# Programmation à Objets avec Java

**Pascal ANDRE**

**LINA - Université de Nantes**
**2, rue de la Houssinière - BP 92208**
**44322 Nantes Cedex 03**

pascal.andre@univ-nantes.fr

*RÉSUMÉ. Java a été rapidement et largement adopté par la communauté du développement du logiciel. Ce cours vise à donner aux auditeurs un aperçu de la programmation à objets, avec un langage représentatif de ce type de programmation :* JAVA. *Après avoir donné les bases du modèle à objets de Java, nous proposons un rapide tour d'horizon des API et librairies standards, ainsi que de l'environnement de développement Eclipse et de la méthode de programmation. Ce cours fait partie d'un cours sur le génie logiciel et la notation UML et de travaux pratiques de développement à objets. Des exemples pratiques illustrent ce cours.*

*MOTS-CLÉS : Génie Logiciel, Modélisation à objets, Programmation à objets, Java.*

*ABSTRACT. Java has been widely adopted by the software development teams. This course is a general overview of object-oriented programming, with a representative object oriented language:* JAVA. *After a short presentation of the language concepts, we explore the Java world, the rich interface library and the development tools of Eclipse. This course is a companion book of a course on Software Engineering and the a deeper UML Notation. It works with practical examples of Java programs.*

*KEY WORDS : Sofware Engineering, Development Methods, Object-Orientation, Java*

# Table des matières

# Chapitre 1

# Introduction

La programmation à objets est issue des travaux sur le langage SIMULA dans les années 1960. La technique n'est pas donc récente mais elle a largement évolué notamment avec des langages comme Smalltalk (1970). Elle a aussi bénéficié des travaux sur représentation des connaissances en intelligence artificielle (Minsky, 1975) et le travail coopératif (systèmes distribués, parallélisme) (Hewitt, 1973). Ces différents courants sous-entendent toutefois des sémantiques variées di concept d'objet : un objet est une unité de connaissance, une unité de calcul, un module, une valeur d'un type, etc. Par exemple, la notion d'objet est utilisée en intelligence articifielle pour représenter des connaissances (par exemple, un objet est un *frame*) et pour mettre en œuvre les mécanismes de raisonnement (par exemple, un objet est un *acteur*). On a aussi vu fleurir des langages hybrides. La technique est devenue mature et reconnue dans les années 80 et aujourd'hui nombre d'applications sont développées en utilisant une approche objet. Dans ce document, nous nous restreignons aux modèles à objet avec classes, ceux qu'on utilise en Génie Logiciel. Selon la classification de [Weg90], les langages à classes sont caractérisés par trois concepts essentiels : les objets, les classes et l'héritage. En ce sens, un langage tel qu'Ada, qui possède des mécanismes d'abstraction de données et d'encapsulation (les paquetages) n'est pas considéré comme un langage à classes. Deux autres "styles" à objets existent : les frames et les acteurs. Consulter [MNC+90] pour une description générale des langages à objets. Le lecteur trouvera aussi dans le chapitre 2 de [And01] un exposé sur les modèles à objets (concepts) et le développement à objets. Enfin, le rapport [AR98] constitue une bonne introduction à la modélisation à objets. Par abus de langage, le complément de nom "*à objets*" désignera une approche avec des classes.

L'approche à objets a marqué une rupture avec l'approche structurée qui dominait le développement du logiciel depuis 1960, et qui est encore largement employée. L'univers de l'application est structuré en termes d'objets et non plus en termes de procédures. L'algorithme et la structure de données de la programmation structurée ont fait place à une collection d'objets qui collaborent, dans la programmation à objets. La programmation à objets est une approche modulaire avec un grain de modularité fine, l'objet, qui encapsule données et traitements. Rappelons que qu'un bon programme modulaire groupe les morceaux de programmes par affinité (cohésion forte) et limite les interactions entre modules (couplage faible). Avoir de petites unités facilite leur écriture, leur compréhension, leur preuve et leur maintenance. La modification du code par ajout d'objets ne perturbe pas fondamentalement l'organisation du système, celui est est donc extensible. Les classes fournissent un mécanisme d'abstraction qui améliore la conception du logiciel en augmentant l'usage possible des composants. L'héritage renforce l'abstraction et l'utilisation verticale du code (autre forme d'extensibilité). L'envoi de message favorise aussi l'abstraction et la concision du code. Le parallélisme est naturellement induit par les objets. Ceci étant, la mise en œuvre dans des architectures systèmes traditionnelles pose le problème de la répartition explicite des objets. Les avantages attendus en termes de qualité du logiciel sont principalement l'extensibilité, la réutilisabilité. Mais la

modularité favorise aussi la lisibilité, l'intégrité, la compatibilité et la vérifiabilité. L'objectif est donc de capitaliser les efforts de développement en favorisant l'évolution et la réutilisation de composants logiciels. Pour cela, il faut une certaine culture de développement du composant [Mey89, Mey97]. L'objet est aussi bien adapté à la programmation à grande échelle (gros programmes, nombreux participants, développement sur la durée, formation importante) car il permet la répartition des activités.

La famille des langages à classes est celle du Génie Logiciel et du développpement. Parmi les plus connus et usités actuellement, citons tout d'abord Java et C++, puis Smalltalk et Eiffel et enfin toutes les variantes de ces langages. C++ est le langage de référence du développement industriel, c'est le standard qui s'est naturellement imposé de part son langage hôte, le langage C. Néanmoins, ce langage est relativement lourd et rigide : c'est un avantage pour la fiabilité des programmes mais certainement pas pour leur construction. Cette remarque devient déterminante dans le développement de petites applications, de prototypes ou de programmés "légers". C'est notamment le cas du développement d'applications distribuées ou en client léger (applets, scripts...). Java s'est largement imposé grâce au développement Web sur Internet. Il suffit de constater le nombre d'ouvrages sur le sujet pour s'en convaincre. Java est le langage de référence pour les applications Web, mais il ne faut pas le confondre avec JavaScript, unlangage de script HTML.

Ce cours présente l'ensemble des caractéristiques de la programmation à objets et classes, en les adaptant à Java, la méthode de développement et un environnement de développement. On s'inspirera de la notation UML pour décrire les concepts. Le plan indicatif du cours est le suivant :

1. Le langage et la conception à objets
2. La gestion des classes (classes abstraites, interface, héritage, généricité), redéfinitions, typage...
3. Les principales hiérarchies de classes de la bibliothèque Java.
4. IHM (Swing) et applets
5. Eclipse 3 (IDE Java)
6. Compléments : les threads, JNI (Java Native Interface)

Le document est structuré comme suit. Dans les sections qui suivent, nous présentons rapidement le langage Java, ses sources, les concepts du langage, et les principales classes qui forment qui l'environnement de développement.

Nous étudions ensuite plus en profondeur les concepts du modèle à objets de Java dans le chapitre 2. Nous présentons la programmation Java dans le chapitre 3 avec un exemple introductif et dans la section 2, nous étudions quelques hiérarchies importantes du système.

Dans le chapitre 4, nous détaillons l'environnement de programmation Eclipse 3 pour Java. Cet environnement est riche et ouvert (personnalisable par modules). Nous l'illustrons par un programme de jeu de Nim.

Nous étudions quelques concepts complémentaires dans le chapitre 5, tels que les entrées-sorties, les exceptions, l'introspection, la documentation. Nous étudions quelques API dans le chapitre 6 (Applets, Threads, JNI...).

Enfin, dans le chapitre 7, nous étudions rapidement les IHM sous Java, et notamment `Swing`, un outil de développement d'interfaces homme/machine de Java. Les expérimentations sont menées avec Eclipse 3, l'IDE ouvert et libre du consortium autour d'IBM.

## 1 Aperçu de Java

Java est un langage de programmation à objets créé en 1991 pour Sun Microsystem par Bill Joy et James Gosling, dont les caractéristiques s'expliquent simplement par l'héritage combiné de C++ et de Smalltalk :

- De C++ [Cha96, Str87], le standard à objet de l'industrie dans les années 90, Java reprend le typage statique sûr (y compris la coercition de type), un certains nombre de types et opérateurs primitifs, une partie de la syntaxe (affectation, séquence, `null`, les définitions...), les règles de visibilité (privé, public, protégé) ou de définition d'héritage (static), la gestion des exceptions, les itérateurs en tant que classe, la pseudo-variable *this*, etc.
  Ces caractéristiques permettent à Java d'être un langage sûr et robuste.
- De Smalltalk [And05, BS96, How95], le langage à objet de référence, Java reprend l'idée et l'implantation de la machine virtuelle qui permet d'assurer le portage du code en deux niveaux, la gestion des identifiants des objets et de la mémoire (les fameux pointeurs C++), l'héritage simple, la pseudo-variable *super*, une bonne partie des classes de base (notamment pour les collections), une inspiration du modèle MVC (Look and Feel, Wrappers...).
  Ces caractéristiques permettent à Java d'être un langage simple, riche et portable.

Les avantages de l'un étant souvent les inconvénients de l'autre, on peut affirmer que la force de Java est d'avoir été un bon compromis entre rigueur et puissance. Par exemple, les programmeur C++ vont louer l'absence de gestion de pointeurs et de mémoire (les destructeurs de classes), l'absence de pré-processeurs et de multiplication de classes (unions, struct n'existent plus). On pourra malgré tout regretter l'absence (relative) de métaclasses et de protocoles de classe dans Java, bien que le Java *réflexif* soit une tentative de gestion de classe comme ensemble d'objets (ses instances).

Java a aussi été un apport dans la programmation à objets avec une adaptation du concept de paquetage pour la gestion de nom et la structuration simple des bibliothèques, le développement de modules distribués communicables, tels que les *applets*, la gestion de la concurrence avec les *threads*, l'utilisation d'interfaces pour gérer le problème de l'héritage multiple, etc. Le langage a évolué et inclue maintenant la généricité, un concept tout aussi important que l'héritage pour la réutilisation de code. Un avantage important de Java est que les spécifications du langage sont libres et que chacun peut ajouter de nouvelles bibliothèques de classes.

Le succès de Java s'explique d'une part par ses qualités intrinsèques énoncées ci-avant mais aussi qu'il a été la réponse, plus ou moins souhaitée, à un besoin crucial d'outils de développement pour le Web et Internet. L'émergence des applications commerciales, des sites de e-commerce et le développement des communications intra- et inter-entreprise ont démultiplié le besoin de nouvelles applications (laborieusement et naïvement appelées nouvelles technologies de l'information) compatibles avec les navigateurs web (HTML). D'autres standards sont apparus au même moment tel XML [Mic98] pour lesquels Java a été rapidement associé. Java est aussi le langage de prédilection (avec Smalltalk, Squeak ou Python) pour les développeurs libres et passionnés du monde entier. Actuellement, on observe un boom du développement Web autour de Java et des serveurs Web (J2EE, JSP, Servlet, EJB, JBoss, Apache Tomcat, ...). Consulter [**?**, Dja05, Dau04] pour plus de détails à ce sujet.

Nous détaillons par la suite une petite histoire du (jeune) langage, les principales caractéristiques et les références bibliographiques actuelles.

## 2 Petit historique

**Avertissement** : *Cette section est une compilation de plusieurs sources :*
*http://fr.wikibooks.org/wiki/Programmation_Java_Introduction*

*http://fr.wikipedia.org/wiki/Java_(langage)*
*http://jmdoudoux.developpez.com/java/*
*http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/java.html*
*http://www.labo-sun.com/.*
*Le lecteur y trouvera aussi des définitions des acronymes (nombreux autour de la technologie Java-Web).*

## 2.1 Généralités

Le 23 mai 1995, Sun Microsystem présentait une nouvelle plateforme, composée d'un langage de programmation et d'une machine virtuelle. Java était né.

L'histoire de Java commence en fait en 1991, lorsque Sun décide de lancer un projet destiné à anticiper l'évolution de l'informatique, confié à James Gosling, Patrick Naughton et Mike Sheridan. Ce projet, appelé "Green Project", eu comme résultat une plate-forme baptisée Oak, indépendante du système, orientée objet et légère. Oak était initialement destinée à la télévision interactive.

Lorsque Java est révélé en 1995 (Oak a été renommé en Java - café en argo anglais - pour de simples raisons de droit d'auteur), Java profite de l'essor d'Internet en permettant l'un des premiers mécanismes d'interactivité au niveau du poste client : l'applet Java.

Langage orienté objet d'usage généraliste, Java est enrichi par des bibliothèques, des outils et des environnements très diversifiés, standardisés par le Java Community Process (JCP), consortium chargé de l'évolution de Java. Ce consortium regroupe des entreprises, comme Sun, IBM, Oracle, Borland, BEA, des organismes de normalisation, comme le NIST, des organismes du monde Open Source, comme la Fondation Apache et le JBoss Group, et des particuliers.

Il est possible d'utiliser Java pour créer des logiciels dans des environnements très diversifiés :

- Applications sur client lourd (JFC)
- Applications Web, côté serveur (servlets, JSP, Struts, JSF)
- Applications réparties (EJB)
- Applications embarquées (J2ME)
- Applications sur carte à puce (JavaCard)

Ces applications peuvent être enrichies par de nombreuses fonctionnalités :

- Accès à des bases de données (JDBC et JDO)
- Accès à des annuaires (JNDI)
- Traitements XML (JAXP)
- Connexion à des ERP (JCA)
- Accès à des traitements en d'autres langages (JNI)
- Web services (JAX-RPC, JAXM, JAXR)
- Multimédia (Java Media)
- Téléphonie (JTAPI)
- Télévision interactive (Java TV)

Ceci n'est bien sûr qu'un petit échantillon. Il existe bien d'autres bibliothèques.

