de lire/écrire des objets sur un flot au moyen des classes `ObjectInputStream` et `ObjectOutputStream`.

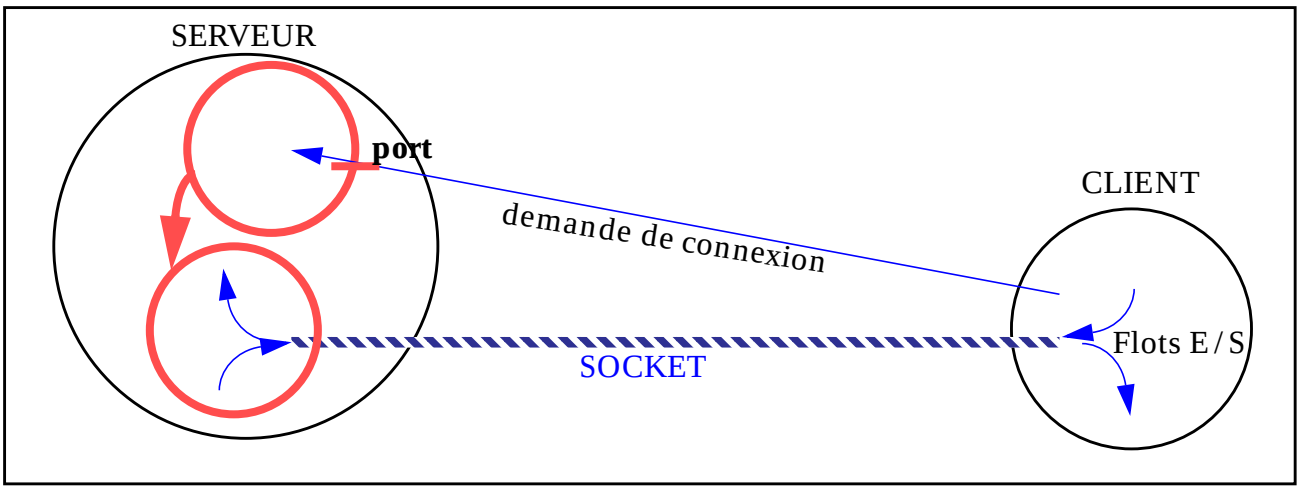
Les objets transférés doivent déclarer la propriété “linéarisable” (`implements Serializable`).

Les mécanismes standard permettent de contrôler la manière dont les objets sont transférés (non-transfert de certaines variables membres, détection des envois multiples de la même instance,...)

Il est également possible de personnaliser la manière dont les instances sont envoyées et reconstituées.

java.net

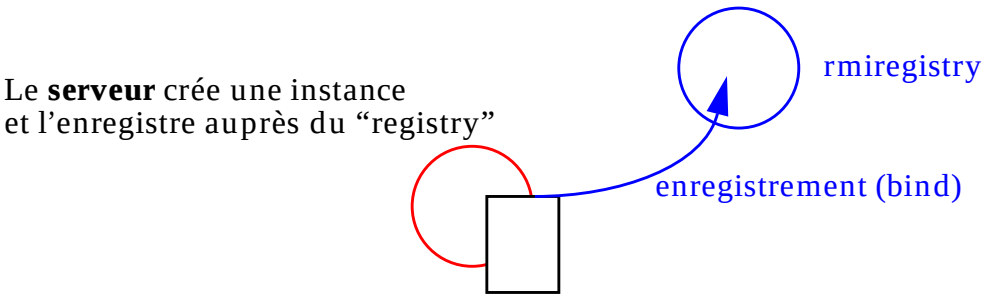
La librairie permet de réaliser des échanges entre des sites distant sur le réseau.

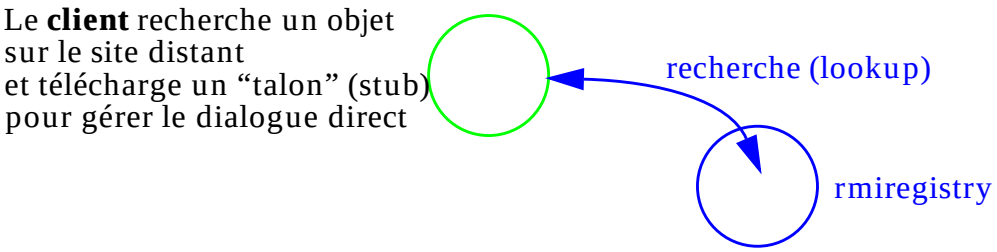


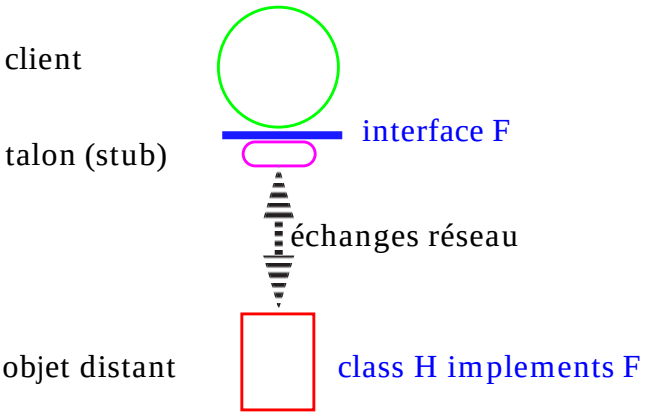
R.M.I

Mécanismes d'appel de méthodes sur des objets distants. Permettent de définir et de mettre en oeuvre des échanges client/serveur entre des programmes Java.

Coté serveur un programme *rmiregistry* sert d'intermédiaire pour permettre au client de rechercher un objet, de télécharger un protocole de communication avec un objet, etc.





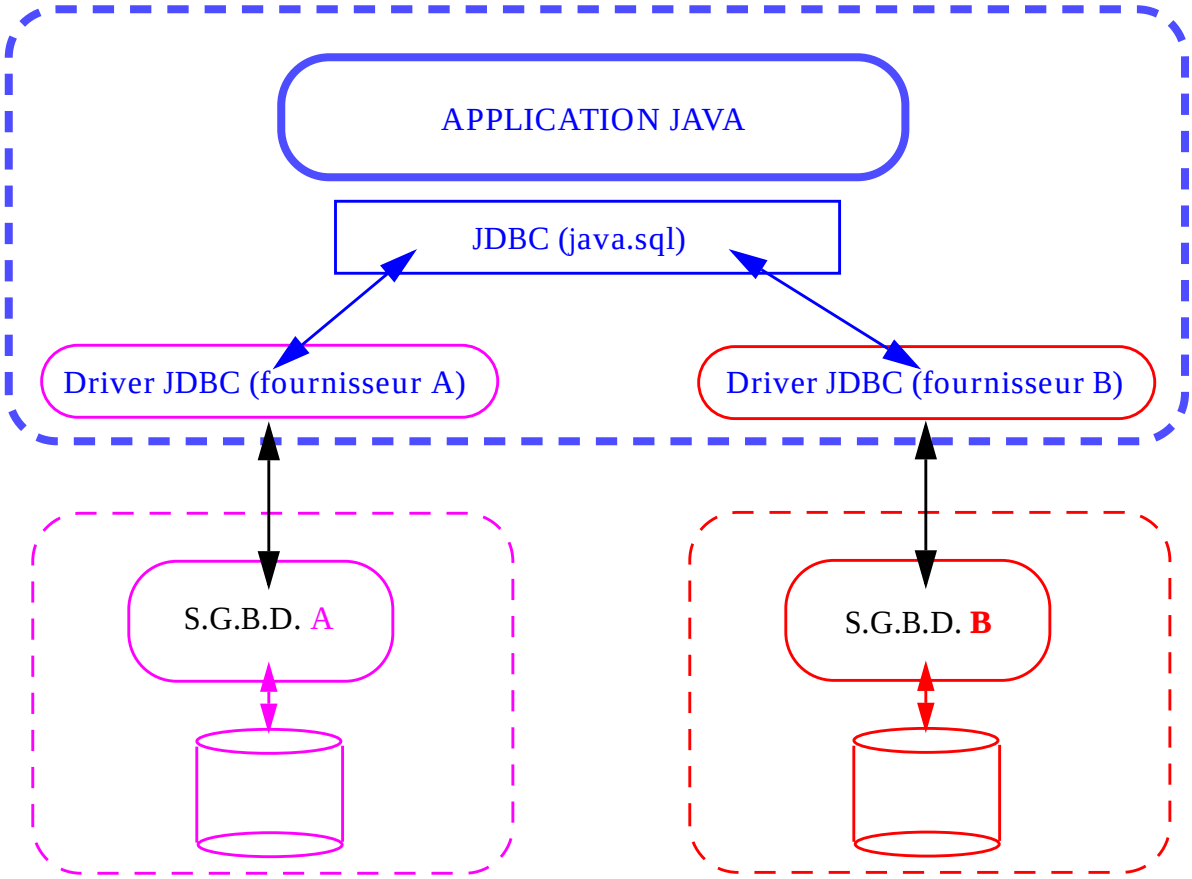


Un autre "démon" (processus système de veille) nommé *rmid* permet d'activer à la demande des objets serveurs qui peuvent être "dormants" (non présents au sein d'une JVM active).

J.D.B.C

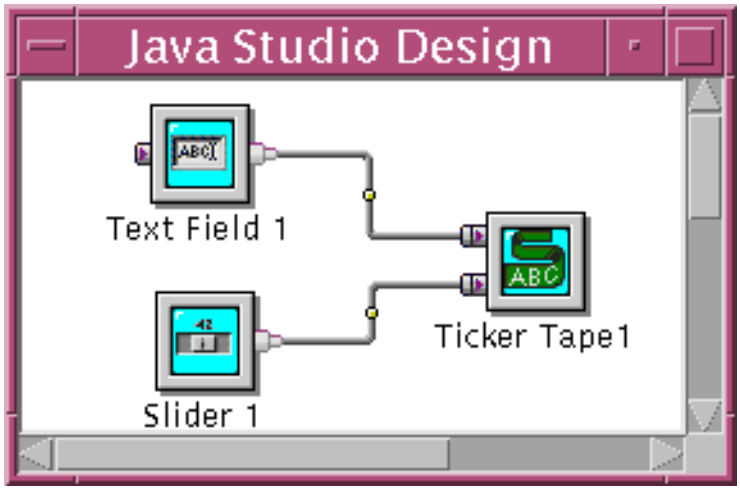
Le package java.sql

Définit essentiellement des interfaces d'accès à des bases de données SQL. Des produits spécifiques (hors JDK) sont chargés d'implanter ces interfaces (drivers JDBC).



Beans

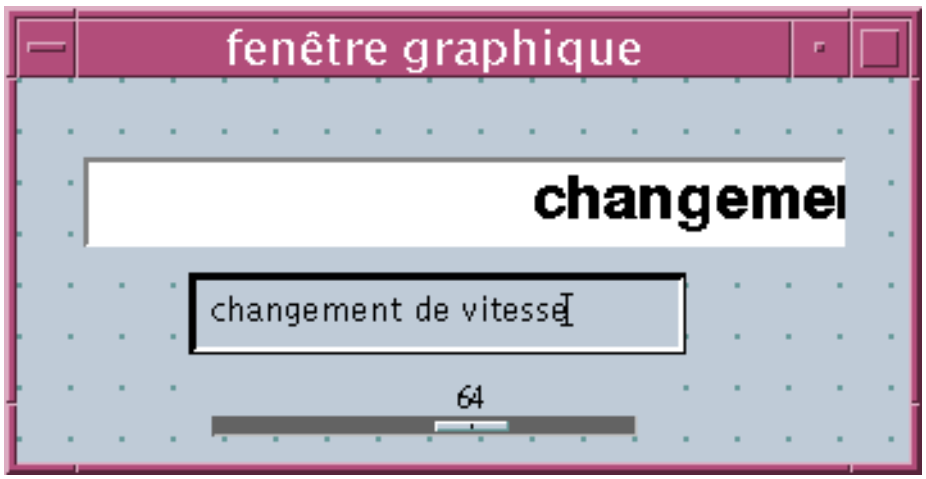
Les “JavaBeans” sont des composants réutilisables destinés à être assemblés dynamiquement (par exemple par un outil interactif “bean builder”).



JAVA STUDIO
un ex. de “bean builder”

assemblage logique des beans

assemblage résultant



La spécification JavaBean permet d’écrire des classes Java selon des conventions qui permettent la “découverte” de certaines capacités (production/consommation de données, etc.).

Un environnement de manipulation de Beans permet donc d'inspecter les capacités d'un classe (*introspection*), de personnaliser un composant en jouant sur ses paramètres, d'assembler des composants en les faisant communiquer aux moyen d'événements, de sauvegarder et de ranimer des instances, et de créer donc de nouveaux composants à partir de composants existants.

Le package **java.beans.beancontext** permet de définir des "containers" qui constituent un environnement d'exécution pour des beans.

`org.omg.CORBA`:

Mécanismes d'appels d'objets distants entre des réalisations hétérogènes (on n'a pas besoin d'avoir un client ET un serveur Java). S'appuie sur le standard CORBA.

Il est fourni une réalisation minimale d'un ORB (Object Request Broker) constitué d'un service de nom: *tnameserv.*

Un autre utilitaire (*idltojava*) permet de générer automatiquement du code à partir de descriptions en langage IDL .

Des packages particuliers (*javax.rmi..*) permettent de réaliser une partie des services RMI en se plaçant dans le monde CORBA.