## 2.2 Frameworks et API

Une Interface de programmation (en anglais Application Programming Interface ou API) définit la manière dont un composant informatique peut communiquer avec un autre. Dans le cas typique d'une bibliothèque, il s'agit généralement d'une liste de fonctions considérées comme utiles pour d'autres composants. Sun fournit un grand nombre de frameworks et d'API afin de permettre l'utilisation de Java pour des usages très diversifiés. On distingue essentiellement 4 grands frameworks :

- J2SE (Java 2 Standard Edition) est le framework destiné aux applications pour poste de travail. Ce framework contient toutes les API de base, mais également toutes les API spécialisées dans le poste client (JFC et donc Swing, AWT et Java2D), ainsi que des API d'usage général comme JAXP (pour le parsing XML) et JDBC (pour la gestion des bases de données).
- J2EE (Java 2 Platform, Enterprise Edition) est une spécification pour le langage de programmation Java de Sun plus particulièrement destiné aux applications d'entreprise. Dans ce but, toute implémentation de cette spécification contient un ensemble d'extension au framework Java standard (J2SE, Java 2 standard edition) afin de faciliter la création d'applications réparties. Voici une liste des API pouvant être contenues dans une implémentation J2EE :
  - Servlets : Conteneur Web
  - JSP : Framework Web
  - JSF : Framework Web, extension des JSP
  - EJB : Composants distribués transactionnels
  - JNDI : API de connexion à des annuaires, notamment des annuaires LDAP
  - JDBC : API de connexion à des bases de données
  - JMS : API de communication asynchrone
  - JCA : API de connexion, servant notamment à se connecter à des PGI
  - JavaMail : API de gestion des mails
  - JMX : Extension d'administration des applications
  - JTA : API de gestion des transactions
  - JAXP : API de parsing XML
  - JAXM : API de communication asynchrone par XML
  - JAX-RPC : API de communication synchrone par XML, par exemple à l'aide du protocole SOAP
  - JAXB : API de serialization par XML
  - JAXR : API de gestion des registres XML, permettant d'enregistrer des Web Services en ebXML
  - RMI : API de communication distante entre des objets java
  - Java IDL : API de communication entre objets Java et objets non-Java, via le protocole CORBA
- J2ME (Java 2 Micro Edition) est le framework Java spécialisé dans les applications mobiles. Des plate-formes Java compatibles avec J2ME sont embarquées dans de nombreux téléphones portables et PDA.
- JavaCard : Ce framework est spécialisé dans les applications liées aux cartes à puces et autres SmartCards

### 2.3 Environnement d'exécution Java

JRE est l'acronyme de Java Runtime Environment ("environnement d'exécution Java") et désigne un ensemble d'outils permettant l'exécution de programmes Java sur toutes les plateformes supportées. JRE est aussi connu sous le nom de JVM (Java Virtual Machine). La machine virtuelle est nécessaire pour exécuter des applets ou des programmes écrits en Java.

Cet outil est à distinguer du JDK (Java Developement Kit) qui permet de compiler du code java et produire du bytecode qui sera interprété par la machine virtuelle sur le poste utilisateur. Les kits de développement Java (Java development kit ou JDK) sont publiés par Sun et utilisables gratuitement.

Habituellement, les environnements de développement intégrés (EDI ou IDE en anglais pour Integrated Development Environment) sont paramétrés par une version de chacun de ces outils.

## 2.4 Historique de Java

Voici les principales dates importantes de Java à ce jour.

* 1991 : Début du projet Oak, qui donnera naissance à Java
* Eté 1992 : première présentation interne des possiblités de Oak. Un appareil appelé "Star Seven" permet de visualiser une animation montrant Duke, l'actuelle mascotte de Java.
* 1994 : Développement de HotJava, un navigateur internet entièrement écrit en Java.
* 23 mai 1995 : Lancement officiel de Java 1.0
* 23 janvier 1996 : Lancement du JDK 1.0
* 29 mai 1996 : Première conférence JavaOne. Java Media et les servlets y sont notamment annoncés.
* 16 août 1996 : Premières éditions des livres "Java Tutorial" et "Java Language Specification" par Sun et Addison-Wesley
* Septembre 1996 : Lancement du site Web Java Developer Connection Web
* 16 octobre 1996 : Spécification des JavaBeans
* 11 janvier 1997 : Lancement des JavaBeans
* 18 février 1997 : Lancement du JDK 1.1
* 28 février 1997 : Netscape annonce que Communicator supporte désormais Java
* 4 mars 1997 : Lancement des servlets
* 10 mars 1997 : Lancement de JNDI
* 5 juin 1997 : Lancement de Java Web Server 1.0
* 5 août 1997 : Lancement de Java Media et de Java Communication API
* Mars 1998 : Lancement des JFC et notamment de Swing
* 8 décembre 1998 : Lancement de Java 2 et du JDK 1.2
* 4 mars 1999 : Support XML
* 27 mars 1999 : Lancement de la machine virtuelle HotSpot
* 2 juin 1999 : Lancement des JSP
* 15 juin 1999 : Formalisation des environnements J2ME, de J2SE et J2EE ; Lancement de Java TV
* Août 1999 : Lancement de Java Phone
* 8 mai 2000 : Lancement de J2SE 1.3
* 9 mai 2002 : Lancement de J2SE 1.4
* 24 novembre 2003 : Lancement de J2EE 1.4
* 30 septembre 2004 : Lancement de J2SE 1.5, nommé également J2SE 5.0

## 2.5 Les versions de Java

Voici les principales versions des spécifications de Java.

| Version | Date | Evolutions |
|---|---|---|
| 0.10 | 15/01/2001 | brouillon : 1ere version diffusée sur le web. |
| 0.20 | 11/03/2001 | - ajout des chapitres JSP et serialization, des informations sur le JDK et son installation, corrections diverses. |
| 0.30 | 10/05/2001 | - ajout des chapitres flux, beans et outils du JDK, corrections diverses. |
| 0.40 | 10/11/2001 | - réorganisation des chapitres et remise en page du document au format HTML (1 page par chapitre) pour faciliter la maintenance<br>- ajout des chapitres : collections, XML, JMS, début des chapitres Swing et EJB |
| 0.50 | 31/04/2002 | - séparation du document en trois parties<br>- ajout des chapitres : logging, JNDI, Java mail, services web, outils du JDK, outils lires et commerciaux, Java et UML, motifs de conception<br>- compléments ajoutés aux chapitres : JDBC, Javadoc, intéraction avec le réseau, Java et xml, bibliothèques de classes |
| 0.60 | 23/12/2002 | - ajout des chapitres : JSTL, JDO, Ant, les frameworks<br>- ajout des sections : Java et MySQL, les classes internes, les expressions régulières, dom4j<br>- compléments ajoutés aux chapitres : JNDI, design patterns, J2EE, EJB |
| 0.65 | 05/04/2003 | - ajout d'un index sous la forme d'un arbre hiérarchique affiché dans un frame de la version HTML<br>- ajout des sections : DOM, JAXB, bibliothèques de tags personnalisés, package .war<br>- compléments ajoutés aux chapitres : EJB, réseau, services web |

| Version | Date | Evolutions |
|---|---|---|
| 0.70 | 05/07/2003 | - ajout de la partie sur le développement d'applications mobiles contenant les chapitres : J2ME, CLDC, MIDP, CDC, Personal Profile, les autres technologies<br>- ajout des chapitres : le multitache, les frameworks de tests, la sécurité, les frameworks pour les app web<br>- compléments ajoutés aux chapitres : JDBC, JSP, servlets, intéraction avec le réseau, application d'une feuille de styles CSS pour la version HTML<br>- corrections et ajouts divers |
| 0.75 | 21/03/2004 | - ajout des chapitres : le développement d'interfaces avec SWT, Java Web Start, JNI<br>- compléments ajoutés aux chapitres : GCJ, JDO<br>nombreuses corrections et ajouts divers notamment dans les premiers chapitres<br>- ajout des chapitres : le JDK 1.5, des bibliothèques open source, des outils open source, Maven et d'autres solutions de mapping objet-relationnel |
| 0.80 | 29/06/2004 | - ajout des sections : Installation J2SE 1.4.2 sous windows, J2EE 1.4 SDK, J2ME WTK 2.1 |
| 0.80.1 | 15/10/2004 | - compléments ajoutés aux chapitres : Ant, Jdbc, Swing, Java et UML, MIDP, J2ME, JSP, JDO |
| 0.80.2 | 07/11/2004 | - nombreuses corrections et ajouts divers |
| 0.85 | 27/11/2005 | - ajout du chapitre : Java Server Faces<br>- ajout des sections : java updates, le composant JTree<br>- nombreuses corrections et ajouts divers |

TABLEAU I– *Historique des versions de Java*

Plus couramment, on trouve les versions du JDK, le kit de développement Java (Java development kit ou JDK) publiés par Sun et utilisables gratuitement :

- JDK 1.0, 1996. Le langage devient stable. La programmation Web y tient une place importante.
- JDK 1.1, 1997. Ajout des JavaBeans, des archives JAR, et des connexions telles que JDBC.
- JDK 1.2, 1998 dite Java 2. Amélioration de JDBC et des collections, ajout de Swing (Java 2D).
- JDK 1.3, 2000.
- JDK 1.4, 2002. Ajout du support XML, des entrées/sorties. Amélioration de JDBC.
- JDK 1.5, 2004 dite Java 5.0 (nom de code Tiger). Ajout de facilités de programmation (autoboxing, annotation, collections typés, généricité, ...). Amélioration des outils XML, JDBC...

## 3 Principales caractéristiques

Le langage Java est dans la lignée directe des langages (à objets) à classes et inspirés de la programmation impérative (variables, état, affectation, séquencement d'actions, procédures). Contrairement à Smalltalk, qui est aussi inspiré de Lisp, les fonctions (le code) ne sont pas des entités à part entière, c'est-à-dire des objets. On y perd en souplesse de codage, mais on y gagne en fiabilité. Les principaux concepts du langage sont les suivants :

- Programmatioin impérative (variables d'état, affectation, séquence et structures de contrôle classiques -alternatives, boucles-, types et opérateurs primitifs),
- Modèle objet (objets, classes, héritage simple, polymorphisme, interfaces, relation d'implantation, classes internes,
- Paquetages, typage et visibilité, généricité (depuis la version 5), coercition (transtypage),
- Gestion des exceptions
- Threads (concurrence)
- Applets (code transportable)
- API d'IHM (AWT, Swing, ...)

– API système (réseau, sécurité, RMI, Web)
– API de connexion (JDBC, XML, JNI)

Java est un langage principalement compilé. En fait, il compile du code pour la machine virtuelle Java, appelé *bytecode*, que celle-ci va interpréter. C'est un concept fondamental, inspiré de Smalltalk. Le principe de la **machine virtuelle** rend le code produit indépendant de la plateforme système (du système d'exploitation hôte) dans la mesure où on dispose pour chaque plateforme système d'un support d'exécution. Le code devient ainsi portable. On peut aussi archiver (et compresser) une ensemble de classes dans un fichier **jar** et définir ainsi une bibliothèque invocable lors de l'exécution (locale ou déportée) d'un programme. On utilise aussi les archives comme composants téléchargeables. L'inconvénient majeur étant que l'interprétation du *bytecode* (cela peut aussi être une compilation à la volée) est plus coûteuse à l'exécution que celle d'un programme binaire, issu d'une compilation classique, comme en C++. De manière générale, un programme Java sera donc un langage moins efficace qu'un programme C++.

Java est un langage **typé fortement et statiquement**, à l'instar de C++ et avec des règles sensiblement équivalentes. Cela signifie que toute exécution d'un programme Java n'engendrera pas d'erreur de typage. Par comparaison, Smalltalk est typé dynamiquement, cela signifie que le type d'une variable n'est connu qu'à l'exécution. Les contrôles ont donc lieu lors du déroulement du programme Smalltalk, ce qui diminue son efficacité et peut donner des erreurs de type à l'exécution. Le prix à payer pour la détection des erreurs de type lors de la compilation (en Java ou C++) est une plus grande rigueur dans le typage des expressions, une mise au point souvent plus pénible, et surtout la multiplication d'opérations de changement de type (le fameux *cast* ou plus poétiquement l'**affectation polymorphique inverse**).

Comme dans UML [AV01b] ou Smalltalk, les objets de Java ont un **identifiant implicite**, géré par le système, qui correspond aux pointeurs de C++. Déléguer la gestion des pointeurs (et donc la mémoire) au système apporte un confort non négligeable au programmeur et un gain de temps dans l'écriture et la mise au point des programmes. La gestion de la mémoire s'inspire de Smalltalk. Le ramasse-miette (*garbage collector*) se charge de récupérer la place mémoire occupée par les objets inutilisés. Il n'y a donc pas besoin d'écrire des fonctions de destruction d'objets, dont la difficulté majeur était la destruction en chaîne d'objets dépendants (ou pire interdépendants).

La plate-forme Java fut l'un des premiers systèmes à offrir le support de l'exécution du code à partir de sources distantes. Une applet peut fonctionner dans le navigateur Web d'un utilisateur, exécutant du code téléchargé d'un serveur HTTP. Le code d'une applet fonctionne dans un espace très restrictif, ce qui protège l'utilisateur des codes erronés ou mal intentionnés. Cet espace est délimité par un objet appelé gestionnaire de sécurité. Un tel objet existe aussi pour du code local, mais il est alors par défaut inactif. Le gestionnaire de sécurité (la classe `SecurityManager`) permet de définir un certain nombre d'autorisations d'utilisation des ressources du système local (système de fichiers, réseau, propriétés système,...). Une autorisation définit

1. un code accesseur (typiquement, une applet - éventuellement signée - envoyée depuis un serveur web) ;
2. une ressource locale concernée (par exemple un répertoire) ;
3. un ensemble de doits (par exemple lire/écrire).

Les éditeurs d'applet peuvent demander un certificat pour leur permettre de signer numériquement une applet comme sûre, leur donnant ainsi potentiellement (moyennant l'autorisation adéquate) la permission de sortir de l'espace restrictif et d'accéder aux ressources du système local.

Java est un des langages adoptés par le monde de l'*Open Source*, ce fut la politique de lancement de Java par Sun, offrir les spécifications et les briques de base (JDK, SDK). Cela

signifie aussi que beaucoup de développeurs contribuent à enrichir les bibliothèques de Java et des outils associés. Ainsi bon nombre d'IDE gratuits et modulaires tels que JBuilder, NetBeans ou Eclipse permettent à tout programmeur de développer en Java et à tout d'éveloppeur de proposer de nouveaux modules.

## 4 Bibliographie de référence

Une bibliographie de référence, à ce jour, est la suivante :
- Les bases du langage [Cha03, Ses05, Fla97b, Fla97a]
- la programmation avec Java [Cla03, SM03, Puy04, NP96, Eck02, BB05]
- le développement avec J2EE et EJB [**?**, CFS03]
- le développement avec les patterns [SM03, DCM03, BM02, AMC01]
- Eclipse et Java [Hol04, Dau04, Dja05, **?**]
- Cours, résumé (non lus) [Che05, GR05]

Pour la webographie, le nombre de références est quasi-infini. Nous conseillons le site de sun (la spécification des API), le très riche site de référence du développement avec de nombreux pointeurs et conseils et le site de l'encyclopédie libre pour les définitions précises.
- `http://java.sun.com/reference/api/index.html`
- `http://java.developpez.com/cours/`
- `http://fr.wikipedia.org/wiki/Java_(langage)`

Nous remercions Gilles Ardourel pour ses conseils sur Java et ses relectures.

# Chapitre 2

# Le modèle à objets de Java

Ce chapitre présente le langage de Java, c'est-à-dire les concepts et la syntaxe des programmes (sources) Java. La référence est celle de l'API de Sun :
http://java.sun.com/j2se/1.5.0/docs/api/
Outre cette référence et les sources bibliographiques de la section 4 du chapitre 1, nous nous sommes aussi inspirés de

- http://www.esil.univ-mrs.fr/ tourai/Java/
- http://java.developpez.com/cours/
- http://www.unix.org.ua/orelly/java-ent/jenut/

## 1 Java, un langage à objets ?

Dans un langage à objets pur, comme Smalltalk, il y a seulement deux concepts de base : l'**objet** et l'**envoi de message**. Tout le reste du langage est dérivé de ces deux concepts. Java est n'est pas un langage à objets pur car il combine des éléments de programmation impérative classique aux concepts de la programmation à objets. Reprenons à ce sujet le texte de http://fr.wikipedia.org/wiki/Java_(langage) :
*Java n'est pas un langage totalement objet pour plusieurs raisons :*

- *Tout n'est pas objet en Java ; on retrouve entre autres des types primitifs qui ne sont pas des objets ("int", "double", etc.), même si depuis J2SE 5.0, l'autoboxing rend interchangeables les types primitifs et leurs wrappers.*
- *Une classe n'est pas un objet (seulement en apparence). Toutes les classes en Java héritent d'"Object", sauf "Object". Les classes sont manipulées comme des objets en Java. Elles peuvent être passées en paramètre, etc. Toutefois, les classes ont un traitement spécifique dans la machine virtuelle.*
- *Il est impossible de redéfinir ou modifier la métaclasse "Class".*
- *L'introspection est la capacité d'un programme de s'observer et de raisonner sur son propre état. L'intercession est la capacité d'un programme de modifier son propre état d'exécution et d'altérer son interprétation ou son sens. Java ne procure qu'une certaine capacité d'introspection. On peut savoir quelle méthode détient une classe, les invoquer, analyser la structure d'un objet, mais il est impossible de modifier la métaclasse "Class" ou d'ajouter d'autre métaniveaux.*

*Des langages comme Smalltalk et ObjVLisp supportent certaines formes d'intercession : on peut contrôler totalement l'interprétation d'un programme, car le compilateur est lui-même écrit en Smalltalk. On peut également changer l'ordre de recherche des méthodes, etc. On peut changer le comportement des classes en modifiant les métaclasses. Tout ceci est impossible en Java. Ceci peut se comprendre parce que Java est un langage industriel. En Java des objets peuvent transiger par internet, etc. Il est donc important dans ce contexte d'avoir une certaine protection contre d'autres programmes pouvant rouler dans la même machine virtuelle. Il y*

*aussi certaines modifications qui ont été faites pour des fins d'optimisation. Ainsi, il existe des types primitifs qui ne sont pas des objets pour assurer de meilleures performances.*

*En Java, il existe des extensions réflexives pour lui procurer de plus grande capacité réflexive. Par exemple, OpenJava, Javassist, Reflective Java, pour n'en nommer que quelques-uns. On dénote principalement trois approches principales parmi ces extensions : un changement du chargeur de classes (class loader), un préprocesseur de code Java et un changement de la machine virtuelle. On note que l'utilisation d'une machine virtuelle modifiée rend le code non portable vers une machine virtuelle standard. Ces approches permettent d'augmenter les capacités de Java, si cela s'avère nécessaire.*

## 2 Modèle Impératif

Dans cette section, nous synthétisons les types primitifs avec les valeurs et opérateurs associés, ainsi que les structures de contrôle classiques de la programmation impérative.

### 2.1 Types primitifs et constructions de base

Comme indiqué dans l'introduction, Java est un langage fortement typé. Dans cette section, nous synthétisons les types primitifs avec les valeurs et opérateurs associés, ainsi que les structures de contrôle classiques de la programmation impérative.

#### Types primitifs et valeurs

Les types primitifs (ou de base) sont essentiellement des types représentant des nombres (`byte`, `short`, `int`, `long`, `double` et `float`), des booléens (`boolean`) et des caractères (`char`).

| nom | type | intervalle | classes | valeurs |
|---|---|---|---|---|
| byte | entier 8 bits | de -128 à127 | Byte | 50 |
| short | entier 16 bits | de -32768 à 32767 | Short | 2500 |
| int | entier 32 bits | de -2.14e9 à 2.14e9 | Integer | 25000 |
| long | entier 64 bits | entiers longs | Long | 160 |
| float | réels 32 bits | virgule flottante simple | Float | 23.65 |
| double | réels 64 bits | virgule flottante double | Double | 3444.555 |
| boolean | booléen | true, false | Boolean | true |
| char | caractère | caractère UNICODE† | Character | 'a' |
| String | chaîne de caractères | chaîne de caractères‡ | String | "abc" |

TABLEAU II– *Types primitifs de Java*

† les caractères ASCII représentent la première partie du code.
‡ c'est une classe mais pouvant être utilisée comme n'importe quel type

Toutes les chaînes de caractères Java mémorisent des caractères de type char, donc des caractères Unicode codés sur 16 bits. Les chaînes de caractères sont représentées en Java par les classes `String` ou `StringBuffer`. Ces deux classes ne jouent pas exactement le même rôle : `String` est utilisé pour représenter les chaînes de caractères constantes, qui peuvent être partagées sans risque par plusieurs threads, puisque leur contenu ne peut pas être changé, `StringBuffer` est utilisé pour les chaînes de caractères dont on veut modifier le contenu.

La classe `String` représente les chaînes de caractères constantes telles que "abc", qui en sont des instances. Les chaînes sont partageables car constantes. La déclaration de chaîne `String str = "abc";` est équivalent à :

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Voici des exemples d'utilisation

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
// + est la concaténation
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

Les notations habituelles en C des valeurs de types caractères sont reprises en Java (`\n`, `\t`, `\b`, `\r`, `\'`, `\"`, `\ddd` , `\uxxxx`) `\ddd` représente un caractère octal, `\uxxxx` représente un caractère unicode.

Une variable est un élément du langage (voir en section 4.4 les différentes sortes de variables) qui contient un identifiant d'objet ou une valeur. La valeur `null` représente une référence indéterminée (pour un objet). Une déclaration de variable se note :

```
type var = expression ;
```

Pour unifier objet et valeurs, on peut convertir des types primitifs en classe (*boxing*) et vice-versa (*unboxing*). Le tableau II met en évidence cette correspondance. Dans Java 1.5, cette conversion devient implicite (*autoboxing*). Le compilateur se chargera de convertir (transtyper) automatiquement la variable dans sa classe. On parle alors, comme en Smalltalk, d'**objets littéraux** sont des objets connus et désignés par leur valeur (unique) : `null` désigne un objet vide, `55` pour un entier, `#toto` pour un symbole, `"toto"` pour une chaîne de caractères, etc.

### Tableaux et énumérations

Un tableau Java est une suite de taille fixe de valeur, on parle de variable indexée. Une déclaration de variable (indexée) de type tableau se définit ainsi :

```
type[] var = expression ;
type var[] = expression ;
```

où l'expression d'initialisation est une instanciation générique de la forme `new Type [nb]` ou une énumération `{v1, v2, ...}`. La seconde forme est à proscrire pour la lisibilité du typage. Par exemple,

```
int tab[] = new int[5];
int [] tab = {1, 2, 3, 4, 5};
int [][] mat = new int[5][3];
for (int i=0;i<mat.length;i++) {
    for (int j=0;j<mat[i].length;j++) {
        System.out.print("mat["+i+","+j+"] = " + mat[i][j]);
    }
}
```

Ce type de tableau correspond à un usage courant pour les paramètres et les tableaux de taille fixe (0..n-1). La classe de prédilection pour les tableaux dynamiques est a priori la classe `Vector` du paquetage `java.lang.Vector` (classe synchronisée au sens des threads), mais la classe `ArrayList` du paquetage `java.util` nous semble plus intéressante car plus proche des collections (conception plus abstraite).

Les énumérations peuvent être implantées par des collections ou des constantes (de classe).

```
public class JourDeLaSemaine
{
    public JourDeLaSemaine(){}

    public static final JourDeLaSemaine lundi = null;
```

```
    public static final JourDeLaSemaine mardi = new Jour();
    public static final JourDeLaSemaine mercredi = new Jour();
    public static final JourDeLaSemaine jeudi = new Jour();
  ...
  }
```

ou

```
public class JourDeLaSemaine
{
   public static final int lundi = 1;
   public static final int mardi = 2;
   public static final int mercredi = 3;
   public static final int jeudi = 4;
   ...
}
```

Cependant, cette version comporte plusieurs désavantages [1]. Elle n'est pas « type-safe » [2], c'est-à-dire qu'elle n'empêche pas d'affecter une valeur inconnue à la variable de type `int`, entraînant des erreurs à l'exécution, qui ne sont donc pas levées à la compilation. Pour ajouter une valeur possible au type, il faut éditer la classe et tout recompiler. L'utilisation de ces constantes est souvent préfixée par le nom du type, sans pour autant posséder d'espace de nommage propre au sein du programme. Elle ne fournit pas de correspondance entre la valeur de la constante et sa description (comme il est d'usage de le faire pour une collection d'exceptions par exemple). D'un point de vue purement objet, ce n'est pas une classe, car un des paradigmes objet définit une classe comme un ensemble qui encapsule des données et des méthodes permettant de traiter ou manipuler celles-ci. Ici, il n'y a pas de méthodes.

Le J2SE 5.0 fournit un type énuméré « type-safe » en standard via le mot clé `enum`:

```
public enum JourDeLaSemaine {lundi, mardi, mercredi, ...};
```

### Opérateurs primitifs et expressions

L'affectation est désignée, comme en C, par le symbole `variable = expression`. La variable peut désigner un objet, une propriété, une valeur d'un type simple, un tableau... Une affectation peut figurer en membre droit d'une autre affectation (par exemple `v1 = v2 = 10`).

Une expression primitive est formée à partir d'opérateurs associés aux types primitifs dans un ordre fixé par des priorités, reconnues par le compilateur. Le tableau III est une synthèse des opérateurs avec leur priorité inspirée de [Cla03] et
http://www.esil.univ-mrs.fr/∼tourai/Java/
Exemples :

```
int i = 4 + 3;
long i = 0x7effffffffffffL ;    // 9151314442816847871
float f = i ;                    // 9.15131e+18
long l = (long)f ;
int i = 90 + (int)2.5;
```

L'évaluation d'une expression peut conduire à une erreur. Dans ces cas, Java lève une **exception**. Par exemple :

1. http ://lroux.developpez.com/article/java/tiger/ ?page=page_2
2. Un type énuméré « type-safe » est un type énuméré qui provoque des erreurs de compilation si l'on tente de lui affecter des valeurs non permises. Il existe bien entendu des patterns permettant de créer ces types énumérés, qui de plus contiennent des méthodes de traitement et leur associent des descriptions.

- `OutOfMemoryError` : Cette erreur est produite lorsque l'espace mémoire requise est insuffisante. Cette erreur est générée lors de la création (dynamique) des objets d'une classe, des tableaux, des chaînes de caractères.
- `ArrayNegativeSizeException` : Une des dimensions d'un tableau est négative
- `NullPointerException` : La valeur d'une référence d'un objet est null.
- `IndexOutOfBoundsException` : la valeur de l'indice d'un tableau est hors des bornes du tableau.
- `ClassCastException` : l'opération de cast n'est pas permise.
- `ArithmeticException` : l'opérande droit d'une division ou d'un modulo est nul.
- `ArrayStoreException` : la référence que l'on veut affecter à un élément d'un tableau n'est pas du bon type.
- etc.

Les exception sont étudiées dans la section 1 du chapitre 5.

| Pré† | Ordre | Opérateur | Type | Description |
|---|---|---|---|---|
| 1 | droite gauche | `++, -` | Arithmétique | Incrémentation et décrémentation |
| 1 | droite gauche | `+, -` | Arithmétique | Plus et moins unaire |
| 1 | droite gauche | $\sim$ | Entier | complément bit à bit |
| 1 | droite gauche | `!` | Booléen | non booléen |
| 1 | droite gauche | `., []` | objet | Accès aux membres |
| 1 | droite gauche | `(type)` | expr | Coercition (transtypage) |
| 2 | gauche droite | `*, /, %` | Arithmétique | Multiplication, division et reste |
| 3 | gauche droite | `+, -` | Arithmétique | Addition et soustraction |
| 3 | gauche droite | `+` | Chaînes de car. | Concaténation de chaînes |
| 4 | gauche droite | `«, », »>` | Entier | Décalage de bits |
| 5 | gauche droite | `<,<=, >, >=` | Arithmétique | Comparaison arithmétique |
| 5 | gauche droite | `instanceof` | objet | Comparaison de type |
| 6 | gauche droite | `==, !=` | objet et valeurs | égalité et différence |
| 7 | gauche droite | `&` | Entier et booléen | ET bit à bit et booléen |
| 8 | gauche droite | `^` | Entier et booléen | OU exclusif bit à bit et booléen |
| 9 | gauche droite | `\|` | Entier et booléen | Ou bit à bit et booléen |
| 11 | gauche droite | `&&` | Booléen | ET conditionnel |
| 11 | gauche droite | `\|\|` | Booléen | OU conditionnel |
| 12 | droite gauche | `(expr) vv : vf` | Bool, expr, expr | Si-alors-sinon |
| 13 | droite gauche | `=,` | Variable, expr | affectation |
| | | `*=, /=, % =, +=, -=, «=, »=, »>=, &=, ^=, \|=` | Variable, expr | Affectation avec opérations |

TABLEAU III– *Opérateurs des types primitifs de Java*

† Précédence.

## 2.2 Structures de contrôle

Les structures de contrôle servent à ordonnancer les instructions. Elles sont similaires à celles de C (ou C++).

**Séquence** La principale structure de contrôle est la séquence `instr`$_1$ `;` `instr`$_2$ qui indique que `instr`$_1$ est exécutée avant `instr`$_2$.

**Résultat** L'instruction `return`, où quelle soit, termine une méthode. Avec `return expr`, `expr` sera renvoyée à la méthode appelante.

**Alternatives** Structures conditionnelles
Instructions gardées.

```
if (<expression booléenne>) {
   instruction(s)
}
```

Alternative simple

```
if (<expression booléenne>) {
    instruction(s)
}
else {
    instruction(s)
}
```

Alternatives multiples

```
if (<expression booléenne>) {
    instruction(s)
}
else if (<expression booléenne>) {
    instruction(s)
}
else if (<expression booléenne>) {
    instruction(s)
}
else {
    instruction(s)
}
```

Alternatives par cas (un type simple est `int` ou `char`).

```
switch (<expression de type simple >) {
   case <valeur de type simple>:
        instruction(s)
        break;
   case <valeur de type simple>:
       instruction(s)
       break;
    [...]
   default:
        instruction(s)
        break;
}
```

La commande `break` sort d'une clause contenue dans un `switch`. Il est fortement conseillé d'utiliser cette commande pour éviter des erreurs de maintenance, c'est l'inverse pour les boucles.