`javax.servlet`:

Ce package ne fait pas partie du "noyau" standard. Il est destiné à être installé avec des serveurs HTTP pour permettre un traitement des requêtes CGI (dans le cas des **HttpServlet**s) .

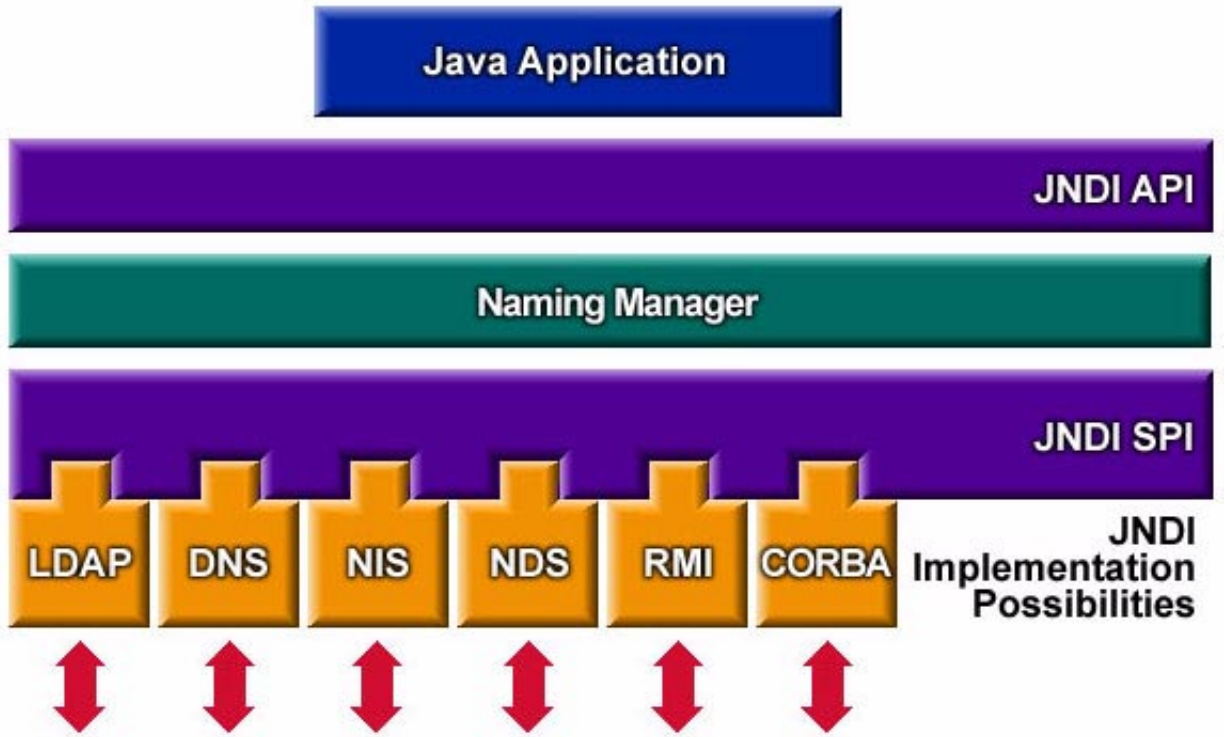
Les classes de cette librairie offrent des facilités pour décoder les arguments CGI, pour préparer le document en réponse, pour gérer des clients en parallèle, organiser des sessions, sauvegarder et restaurer leur état, etc.

javax.naming:

(dans le noyau à partir de Java1.3).

Unification des services d'annuaire : permet de rechercher des objets ou des références de ressources répertoriées par des mécanimes d'annuaire de natures très différentes.

Les services concrets de l'API de haut niveau offerte par J.N.D.I sont rendus au travers de "Service Provider Interfaces" (S.P.I). En standard les services de recherche d'objets RMI et CORBA et les services d'accès à l'annuaire généralisé (X500) de LDAP sont implantés dans le SDK.



Java et le Web: les applets 16

Points essentiels

- Les Applets constituents des petites applications Java hébergées au sein d'une page HTML, leur code est téléchargé par le navigateur.

- Une Applet est, à la base, un panneau graphique. Un protocole particulier la lie au navigateur qui la met en oeuvre (cycle de vie de l'Applet).

- Les codes qui s'exécutent au sein d'une Applet sont soumis à des restrictions de sécurité.

Applets

Une *Appliquette* (*Applet*) est une portion de code Java qui s'exécute dans l'environnement d'un navigateur au sein d'une page HTML. Elle diffère d'une application par son mode de lancement et son contexte d'exécution.

Une application autonome est associée au lancement d'un processus JVM qui invoque la méthode `main` d'une classe de démarrage.

Une JVM associée à un navigateur peut gérer plusieurs Applets, gérer leur contexte (éventuellement différents) et gérer leur "cycle de vie" (initialisation, phases d'activité, fin de vie).

Lancement d'une Applet

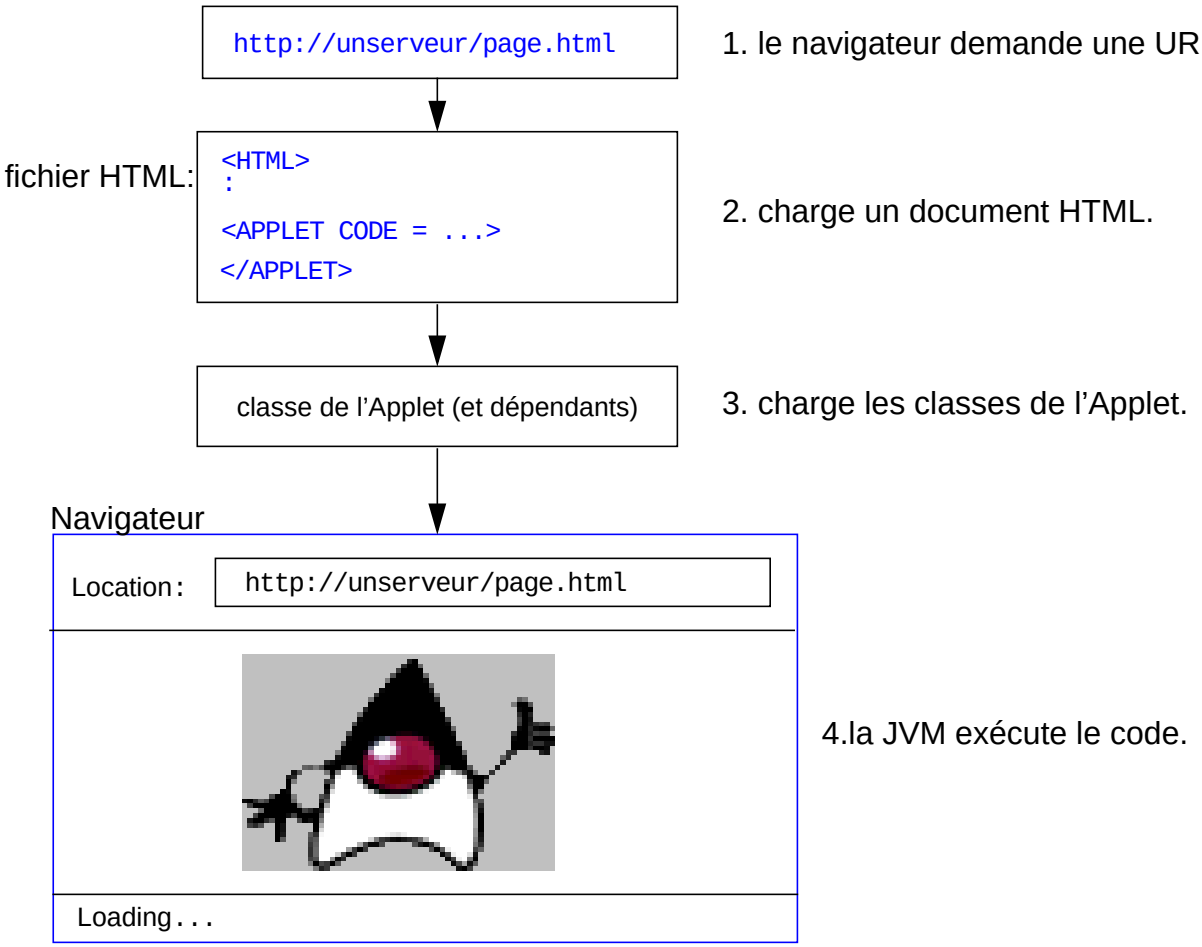
Une applet s'exécutant dans le cadre d'une "page" HTML, la requête de lancement et les paramètres associés sont en fait contenus dans le fichier source décrivant la page.

Le langage HTML (voir `http://www.w3.org/MarkUp`) permet de définir une présentation à partir d'un texte contenant des *balises* (qui sont des instructions de mise en page). Certaines de ces balises demandent le chargement de données non-textuelles comme des images ou des Applets Java.

Le lancement d'une Applet suppose donc :

- La désignation d'un document au navigateur au travers d'une adresse URL (voir `http://www.w3.org/Adressing`)
- Le chargement et la mise en page du document HTML associé
- Le chargement du code de l'Applet spécifié dans le document (et éventuellement le chargement des codes des classes distantes appelées par l'Applet).
- La gestion, par la JVM du navigateur, du cycle de vie de l'Applet au sein du document mis en page.

Applets: lancement



Exemple d'utilisation de la balise APPLET :

```
<P> et maintenant notre Applet :

<APPLET code="fr.gibis.applets.MonApplet.class"
        width=100 height=100>
</APPLET>
```

Applets: restrictions de sécurité

L'applet est chargée par un `ClassLoader` particulier. On a un contexte d'exécution lié à ce ClassLoader et associé au site distant dont est originaire l'Applet.

Dans ce contexte on va exécuter des codes de classe : classses "distantes" chargées par le ClassLoader et classes "système" de la librairie de la J.V.M locale. Pour éviter qu'un code distant puisse réaliser des opérations contraires à une politique de sécurité élémentaire, des règles par défaut s'appliquent (*sandbox security policy*):

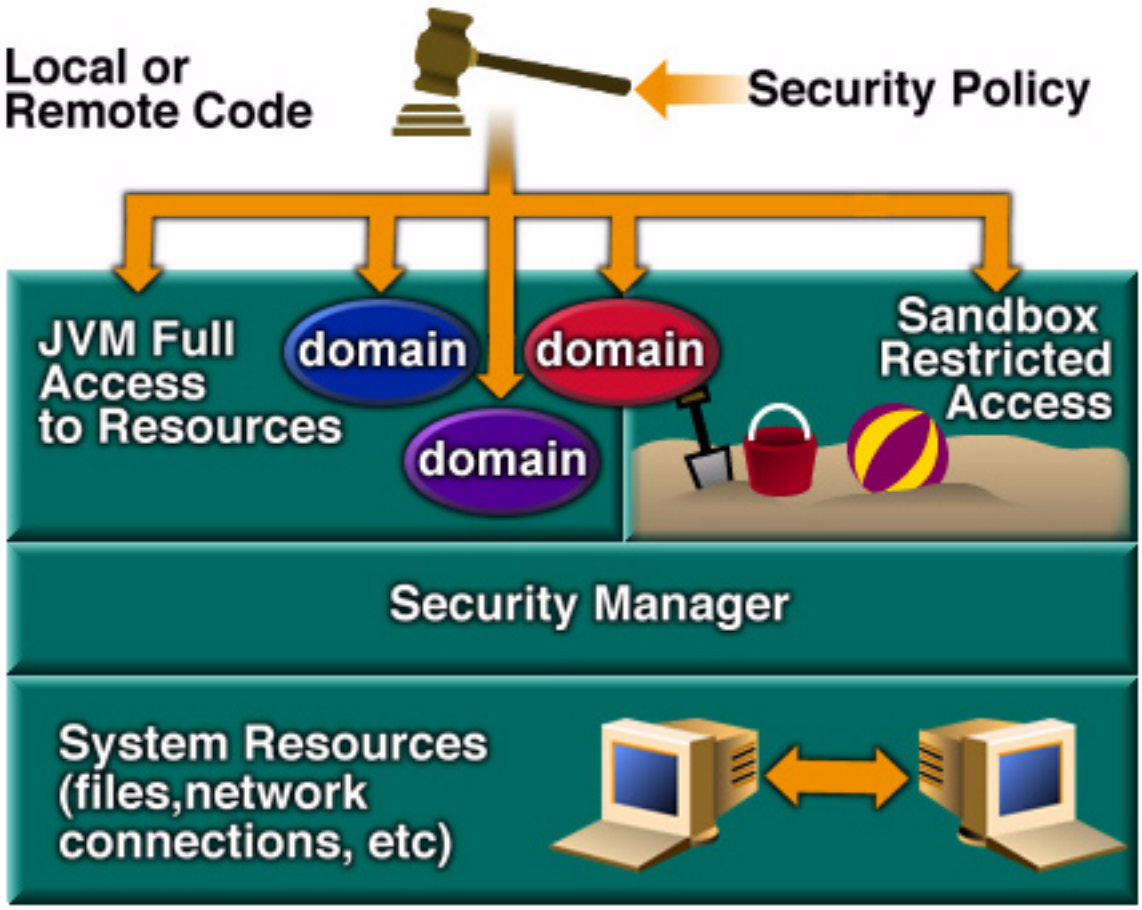
- L'Applet ne peut obtenir des informations sur le système courant (hormis quelques informations élémentaires comme la nature du système d'exploitation, le type de J.V.M., etc.).
- L'Applet ne peut connaître le système de fichiers local (et donc ne peut réaliser d'entrée/sortie sur des fichiers).
- L'Applet ne peut pas lancer de processus
- Les communications sur réseau (par *socket*) ne peuvent être établies qu'avec le site d'origine du code de l'Applet.