**Répétitives**

Boucle tant que :

```
while (<expression booléenne>) {
    instruction(s)
}
```

Boucle répéter :

```
do {
    instruction(s)
}
while (<expression booléenne>);
```

Boucle pour :

```
for (< initialisation > ; <condition> ; <expression d'incrémentation>) {
   instruction(s)
}
```

Boucle pour générique (java 1.5) :

```
for(<variable de stockage de l'occurrence suivante de la collection> : <Collection d'objets>){
    instruction(s)
}
```

La commande `break` sort immédiatement la boucle en cours (`for`, `while`, `do`). La commande `continue` termine l'itération en cours et reprend à la prochaine.

Illustrons ces structures répétitives par quelques exemples inspirés de
http://www.developpez.org/club/bkostrzewa/td-bases/epargne.html

*Vous avez déposé 10000 euros à la banque et vous souhaitez savoir comment cette somme va évoluer. Chaque année elle rapporte 4,5/Essayons de connaitre le montant de votre capital après 5 ans. Il faut multiplier le capital initial par 1,045 cinq fois de suite. Nous pouvons utiliser une boucle `for`.*

```
double capital=10000;
for (int i=1; i≤5; i++) capital=capital*1.045;
```

L'initialisation consiste à mettre le compteur `i` à 1. La condition pour laquelle on exécute les instructions est `i<=5`. La méthode de transition d'un passage à l'autre dans la boucle consiste à ajouter une unité au compteur `i`, ce qui se traduit par `i++`.

```
double capital=10000;
double taux=1.045;
//traduire l'argument en entier
int n=Integer.parseInt(args[0]);
//boucle pour répéter n fois le calcul
for (int i=1; i≤n; i++)
    capital=capital*taux;
```

Modifions légèrement notre problème. Nous avons déposé 10000 euros F à la banque et nous souhaitons savoir combien d'années il faudra attendre pour que le capital obtenu dépasse 15000 euros. Nous allons utiliser une boucle `while` pour indiquer que tant que le capital ne dépasse pas 15000 euros il faut attendre une année de plus. Cela se traduit par :

```
int n=0; //nombre d'années
double capital=10000;
while (capital<15000) {
  capital=capital*1.045;
  n++;
}
```

La condition à vérifier pour que la boucle s'exécute est simplement capital<15000 (on n'a pas encore atteint les 15000 euros attendus). A chaque passage dans la boucle on calcule la nouvelle valeur du capital et on augmente d'une unité le nombre d'années.

### 2.3 Structure d'un programme

Comme tous les programmes impératifs, en Java le programme principal est une fonction `main`. La différence est que cette fonction se trouve dans une classe, et qu'il peut y en avoir plusieurs dans un programme (de classe différente). En cas d'ambiguïté à l'exécution, on doit déterminer celle qu'il faut exécuter. Reprenons l'exemple du code précédent.

Listing 2.1 – Code de la classe Epargne

```
public class Epargne {

    public static void main(String args[]) {
     double capital=10000;
     double taux=1.045;
     //traduire l'argument en entier
     int n=Integer.parseInt(args[0]);
     //boucle pour répéter n fois le calcul
     for (int i=1; i≤n; i++)
      capital=capital*taux;
     //fonctions d'entrée-sortie qui ne nécessite pas
     //d'inclusion de paquetage
     System.out.print("Après "+n+" années, le capital vaut ");
     System.out.println(capital+" F.");
    }
}
```

Le tableau d'arguments `args` sont ceux d'une commande en ligne dans le système d'exploitation hôte, comme pour un programme C.

### 2.4 Compléments sur les types primitifs

Consulter l'archive jointe.

## 3 Modèle à classes

Dans cette section, nous présentons les éléments principaux du modèle à objets de Java, qui est un modèle à classes, typique du Génie Logiciel.

NB : L'étude de différentes sources bibliographiques montre que le vocabulaire utilisé pour la présentation varie d'un auteur à l'autre. Plusieurs raisons l'expliquent : tradutions variables, erreurs de tansmission ou de retranscription et surtout expérience de l'auteur dans différents langages de programmation (et donc influence des langages connus sur l'acquisition des nouveaux langages). Nous ne dérogerons sans doute pas à cette règle. Néanmoins nous conseillons vivement de consulter la page web suivante,pour bien parler le Java.
http://java.sun.com/docs/glossary.html

### 3.1 Objet et classe

Java est un langage à objets.

**Principe 3.1 (objet)** *Toute entité en Java est soit une valeur d'un type primitif soit un objet. Tout objet a une* ***identité*** *propre et implicite.*

Chaque objet a un numéro (une identité) distinct dans le système. Ce qui différencie deux objets, c'est leur identité. Un **objet** sert à stocker et accéder à des informations propres et à envoyer ou répondre à des messages. Un objet a un état et un comportement . L'état

est l'ensemble des valeurs que détient l'objet. Le comportement est l'ensemble des opérations (procédures ou fonctions), appelées **méthodes** que l'objet peut réaliser. Par exemple, prenons un point `p1 = (5,6)` défini par ses coordonnées cartésiennes. L'état est décrit par les deux coordonnées `5` et `6` dans le plan. Le comportement est `déplacer`, `distance`, etc. On dit qu'un objet **encapsule** des données (l'état) et des traitements sur ces données (le comportement) au sein d'une même entité.

**Principe 3.2 (classe)** *Tout objet est instance d'une classe. On dit qu'un objet est instancié par la classe. Les termes instance et objet sont synonymes.*

La **classe** regroupe les objets ayant même structure (même forme d'état) et même comportement. Par exemple, notre point `p1` est défini dans une classe `Point`. La **structure** est définie par un ensemble de **variables d'instances**[3]. Par exemple, la structure du point `p1 = (5,6)` est formée de deux variables d'instance `x` et `y`. Le **comportement** est décrit par un ensemble de méthodes Une **méthode** est une abstraction procédurale (procédure, fonction) définie par un **profil** et un **corps**. Le profil comprend un nom de méthode, appelé **sélecteur** de la méthode et des noms de variables en paramètres. Syntaxiquement, une méthode se présente comme une fonction C :

```
// Déclarations de méthodes
  Visibilite  TypeDeRetour nom\_methode (Type_1 nom_Param1 ,... , Type_n nom_Param_n)
 {
   // Corps de la methode
 }
```

La visibilité correspond aux droits d'utilisation (section 7.3). Les types sont des types primitifs ou des classes. Contrairement au langage C++ , la définition effective des méthodes de la classe doit se faire dans la définition de la classe elle-même.

**Définition 3.3 (propriété)** *On appelle **propriété** l'ensemble des variables et méthodes d'un objet.*

Chaque objet détient les valeurs de ses variables d'instances mais son comportement est situé au niveau de sa classe. L'objet est relié à sa classe par la **relation d'instanciation** Nous avons donc schématiquement l'implantation suivante :

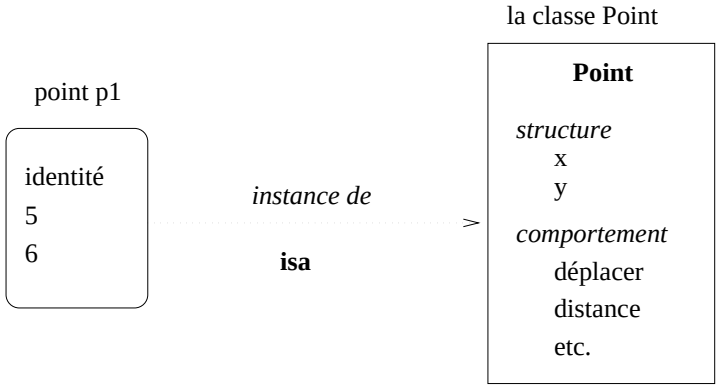



Figure 1 : *Relation d'instanciation du point p1*

Le point `p1` a pour coordonnées `x = 5` et `y = 6`. Ses méthodes sont regroupées dans la classe `Point` avec qui il est relié par une relation d'instanciation. L'état d'un objet est alors défini par la valeur de ses variables d'instance. C'est ce qui permet de comparer deux objets

3. Appelé aussi **champ** (*field*) par inspiration du langage C. Le terme **attribut** est lui inspiré d'UML.

d'une même classe. En fait, chaque objet est unique dans le système : l'objet possède une identité.

La classe représente à la fois l'ensemble de ses instances et un modèle des instances. Dans le comportement on distingue les méthodes applicables aux objets, ce sont les **méthodes d'instance**, des méthodes applicables à la classe elle-même, appelées **méthodes de classe**. Les méthodes de classe servent notamment à créer les objets de la classe. Contrairement à Smalltalk, dans lequel la méthode de classe est une méthode d'instance de la (méta)classe, les méthodes de classes en Java sont (implantées par) des méthodes annotées par le mot-clé `static` (qui correspond à une option de compilation).

Les méthodes sont classées (par convention) en constructeurs, accesseurs, modificateurs, destructeurs (voir section 7.4). Les constructeurs sont des méthodes de classe obligatoires qui servent à instancier des objets de la classe en question. Chaque classe définit au moins un constructeur par défaut (sans paramètres). Les accesseurs servent à accéder aux variables et préserver l'encapsulation. Le destructeur est fondamental dans la gestion de la mémoire en C++. En Java, il est implanté par une méthode protégée `finalize` utilisé par le ramasse-miette uniquement dans certaines situations spécifiques telles que la terminaison de processus.

Définissons par exemple une classe `Point`[4], dont le code est donnée dans le *listing 2.2*. Cet exemple nous servira aussi à illustrer d'autres concepts non encore abordés.

Ce code produit le résultat suivant :

```
p0 : PointPA@16f0472 (x = 0.0 ; y = 0.0)
p1 : PointPA@1e5e2c3 (x = 5.0 ; y = 0.0)
p2 : PointPA@18a992f (x = 10.2 ; y = 5.0)
```

Une classe qui ne peut être instancié est qualifiée de **classe abstraite**. De même, une méthode dont le corps n'est pas défini est dite **méthode abstraite** . Leur déclaration est préfixée par le mot-clé `abstract`. Les classes abstraites servent à structurer le graphe d'héritage et factoriser le code commun. Les méthodes abstraites servent au typage et au polymorphisme (voir section 6.6).

## 3.2 Envoi de message

Un envoi de message est un appel d'une méthode d'un objet.

**Définition 3.4 (envoi de message)** *L'**envoi de message** est le mécanisme de la programmation à objets qui permet d'invoquer une méthode sur un objet. On parle aussi d'invocation de méthode, d'appel de méthode.*

Comme en C++, l'accès aux propriétés d'un objet se note `objet.propriété` (notation pointée). Ces accès sont soumis aux règles de visibilité définies pour la propriété en question (section 7.3). Néanmoins, nous conseillons de respecter le principe d'encapsulation, fondamental en conception à objets. Implicitement cela signifie que que les variables doivent être privées (ou protégées). Ce principe a été respecté dans la classe `PointPA` du *listing 2.2*.

**Principe 3.5 (encapsulation)** *Un objet n'est accessible que par **envoi de message**.*

L'objet destinataire de l'envoi de message est appelé **receveur**. On parle de **sélection simple** car il n'y a qu'un seul receveur. Syntaxiquement, l'envoi de message s'écrit comme un appel de fonction en programmation impérative.

var_resultat = expr_obj.selecteur(par_effect_1,...,par_effect_n)

4. Il y en a plusieurs dans le paquetage `java.awt` : Point2D, Point, Point2D.Double, Point2D.Float).

Listing 2.2 – Code de la classe PointPA

```
package points;

public class PointPA {
    // Les variables d'instance
    private double x;
    private double y;

    /* constructeurs */
    // constructeur par défaut
    public PointPA() {
        x = 0.0;
        y = 0.0;
    }
    // constructeur avec initialisation
    public PointPA(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /* méthodes d'instance */

    // Un modificateur
    public void setXY(double px, double py) {
        x = px;
        y = py;
    }
    // les accesseurs
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }

    // les convertisseurs
    public String toString() {
        String res = super.toString()+" (";
        res = res + "x = " + this.getX()+ " ; y = " + y + ")";
        return res;
    }

     public static void main(String args[]) {
        PointPA p0, p1, p2;
          p0 = new PointPA();
          p1 = new PointPA(5.0, 0.0);
          p2 = new PointPA(10.2, 5);
          System.out.println("p0 : "+p0.toString());
          System.out.println("p1 : "+p1.toString());
          System.out.println("p2 : "+p2.toString());
      }
}
```

Le receveur est le résultat de l'évaluation de l'expression `expr_obj`. Le sélecteur est le nom de la méthode définie pour l'objet receveur. Par exemple, dans la méthode `main` de la classe `PointPA` du *listing 2.2*, la méthode `asString` est envoyée à chaque point, qui renvoie une chaîne de caractères (`return res`).

### 3.3 Interface

Une **interface** est une classe abstraite particulière qui ne définit pas de variables d'instance mais uniquement des méthodes abstraites et des constantes de classe (static + final). Le concept d'interface est similaire à celui de la programmation modulaire (C, C++) qui sépare les déclarations (header) de l'implantation (body). En Java, les interfaces servent surtout comme substitut à l'héritage multiple pour satisfaire aux règles de typage dans le polymorphisme.

## 4 Eléments du langage

Cette section détaille les principaux éléments du langage et de la programmation Java.

### 4.1 Paquetage

Le code Java est organisé en **paquetages** (*package*) et fichier source (`.java`), qui forment un espace de nommage unique et global. L'originalité de l'organisation est dans sa simplicité de mise-en-œuvre. Les paquetages sont organisés (logiquement) comme des répertoires (du système de fichier du système d'exploitation hôte). Par défaut, les noms de répertoires servent de noms de paquetage.

Par convention, on place chaque classe `MaClasse` dans un fichier source qui porte son nom `MaClasse.java`. Pour faire référence à une classe ou un ensemble de classe, il suffit d'importer le paquetage correspondant (clause `import`). Le nommage hiérarchique des paquetages utilise le point comme séparateur. Par exemple, `java.awt.geom.Point2D` indique que la classe `Point2D` se trouve dans le paquetage `geom` qui est lui-même dans le paquetage graphique de base `awt` qui est dans le paquetage `java`, la racine des classes de base du langage.

Le compilateur génère pour chaque classe `MaClasse` un fichier compilé `MaClasse.class` qui contient sous forme *bytecode* une description de la classe. On les trouve aussi dans les archives `.jar`. La variable d'environnement `CLASSPATH` permet au compilateur et à la machine virtuelle de chercher les programmes `.class`. Le paquetage par défaut est le répertoire courant. Pour plus de détails, consulter ([Cla03], p. 79) et
http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html

### 4.2 Mots réservés

Voici une liste de mots-clés proposée par http://fr.wikipedia.org/wiki/Java_(langage)

| | | | | |
|---|---|---|---|---|
| abstract | else | instanceof | strictfp | boolean |
| assert (JDK 1.4) | enum (JDK 1.5) | interface | super | byte |
| break | extends | native | switch | char |
| case | final | new | synchronized | double |
| catch | finally | package | this | float |
| class | for | private | throw | int |
| const (Non utilisé) | goto (Non utilisé) | protected | throws | long |
| continue | if | public | transient | short |
| default | implements | return | try | void |
| do | import | static | volatile | while |
| false | true | null | | |

Voir aussi http://cui.unige.ch/java/JAVAF/BNFidxkw.html pour plus de détails sur leur signification.