Bien entendu un code d'Applet ne peut pas contenir de code "natif" (par essence non portable).

A partir de la version 2 de Java un système standard de mise en place de domaines de sécurité (*Protection Domain*) permet d'assouplir ces règles pour permettre certaines opérations à des codes dûment authentifiés venus de sites connus. On peut ainsi imaginer que l'Applet qui vous permet de consulter votre compte en banque vous permet aussi de rapatrier des données à un endroit du système de fichier défini par vous.

Applets: restrictions de sécurité

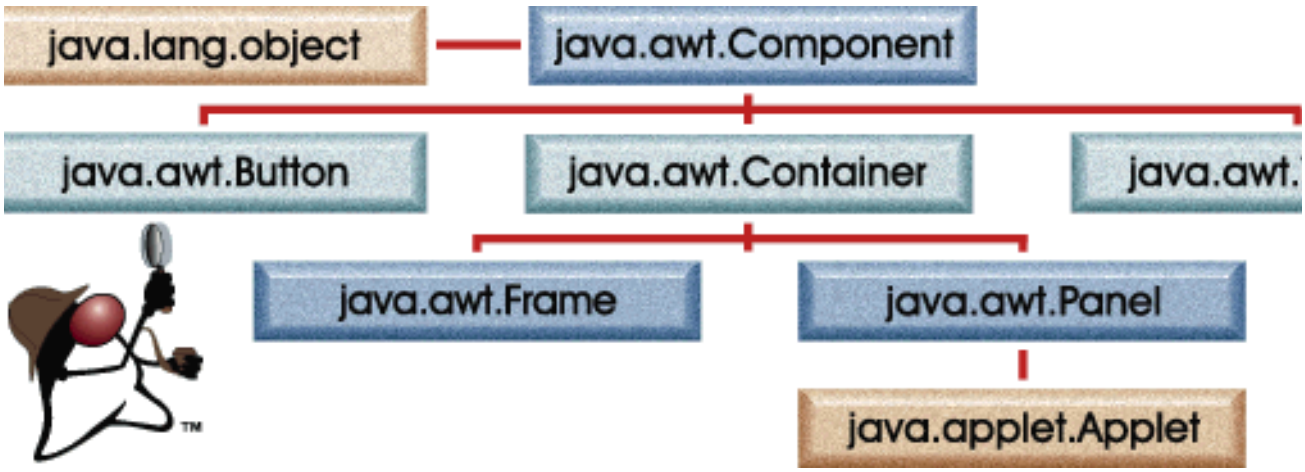
Le modèle de sécurité Java 2 s'applique dès qu'un `SecurityManager` est présent et ce aussi bien pour du code téléchargé que pour du code local



Hiérarchie de la classe Applet

De par l'incorporation d'une Applet à une présentation graphique (la "page" du navigateur) toute Applet est, de manière inhérente, une classe graphique (même si on peut créer des Applets qui n'ont aucune action de nature graphique). A la base l'incorporation d'une Applet dans une page HTML revient à signaler au navigateur qu'une zone de dimensions données n'est plus gérée directement par l'interpréteur de HTML mais par un programme autonome qui prend le contrôle de la représentation graphique de cette zone.

Toute classe qui est une Applet doit dériver de `java.applet.Applet` qui est elle-même une classe dérivée de `java.awt.Panel` qui représente un "panneau" graphique dans la librairie graphique AWT.



Applets: groupes de méthodes

Une Applet étant un `Container` graphique, elle hérite des méthodes qui permettent de disposer des composants graphiques dans la zone gérée par la classe. Ce point sera vu dans des chapitres ultérieurs, et, pour l'instant, nous allons nous attacher à trois groupes de méthodes:

- Méthodes qui sont appelées par le système graphique pour la notification d'une demande de rafraîchissement d'une zone de l'écran:
  `repaint(), update(graphics), paint(Graphics)`
  Dans le cas où l'Applet souhaite gérer le graphique de bas niveau (au lieu de laisser agir le système automatique attachés à des composant AWT de haut niveau) elle doit conserver un modèle logique des éléments graphiques qu'elle gère et être capable de les redessiner à la demande.

- Méthodes spécifiques aux Applets et qui leur permettent de demander au navigateur des informations ou des actions liées au contexte: informations sur la page HTML courante, chargement d'une ressource sur le site d'origine de la page, etc.

- Méthodes spécifiques aux Applets et à leur cycle de vie : le navigateur doit pouvoir notifier à l'Applet qu'il veut l'initialiser (`init()`), qu'il veut la rendre active ou inactive (`start(), stop()`), ou qu'il veut la "détruire" (`destroy()`).
  Par défaut ces méthodes ne font rien et il faut en redéfinir certaines d'entre elles pour obtenir un comportement de l'Applet.

H.T.M.L.: la balise `Applet`

Syntaxe des balises HTML :

```
<APPLET
        [archive=ListeArchives]
        code=package.NomApplet.class
         width=pixels height=pixels
        [codebase=codebaseURL]
        [alt=TexteAlternatif]
        [name=nomInstance]
        [align=alignement]
        [vspace=pixels]  [hspace=pixels]
>
  [<PARAM name=Attribut1 value=valeur>]
  [<PARAM name=Attribut2 value=valeur>]
  . . . . .
  [HTMLalternatif]
</APPLET>
```

Exemple :

```
<APPLET
        code=fr.acme.MonApplet.class
        width=300 height=400
>
</APPLET>
```

Evolutions (HTML 4) :

```
<OBJECT codetype=”application/java”
        classid="fr.acme.MonApplet.class"
        width=300 height=400>
        >
</OBJECT>
```

HTML: la balise `Applet`

. La balise HTML APPLET comporte les attributs suivants :

- code=*appletFile.class* - Cet attribut *obligatoire* fournit le nom du fichier contenant la classe compilée de l'applet (dérivée de `java.applet.Applet`). Son format pourrait également être *aPackage.appletFile.class.*

  Note – La localisation de ce fichier est relative à l'URL de base du fichier HTML de chargement de l'applet.

- width=*pixels* height=*pixels* - Ces attributs *obligatoires* fournissent la largeur et la hauteur initiales (en pixels) de la zone d'affichage de l'applet, sans compter les éventuelles fenêtres ou boîtes de dialogue affichées par l'Applet.

- codebase=*codebaseURL* - Cet attribut facultatif indique l'URL de base de l'applet : le répertoire contenant le code de l'applet. Si cet attribut n'est pas précisé, c'est l'URL du document qui est utilisé.

- name=*appletInstanceName* -- Cet attribut, facultatif, fournit un nom pour l'instance de l'applet et permet de ce fait aux applets situées sur la même page de se rechercher mutuellement (et de communiquer entre-elles).

- archive=*ListeArchives* permet de spécifier une liste de fichiers archive .jar contenant les classes exécutables et, éventuellement des ressources. Les noms des archives sont séparés par des virgules. Ce point ne sera pas abordé dans ce cours.

- object=*objectFile.ser* permet de spécifier une instance d'objet à charger. Ce point ne sera pas abordé dans ce cours.

`<param name=`*appletAttribute1* `value=`*value*`>` -- Ces éléments permettent de spécifier un paramètre à l'applet. Les applets accèdent à leurs paramètres par la méthode `getParameter()`.

Méthodes du système graphique de bas niveau

Applet héritant de Panel il est possible de modifier l'aspect en utilisant les méthodes graphiques des composants.

Voici un exemple minimal d'Applet qui écrit du texte de manière graphique:

```
package fr.gibis.applets ;
import java.awt.* ;
import java.applet.Applet ;

public class HelloWorld extends Applet {

        public void paint(Graphics gr) {
                gr.drawString("Hello World?", 25 ,25) ;
        }
}
```

Les arguments numériques de la méthode drawString() sont les coordonnées x et y du début du texte. Ces coordonnées font référence à la "ligne de base" de la police. Mettre la coordonnée y à zero aurait fait disparaître la majeure partie du texte en haut de l'affichage (à l'exception des parties descendantes des lettres comme "p", "q" etc.)

Méthodes d'accès aux ressources de l'environnement

L'applet peut demander des ressources (généralement situées sur le réseau). Ces ressources sont désignées par des URLs (classe `java.net.URL`). Deux URL de référence sont importantes :

- l'URL du document HTML qui contient la description de la page courante. Cette URL est obtenue par `getDocumentBase()` .
- l'URL du code "racine" de l'Applet (celui qui est décrit par l'attribut "code"). Cette URL est obtenue par `getCodeBase()` .

En utilisant une de ces URLs comme point de base on peut demander des ressources comme des images ou des sons :

- `getImage(URL base, String désignation)` : permet d'aller rechercher une image; rend une instance de la classe **`Image`**.
- `getAudioClip(URL base, String désignation)` : permet d'aller rechercher un son; rend une instance de la classe **`AudioClip`**.

Les désignations de ressource par rapport à une URL de base peuvent comprendre des cheminoms relatifs (par ex: "../../images/truc.gif"). Attention toutefois : certaines configurations n'autorisent pas des remontées dans la hiérarchie des répertoires.

Le moteur son de la plateforme Java2 sait traiter des fichiers .wav, .aiff et .au ainsi que des ressources MIDI. Une nouvelle méthode `newAudioClip(URL)` permet de charger un AudioClip.

La méthode `getParameter(String nom)` permet de récupérer, dans le fichier source HTML de la page courante, la valeur d'un des éléments de l'applet courante, décrit par une balise `<PARAM>` et ayant l'attribut `name=`*nom* . Cette valeur est une chaîne `String`.

Méthodes d'accès aux ressources de l'environnement

Utilisation de la récupération de paramètres. Exemple de source HTML:

```
<APPLET code="fr.gibis.graph.Dessin.class" width=200 height=200>
        <PARAM name="image" value="duke.gif">
</APPLET>
```

Le code Java correspondant :

```
package fr.gibis.graph ;
import java.awt.*;
import java.applet.Applet;

public class Dessin extends Applet {
        Image img ;

        public void init() {// redef. méthode standard cycle vie
                String nomImage = getParameter("image") ;
                if ( null != nomImage) {
                        img = getImage(getDocumentBase(),
                                nomImage);
                }
        }

        public void paint (Graphics gr) {
                if( img != null) {
                        gr.drawImage(img,50,50, this) ;
                } else {
                        gr.drawString("image non chargée",
                                25,25) ;
                }
        }
}
```

Les méthodes de chargement de media comme `getImage()` sont des méthodes *asynchrones*. On revient de l'appel de la méthode alors que le chargement est en cours (et, possiblement, non terminé). Il est possible que `paint()` soit appelé plusieurs fois au fur et à mesure que l'image devient complètement disponible.

Méthodes du cycle de vie

`init()` : Cette méthode est appelée au moment où l'applet est créée et chargée pour la première fois dans un navigateur activé par Java (comme AppletViewer). L'applet peut utiliser cette méthode pour initialiser les valeurs des données. Cette méthode n'est pas appelée chaque fois que le navigateur ouvre la page contenant l'applet, mais seulement la première fois juste après le changement de l'applet.

La méthode `start()` est appelée pour indiquer que l'applet doit être "activée". Cette situation se produit au démarrage de l'applet, une fois la méthode `init()` terminée. Elle se produit également lorsque le navigateur est restauré après avoir été iconisé ou lorsque la page qui l'héberge redevient la page courante du navigateur. Cela signifie que l'applet peut utiliser cette méthode pour effectuer des tâches comme démarrer une animation ou jouer des sons.

```
public void start() {
        musicClip.play();
}
```

La méthode `stop()` est appelée lorsque l'applet cesse d'être en phase active. Cette situation se produit lorsque le navigateur est icônisé ou lorsque le navigateur présente une autre page que la page courante. L'applet peut utiliser cette méthode pour effectuer des tâches telles que l'arrêt d'une animation.

```
public void stop() {
        musicClip.stop();
}
```

Les méthodes `start()` et `stop()` forment en fait une paire, de sorte que `start()` peut servir à déclencher un comportement dans l'applet et `stop()` à désactiver ce comportement.

`destroy()` : Cette méthode est appelée avant que l'objet applet ne soit détruit c.a.d enlevé du cache du navigateur.

Méthodes du cycle de vie

```
package fr.gibis.applets;
// Suppose l'existence du fichier son "cuckoo.au"
// dans le répertoire "sounds" situé dans
// le répertoire du fichier HTML d'origine

import java.awt.Graphics;
import java.applet.*;

public class HwLoop extends Applet {
        AudioClip sound;

        public void init() {
                sound = getAudioClip(getDocumentBase(),
                        "sounds/cuckoo.au");
                // attention syntaxe d'URL!!!
        }

        public void paint(Graphics gr) {
                // méthode de dessin de java.awt.Graphics
                gr.drawString("Audio Test", 25, 25);
        }

        public void start () {
                sound.loop();
        }

        public void stop() {
                sound.stop();
        }
}
```

Exercices :

En règle générale une Applet est exécutée au sein d'un navigateur comme Netscape Navigator. Néanmoins pour simplifier et accélérer le développement votre SDK est fourni avec un outil simple conçu pour ne visualiser que les Applets.

Cet outil est appletviewer et vous pouvez le lancer en lui passant un fichier HTML simplifié par la commande :

```
appletviewer url_ou_fichier
```

Attention: Appletviewer a un comportement hybride car il simule à la fois le comportement d'un navigateur et celui d'un serveur. Il s'ensuit que si le document à charger est accédé par une URL désignant un document local (et non une URL "distante" de type "http:...") il risque de considérer que les ressources accédées doivent être soumises aux contrôles de sécurité. Dans ce cas, pour éviter d'avoir à modifier les autorisations de sécurité, faire le test avec `getCodebase()` plutôt qu'avec `getDocumentBase()`.

Exercice * un message en couleur :

Réaliser une Applet qui affiche un rectangle plein de couleur rouge de largeur 200, de hauteur 50 en coordonnées (50,25). Afficher dans ce rectangle un message de couleur verte "BONJOUR DE MON APPLET".

Consulter la documentation pour pouvoir utiliser les méthodes de la classe Graphics :

```
fillRect(...)
setColor(...)
drawString(...)
```

générer un fichier html associé prévoyant pour l'Applet un emplacement de 300 de largeur et de 100 de hauteur

Exercice * image et son:

Ecrire une Applet qui recoive en paramètre :

- La référence d'une image à charger (et à afficher).
- La référence d'un son à charger.

Ce son doit démarrer au moment du démarrage de l'applet et doit s'arréter si on change de page.

Les composants AWT 17

Contenu

Cette annexe donne un aperçu sur les composants AWT courants et sur leur manipulation.

Button

C’est un composant d’interface utilisateur de base de type "appuyer pour activer". Il peut être construit avec un étiquetage texte précisant son rôle .

```
Button bt = new Button(“Sample”);
bt.addActionListener(....);
```

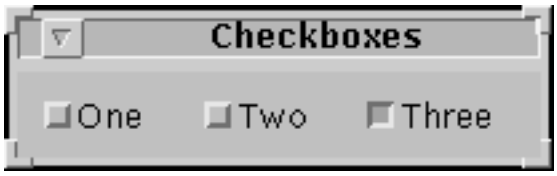
L’interface `ActionListener` doit pouvoir traiter un clic d’un bouton de souris. La méthode `getActionCommand()` de l’événement action (`ActionEvent`) activé lorsqu’on appuie sur le bouton rend par défaut la chaîne d’étiquetage (pour une meilleure internationalisation récupérer plutôt une chaîne positionnée par `setActionCommand()`)

Pour des besoins de repérage programmatique une chaîne symbolique peut-être rattachée à tout composant pour l’identifier: utiliser `setName()` et `getName()`.

Checkbox

La case à cocher fournit un dispositif d'entrée "actif/inactif" accompagné d'une étiquette texte.

```
Checkbox one = new Checkbox("One", false);
Checkbox two = new Checkbox("Two", false);
Checkbox three = new Checkbox("Three", true);
one.addItemListener(new Handler());
two.addItemListener(new Handler());
three.addItemListener(new Handler());
```



La sélection ou désélection d'une case à cocher est notifiée à un objet rélaisant l'interface `ItemListener`. Pour détecter une opération de sélection ou de déselection, il faut utiliser la méthode `getStateChange()` sur l'objet `ItemEvent`. Cette méthode renvoie l'une des constantes ItemEvent.`DESELECTED` ou `ItemEvent.SELECTED`, selon le cas. La méthode `getItem()` renvoie un objet de type chaîne (`String`) qui représente la chaîne de l'étiquette de la case à cocher considérée.

```
class Handler implements ItemListener {
        public void itemStateChanged(ItemEvent ev) {
                String state = "deselected";
                if (ev.getStateChange() == ItemEvent.SELECTED){
                        state = "selected";
                }
                System.out.println(ev.getItem() + " " + state);
        }
}
```

CheckboxGroup

On peut créer des cases à cocher à l'aide d'un constructeur spécial qui utilise un argument supplémentaire de type CheckboxGroup. Si on procéde ainsi, l'aspect des cases à cocher est modifié et toutes les cases à cocher liées au même groupe adoptent un comportement de "bouton radio".

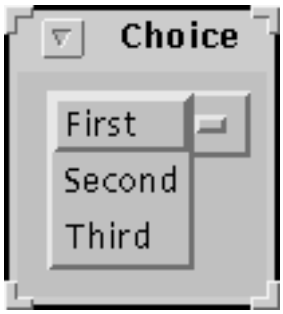
```
CheckboxGroup cbg = new CheckboxGroup();
Checkbox one = new Checkbox("One", cbg, false);
Checkbox two = new Checkbox("Two", cbg, false);
Checkbox three = new Checkbox("Three", cbg, true);
```

Choice

Le composant `Choice` fournit une outil simple de saisie de type "sélectionner un élément dans cette liste".

```
 Choice c = new Choice();
c.addItem("First");
c.addItem("Second");
c.addItem("Third");
c.addItemListener(. . .);
```

Lorsqu'un composant `Choice` est activé il affiche la liste des éléments qui lui ont été ajoutés. Notez que les éléments ajoutés sont des objets de type chaîne (`String`).



L'interface `ItemListener` sert à observer les modifications de ce choix. Les détails sont les mêmes que pour la case à cocher. La méthode `getSelectedIndex()` de Choice permet de connaître l'index selectionné.

List

Une liste permet de présenter à l'utilisateur des options de texte affichées dans une zone où plusieurs éléments peuvent être visualisés simultanément. Il est possible de naviguer dans la liste et d'y sélectionner un ou plusieurs éléments simultanément (mode de sélection simple ou multiple).

```
List l = new List(4, true);
```

L'argument numérique transmis au constructeur définit le nombre d'items visibles. L'argument booléen indique si la liste doit permettre à l'utilisateur d'effectuer des sélections multiples.

Un `ActionEvent`, géré par l'intermédiaire de l'interface `ActionListener`, est généré par la liste dans les modes de sélection simple et multiple. Les éléments sont sélectionnés dans la liste conformément aux conventions de la plate-forme. Pour un environnement Unix / Motif, cela signifie qu'un simple clic met en valeur une entrée dans la liste, mais qu'un double-clic déclenche l'action correspondante.

Récupération: voir méthodes `getSelectedObjects()`, `getSelectedItems()`

Label

Un label affiche une seule ligne de texte. Le programme peut modifier le texte. Aucune bordure ou autre décoration particulière n'est utilisée pour délimiter un label.

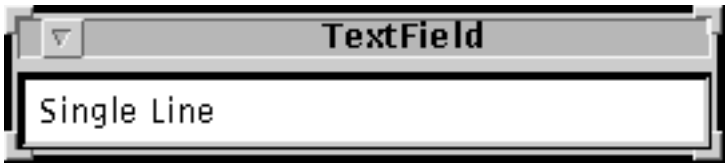
```
Label lab = new Label("Hello");
```

En général, on ne s'attend pas à ce que les `Labels` traitent des événements, pourtant on peut effectuer cette opération de la même façon que pour un `Canvas` (ou une classe utlisateur dérivée de Component). Dans ce cas, on ne peut capter les événements clavier de façon fiable qu'en faisant appel à `requestFocus()`.

TextField

Le `TextField` est un dispositif de saisie de texte sur une seule ligne.

```
TextField tf = new TextField(“Single line”, 30);
```



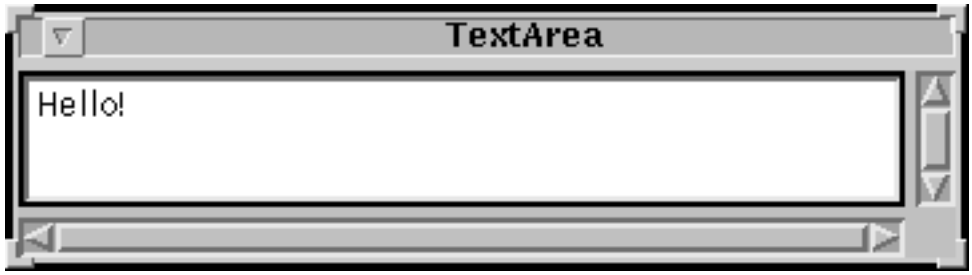
Du fait qu’une seule ligne soit possible, un veilleur d’événement Action (`ActionListener`) peut être informé, lorsque la touche <Entrée> ou <Retour> est activée.

Le champ texte peut être en lecture seule. Certains constructeurs prennent en argument un nombre de caractères visibles. Un TextField n’affiche pas de barres de défilement dans l’une ou l’autre direction mais permet, si besoin est, un défilement de gauche à droite d’un texte trop long.

TextArea

Le `TextArea` est un dispositif de saisie de texte multi-lignes, multi-colonnes. On peut le rendre non éditable par l'intermédiaire de la méthode `setEditable(boolean)`. Il affiche des barres de défilement horizontales et verticales.

```
TextArea tx = new TextArea("Hello!", 4, 30);
;
```



On peut ajouter des veilleurs d'événements de divers type sur un `TextArea`.

Le texte étant multi-lignes, le fait d'appuyer sur <Entrée> place seulement un autre caractère dans la mémoire tampon. Si on a besoin de savoir à quel moment une saisie est terminée, on peut placer un bouton de validation à côté d'un TextArea pour permettre à l'utilisateur de fournir cette information.

Un veilleur `KeyListener` permet de traiter chaque caractère entré en association avec la méthode `getKeyChar()`, `getKeyCode()` de la classe `KeyEvent`.

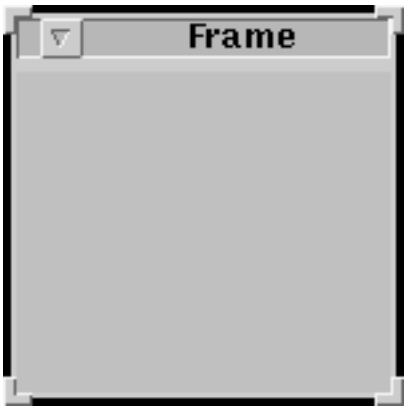
TextComponent

La classe `TextComponent` dont dérivent `TextField` et `TextArea` fourni un grand nombre de méthodes.On a vu que les constructeurs des classes `TextArea` et `TextField` permettent de définir un nombre de colonnes pour l'affichage. Le nombre de colonnes est interprété en fonction de la largeur moyenne des caractères dans la police utilisée. Le nombre de caractères effectivement affichés peut varier radicalement en cas d'utilisation d'une police à chasse proportionnellel.

Frame

C’est la fenêtre générale de "plus haut niveau". Elle possède des attributs tels que : barre de titre et zones de contrôle du redimensionnement.

```
Frame fr = new Frame(“Frame”);
```



La taille d’un `Frame` peut être définie à l’aide de la méthode `setSize()` ou avec la méthode `pack()`. Dans ce cas le gestionnaire de disposition calcule une taille englobant tous les composants du `Frame` et définit la taille de ce dernier en conséquence.

Les événements du Frame peuvent être surveillés à l’aide de tous les gestionnaires d’événements applicables aux composants généraux. `WindowListener` peut être utilisé pour réagir, via la méthode `windowClosing()`, lorsque le bouton Quit a été activé dans le menu du gestionnaire de fenêtres.

Il n’est pas conseillé d’écouter des événements clavier directement à partir d’un `Frame`. Bien que la technique décrite pour les composants de type `Canvas` et `Label`, à savoir l’appel de `requestFocus()`, fonctionne parfois, elle n’est pas fiable. Si on a besoin de suivre des événements clavier, il est plutôt recommandé d’ajouter au `Frame` un `Canvas`, `Panel`, etc., et d’associer le gestionnaire d’événement à ce dernier.

Dialog

Un `Dialog` est une fenêtre (qui, comme `Frame`, hérite de `Window`) elle diffère toutefois d'un `Frame` :

- elle est destinée à afficher des messsages
- elle peut être modale: elle recevra systématiquement toutes les saisies jusqu'à fermeture.


```
Dialog dlg = new Dialog(fen, "Dialog", false);
dlg.add(new Label("Hello, I'm a Dialog"),
                        BorderLayout.CENTER);
dlg.pack();
```

Un dialog dépend d'une Frame : cette Frame apparait comme premier argument dans les constructeurs de la classe Dialog.