### 4.3 Commentaires

Les commentaires lignes sont précédés de //.
Les commentaires multilignes sont cernés par /* */.
Les commentaires JavaDoc sont cernés par /** */ et contiennent un texte HTML. Nous y reviendrons à la page 127.

### 4.4 Variables

En Java, les **variables** sont déclarées par un nom, un type et des attributs. Elle peut être initialisée à la déclaration.

```
Attributs*  Type nom_var = expr_init ;
```

Le nom est constitué de lettres, chiffres et du caractère '_' (varie selon le compilateur). Il débute par une lettre (par convention, c'est une minuscule pour les variables d'instance) ou '_'. Les types sont des types primitifs ou des classes. Les attributs correspondent à la visibilité, au niveau et aux options de compilation... L'accessibilité et l'utilisation d'une variable sont étudiés en détail dans la section 7.3.

On distingue plusieurs sortes de variables :

- les variables liées à une expression en cours d'évaluation : **variables locales**,
- les variables liées à une méthode : **variables locales**,
- les variables propres à une instance : **variables d'instance**,
- les variables partagées par les instances d'une classe (et des sous classes) : **variables de classes**.
- les variables globales.

Il existe deux noms de variables prédéfinies `this` et `super`. La pseudo-variable `this` (vient de C++) désigne le receveur dans le contexte d'une méthode. Elle permet au receveur d'un message de s'envoyer un message. La pseudo-variable `super` (vient de Smalltalk) est relative à l'usage combiné de deux méthodes de même sélecteur dans des classes liées par héritage. Nous les utilisons dans le programme du *listing 2.2* mais leur rôle ne sera précisé que dans la section 6, relative à l'héritage et au polymorphisme. Dans la `PointPA`, l'envoi de message `super.toString()` invoque la méthode d'instance `toString()` de la classe `Object` du paquetage `java.lang` et définie comme suit :

```
public String toString() {
return getClass().getName() + "@" + Integer.toHexString(hashCode());
 }
```

Une **constante** est déclarée dans des classes ou des méthodes en utilisant le mot-clé `final`. Par convention, le nom des constantes est en majuscules afin de les distinguer sans équivoques des variables.

```
final double PI = 3.1415;
final double EURO_FR = 6.55957;
final String BLACK_COLOR = "#000000";
```

Nous reviendrons sur ce modificateur dans la section 7.2.

## 5 Affectation, égalité, copie

L'affectation d'un objet à une variable se fait par `variable = objet`, où `objet` est un objet résultat d'une expression. En fait, la variable reçoit une valeur qui est l'identité de

l'objet résultat. Cette identité est l'adresse de l'objet dans le système (adressage unique et global). L'accès à l'objet se fait donc indirectement par son adresse. Cette indirection a un effet sur la copie et la comparaison d'objets. Ces 'protocoles' sont définis dans la classe `Object`.

La copie d'objet est soit uniquement sur la valeur des variables d'instances (copie de la structure et valuation des variables d'instance par les identités) soit une copie complète et en profondeur des valeurs de l'objet (récursif). La première est appelée `shallow copy`, elle est définie par la méthode `clone` dans la classe `Object`.

```
protected native Object clone() throws CloneNotSupportedException;
```

La seconde est appelée `deep copy` et est en principe à la charge du programmeur. Plus précisément, la copie profonde est en principe assurée pour les objets dont la classe implante l'interface `Cloneable`, ce qui n'est pas le cas de la classe `Object`, qui génère une exception. Le problème se pose en particulier pour les chaînes de caractères. Pour éviter des soucis, on peut aussi définir (et redéfinir) une méthode `copy` ou `deepCopy` pour éviter toute ambiguïté. Une autre solution consiste à définir un constructeur de copie qui prend un paramètre un objet de la même classe.

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

De même l'égalité entre deux variables désignant des objets porte sur l'identité (`==`) ou une notion redéfinissable de l'égalité (`equals`). Cette redéfinition de `equals` permet de prendre en compte la copie superficielle ou la copie en profondeur.

NB : cette méthode s'applique aux chaînes de caractères sous peine d'erreur de test/

Prenons l'exemple d'un ensemble de points, déclarés comme suit.

Listing 2.3 – Code de la classe Copies

```
package copies;
import java.util.HashSet;
import points.PointPA;

public class Copies {
   /* test de copies d'objets */
    public static void main(String args[]) {
       PointPA p1 = new PointPA(5.0, 6.0);
       PointPA p2 = new PointPA(7.0, 8.0);
       PointPA p3 = p1;
       HashSet s1 = new HashSet();
       HashSet s2 = new HashSet();
       HashSet s3 = new HashSet();
       HashSet s4 = new HashSet();
       s1.add(p1); s1.add(p2); s1.add(p3);
       s2 = (HashSet) s1.clone();
       s3 = (HashSet) s2.clone();
       s4.add(s1);
       System.out.println("Résultat : "+s4.toString()+ " →" + s4.size());
       s4.add(s2);
       System.out.println("Résultat : "+s4.toString()+ " →" + s4.size());
       s4.add(s3);
       System.out.println("Résultat : "+s4.toString()+ " →" + s4.size());
    }
}
```

Ce code produit le résultat suivant :

```
Résultat : [[PointPAClone@45a877 (x = 7.0 ; y = 8.0),
   PointPAClone@360be0 (x = 5.0 ; y = 6.0)]] -> 1
Résultat : [[PointPAClone@45a877 (x = 7.0 ; y = 8.0),
   PointPAClone@360be0 (x = 5.0 ; y = 6.0)]] -> 1
Résultat : [[PointPAClone@45a877 (x = 7.0 ; y = 8.0),
   PointPAClone@360be0 (x = 5.0 ; y = 6.0)]] -> 1
```

L'ajout de l'occurrence `p3` dans `s1` n'a pas eu lieu car `p1` y était déjà et que les ensembles ont des éléments en un seul exemplaire. On constate aussi qu'il n'y a qu'un seul objet dans l'ensemble `s4`, cela vient du fait que le test d'ajout dans l'ensemble porte, non pas sur l'identifiant mais sur les valeurs. Ceci est du à l'implantation des ensembles par des dictionnaires en Java (`HashMap`). La méthode `clone` est redéfinie comme suit dans la classe `HashSet`.

```
    public Object clone() {
   try {
       HashSet<E> newSet = (HashSet<E>) super.clone();
       newSet.map = (HashMap<E, Object>) map.clone();
       return newSet;
   } catch (CloneNotSupportedException e) {
       throw new InternalError();
   }
    }
```

Noter que `<E>` correspond à une définition générique (voir section 9). Noter aussi que le type ensemble est générique et peut contenir toutes sortes d'objets. Néanmoins, seules les opérations de la classe `Object` sont applicables aux éléments de l'ensemble. Dans le cas contraire, il faut *transtyper* l'éléments, comme on peut l'observer dans l'exemple ci-dessous.

Essayons maintenant de définir plus précisément les différences entre copies en définissant uen interface `Copiable` et des sous-classes pour les points et les ensembles (bien que l'héritage n'ait pas encore été abordé).

Listing 2.4 – Code de l'interface Copiable

```
package copies;

public interface Copiable extends Cloneable {
   public Copiable copy ();
   public Copiable shallowCopy ();
   public Copiable deepCopy ();
}
```

Cette interface permet de définir les trois méthodes de copies. Pour bénéficier du polymorphisme et permettre la récurrence en profondeur, nous devons redéfinir les ensembles et les points pour les rendre 'copiables'.

Listing 2.5 – Code de la classe CopiablePointPA

```
package copies;

import points.PointPA;

public class CopiablePointPA extends PointPA implements Cloneable, Copiable {

   public CopiablePointPA() {
      // TODO Auto-generated constructor stub
      super();
   }

   public CopiablePointPA(double x, double y) {
```

```
        super(x, y);
        // TODO Auto-generated constructor stub
    }

    public CopiablePointPA(CopiablePointPA p) {
        // Constructeur de copie
        super();
        this.setXY(p.getX(), p.getY());
    }

    protected Object clone() throws CloneNotSupportedException {
        // redéfinition de la méthode
        CopiablePointPA copie = (CopiablePointPA)super.clone();
         copie.setXY(this.getX(), this.getY());
         return copie;
    }

    public Copiable deepCopy() {
        // redéfinition de la méthode et arret de la récursion
        return this.shallowCopy();
    }

    public Copiable shallowCopy() {
        // redéfinition de la méthode
        CopiablePointPA copie = new CopiablePointPA();
        copie.setXY(this.getX(), this.getY());
         return copie;
    }

    public Copiable copy() {
        // redéfinition de la méthode
        return this.deepCopy();
    }

    public String toString() {
        String res = Integer.toHexString(hashCode())+" (";
        res = res + this.getX()+ "@" + this.getY() + ")";
        return res;
    }

}
```

Listing 2.6 – Code de la classe CopiableSet

```
package copies;
import java.util.HashSet;
import java.util.Iterator;

public class CopiableSet extends HashSet implements Copiable {
    private String name; // pour l'affichage

    public CopiableSet() {
        super();
        // TODO Auto-generated constructor stub
    }

    public CopiableSet(String n) {
        // constructeur avec nom
```

```
        super();
        name = n;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Copiable deepCopy() {
        // redéfinition de la méthode
        CopiableSet copie = new CopiableSet();
        for (Iterator it = this.iterator(); it.hasNext();)
        {
            Copiable obj = (Copiable) it.next();
            copie.add(obj.deepCopy());
        }
         return copie;
    }

    public Copiable shallowCopy() {
        // redéfinition de la méthode
        CopiableSet copie = new CopiableSet();
        copie.addAll(this);
        return copie;
    }

    public Copiable copy() {
        // redéfinition de la méthode
        return this.deepCopy();
    }

    public String toString() {
        // redéfinition de la méthode ((Object)this).toString()+
        return (name+super.toString());
    }
}
```

Enfin la classe `Copies` reprend le code de la classe `CopiesClones` en l'adaptant.

Listing 2.7 – Code de la classe CopiesCopiables

```
package copies;
import java.util.HashSet;
import java.util.Vector;

public class CopiesCopiables {
    /* test de copies d'objets */

     public static void main(String args[]) {
         CopiablePointPA p1 = new CopiablePointPA(5.0, 6.0);
         CopiablePointPA p2 = new CopiablePointPA(7.0, 8.0);
         CopiablePointPA p3 = p1;
         CopiableSet s1 = new CopiableSet("s1");
         CopiableSet s2, s3 ;
         HashSet s5 = new HashSet(); // ancien s4
         Vector s4 = new Vector(); // pour mieux distinguer
         s1.add(p1); s1.add(p2); s1.add(p3);
         s2 = (CopiableSet) s1.shallowCopy(); //shallow copy
         s2.setName("s2");
```

```
        s3 = (CopiableSet) s2.deepCopy(); //deep copy
        s3.setName("s3");
        s4.add(s1); s4.add(s2); s4.add(s3);
        s5.add(s1); s5.add(s2); s5.add(s3);
        System.out.println("Résultat : "+s4.toString()+ " →" + s4.size());
        System.out.println("Résultat : "+s5.toString()+ " →" + s5.size());
    }
}
```

Ce code produit le résultat suivant :

```
Résultat : [s1[ad3ba4 (5.0@6.0), 126b249 (7.0@8.0)],
            s2[ad3ba4 (5.0@6.0), 126b249 (7.0@8.0)],
            s3[5224ee (7.0@8.0), 1c78e57 (5.0@6.0)]] -> 3
Résultat : [s3[5224ee (7.0@8.0), 1c78e57 (5.0@6.0)],
            s1[ad3ba4 (5.0@6.0), 126b249 (7.0@8.0)]] -> 2
```

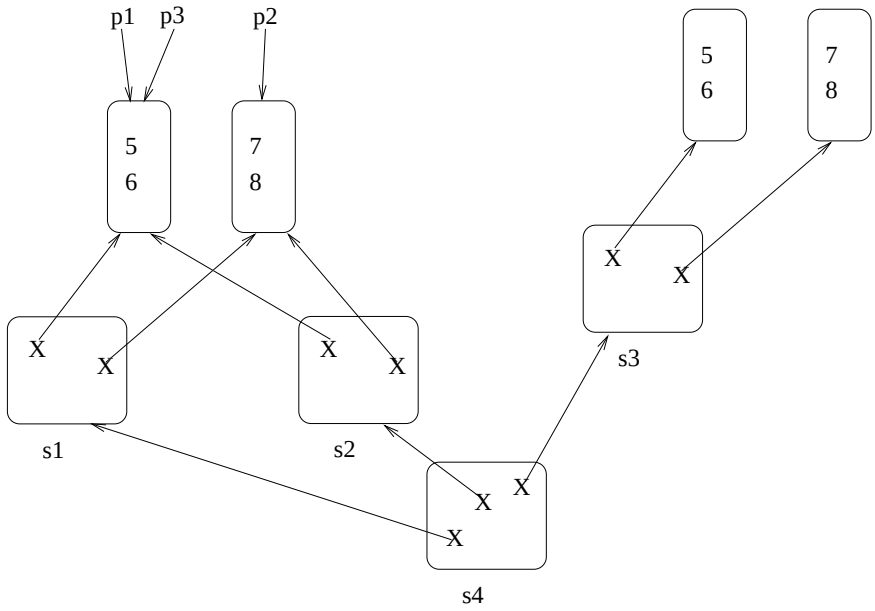



Figure 2 : *Un exemple de copie en Java*

Examinons dans la figure 2 l'effet des différentes copies et affectations dans `s4`. nLes numéros d'objets permettent de bien distinguer les copies et les originaux.

Pour des compléments sur la copie, consulter ([Cla03], p. 129) et
http://java.sun.com/developer/JDCTechTips/2001/tt0410.html
http://www.infres.enst.fr/ charon/coursJava/avance/dupliquer.html

## 6 Héritage

L'innovation la plus importante du modèle objet est l'héritage. Voici une définition extraite de [Mey97].

**Définition 6.1 (héritage)** *L'**héritage** est un mécanisme permettant de définir une nouvelle classe (la sous-classe) à partir d'une classe existante (la super-classe) par extension ou restriction.*

L'extension se fait en rajoutant des méthodes dans le comportement ou des variable d'instances dans la structure. Par exemple, un employé est une personne qui a un contrat de travail. La restriction consiste à réduire l'espace des valeurs définies par la classe en posant

des contraintes (conditions logiques à vérifier) sur ces valeurs. Par exemple, un carré est un rectangle dont les quatre côtés sont égaux.