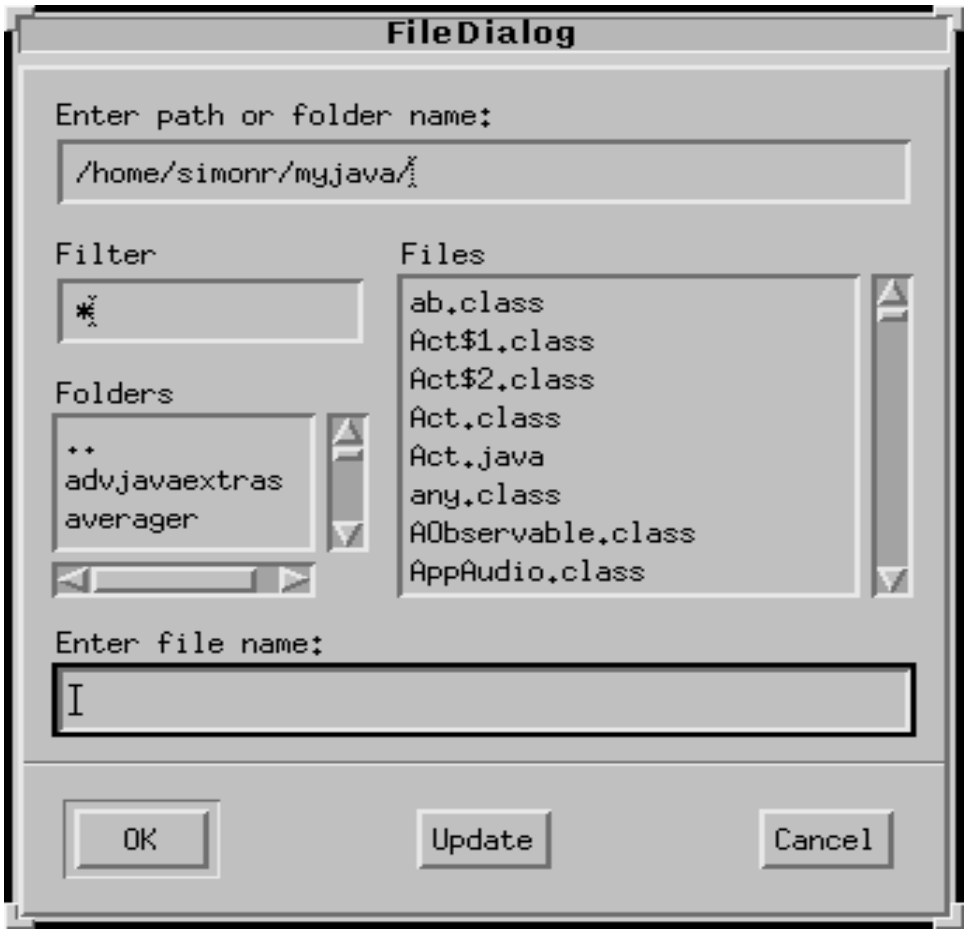
Les boîtes de dialogue ne sont pas visibles lors de leur création (utiliser `setVisible(true)`). Elles s'affichent plutôt en réponse à une autre action au sein de l'interface utilisateur, comme le fait d'appuyer sur un bouton.

Il est recommandé de considérer une boîte de dialogue comme un dispositif réutilisable. Ainsi, vous ne devez pas détruire l'objet individuel lorsqu'il est effacé de l'écran, mais le conserver pour une réutilisation ultérieure.

FileDialog

C'est une implantation d'un dispositif de sélection de fichier. Elle comporte sa propre fenêtre autonome et permet à l'utilisateur de parcourir le système de fichiers et de sélectionner un fichier spécifique pour des opérations ultérieures.

```
FileDialog d = new FileDialog(f, "FileDialog");
d.setVisible(true);
String fname = d.getFile();
```



En général, il n'est pas nécessaire de gérer des événements à partir de la boîte de dialogue de fichiers. L'appel de `setVisible(true)` se bloque jusqu'à ce que l'utilisateur sélectionne OK. Le fichier sélectionné est renvoyé sous forme de chaîne .Voir `getDirectory(), getFile()`

Panel

C'est le conteneur de base. Il ne peut pas être utilisé de façon isolée comme les Frames, les fenêtres et les boîtes de dialogue.

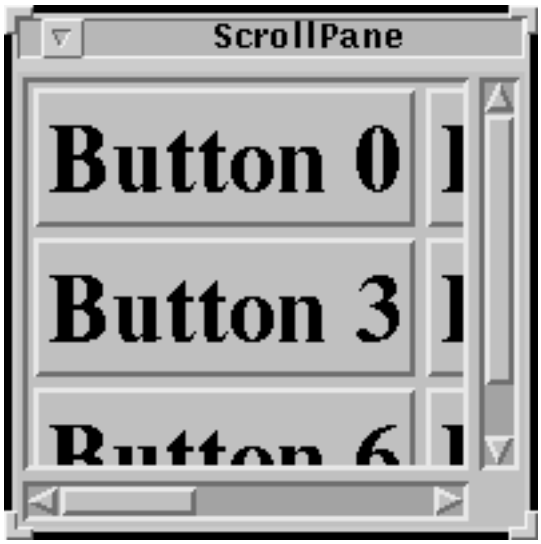
```
Panel p = new Panel();
```

Les Panels peuvent gérer les événements (rappel : le focus clavier doit être demandé explicitement).

ScrollPane

Fournit un conteneur général ne pouvant pas être utilisé de façon autonome. Il fournit une vue sur une zone plus large et des barres de défilement pour manipuler cette vue.

```
Frame fr = new Frame("ScrollPane");
Panel pan = new Panel();
ScrollPane sp = new ScrollPane();
pan.setLayout(new GridLayout(3, 4));
....
sp.add(pan);
fen.add(sp);
fen.setSize(200, 200);
fen.setVisible(true);
```



Le `ScrollPane` crée et gère les barres de défilement selon les besoins. Il contient un seul composant et on ne peut pas influer sur le gestionnaire de disposition qu'il utilise. Au lieu de cela, on doit lui ajouter un Panel, configurer le gestionnaire de disposition de ce Panel et placer les composants à l'intérieur de ce dernier.

En général, on ne gère pas d'événements dans un `ScrollPane`, mais on le fait dans les composants qu'il contient.

Canvas

Un canvas fournit un espace vide (arrière-plan coloré). Sa taille par défaut étant zéro par zéro, on doit généralement s'assurer que le gestionnaire de disposition lui affectera une taille non nulle.

Cet espace peut être utilisé pour dessiner, recevoir du texte ou des saisies en provenance du clavier ou de la souris.

Le canvas est généralement utilisé tel quel pour fournir un espace de dessin général.

Le canvas peut écouter tous les événements applicables à un composant général. On peut, en particulier, lui associer des objets `KeyListener`, `MouseMotionListener` ou `MouseListener` pour lui permettre de répondre d'une façon ou d'une autre à une interaction utilisateur.

Pour réaliser un tel espace "libre" on peut aussi créer un composant dérivé de Component ou de Container: dans ce cas le "fond" sera transparent. Pour donner une taille à un tel composant on définira ses méthodes `get***Size()` .

Menus

Les menus diffèrent des autres composants par un aspect essentiel. En général, on ne peut pas ajouter de menus à des conteneurs ordinaires et laisser le gestionnaire de disposition les gérer. On peut seulement ajouter des menus à des éléments spécifiques appelés conteneurs de menus. Généralement, on ne peut démarrer une "arborescence de menu" qu'en plaçant une barre de menus dans un Frame via la méthode `setMenuBar()`. A partir de là, on peut ajouter des menus à la barre de menus et incorporer des menus ou éléments de menu à ces menus.

L'exception est le menu `PopUpMenu` qui peut être ajouté à n'importe quel composant, mais dans ce cas précis, il n'est pas question de disposition à proprement parler.

Menu Aide

Une caractéristique particulière de la barre de menus est que l'on peut désigner un menu comme le menu Aide. Cette opération s'effectue par l'intermédiaire de la méthode `setHelpMenu(Menu)`. Le menu à considérer comme le menu Aide doit avoir été ajouté à la barre de menus, et il sera ensuite traité de la façon appropriée pour un menu Aide sur la plate-forme locale. Pour les systèmes de type X/Motif, cela consiste à décaler l'entrée de menu à l'extrémité droite de la barre de menus.

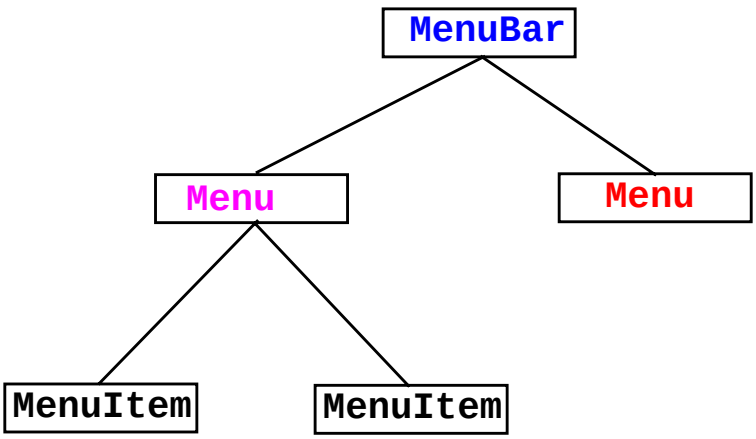
MenuBar

C'est la barre de menu horizontale. Elle peut seulement être ajouté à l'objet `Frame` et constitue la racine de toutes les arborescences de menus.

```
Frame fr = new Frame("MenuBar");
MenuBar mb = new MenuBar();
fr.setMenuBar(mb);
```

On n'abonne pas de veilleur d'événements à une `MenuBar` tout est automatique à ce niveau.

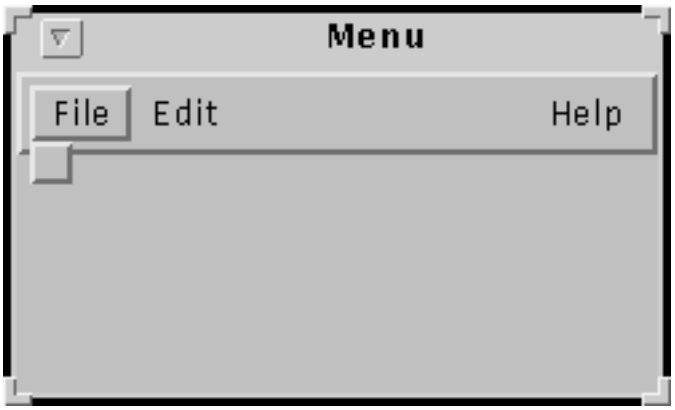
Hiérarchie d'accrochage des éléments de menu:



Menu

La classe `Menu` fournit le menu déroulant de base. Elle peut être ajoutée à une barre de menus ou à un autre menu.

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
mb.setHelpMenu(m3);
```



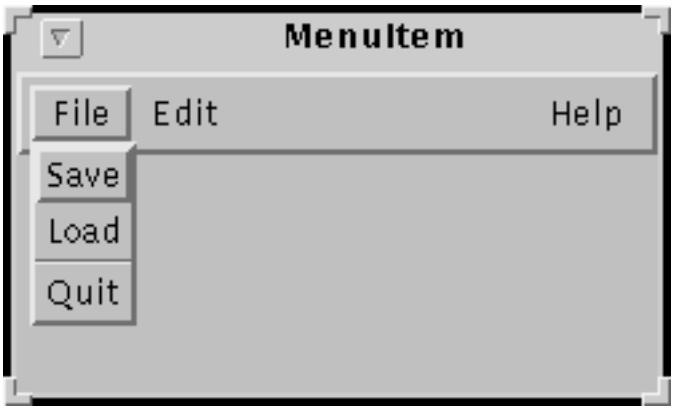
Les menus présentés ici sont vides ce qui explique l'aspect du menu File.

On peut ajouter un *`ActionListener`* à un objet Menu, mais c'est assez inhabituel. Normalement, les menus servent seulement à disposer des `MenuItem` décrits plus loin.

MenuItem

Les éléments de menu `MenuItem` sont les “feuilles” d’une arborescence de menu.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
MenuItem mi2 = new MenuItem("Load");
MenuItem mi3 = new MenuItem("Quit");
m1.add(mi1);
m1.add(mi2);
m1.addSeparator();
m1.add(mi3);
```

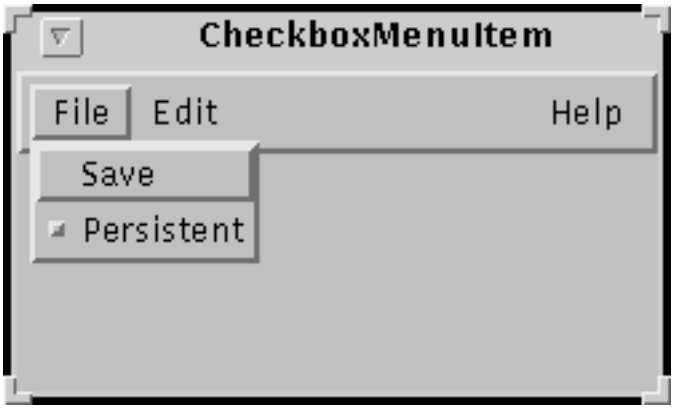


En règle générale, on associe un `ActionListener` aux objets `MenuItem` afin d’attribuer des comportements aux éléments de menu.

CheckboxMenuItem

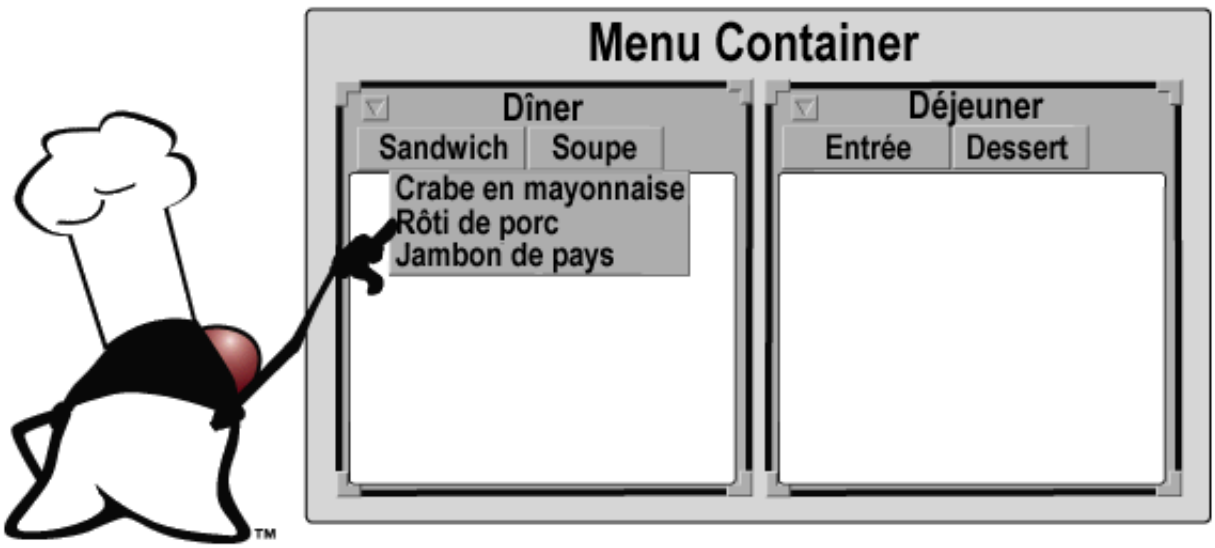
Les éléments de menu à cocher permettent de proposer des sélections (activé / désactivé) dans les menus.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
CheckboxMenuItem mi2 =
          new CheckboxMenuItem("Persistent");
m1.add(mi1);
m1.add(mi2);
```



L'élément de menu à cocher doit être surveillé via l'interface ItemListener. La méthode `itemStateChanged()` est appelée lorsque l'état de l'élément à cocher est modifié.

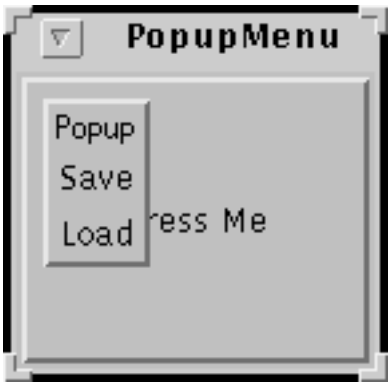
Menus


Menu Container
Dîner
Sandwich
Soupe
Crabe en mayonnaise
Rôti de porc
Jambon de pays
Déjeuner
Entrée
Dessert

PopupMenu

Fournit un menu autonome pouvant s'afficher instantanément sur un autre composant. On peut ajouter des menus ou éléments de menu à un menu instantané.

```
Frame fr = new Frame("PopupMenu");
Button bt = new Button("Press Me");
bt.addActionListener(...);

PopupMenu pp = new PopupMenu("Popup");
MenuItem sit = new MenuItem("Save");
MenuItem lit = new MenuItem("Load");

sit.addActionListener(...);
lit.addActionListener(...);

fr.add("Center", bt);
pp.add(sit);
pp.add(lit);
fr.add(pp);
```



PopupMenu (suite)

Le menu PopUp doit être ajouté à un composant "parent". Cette opération diffère de l'ajout de composants ordinaires à des conteneurs. Dans l'exemple suivant, le menu instantané a été ajouté au Frame englobant.

Pour provoquer l'affichage du menu instantané, on doit appeler la méthode show. L'affichage nécessite qu'une référence à un composant joue le rôle d'origine pour les coordonnées x et y.

Dans cet exemple c'est le composant bt qui sert de référence.

```
public void actionPerformed(ActionEvent ev) {
          pp.show(bt, 10, 10);
}
```

Composant de référence pour l'affichage du popup menu

Coordonnées % composantes d'origines

Le composant d'origine doit être sous (ou contenu dans) le composant parent dans la hiérarchie des composants.

Contrôle des aspects visuels

On peut contrôler l'apparence des composants AWT en matière de couleur de fond et de premier plan ainsi que de police utilisée pour le texte.

Couleurs

Deux méthodes permettent de définir les couleurs d'un composant :

- setForeground(...)
- setBackground(...)

Ces deux méthodes utilisent un argument qui est une instance de la classe `java.awt.Color`. On peut utiliser des couleurs de constante désignées par `Color.red` `Color.blue` etc. La gamme complète de couleurs prédéfinies est documentée dans la page relative à la classe `Color`.

Qui plus est, on peut créer une couleur spécifique de la façon suivante :

```
int r = 255, g = 255, b = 0;
Color c = new Color(r, g, b);
```

Un tel constructeur crée une couleur d'après les intensités de rouge, vert et bleu spécifiées sur une échelle allant de 0 à 255 pour chacune.

Polices

La police utilisée pour afficher du texte dans un composant peut être définie à l'aide de la méthode `setFont()`. L'argument utilisé pour cette méthode doit être une instance de la classe `java.awt.Font`.

Aucune constante n'est définie pour les polices, mais on peut créer une police en indiquant son nom, son style et sa taille en points.

```
Font f = new Font("TimesRoman", Font.PLAIN, 14);
```

Si la portabilité est recherchée il vaut mieux utiliser des noms de polices abstraites :

- Dialog, DialogInput
- SansSerif (remplace Helvetica)
- Serif (remplace TimesRoman)
- Monospaced (remplace Courier)
- Symbol

On peut obtenir la liste complète des polices en appelant la méthode `getFontList()` de la classe `Toolkit`. La boîte à outils (toolkit) peut être obtenue à partir du composant, une fois ce dernier affiché on appelle la méthode `getToolkit()`. On peut aussi utiliser le ToolKit par défaut obtenu par `Toolkit.getDefaultToolkit()`.

Les constantes de style de police sont en réalité des valeurs entières (int), parmi celles citées ci-après :

- `Font.BOLD`
- `Font.ITALIC`
- `Font.PLAIN`
- `Font.BOLD + Font.ITALIC`

Les tailles en points doivent être définies avec une valeur entière.

Impression

L'impression de documents est très liée à la représentation graphique.

Pour une documentation à jour voir l'URL: `http://java.sun.com/printing` et le package `java.awt.print`.

Rappels: variables, méthodes, constructeurs 18

Contenu:

Condensé des spécifications d'utilisation des variables, méthodes, constructeurs.

Pour des descriptions plus completes voir le document de spécification du langage (J.L.S.)

Plan général d'une classe

Plan général des déclarations de classe premier niveau:

```
package yyy;
```

Voir : "packages", page 187

```
import z.UneCLasse ; // A préférer
import w.* ;
```

Voir: "Déclaration de visibilité (import)", page 191

```
public class XX extends YY implements AA, BB {
        // public ou "visibilité package"
                  // une classe publique par fichier
        // si méthode "abstract" la classe doit etre "abstract"
```

Voir: "Héritage: mot-clef extends", page 166; "Réalisations d'une interface", page 290

```
        // ordre indifférent sauf pour les blocs et expressions
        // d'initialisation de premier niveau

        //MEMBRES STATIQUES
        variables statiques
                  var. statiques initialisées
                  var. statiques "final"/constantes de classe
```

Voir: "variables partagées", page 197;,"variables de classe", page 386,

```
        méthodes statiques
```

Voir: "Méthodes de classe :", page 110;"méthodes de classe", page 199;"Méthodes de classe", page 393;

```
        classes ou interfaces statiques
                  classes statiques avec accès privilégié
                            avec la classe englobante.
```

Voir: "Classes et interfaces membres statiques", page 309

//MEMBRES D'INSTANCE
variables d'instance

Voir: "Objets: agrégats, création explicite par new", page 87; "Classes et objets", page 143; "variables d'instance", page 384; ;

var.d'instance initialisées

Voir: "Opérations d'initialisations", page 156; "Opérations d'initialisation des instances", page 179; "variables d'instance", page 384

var d'instance *"blank final"*

Voir: "Variables marquées final", page 196; "variables d'instance", page 384

méthodes d'instance

Voir: "Méthodes d'instance", page 108; "Méthodes d'instance", page 144;"Méthodes d'instance", page 391

"patrons" de méthodes d'instance (abstract,...)

Voir:"Les classes abstraites", page 293; "Méthodes d'instance", page 391

classes d'instance
classes avec accès instance englobante

Voir: "Classes membres d'instance", page 311

```
//BLOCS DE PREMIER NIVEAU
blocs statiques
 /* évalués au moment du chargement de la classe */
```

Voir: “bloc d’initialisation de classe”, page 204; “Blocs statiques”, page 397

```
blocs d’instance
 /* évalués au moment de la création de l’instance */
```

Voir: “Blocs d’instance”, page 398

```
// CONSTRUCTEURS
constructeurs
```

Voir: “Initialisations : constructeur”, page 151; “Le constructeur par défaut”, page 156; “Perte du constructeur par défaut”, page 179; “Constructeurs”, page 394

```
}
```

Plan général d'une interface

```
public interface JJ extends BB, CC {
```

Voir: “Réalisations d'une interface”, page 290

var. statiques “final”/constantes de classe

Voir: “Réalisations d'une interface”, page 290; “variables partagées”, page 197;,“variables de classe”, page 386

classes ou interfaces statiques

Voir: “Classes et interfaces membres statiques”, page 309

“patrons” de méthodes

Voir: “Réalisations d'une interface”, page 290

```
}
```

Variables

Deux aspects : déclaration , utilisation.

On ne peut utiliser une variable qui n'a pas été déclarée. Dans de nombreux cas on ne peut pas utiliser une variable qui n'a pas été initialisée (vérification de *compile-time*).

Critères à controler :

- Nom : un symbole (un nom) va désigner la variable. Il faut que ce nom ne soit pas un mot réservé, que ce "mot" ne constitue pas une gêne pour une variable déclarée avec le même "nom" (conflit de portée). Suivre les règles de nommage pour rendre votre code plus lisible.

- Type : toute déclaration d'un nouveau symbole doit s'accompagner d'une déclaration de type qui définit les droits et obligations liés à l'usage du nom. Attention les types scalaires primitifs et les types objets n'ont pas les mêmes comportementa, ni les mêmes contraintes.

- Initialisation : à vérifier soigneusement. Le compilateur fait une vérification d'initialisation sur les variables automatiques (variables de bloc).

Forme générale des déclarations:

```
Type nomSymbole
        // exemples
        int ix              // un scalaire
        Integer iix         // un objet
        double[] tbd        // un tableau de scalaires
        UneClasse[] tbo     // un tableau d'objets
        UneClasse[][] tb2   // un tableau de tableaux
```

variables d'instance

Variables membres de l'instance. Chaque objet créé sur le modèle de la classe gère sa propre combinaison de valeurs associées à ces variables.

```
public class Compte {
        int numero ;
        BigDecimal solde ;
}
```

Les objets créés sur ce modèle: `compteDupond`, `compteDurand` ont des valeurs différentes pour `numero` ou `solde`.

- Déclaration : sous la forme `Type nom ;`
  (autres formes possibles : `Type nom1, nom2`, etc. et déclaration +initialisations)
  Ordre et emplacement des déclarations indifférent (sauf si initialisations)

- Modificateurs d'accès possibles: `public`, `private`, `protected`

- Initialisations : par défaut les variables d'instance sont initialisées à zéro au moment de la création de l'instance. Selon les types des variables cette initialisation peut avoir des signications différentes (*false* pour les `booleans`, *null* pour les références, etc.).
  En général il est conseillé de réaliser l'initialisation d'une variable d'instance soit dans un constructeur (valeur passée en paramètre du constructeur), soit dans une déclaration avec initialisation (par exemple pour l'allocation de structures de données):

```
public class Compte {
        int numero ;
        BigDecimal solde ;
        // "structure de donnée" initialisée
        Arraylist listOpérations = new ArrayList() ; \

        public Compte( int num, String depotInitial) {
                numero = num ;
                solde = new BigDecimal(depotInitial) ;
        }
}
```

Dans le cas où la variable d'instance est déclarée et initialisée l'ordre de déclaration des initialisations est important. Par ailleurs ces initialisations sont exécutées avant la partie du code du constructeur ne faisant pas appel à un autre constructeur (appel de `this(...)` ou `super(...)`).
Attention: aucune de ces opérations d'initialisations hors constructeur ne doit être susceptible de propager une exception controlée (voir également bloc d'initialisation d'instance).
Une variable d'instance peut être marquée `final` ("*blank final*"): dans ce cas elle doit être initialisée dans le constructeur (ou par une initialisation immédiate); dans un constructeur on ne peut initialiser une variable `final` héritée (puique celle-ci a deja été initialisée dans la construction d'une super-classe).

- Accès : on ne peut consulter une variable d'instance que si l'on dispose d'une instance :
  `compteDupond.numero ;`
  Dans le code de la classe on a implicitement une référence à l'instance courante (this) :

```
double getSolde() {// dans classe Compte
          return solde.doubleValue() ;
          // implicitement this.solde
}
```

Attention: ceci n'est pas vrai dans le code d'une méthode statique (comme main).
Autres cas : désignation d'une instance d'une classe englobante : `ClasseEnglobante.this.champ` et désignation d'un champ "caché" de la super-classe : `super.champ` (conseil: ne pas se mettre dans cette situation qui consiste à redéfinir un champ d'instance dans une sous-classe).

- Autres modificateurs : `transient`, `volatile` (sortent du périmètre de ce cours)

variables de classe

Variables membres de la classe. Dans le contexte d'un ClassLoader toutes les instances ont accès à ces **variables partagées**.