L'héritage induit le **polymorphisme** (voir aussi section 6.6) : une instance de la sous-classe est aussi une instance de la super-classe. Cette instance peut donc utiliser (sans les définir) les méthodes de la super-classe. Inversement, on peut donner comme paramètre effectif un objet instance d'une sous-classe du paramètre formel.

Dans la pratique, l'héritage peut être utilisé à tort et à travers (il y a parfois même confusion entre héritage et utilisation). On peut **redéfinir** les méthodes et/ou changer leur profils. On peut **surcharger** les méthodes par un profil différent (des types différents). On doit donc donner des règles précises de contrôle de l'héritage pour conserver un contrôle de type sûr. Des règles ont été données dans [Car88, Mey97, ACR94].

Java autorise uniquement l'**héritage simple** : chaque classe hérite d'au plus une super-classe par la clause `extends`. La structure est héritée. le comportement est redéfinissable.

L'unique **source** du graphe d'héritage est la classe `Object` : toutes les classes héritent de cette classe. Elle même hérite de l'objet indéfini `nil`.

### 6.1 Redéfinition de propriétés

Les variables sont redéfinissables par **masquage**. La portée de la variable redéfinie est localisée à la sous-classe. Les types de la variable doivent être compatibles. Mais nous ne conseillons d'utiliser avec précaution cette redéfinition car il risque d'y avoir confusion lors de l'usage combiné de méthodes de la super-classe et de la sous-classe. Tout dépend de votre écriture de code. Par exemple, redéfinir la coordonnée `x` d'un point par une "String" génère des erreurs.

Une méthode peut porter le même nom (son sélecteur) et avoir les mêmes paramètres, on dit qu'elle la redéfinit. En Java, une méthode est redéfinie avec le même profil exactement (même types en paramères et en résultat). C'est l'application de la règle de **novariance** (ou invariante), comme en C++. La règle de covariance autorise un sous-type pour les paramètres et/ou le résultat. **contravariance**. La règle de contravariance autorise un super-type pour les paramètres et/ou le résultat. On peut appliquer des règles mixtes. Pour simuler la redéfinition covariante on utilise la surcharge statique (redéfinis pour de nouveaux types de données) et la coercition de types.
Attention, il y a des risques d'erreurs de compilation ou d'interprétation par un autre programmeur. Exemple :

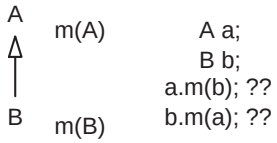



Figure 3 : *Redéfinition de méthode en Java*

Les langages typés dynamiquement comme Smalltalk ou CLOS règlent le problème à l'exécution (et si c'est bien programmé, il n'y a pas de problème... car le typage statique est souvent contraignant.). En redéfinissant une méthode, il est possible d'étendre sa zone de visibilité mais non de la restreindre.
La surcharge de méthode (plus précisément de sélecteur) est pratique courante pour les constructeurs, les exemples précédents le montrent bien.

La redéfinition doit aussi être cohérente avec les règles de visibilité des propriétés redéfinies (voir section 7.3).

### 6.2 Héritage, interfaces et redéfinitions

Une classe Java hérite d'une seule classe mais peut implanter différentes interfaces. Lorsqu'il y a conflit dans les définitions, une redéfinition s'impose pour résoudre les conflits.

### 6.3 Recherche de méthode

Lors d'un envoi de message `expr sél args`, le mécanisme suivant est exécuté.

1. le receveur est calculé par évaluation de l'expression `expr`
2. la méthode de sélecteur `sél` est recherchée à partir de la classe du receveur (classe courante) en tenant compte de son profil (c'est la signature avec le sélecteur et les types uniquement).
   (a) si une méthode ayant des types compatibles est trouvée, elle est appliquée avec les arguments `arguments`
   (b) sinon la méthode de sélecteur `sél` est recherchée dans la super-classe de la classe courante et ainsi de suite jusqu'à la trouver ou arriver dans la classe `Object`.
   (c) Le typage statique nous assure qu'un programme compilé devra trouver cette méthode, sinon il y aurait eu une erreur de type. Dans un langage typé dynamiquement comme Smalltalk, si la recherche échoue dans la classe `Object`, alors le message `doesNotUnderstand:` est envoyé avec comme paramètre le sélecteur `sél`. Nous rejoignons là le problème du **traitement des exceptions**.

L'utilisation de récursion par la pseudo variable `this` apporte ici toute sa puissance : la méthode la plus spécifique sera celle utilisée à l'exécution.

### 6.4 Super méthode

La super méthode est une simplification du concept de méthode qualifiée en CLOS [BDG+88]. Elle permet de combiner les différentes définitions d'une méthode par la pseudo-variable `super`. Cette caractéristique est liée à la recherche de méthode : la recherche de la méthode est double. La première recherche fournit la méthode principale qui se trouve dans une classe `Classe` qui est soit la classe du receveur soit une super-classe de la classe du receveur. Avec la pseudo-variable `super`, l'interpréteur commence alors une seconde recherche de la méthode à appliquer. L'évaluation des paramètres se fait sur le résultat de cette seconde recherche. Par exemple supposons la hiérarchie de classes et d'instances de la figure 4.

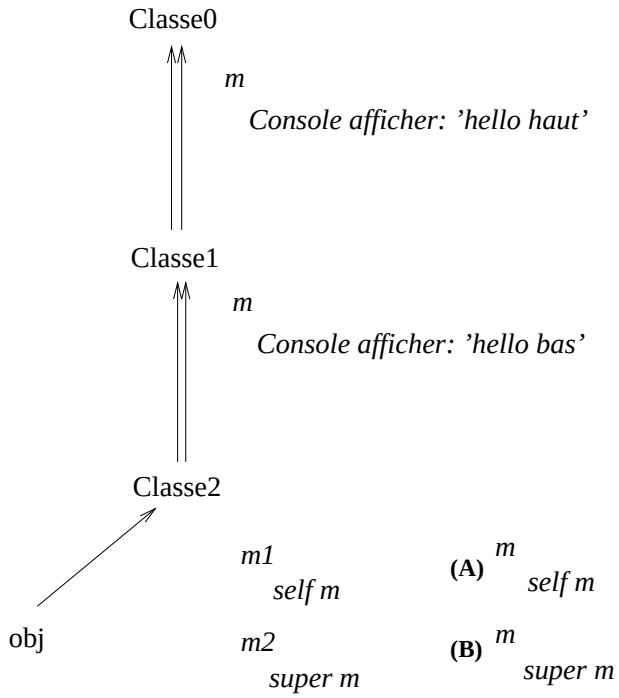



Figure 4 : *La super-méthode en Java*

La flèche simple représente la relation d'instanciation. La flèche simple représente la relation d'héritage. L'évaluation de l'envoi de message `obj m1` affiche `hello bas` sur la console. L'évaluation de l'envoi de message `obj m2` affiche `hello haut` sur la console. Dans l'alternative **(A)**, l'envoi de messages `obj m` boucle. Dans l'alternative **(B)**, l'envoi de messages`obj m` affiche `hello bas` sur la console.
A noter, on utilise ces variables `this` et `super` avec des arguments dans les constructeurs pour invoquer d'autres constructeurs. Dans ce cas, ils se trouvent en première instruction du constructeur.

Reprenons le code de la méthode `toString` de la classe `PointPA` du *listing 2.2* de la page 25.

```
public String toString() {
    String res = super.toString()+" (";
    res = res + "x = " + this.getX()+ " ; y = " + y + ")";
    return res;
}
```

Cette méthode invoque la méthode `toString` définie dans une superclasse, ici la classe `Object` pour afficher les informations par défaut d'un objet, notamment son indentifiant. L'expression `this.getX()` est équivalente dans cette classe à `x` mais si on la redéfinit dans une sous-classe, alors l'envoi d'un message `sousPt.toString()` à une instance `sousPt` de cette sous-classe ne donnerait pas le même résultat. C'est toute la puissance du polymorphisme.

### 6.5 Instanciation et classes abstraites

La méthode d'instanciation par défaut est le constructeur par défaut. Il est généré implicitement pour chaque classe lorsqu'aucun constructeur n'a été défini pour cette classe. Habituellement cette méthode prend en charge l'initialisation des variables d'instance.

Les classes abstraites sont utiles à la structuration du graphe d'héritage. Elles permettent de factoriser des comportements communs de haut-niveau. Par exemple, la classe `Magnitude` en Smalltalk ou `Comparable` en Eiffel rassemble tous les objets munis des opérations de comparaison (`=`,`>`, `>=`, etc.).

**Définition 6.2 (classe abstraite)**
*Une **classe abstraite** est une classe non feuille du graphe d'héritage et qui n'est jamais instanciée dans le système.*

Par exemple, considérons des étudiants et des enseignants. La classe des personnes est une classe abstraite, en effet un individu particulier sera instanciée à partir d'une des sous-classes spécifiques. La classe `Personne` définit le comportement commun à toutes ses sous-classes. Classe abstraite et métaclasse sont parfois confondus, à tort : une classe abstraite peut être une métaclasse ou n'être qu'une classe ordinaire et vice-versa. Nous pouvons dire qu'une classe abstraite est un nœud interne du graphe d'héritage et une métaclasse un nœud interne du graphe d'instanciation.

En java, une classe abstraite est annotée par l'attribut `abstract`. Les **interfaces** jouent aussi ce rôle en Java, notamment pour l'implantation de l'héritage multiple (voir section 3.3).

## 6.6 Typage, Polymorphisme et Coercition

Habituellement, dans un langage à classes, toute classe représente un type (pour le système de types). L'inverse est vrai dans un langage comme Smalltalk, mais pas en Java ou C++, qui proposent des types qui ne sont pas des classes.

La coercition ou transtypage est le changement de type d'une expression. Elle permet d'affiner le type statique d'une expression par rapport à son contexte d'utilisation. Par exemple, on transforme un entier en réel ou l'inverse. Il peut y avoir perte, par exemple le passage des réels aux entiers « oublie » la partie décimale. Certaines conversions de type sont implicites (comme l'*autoboxing* de la page 17) d'autres doivent être explicitées, ce qui se note `(NouveauType) expr`. Prenons quelques exemples :

```
double d = 12.08;
float  f  = 35.0;
long l  = 20;
byte b = 8;

// transtypages implicites

d = f;
f  = l;
l  = b;
d = (f*l)  + d;

// transtypages explicites

b = (byte) f;
l  = (long) d;
l  = (long) f  + d;
```

Cette opération, appelée en Eiffel, l'affectation polymorphique inverse, va de pair avec le polymorphisme et la surcharge. Par exemple, vous pouvez passer en paramètre ou affecter à un objet, un objet d'une sous-classe du type demandé. Cela ne pose aucun problème, compte-tenu des règles de redéfinition des propriétés.

Par exemple, dans le *listing 2.7*, les méthodes `shallowCopy`, `shallowCopy` rendent, selon leur profil une instance de la classe `Copiable`, mais l'examen du code de ces méthodes dans la classe `CopiableSet` montre qu'il s'agit en fait (dynamiquement) d'une instance de la classe `CopiableSet`. Le code `s2 = (CopiableSet) s1.shallowCopy();` permet de respecter le type de `s2` et donc de lui appliquer des méthodes de la classe `CopiableSet`.

### Polymorphisme

Revenons sur la notion de polymorphisme.

**Définition 6.3 (Polymorphisme)** *Une variable est **polymorphe** si elle a plusieurs types. Une méthode est **polymorphe** si elle est applicable à des paramètres (y compris le receveir) de plusieurs types.*

On distingue en fait deux cas de polymorphisme : le vrai et le faux.
- Le vrai polymorphisme est celui qui est induit par l'héritage (voir section 6) et la généricité (voir section 9). Par exemple, une variable `v:C` définie avec une classe `C` est aussi d'un type plus grand que `C` : les super-classes de `C` et les interfaces qu'implante `C`. Dans le cas de l'héritage et de la généricité, les codes sont applicables **sans transformation** aux instances des sous-classes.
- Le faux polymorphisme (ou polymorphisme *ad hoc* est celui de la surcharge et de la conversion (coercition, transtypage). Une variable d'un type primitif n'est pas polymorphe, pour lui chnger de type il faut la convertir (implicitement ou explicitement). La surcharge est une définition d'opérateurs ou de méthodes avec des paramètres différents (et un code différent).

## 7 Autres

### 7.1 Variables et méthodes de classe

Dans les langages de classes on utilise parfois la terminologie méthode et variable d'instance ou méthode et variable de classe. Une méthode de classe, par opposition à une méthode d'instance est une méthode prévue pour s'appliquer à une classe et non pas à une instance. C'est le cas par exemple des méthodes d'instanciation (constructeurs). Les variables de classes sont utilisées pour représenter des informations commune à l'ensemble des instances d'une classe. Si l'information n'est visible qu'en lecture alors une méthode d'instance est une solution suffisante. Une **variable de classe** est une variable qui est partagée (lecture et/ou écriture) par toutes les instances de cette classe. Il est possible de la voir comme appartenant à la classe et non plus à ses instances. La présence de méta-objets facilite la description de variables et de méthodes de classes. En l'absence de méta-objets, les variables et méthodes de classe ne sont pas accessibles par envoi de message habituel : il s'agit d'opérations primitives. Nous approfondissons ces concepts dans la section 8 et la section 4.3 du chapitre 5.

En Java, les propriétés de classes (variables, méthodes) sont annotées par l'attribut (modificateur) `static`. Une propriété est statique signifie qu'elle n'existe qu'en un seul exemplaire.

Le nombre de côtés d'un quadrilatère (initialisé à 4) est un exemple de variable statique. Les méthodes `main()` sont statiques.

### 7.2 Attributs qualifiants des éléments du langage

La sémantique des éléments du langage (classe, propriétés, etc.) varie en fonction d'annotations, que nous appelons **attributs** ou **modificateur**. Certains attributs on déjà été présentés, nous en proposons une liste et une synthèse ici.
- `static` : ce mot-clé, associé à une propriété (variable, méthodes) d'une classe, signifie que la propriété est une propriété de classe et non d'instance.
- `final` : ce mot-clé, associé à une classe ou une propriété d'une classe, signifie que l'élément ne doit pas évoluer : classe non héritable, variable non modifiable (= constante), méthode non redéfinissable.
- `public/protected/private` : cet attribut précise la visibilité de l'élément auquel il est associé (voir ci-après).

- `transcient` attribut non inclus en sérialisation.
- `volatile` modifié simultanément par différents *threads*.
- `synchonized` méthode en exclusion pour les threads.

## 7.3 Accessibilité et visibilité des propriétés

Les règles d'utilisation des propriétés (variables et méthodes) s'inspirent de celles de C++ et sont relativement complexes à manipuler. Elles dépendent de la porté de la propriété (l'endroit où elle est déclarée) et des attributs qui lui sont associés (visibilité, niveau, compilation).