```
public class Compte {
        static int compteurDInstances ; // partagé
        int numero ;
        BigDecimal solde ;
}
```

- Déclaration : sous la forme `Type nom ;`
  (autres formes possibles : `Type nom1, nom2`, etc. et déclaration +initialisations)
  Ordre et emplacement des déclarations indifférent (sauf si initialisations)

- Modificateurs d'accès possibles: `public, private, protected`

- Initialisations : par défaut les variables de classe sont initialisées à zéro au moment du chargement de la classe par le ClassLoader. Selon les types des variables cette initialisation peut avoir des signications différentes (*false* pour les `booleans`, *null* pour les références, etc.).
  Pour réaliser l'initialisation d'une variable de classe il faut soit utiliser un bloc statique de premier niveau , soit une déclaration avec initialisation :

```
public class Contexte {
        public static String osName =
                              System.getProperty("os.name");
        public static URL urlDoc ;
        static { // bloc statique
                  try { urlDoc = new URL("http://java.sun.com")
                  } catch(MalformedURLException exc){}
        }//ATTENTION si "urlDoc" est final : problèmes possibles
}
```

Les variables membres d'une interface sont implicitement `public static final`.

Dans le cas où la variable de classe est déclarée et initialisée l'ordre de déclaration des initialisations et des blocs statiques est important: les évaluations se font dans l'ordre au moment du chargement de la classe.
Attention : les opérations d'initialisation ne peuvent faire appel à des méthodes d'instance de la classe courante (sans création d'une telle instance) et ne doivent pas être susceptibles de propager des exceptions controlées.

Une variable de classe peut être marqueé `final` et initialisée (ce qui ne veut pas nécessairement dire que ce sont des constantes évaluables au moment de la compilation).

- Accès : on peut accéder à une variable de classe de 3 manières:
  - en la qualifiant par le nom de la classe
    `MaClasse.maVariableDeClasse`
    `MaClasse.CONSTANTE_DE_CLASSE`
    ce sont les notations à préférer.
  - en la qualifiant par une référence de la classe
    `instanceDeMaClasse.maVariableDeClasse`
    (ou en utilisant implicitement ou explicitement `this`)
    Cette notation est à éviter car elle ne facilite pas la compréhension du code.
  - en la qualifiant par désignation de la super-classe :
    `super.variableDeClasse`
    ceci correspond à une situation de redéfinition d'une variable de classe dans une sous-classe (*data hiding*): à éviter
- Autre modificateur : `volatile` (sort du périmètre de ce cours)

variables de bloc

Il s'agit de variables temporaires: elles sont créées au moment où l'exécution atteint cette partie du code et détruites quand on sort de la portée du bloc courant (variables ou références allouées dans la pile).

- Déclarations :
  - instruction de déclaration dans un bloc

```
{
    int ix ;// n'importe où dans le bloc
    String s1, s2, s3 ;
    double dz = 3.14 ;
```

  - cas particuliers assimilés aux précédents :

```
for(int ix = 0, iy = tb.length; ix<tb.length; ix++, iy--) {
    // ix et iy n'existent que dans la portée du bloc "for"
```

  - paramètres

```
void methode(int ix, String nom) {
        // ix et nom en portée dans le bloc
....
Maclasse(String parm) { // paramètre constructeur
...
try {....
} catch (IOException exc){//exc en portée dans bloc catch
}
```

Attention: un symbole local dans un bloc peut "cacher" une variable d'instance ou une variable de classe, le compilateur ne le signale pas mais le code devient illisible. Par contre il ne peut pas porter le même nom qu'un symbole du bloc local ou d'un bloc englobant.

- Modificateurs :
  possibles : `final`,
  impossibles : `static`, `public`, `private`, `protected`, `volatile`, `transient` (attention: erreur de syntaxe difficile à comprendre, certains compilateurs donnent un message obscur!)

- Initialisations :

  - Les paramètres sont automatiquement initialisés (par l'appel de la méthode, ou la récupération d'une exception).
    La modification de la valeur d'un paramètre dans le bloc est sans effet sur le code appelant (voir cas particulier des "effets de bord").

  - Les autres variables de bloc dovient impérativement être initialisés avant d'être utilisées (condition vérifiée par le compilateur)

Méthodes

Deux aspects: définition d'un coté, utilisation de l'autre.

L'utilisation d'une méthode fait partie du contrat d'un type donné. Sauf dans le cas très particulier de l'exécution dynamique par introspection (package `java.lang.reflect`), le compilateur va s'efforcer de vérifier que l'appel est conforme à la définition (il consulte le fichier binaire ".class" pour cela).

Critères à controler :

- Nom : un symbole (un nom) va désigner la méthode. Plusieurs méthodes peuvent porter le même nom -avec des paramètres différents-(*surcharge*): il est préférable dans ce cas que l'intention sémantique soit la même.
  Pour éviter toute confusion ne pas utiliser le même nom qu'une variable ou qu'une classe (c'est possible!)
  Suivre les règles de nommage pour rendre votre code plus lisible.

- Signature : c'est la combinaison caractéristique d'une méthode comprenant le nom et la liste ordonnée des types des paramètres. Une classe ne peut pas avoir deux méthodes avec la même signature.

- Résultats : les méthodes de Java sont "fonctionnelles" par nature, elles prennent des paramètres en entrée et rendent un seul résultat pour l'exécution "normale". Si il n'y a pas de résultat on déclare un type `void`.
  Dès qu'une méthode déclare un résultat non-`void` on ne doit sortir "normallement" de la méthode que par l'appel d'un ou plusieurs `return` accompagné d'une valeur du type de retour demandé.
  Les résultats "anormaux" sont ceux qui découlent d'une erreur d'exécution: leur déclaration fait partie du "contrat" de la méthode.

```
Object[] tableauTrié(Object[] tb)
                    throws ClassCastException {
          Object[] res = (Object[]) Array.newInstance(
                    tb.getClass().getComponentType(), tb.length);
                    ...// copier dans res et trier
          return res ;
}
```

Méthodes d'instance

Méthodes membres de l'instance. Les instructions dans le corps de définition peuvent agir sur les variables propres à la méthode, les variables membres de l'instance courante, les variable de classe de la classe courante.

- Définition : sous la forme

```
TypeRésultat nom(liste_paramètres) throws liste_exceptions {
        //corps de définition
}
```

Les méthodes marquées `abstract` (et celles définies dans les interfaces) ou marquées `native` n'ont pas de corps de définition.

Des règles strictes gouvernent la déclaration d'une méthode comme redéfinition d'une méthode de la super-classe:

- on ne peut redéfinir une méthode de classe en méthode d'instance.
- on ne peut aggraver le "contrat" : rendre la méthode plus "privée" que la méthode originale, déclarer propager plus d'exceptions (ou des super-classes des exceptions déclarées).
- on ne peut modifier le type de résultat d'une méthode de même signature.

Nota: la définition d'un paramètre peut Être d'une des formes:

```
Type nom
final Type nom
```

- Modificateurs :
  `public, private, protected, final, abstract`
  et `native, strictfp, synchronized` (hors du périmètre de ce cours)
  (`abstract` est incompatible avec `final` et `native`)
  Les en-têtes de méthodes de déclaration des interfaces sont implicitement `public abstract`.

- Appels : on ne peut appeler une méthode d'instance que si l'on dispose d'une instance (on "envoie un message" à l'instance):

```
monInstance.méthode(arg1, arg2, argn) ;
```

Dans le code de la classe on a implicitement une reférence à l'instance courante :

```
public class MaClasse {
        public void meth1() {
                ....
        }
        public void meth2() {
                ...
                meth1() ;
                // implicitement this.meth1()
        }
}
```

Autres cas :

- désignation d'une instance englobante :

```
ClasseEnglobante.this.methode() ;
//utile si methode est aussi defini dans la classe locale
```

- désignation de la méthode de la super-classe (en cas de redéfinition locale)

```
super.toString() + "champ local:" + champlocal ;
// redefinition de toString() dans la classe fille
```

Les paramètres sont passés par valeur: c'est à dire qu'ils ont évalués, puis copiés dans la pile et c'est cette copie qui est récupérée par le code de la méthode. Les règles de promotion de type sont appliquées.

Méthodes de classe

Méthodes membres de la classe. Les instructions dans le corps de définition peuvent agir sur les variables propres à la méthode et les variable de classe de la classe courante. Elles ne peuvent agir sur des variables d'instance si ces variables ne sont pas qualifiées par rapport à une instance dûment créée.

- Définition : sous la forme

```
static TypeRes nom(liste_paramètres) throws liste_exceptions {
        //corps de définition
}
```

On peut redéfinir ainsi dans une classe une méthode statique de la super-classe (rupture du lien statique : *hiding*) -dispositif rare et peu conseillé-

- Modificateurs :
  `public, private, protected, final`
  et `native, strictfp, synchronized` (hors périmètre).

- Appels : on peut appeler une méthode de classe:

  - en la qualifiant par le nom de la classe
    `MaClasse.maMéthodeDeClasse()`
    c'est la notation à préférer.

  - en la qualifiant par une référence de la classe
    `instanceDeMaClasse.maMéthodeDeClasse()`
    (ou en utilisant implicitement ou explicitement `this`)
    Cette notation est à éviter car elle ne facilite pas la compréhension du code (+ évaluation non polymorphique).

  - en la qualifiant par désignation de la super-classe :
    `super.maMéthodeDeClasse()`
    `((SuperClasse)instance).méthodeDeClasse()`
    ceci correspond à une situation de redéfinition d'une méthode de classe dans une sous-classe (*hiding*): à éviter

Les paramètres sont passés par valeur: c'est à dire qu'ils ont évalués, puis copiés dans la pile et c'est cette copie qui est récupérée par le code de la méthode. Un paramètre peut être marqué `final`. Les règles de promotion de type sont appliquées.

Constructeurs

Ce ne sont pas des “membres” de la classe (on ne qualifie pas un constructeur par rapport à une instance ou par rapport à une classe: on l’appelle par `new`).

Critères à controler lors de la définition :

- Nom : obligatoirement le nom de la classe.
  Noter que les constructeurs n’étant pas hérités. On est obligé de les définir pour chaque niveau de la hiérarchie de classes: on a bien une correspondance nomClasse-nomConstructeur.

- Paramètres : mêmes principes de définition des paramètres que pour les méthodes.
  Les constructeurs peuvent être surchargés (plusieurs contructeurs possibles avec des paramètres différents).
  Une classe dépourvue de définition est dotée automatiquement d’un constructeur par défaut (constructeur sans paramètres) à condition que sa super-classe lui rende accessible un constructeur sans paramètres (implicite ou non).
  Dès qu’un constructeur est défini la classe perd son consturcteur implicite (on peut alors définir explicitement un constructeur sans paramètres).

- Structure interne : le bloc de définition a une architecture particulière:

  - une première instruction doit concerner l’appel d’un autre constructeur : soit de la même classe (par `this`) soit de la super-classe (par `super`). Si le constructeur à appeler est le constructeur sans paramètres de la super-classe cette instruction peut être implicite.

  - après cette première instruction le système évalue (dans l’ordre défini par le code source) toutes les initialisations explicites de variables d’instance (ou de blocs d’instance).

  - Les autres instructions d’initialisation viennent ensuite. Si une variable membre de l’instance est marquée final sans avoir été initialisée, son initialisation doit être réalisée au travers du code de tous les constructeurs.

(par ailleurs éviter d'appeler dans le code du constructeur des méthodes de l'instance courante qui ne soient pas marquées `final` ou `private`).

- Résultats: Un constructeur ne doit déclarer aucun résultat (même pas `void`). Son rôle est d'allouer et initialiser une instance.
  Par contre un constructeur peut propager des exceptions et donc refuser de construire / initialiser une instance.

- Modificateurs : `public, protected, private`
  (pas de static, final, native, strictfp, synchronized)

- Cas particuliers :

  - On ne peut pas créer une instance d'une classe marquée `abstract` mais celle-ci peut tout de même disposer de constructeurs qui seront utilisées par les sous-classes par l'invocation de super() (ou au travers de la définition / invocation d'une classe anonyme)

  - classes "internes" :

    - on ne peut pas définir de constructeur dans une classe anonyme mais on peut invoquer le constructeur de la super-classe de la classe anonyme en lui passant des paramètres.

    - les constructeurs de classes membres d'instance on besoin d'être invoqué à partir d'une référence du type de la classe englobante dont la classe est membre.

    - cas extrême : il est possible dans la définition du constructeur d'une classe interne d'instance d'appeler `this` ou `super` comme membre d'un paramètre du type de la classe englobante:

```
class Interne2 extends Interne1 {
                    ....
          public Interne2(Englobante inst, int x){
                inst.super(x) ; //dans le contexte
                     // de l'instance paramètre
```

Blocs

Les blocs permettent de regrouper un ensemble d'instructions. Ils déterminent également des zones hiérarchisées de portée des variables:

```
{// bloc englobant
        int nombre ;
        .....
        { // bloc interne
                int autreNombre ;
                ....
                nombre = 10 ; // correct
        } // fin bloc interne
        ...
        autreNombre = 7 ;    // ERREUR ! IMPOSSIBLE
        nombre = 20 ;       // possible
}
```

Certains dispositifs syntaxiques permettent de déclarer également des variables dans la portée du bloc associé : paramètre de méthode ou de constructeur, paramètre d'un bloc `catch`, déclaration de variable dans la partie "initialisation" d'une boucle `for`...

En dehors des blocs associés aux déclarations de classe, aux "corps" des méthodes ou des constructeura, aux structures de controles, aux blocs `try/catch/finally`, et aux blocs `synchronized` (hors périmètre du cours) il existe des blocs "isolés" : blocs d'initialisations statiques, blocs d'initialisations d'instance et blocs isolés dans un autre bloc.

Blocs statiques

Un bloc de premier niveau marqué `static` est évalué au moment du chargement de la classe.

Il peut y avoir plusieurs blocs “d’initialisation de classe”: ils sont évalués dans l’ordre d’apparition dans le code.

```
public class Maintenance{
static URL urlMaintenance ;

   static {
         String url = System.getProperty("produitX.url") ;
         URL secours = null ;
         try {
            secours = new URL ("http://java.sun.com") ;//testé
            urlMaintenance = new URL(url) ;
         } catch (Exception exc)  {
            urlMaintenance = secours;
         }
   }

....
```

Blocs d'instance

Un bloc de premier niveau non marqué `static` est évalué au moment de la création de l'instance (après initialisation de la super-classe et avant l'exécution du reste du code du constructeur).

Il peut y avoir plusieurs blocs "d'initialisation d'instance": ils sont évalués dans l'ordre d'apparition dans le code. Ces blocs, rares, peuvent être utilisés soit pour réaliser des initialisations nécessitant des blocs `try/catch`, soit pour remplacer des opérations d'initialisation d'instance dans une classe anonyme (qui ne dispose pas de système de définition de constructeur).

Blocs non associés à une structure

Dans un bloc il est possible d'ouvrir un nouveau bloc de contexte. ceci permet d'isoler des déclarations de variables, d'améliorer la lisibilité et la structure du code.

```
...
{ // PANEL HAUT (nouveau bloc)
          panelHaut = new Panel(new FlowLayout(
                                  FlowLayout.LEFT));
          Button button1 = new Button("1") ;
          panelHaut.add(button1 ;
}
add(panelHaut, BorderLayout.NORTH) ;
```

Un tel bloc de code peut aussi être étiqueté :

```
__PANEL_HAUT__: {
          ....
          if(...) break __PANEL_HAUT__ ;
           //
          ....
}
```

Ces dispositifs (peu usités) peuvent être utiles pour attirer l'attention du lecteur sur des structures implicites :

Aide-mémoire 19

Contenu:

- les outils du SDK
- aide-mémoire HTML pour javadoc
- glossaire
- adresses utiles

Le SDK

Le SDK s'installe en deux phases :

- L'installation de la machine virtuelle spécifique à votre machine / système
- L'installation de la documentation (commune à toutes les plate-formes)

Une installation typique du répertoire "racine" de Java contient les répertoires ou fichiers:

- **`bin`** : exécutables de lancement des utilitaires liés à Java : `java`, javac, etc.
- **`jre`** : l'installation de la machine virtuelle proprement dite. Exécutables, librairies dynamiques, librairies "système" ou librairies d'extension de Java, ressources générales de configuration de l'exécution.
- **`lib`** : librairies utilitaires spécifiques à l'implantation (par ex. librairies Java des utilitaires comme le compilateur, javadoc, etc.)
- **`include`** : les ressources pour compiler du code natif lié à Java.
- **`demo`** : ressources pour des démonstrations de programmes et leur code source.
- **`docs`**: le répertoire de la documentation générale en HTML (fichier index.html)
- **`src.jar`** : les sources des librairies standard de Java dans une archive Jar. Pour consulter ces sources faire

```
jar xvf src.jar
```

qui crée un répertoire **`src`** contenant les fichiers source.

Les outils

La documentation des outils est liée à la livraison du SDK.

voir `docs/tooldocs/tools.html`

outre **java**, **javac**, **javadoc** quelques autres outils utiles pour débuter:

- **jar** : permet d'archiver un ensemble de binaires java (et d'autres fichiers associés). C'est une archive dans un format compressé (ZIP) qui peut être passée comme élément de CLASSPATH. C'est de cette manière que des librairies ou des applications Java sont livrées. Ce point est developpé dans notre cours "maitrise de la programmation java".

- **javap** : "décompile" un binaire Java en assembleur JVM. Pratique pour vérifier rapidement le véritable contenu d'un fichier ".class". Pour un vrai décompilateur (binaire->source) il existe des logiciels domaine public.

- **jdb** : debuggueur "rustique" en mode commande.

- **appletviewer**: permet de tester rapidement la présentation d'une applet.

- **native2ascii** : utilitaire de conversion de fichiers texte (selon le mode de codage des caractères)

javadoc et HTML

Voir usage de javadoc dans la documentation spécifique : `docs/tooldocs/`*plate-forme*`/javadoc.html`.

Point particulier pour les auteurs francophones: le source Java de la documentation doit être lisible ce qui pose un problème pour les caractères accentués qui peuvent être différents selon les plate-formes de codage. On veillera à ce que le HTML généré soit portable (option -charset).

Pour HTML voir les descriptions de référence à `http://www.w3.org/MarkUp` et `http://www.w3.org/TR/html4`

Aide mémoire HTML

Structures principales

Eléments principaux de la structuration du fichier html : n'ont que peu ou pas d'intérêt pour la documentation javadoc puisque le programme génère automatiquement les grandes structures de présentation du document HTML.

Structure générale :

```
<HTML>
        <HEAD>
                titre interne
                <TITLE> titre externe </TITLE>
        </HEAD>

        <BODY>
                corps du document (texte à balises)
        </BODY>
</HTML>
```

Hiérarchie des sections:

```
<H1> premier niveau de titre de paragraphe </H1>
          <H2> second niveau </H2>
                    <H3> niveau suivant </H3>
<H1> encore premier niveau </H1>
```

Paragraphes et listes

```
<!-- un commentaire -->
<P> un paragraphe
</P>
<P> un autre paragraphe
<BR> un retour à la ligne forcé
</P>
<BR>
<HR> <!------------ un filet horizontal------------------------>
```

Liste simple :

```
<UL>
        <LI> un élément de la liste </LI>
        <LI> un autre élément </LI>
</UL>
```

Liste numérotée:

```
<OL>
        <LI> premier item </LI>
        <LI> deuxième item, etc. </LI>
</OL>
```

Liste descriptive:

```
<DL COMPACT>
        <DT> entrée dans la liste </DT>
                <DD> description de l'entrée </DD>
        <DT> un autre terme </DT>
                <DD> sa description </DD>
</DL>
```

Citation de texte :

```
<BLOCKQUOTE>
        La culture c'est ce qui reste quand on a tout oublié
</BLOCKQUOTE>
```

Texte préformaté :

```
<PRE>
        for( int ix = 0; ix < tb.length; ix++) {
                System.out.println(tb[ix]);
        }
</PRE>
```

Tableaux

```
<TABLE >
        <TR> <!--ligne-->
                <TH> cellule de titre
                <TD> cellule normale
                <TD> cellule suivante
        <TR> <!-- ligne suivante>
                <TH>
                <TD>
</TABLE>
```

Hypertexte et éléments allogènes

Lien vers un autre document:

```
voir <A HREF="http://java.sun.com">
```

Définition d'une destination de lien à l'intérieur d'un document:

```
<A NAME="cible"> c'est ici
```

utilisation dans une reférence hypertexte:

```
voir <A HREF="#cible"> dans le même document
... <!-- dans un autre fichier HTML -->
voir aussi <A HREF="http://www.truc.fr/doc.html#cible">
```

Images

```
voici une image : <IMG SRC="image.gif">
ou <OBJECT data="image.gif" type="image/gif">
```

Styles physiques

```
dans le texte lui même <B>ceci est en caractères gras</>
et <I>ceci en italiques</I>
<U>ceci est souligné</U>
et <TT> ceci en caractères à chasse fixe (comme fonte
"courier")</TT>
```

Glossaire

Quelques définitions dans ce glossaire sont marquées “terminologie non-standard” car il n’y a pas un large accord sur la pratique terminologique en Français. Le terme décrit a toutefois été choisi pour être utilisé dans ce support de cours.

accesseur

(*accessor*) méthode permettant d’interroger une instance sur la valeur d’un champ dont l’accès est protegé. Par convention les accesseurs sont de la forme : `type getXXX()` . où XXX est le nom du champ. Voir “mutateur” et la convention correspondante.

adaptateur

(*adapter*) dans le contexte particulier du traitement des événements en Java: une classe qui réalise toutes les méthodes d’un interface avec un code vide. On doit créer une classe qui hérite de l’adaptateur et qui redefinit une ou plusieurs de ces méthodes.

agregation

Ce terme désigne le fait qu’un objet existe essentiellement comme la réunion d’autres objets dont il possède les références à un moment donné, A COMPLETER.

allocation

Réservation d’un emplacement en mémoire. (par exemple pour contenir les données liées à un objet). En java on distingue les allocations “automatiques” (variables scalaires ou références sur la pile), les allocations dans le tas (par `new`), le pool de constantes, et les allocations des classes elle-mêmes par les ClassLoaders. Contrairement à d’autres langages on n’a, en Java, aucun accès à la gestion de ces emplacements mémoire.

analyse

A COMPLETER

`API`

(*Application Programming Interface*) la liste des accès programmatiques à un objet d'une classe donnée. En pratique on a tendance à restreindre cette description aux membres et constructeurs `public` ou `protected` qui constituent le "contrat" de type défini par la classe vis à vis des utilisateurs externes à l'équipe de développement.

`applet`

(Littéralement "Appliquette" -petite application-). Code Java téléchargé et exécuté par un navigateur WEB.

`architecture à trois parties`

(*three tiers architecture*) Applications(s) éclatée en trois parties distinctes (chaque partie étant éventullement accessible au travers du réseau). Une partie est chargée de l'interaction utilisateur (avec parfois un peu de logique déportée), une partie centrale est chargée de régler la logique applicative et une autre est constituée du fond de données de l'entreprise.

`association`

A COMPLETER

`attribut`

A COMPLETER

`automatique`

Variables propres à une méthode (paramètres, variables locales). Ces variables (scalaires, références vers des objets) sont allouées sur la pile et ont une "durée de vie" qui ne dépasse pas le temps d'exécution de la méthode (exception: les variables automatiques marquées "final" et copiées dans le cadre de classes locales).

`bean`
`champ`

voir "variables d'instance"

`chargeur de classe`

(*ClassLoader*)

`classe`
`classe abstraite`

Une classe qui est utilisée pour représenter un concept abstrait, et à partir de laquelle on ne peut pas créer d'objets. Les méthodes qui sont déclarées sans fournir de définition sont appelées méthodes abstraites. .
Ces classes sont souvent le résultat d'une décision de mutualisation de services entre des classes analogues.

classe anonyme
classe d'encapsulation

(terminologie non-standard pour *wrapper class*) Classes liées à des types scalaires primitifs et permettant : de créer des objets immuables qui contiennent un scalaire du type correspondant (par exemple de type `int` pour la classe `Integer`); d'obtenir des services (méthodes de classe) liés à ces types scalaires (par exemple conversion). Si une méthode prend un argument de type générique `Object` et qu'on veut l'utiliser avec des types primitifs on met alors en place une convention qui permet de passer une instance du type d'encapsulation correspondant. Le champ `TYPE` de ces classes est un objet de type `Class` qui décrit le type scalaire associé.

classe interne

(terminologie non-standard: englobe ici les termes *inner class* et *embedded class*)

classe locale
classe membre
CLASSPATH
code natif
compilation conditionnelle
composition
conception
constante
constructeur
container
conversion
délégation
directive
doclet
dynamicité
encapsulation
espace de nom

(terminologie non-standard : décalque de *namespace*)

événement
exception
expression

filtre
flot
fonction
forçage de type

(terminologie non-standard: rend un des sens de l'opérateur *cast*)

garbage collector
gestionnaire d'événement
gestionnaire de disposition
gestionnaire de sécurité
glaneur de mémoire

(terminologie non-standard: extension de "glaneur de cellules" de Lisp)

héritage
héritage multiple
Hotspot
Ieee754
implémenter

(terminologie non-standard: le mot n'existe pas en français mais il est retenu ici dans le sens très étroit de la déclaration du mot-clef *implements*)

instance
interface
interface homme/machine
introspection
JIT
JVM
liaison dynamique
liaison statique
linearisation
litteral
membre
message
méthode d'instance
méthode de classe
modèle structurel

(terminologie non-standard pour rendre *pattern*)

modèle/vue/controleur
modificateur
mutateur

(terminologie non-standard : décalque de *mutator*)

NaN
objet
opérateur
package
passage par référence
passage par valeur
pattern
persistence
petit-boutien
pile
poids fort/poids faible
pointeur
polymorphisme
pool
portée
primitif
propagation d'une exception
propriété
pseudocode
récursivité
redéfinition
reférence
reflection/introspection
rupture du lien statique

(terminologie non-standard pour rendre le terme *hiding* dans son acceptation par JLS)

sandbox security model
scalaire

(terminologie non-standard empruntée à d'autres langages comme APL)

serialization
spécification
statique
structure de contrôle
surcharge
tas
thread

(pas d'accord sur la traduction du terme en français)

transtypage

(terminologie non-standard: rend les différents sens de *cast*)

UML
variable d'instance
variable de classe
variables partagées
veilleur

(terminologie non-standard: idée de *Listener*)

XML

Adresses utiles

Java

site fondamental : `http://java.sun.com`

spécifications :

Approche "objet"

Un livre: "*Object-Oriented Software Construction, Second Edition* " (Bertrand Meyer - ed. Prentice-Hall-)

UML

`http://uml.free.fr` (en français!)

Unicode

`http://www.unicode.org`

HTML et XML

`http://www.w3.org`

INDEX

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

notes