### Portée

La portée est le bloc de déclaration : local à une méthode, lié à une classe, un paquetage. Le paquetage par défaut est le répertoire courant. Sans restriction, chaque variable est accessible dans la portée sous-entendue par son espace de nom : une variable locale est visible dans le bloc qui la déclare, une variable d'instance est visible par une instance, une variable de classe est visible par la classe, ses sous-classes et ses instances.

### Visibilité

La **visibilité** correspond aux droits d'utilisation d'un élément du langage. On parle aussi de modificateur d'accès.

La visibilité est liée à la portée. Une classe publique du paquetage `java.lang` est utilisable partout. Une classe publique d'un paquetage `p` est utilisable si la classe ou le paquetage est importé. Une classe définie sans visibilité est utilisable à l'intérieur de son paquetage.

Prenons maintenant le cas des propriétés définies dans une classe `C`.

- `public` : une propriété publique est accessible partout où la classe `C` est accessible. Quand une variable d'instance ou une méthode d'instance est publique, elle est applicable à toute instance de la classe en question.
- `package` (par défaut) : un élément paquetage est visible et utilisable dans son paquetage.
- `protected` : une propriété privée est accessible dans la classe `C` ou ses sous-classes, ou une classe du même paquetage.
- `private` une propriété privée est accessible uniquement dans la classe `C`. Quand une variable d'instance est privée, elle n'est accessible que par une méthode de la classe de définition.

Le respect du principe d'encapsulation limite la visibilité des variables d'instance à privé (ou plus rarement protégé).

## 7.4 Rangement

Les méthodes sont classées (par convention) en constructeurs, accesseurs, modificateurs, destructeurs (voir section 7.4).

On peut aussi s'inspirer d'autres classements tels ceux de Smalltalk.

- `initialize-release` : contient les méthodes d'initialisation et suppression des objets (les constructeurs et la méthode `finalize`).
- `accessing` : contient les méthodes d'accès en lecture ou écriture pour les variables d'instance.
- `comparing` : contient les méthodes de comparaison avec d'autres objets.
- `testing` : contient les méthodes de tests divers (appartenance...).
- `copying` : contient les méthodes de copies (redéfinies).
- `converting` : contient les méthodes de conversion ou de changement de classe pour un objet (`as<Classe>`)

- `displaying` : contient les méthodes d'affichage (classes vues).
- `adding-removing` : contient les méthodes d'ajout suppression d'éléments (classes collections).
- `...` : protocoles propres à chaque classe
- `printing` : contient les méthodes de représentation sous forme de chaînes de caractères (description).
- `private` : contient les méthodes non utilisables de l'extérieur (c'est un artifice car la notion de méthode privée n'existe pas en Java).

Les méthodes de classess sont aussi rangées dans des protocoles en Smalltalk. On retrouve habituellement les protocoles suivants :

- `instance creation` : contient les méthodes d'instanciation
- `class initialization` : contient les méthodes d'initialisation des variables de classe ou d'instance de la méta-classe.
- `defaults` : contient les méthodes implantant des valeurs par défaut (soit des variables de classe, ou d'instance de la méta-classe ou soit des calculs externes cachés).
- `Signal constants` : contient les méthodes de gestion de signaux d'exceptions.
- `general inquiries` : contient les méthodes générales de la classe.
- `backward compatibility` : contient les méthodes de compatibilité avec des versions antérieurs.
- `examples` : contient les méthodes d'exemples d'utilisation de la classe.
- `private` : contient les méthodes de classe non utilisables.

### 7.5 Classes internes

Les `classes internes` (ou `inner classes internes`) sont des classes définies dans des classes. Elles sont régies par des règles similaires à celles des variables et leur portée est donc celle de la classe qui les contient. L'intérêt d'une classe interne est de pouvoir définir des structures locales ou des contextes locaux qui ne sont pas des classes à part entière. Ainsi, nous avons utilisé des classes internes dans un traducteur pour stocker les éléments à différents stades de la traduction.

Une classe interne peut être déclarée avec n'importe lequel des attribut de visibilité (public, protected, par défaut ou private) et un autre attribut (abstract, final ou static). Elle est accessible par un nommage hiérarchique contenant le nom de la classe parente (la classe qui la définit). On en distingue quatre sortes : les classes internes membres (statiques ou non), les classes internes locales et les classes internes anonymes.

Pour illustrer le propos, nous prenons les explications de
http://www.laltruiste.com/coursjava/classe_insimple.html

Les classes possèdant le modificateur static deviennent des classes internes statiques. Une classe interne déclarée avec le modificateur d'accès private implique que cette classe ne pourra être utilisée que dans sa classe parente. Les classes internes ne peuvent pas être déclarées à l'intérieur d'initialisateurs statiques ou de membres d'interface. Les classes internes ne doivent pas déclarer de membres statiques, hormis s'ils comportent le modificateur final, dans le cas contraire, une erreur de compilation se produit.

```
class Classeinterne {
  static final x = 10; // constante statique
  static int y = 12; // erreur de compilation
  ...
}
```

Toutefois, les membres statiques de la classe externe peuvent être hérités sans problème par la classe interne. Une classe interne statique est appelée ń classe principale imbriquée ż. Une

telle classe est uniquement accessible depuis sa classe englobante et permet la construction dŠun ensemble de classes cohérentes. Les classes imbriquées sont capables d'accéder à toutes les variables et méthodes de la classe parente, y compris celles déclarées avec un modificateur private.

```
class ClasseParente {
  int x = 10;
  int y = 12;
  private int addition(){
    return (x + y);
  }
  class ClasseInterne {
    void verification (){
      if ((x + y)== addition())
        System.out.println("La classe interne a bien accédé "
                           + "aux membres de sa classe parente.");
    }
  }
  public static void main(String[] args){
    ClassParente obj_out = new ClasseParente();
    ClasseInterne obj_in = obj_out.new ClasseInterne();
    obj_in. verification ();
  }
}
```

Cette notation particulière spécifie que l'objet créé est une instance de la classe interne associée à l'objet résultant de l'instanciation d'une classe de plus haut niveau.

L'instanciation de la classe interne passe obligatoirement par une instance préalable de la classe d'inclusion. La classe parente est d'abord instanciée, puis c'est au tour de la classe interne de l'être par l'intermédiaire de l'objet résultant de la première instance.

```
ClassParente obj_out = new ClasseParente();
ClasseInterne obj_in = obj_out.new ClasseInterne();
// est équivalent à
ClasseInterne obj_in = (new ClassParente()).new ClasseInterne();
```

Il est possible d'utiliser une méthode de la classe parente pour créer directement une instance de la classe interne. Toutefois, lors de l'appel de la méthode, il sera nécessaire de créer une instance de la classe d'inclusion.

```
class ClasseParente {
  int x = 10;
  int y = 12;
  private int addition(){
    return (x + y);
  }
  class ClasseInterne {
    public void verification (){
      if ((x+y)== addition())
        System.out.println("La classe interne a bien accédé "
                           + "aux membres de sa classe parente.");
    }
  }
  void rendDisponible(){
    // Création d'une instance de la classe interne
    ClasseInterne obj_in = new ClasseInterne();
    // Appel d'une méthode de la classe interne
    obj_in. verification ();
```

```
  }
  public static void main(String[] args){
    // Création préalable d'une instance de la classe parente
    ClasseParente obj_out = new ClasseParente();
    // Appel de la méthode instanciant la classe interne
    obj_out.rendDisponible();
  }
}
```

Cette manière de créer une instance de la classe interne est possible grâce à la référence d'objet this désignant une instance de la classe parente.

```
ClasseInterne obj_in = new ClasseInterne();
// est équivalent à
this.ClasseInterne obj_in = this.new ClasseInterne();
```

L'utilisation implicite du mot-clé this permet au constructeur de la classe parente de créer une instance de la classe interne. Le constructeur de cette dernière pourra alors exécuter ses propres instructions.

```
class ClasseParente {
  int x = 10;
  int y = 12;
  // constructeur externe
  ClasseParente(){
    // Création d'une instance de la classe interne
    new ClasseInterne();
  }
  public static void main(String[] args){
    // Création préalable d'une instance de la classe parente
    new ClasseParente();
  }
  private int addition(){
    return (x + y);
  }
  class ClasseInterne {
    // constructeur interne
    ClasseInterne(){
      if ((x + y)== addition())
        System.out.println("La classe interne a bien accédé "
                           + "aux membres de sa classe parente.");
    }
  }
}
```

Il est possible d'imbriquer des classes sur plusieurs niveaux. Une classe normale peut contenir une à plusieurs classes internes et ces dernières peuvent contenir également une à plusieurs autres classes intérieures.

```
class ClasseParente {
  int x = 10;
  int y = 12;

  public static void main(String[] args){
      ClasseInterne obj_in =
              (new ClasseParente()).new ClasseInterne();
      obj_in.verification ();
  }
  private int addition(){
```

```
    return (x + y);
  }
  class ClasseInterne {
    public void verification (){
      if ((x + y)== addition())
        System.out.println("La classe interne a bien accédé "
                           + "aux membres de sa classe parente.");
        ClasseInterneInterieure obj_inin =
                          new ClasseInterneInterieure();
        System.out.println(obj_inin.ajouter());
    }
    class ClasseInterneInterieure {
      int z = 10;
      int ajouter(){
        return (z + x);}
    }
  }
}
```

La classe doublement imbriquée ne peut être instanciée que dans sa classe parente directe. Les méthodes et les variables d'instance d'une classe interne ne deviennent disponibles dans la classe parente qu'après l'instanciation de cette classe imbriquée.

Parfois, il peut être nécessaire de distinguer les variables situées dans les classes interne et externe.

```
class Externe {
  int x = 10;
  int y = 12;
  Externe(){
    new Interne();
  }
  public static void main(String[] args){
    new Externe();
  }
  class Interne {
    int x = 8;
    int y = 14;
    Interne(){
      if (this.x + this.y == Externe.this.x + Externe.this.y)
        System.out.println("La classe interne a bien "
                           + "accédé à l'ensemble des membres "
                           + "des classes imbriquées.");
    }
  }
}
```

Le mot-clé this permet d'accéder à un membre de la classe en cours, c'est-pourquoi `this.variable` accède au membre de la classe interne et `Externe.this.variable` à celui de la classe parente spécifiée.

Il existe aussi des classes internes anonymes, c'est-à-dire des classes dont l'utilisation fait qu'un nom n'est pas nécessaire, par exemple quand une seule instance est nécessaire, quand le nom de la classe interne locale n'est pas utile, ou pour écrire un code concis.

Voir aussi [Eck02].

## 8 Définition d'une classe

La définition d'une classe comprend :

- un éventuel paquetage de définition et des importations de paquetages ou de classes,
- la super-classe et les interfaces implantées,
- le commentaire de la classe,
- les classes internes,
- les déclarations de variables d'instances,
- les déclarations de variables de classes,
- les méthodes d'instances
- les méthodes de classe

```
package ppp;
import java.util.*;
import pp.pp.classe;

<modificateur> class <NomDeClasse> {
   /* définitions des variables */
   <modificateur> <attributs> <Type> <NomDeVariable> [= <expr.initiale>]
   /* définitions des méthodes */
   <modificateur> <attributs> <TypeResultat> <NomDeMéthode> (param) {
            <code de la méthode>
   }
}
```

Des exemples figurent dans les listings 2.2, 2.3, 2.4, 2.5...

## 9 Généricité

La généricité permet de paramétrer des types par des types et ainsi de réutiliser du code sans le modifier. L'exemple classique de la généricité est la collection. On peut utiliser des collections de voitures, de livres... Les opérations sont les mêmes quels que soient les objets de la collection (ajouter, enlever, tester, itérer...).

Sans généricité, une collection reste générique mais contient des objets, l'utilisation précise des objets nécessite une conversion de type. Par exemple, dans le *listing 2.6*, pour indique que les éméments de la collection sont des instances de `Copiable` nous devons utiliser un opérateur de conversion `(Copiable) it.next()`.

```
   public Copiable deepCopy() {
      // redéfinition de la méthode
      CopiableSet copie = new CopiableSet();
      for (Iterator it = this.iterator(); it.hasNext();)
      {
         Copiable obj = (Copiable) it.next();
         copie.add(obj.deepCopy());
      }
       return copie;
   }
```

La généricité a été ajoutée dans Java 1.5. Elle est définie pour toutes les collections. On peut donc écrire une version simplifiée de l'ensemble de points, qui accepte des `PointPA` (*listing 2.2*) ou ds instances de sous-classes tels que des `CopiablePointPA` (*listing 2.5*), etc.

Listing 2.8 – Code de la classe SetPoints

```
package genericite;
import java.util.HashSet;
import copies.CopiablePointPA;
import points.PointPA;
```

```
public class SetPoints {

    public static void main(String args[]) {
        PointPA p0, p1;
          p0 = new PointPA();
          p1 = new PointPA(5.0, 0.0);
          //p2 = new PointPA(10.2, 5);
        CopiablePointPA p2 = new CopiablePointPA(10.2, 5);
          HashSet<PointPA> maCollection;
          maCollection= new HashSet<PointPA>();
          maCollection.add(p0);maCollection.add(p1);maCollection.add(p2);
          System.out.println("ma Collection : "+maCollection.toString());
      }

}
```

Ce code produit le résultat suivant :

```
ma Collection : [points.PointPA@ad3ba4 (x = 5.0 ; y = 0.0),
126b249 (10.2@5.0), points.PointPA@1372a1a (x = 0.0 ; y = 0.0)]
```

On notera que l'affichage invoque la méthode `toString()` de la classe `HashSet<T>` qui elle-même invoque la méthode `toString()` sur chacun des éléments de la collection. Cette méthode varie entre les instances de `PointPA` et celles de `CopiablePointPA`. Une fois de plus, on montre la puissance du polymorphisme.

Définissons maintenant un type générique `CopiableSetGen`, paramétré par l'interface `Copiable`, qui remplace le type `CopiableSet`. L'intérêt majeur est que maintenant les coecitions ont disparu du code.

Listing 2.9 – Code de la classe CopiableSetGen

```
package genericite;
import java.util.HashSet;
import java.util.Vector;

import copies.Copiable;
import copies.CopiablePointPA;
import copies.CopiableSet;

public class CopiableSetGen extends HashSet<Copiable> implements Copiable {
   private String name; // pour l'affichage

   public CopiableSetGen() {
      super();
      // TODO Auto-generated constructor stub
   }

   public CopiableSetGen(String n) {
      // constructeur avec nom
      super();
      name = n;
   }

   public void setName(String name) {
      this.name = name;
   }

   public Copiable deepCopy() {
      // redéfinition de la méthode
```

```
        CopiableSetGen copie = new CopiableSetGen();
        for (Copiable copiable : this) {
            copie.add(copiable.deepCopy());
        }
         return copie;
    }

    public Copiable shallowCopy() {
        // redéfinition de la méthode
        CopiableSetGen copie = new CopiableSetGen();
        copie.addAll(this);
        return copie;
    }

    public Copiable copy() {
        // redéfinition de la méthode
        return this.deepCopy();
    }

    public String toString() {
        // redéfinition de la méthode ((Object)this).toString()+
        return (name+super.toString());
    }

     public static void main(String args[]) {
         CopiablePointPA p1 = new CopiablePointPA(5.0, 6.0);
         CopiablePointPA p2 = new CopiablePointPA(7.0, 8.0);
         CopiablePointPA p3 = p1;
         CopiableSet s1 = new CopiableSet("s1");
         CopiableSet s2, s3 ;
         HashSet s5 = new HashSet(); // ancien s4
         Vector s4 = new Vector(); // pour mieux distinguer
         s1.add(p1); s1.add(p2); s1.add(p3);
         s2 = (CopiableSet) s1.shallowCopy(); //shallow copy
         s2.setName("s2");
         s3 = (CopiableSet) s2.deepCopy(); //deep copy
         s3.setName("s3");
         s4.add(s1); s4.add(s2); s4.add(s3);
         s5.add(s1); s5.add(s2); s5.add(s3);
         System.out.println("Résultat : "+s4.toString()+ " →" + s4.size());
         System.out.println("Résultat : "+s5.toString()+ " →" + s5.size());
     }
}
```

On notera l'utilisation de l'itérateur générique, qui n'est plus une instance de la classe `java.util.Iterator`.

```
    for (Type_iteration var_iteration : collection_gen) {
        <traitement sur var_iteration >
    }
```

Ce code produit le résultat suivant :

```
Résultat : [s1[ad3ba4 (5.0@6.0), 126b249 (7.0@8.0)],
            s2[ad3ba4 (5.0@6.0), 126b249 (7.0@8.0)],
            s3[5224ee (7.0@8.0), 1c78e57 (5.0@6.0)]] -> 3
Résultat : [s3[5224ee (7.0@8.0), 1c78e57 (5.0@6.0)],
            s1[ad3ba4 (5.0@6.0), 126b249 (7.0@8.0)]] -> 2
```

Noter qu'un type peut être paramétré par plusieurs autres, par exemple une association est un couple `Association<Key,Value>`. Voici un autre exemple, du à Gilles Ardourel.

```
public class StockBizarre<T> {
   T unAttributDeTypeT;
   T unAutreAttributDeTypeT;
   public StockBizarre<T>(T unparamdetypeT){
      this.unAttributDeTypeT=unparamdetypeT;
   }
   public void add(T t) {
      this.unAutreAttributDeTypeT=this.unAttributDeTypeT;
      this.unAttributDeTypeT=t;
   }
   public T get(){
      T varlocale=this.unAutreAttributDeTypeT;
      this.unAutreAttributDeTypeT=this.unAttributDeTypeT;
      this.unAttributDeTypeT=varlocale;
      return varlocale;
   }
}
```

D'autres langages permettent aussi de paramétrer les types par des constantes. Pour plus de détails sur la généricité (paramétrique) en Java, consulter
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
http://java.sun.com/developer/technicalArticles/J2SE/generics/

# Chapitre 3

# La programmation avec Java

## 1 Exemple introductif

Cette section est à la fois une introduction à la programmation à objets et à la programmation avec Java. Nous partons d'un algorithme classique de programmation structurée pour aboutir à une programme à objets. Nous passons par une version intermédiaire, qui est quasiment une traduction de programme Pascal en Java. Cette étape intermédiaire illustre la syntaxe du langage et la représentation des structures de contrôles de la programmation structurée et par la structure de contrôle spécifique qu'est l'envoi de message. Le lecteur trouvera dans d'autres ouvrages des exemples introductifs similiaires et instructifs : jeu de Nim [Roy94], tournoi de tennis [Ner90].

### Enoncé informel et programme structuré

Le but du programme est de calculer des niveaux pluviométriques annuels dans des villes à partir de relevés mensuels saisis au préalable. Les résultats sont affichés à l'écran. Ce programme est un exercice sur l'utilisation des procédures et des vecteurs donné en Deug B à Nantes. Voici le programme Pascal correspondant.

```
program pluvio;                                    (* Deug B Nantes *)
   const nblieux=20;
   type vect_char=array[1..nblieux] of string;
        vect_int=array[1..nblieux] of integer;
        vect_ps=array[1..12] of integer;
   var lieux:vect_char;
       mois:vect_int;
       hauteurs:vect_int;
       nbsaisis:integer;
       somme:vect_ps;

 procedure saisir_releves(var lieux:vect_char; var mois:vect_int;
                          var hauteurs:vect_int; var nbsaisis:integer);
   var l:string;
       h,m:integer;

   begin
     writeln('donnez votre fiche: lieu, mois, hauteur. ');
     readln(l,m,h);
     nbsaisis:=0;
     while (l<>'z') and (nbsaisis < nblieux) do begin
         nbsaisis:=nbsaisis+1;
         lieux[nbsaisis]:=l;
         while (m>12) or (m<1) do begin
```

```
            writeln(' E R R E U R  mois, recommencer ');
            readln(m)
        end;
        mois[nbsaisis]:=m;
        hauteurs[nbsaisis]:=h;
        writeln(' fiche suivante (z pour sortir) ');
        readln(l,m,h)
    end
  end;    (* saisir_releves *)

 procedure addition(mois:vect_int; haut:vect_int; nbrel:integer;
                    m:integer; var som:integer);
   var j:integer;

   begin
     som:=0;
     for j:=1 to nbrel do
       if mois[j]=m then som:=som+haut[j]
   end;    (* addition *)

 procedure exploiter_releves(mois:vect_int; haute:vect_int;
                             nbrel:integer; var somm:vect_ps);
   var i,sm:integer;

   begin
     for i:=1 to 12 do begin
         addition(mois,haute,nbrel,i,sm);
         somm[i]:=sm
     end
   end;    (* exploiter_releves*)

 procedure affiche_so(s:vect_ps);
   var i:integer;

   begin
     for i:=1 to 12 do
       writeln(' somme du mois ',i,' est ',s[i])
   end;    (* affiche_so *)

begin      (* Début du programme *)
  saisir_releves(lieux,mois,hauteurs,nbsaisis);
  exploiter_releves(mois,hauteurs,nbsaisis,somme);
  affiche_so(somme)
end.       (* Fin du programme *)
```

Figure 5 : *Programme Pascal de calcul pluviométrique*

### Evolution de l'énoncé

Les modifications suivantes sont envisagées par le demandeur :

1. Les relevés doivent pouvoir être lus dans des fichiers textes en plus de la possibilité de les saisir au clavier.
2. Le nombre maximal de relevés risque d'évoluer selon les années.
3. L'auteur des relevés sera ajouté plus tard pour établir des statistiques.
4. Actuellement la saisie des fiches est faite en une fois, mais elle pourra se faire en plusieurs fois, directement à partir des stations.

5. On souhaite appliquer les mêmes traitements pour les relevés de température pour calculer les moyennes annuelles. Deux cas sont envisagés : saisie des deux valeurs sur une même fiche ou saisie de deux fiches différentes.
6. On souhaite établir maintenant des statistiques sur deux ans.

### Traduction du programme en Java

Une seule classe, appelée `PluvioSimple`, est nécessaire pour implante ler programme principal. La figure 6 synthétise, via la notation UML, cette classe. Les attributs et opérations soulignées font partie des protocoles de classe. Les attributs sont des variables en Java (privées par défaut). Les opération sont des méthodes en Java (publiques par défaut). L'exécution du programme est simplement une instanciation de la classe.


Figure 6 : *Classe* `PluvioSimple`

Les types sont déclarés explicitement en Java (le typage est statique). Les constantes sont représentées

- par des variables d'environnement du système hôte. Cette solution est à proscrire car la constante subsisterait après la fin du programme.
- par des variables de classe (ou d'instance de la métaclasse). Cette solution convient car, pour toute exécution (une instance de la classe est une exécution du programme) la valeur de la constante ne varie pas.
- par des variables d'instances (en fait comme une variable globale du programme Pascal initialisée dans la procédure principale). Cette solution est moins pratique car il faut initialiser la variable (qui représente la constante globale) à chaque lancement de programme (chaque instanciation de la classe).

Les variables globales sont représentées par des variables d'instance : elle ont une nouvelle valeur à chaque exécution du programme (chaque création d'une instance de la classe). Les procédures sont représentées par des méthodes d'instance de la classe. Le programme principal est représenté par une méthode de classe `main` "à la C", qui est la méthode d'instanciation de la classe (le constructeur par défaut).

Le code du programme Java se trouve dans le fichier `PluvioSimple.java`. Un programme s'écrit par au moins une classe. La classe étant un objet à part entière, les données et le code sont répartis dans les différentes valeurs d'objet de la classe. La représentation suivante du code est donc une présentation personnalisée du code.

La traduction ne pose pas de difficultés majeures hormis le traitement des entrées/sorties. La sortie standard (`System.out.println(...)`) est simple et a déjà été utilisé dans les programmes précédents. La lecture est plus compliquée car l'entrée standard (`System.in.read(...)`) ne lit pas de chaînes de caractères. Nous y reviendrons dans la section 2. On doit passer par des tampons divers tels que `BufferedReader` ou `DataInputStream`. La méthode `readLine` n'est plus disponible (*deprecated*) dans la classe `DataInputStream`. Puis nous utilisons la fonction `split` des chaînes de caractères pour récupérer les trois valeurs dans un tableau de chaînes. Le paramètre est une expression régulière qui élimine les blancs et le séparateur ','. On utilise la fonction `parseInt` pour convertir une chaîne en un nombre entier.

Listing 3.1 – Traduction Java du programme de calcul pluviométrique

```
package pluviometrie;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * @author pascal andre
 * traduction du programme pluvio de Pascal en Java
 */

public class PluvioSimple {
    // les variables sont privées
    private static final int NBLIEUX = 20;
    private String lieux [];
    private int mois [];
    private int hauteur [];
    private int somme[];
    private int nbSaisies;

    public PluvioSimple() {
        // initialisation par défaut des variables d'instance
        super();
        lieux = new String[NBLIEUX];
        mois = new int[NBLIEUX];
        hauteur = new int[NBLIEUX];
        somme = new int[12];
        nbSaisies = 0;
    }

    public void saisir_releves () {
        /*
         * saisit les relevés au clavier et les stocke dans les variables
         * d'instance. On essaie d'être aussi proche que possible du code pascal.
         */
        String line []; // variables locales
        String l;
        int h, m;
        nbSaisies = 0;
        BufferedReader stdin = new BufferedReader(new InputStreamReader(
                System.in));
```

```
    // déclare le tampon de lecture (pour types primitifs)
    try {
        System.out.print("donner votre fiche: lieu, mois, hauteur : ");
        // lecture et extraction des trois valeurs
        line = stdin.readLine(). split ("( )*,( )*");
        // ou line = StringTokenizer(stdin.readLine())
        l = line[0];
        while (! l.equals("z") && (nbSaisies < NBLIEUX)) {
            m = Integer.parseInt(line[1]);
            h = Integer.parseInt(line[2]);
            while (m<0 || m>12) {
                System.out.print("Erreur de mois, redonnez-le :");
                line = stdin.readLine(). split ("( )*,( )*");
                m = Integer.parseInt(line[0]);
            }
            lieux[nbSaisies] = l;
            mois[nbSaisies] = m;
            hauteur[nbSaisies] = h;
            nbSaisies++;
            System.out.print("donner votre fiche : lieu, mois, hauteur : ");
            line = stdin.readLine(). split ("( )*,( )*");
            l = line[0];
        }
    } catch (IOException e) {
        System.err.println("IOException thrown " + e.toString());
        // return false;
    }
}

public int addition(int m) {
    /*
     * calcule les sommes par mois
     */
    int som = 0;
    for (int i=0;i< nbSaisies;i++) {
        if (mois[i] == m) som += hauteur[i];
    };
    return som;
}

public void exploiter_releves() {
    /*
     * calcule et stocke les sommes par mois
     */
    for (int i=0;i<12;i++) {
        somme[i]= this.addition(i+1);
    }
}

public void afficher_releves() {
    /*
     * affiche les relevés
     */
    System.out.println("Relevés :");
    for (int i=0;i<nbSaisies;i++) {
        System.out.print("Fiche ("+(i+1)+") lieu " + lieux[i]);
        System.out.print(" - mois " + mois[i]);
        System.out.println(" - hauteur " + hauteur[i]);
```

```
        }
    }

    public void afficher_so() {
        /*
         * affiche les résultats annuels
         */
        System.out.println("Totaux :");
        for (int i=0;i<12;i++)
            System.out.println("somme du mois "+(i+1)+" est " + somme[i]);
    }

    public static void main(String args[]) {
        /* programme principal */
        PluvioSimple pluvio = new PluvioSimple();
        pluvio.saisir_releves();
        pluvio.afficher_releves();
        pluvio.exploiter_releves();
        pluvio.afficher_so();
    }
}
```

Le programme principal est similaire à celui du programme Pascal, hormis le passage des paramètres qui est inutile car les informations sont stockées dans les variables d'instance. Noter que la saisie nécessite un traitement d'exception (voir la section 1 du chapitre 5).

Le premier commentaire est un commentaire JavaDoc (voir la section 3 du chapitre 5).

Nous n'avons créé de méthodes d'accès en lecture ou écriture aux variables. Ce qui pose au moins le problème d'une spécialisation de la classe.

Ce code produit le résultat suivant :

```
donner votre fiche: lieu, mois, hauteur : paris, 12, 10
donner votre fiche: lieu mois hauteur : nantes, 10, 234
donner votre fiche: lieu mois hauteur : paris, 12, 10
donner votre fiche: lieu mois hauteur : paris, 12, 10
donner votre fiche: lieu mois hauteur : nantes, 5, 123
donner votre fiche: lieu mois hauteur : nantes, 10, 6
donner votre fiche: lieu mois hauteur : z
Relevés :
Fiche (1) lieu paris - mois 12 - hauteur 10
Fiche (2) lieu nantes - mois 10 - hauteur 234
Fiche (3) lieu  paris - mois 12 - hauteur 10
Fiche (4) lieu paris - mois 12 - hauteur 10
Fiche (5) lieu nantes - mois 5 - hauteur 123
Fiche (6) lieu nantes - mois 10 - hauteur 6
Totaux :
somme du mois 1 est 0
somme du mois 2 est 0
somme du mois 3 est 0
somme du mois 4 est 0
somme du mois 5 est 123
somme du mois 6 est 0
somme du mois 7 est 0
somme du mois 8 est 0
somme du mois 9 est 0
somme du mois 10 est 240
somme du mois 11 est 0
somme du mois 12 est 30
```

### Synthèse de la traduction

Le programme précédent a mis en évidence la traduction des fonctions de bases d'un programme structuré.

- Les structures de contrôle sont définies par structures de contrôle Java.
- Les entrées/sorties utilisent les interfaces I/O de Java (console, souris, clavier).
- Les procédures sont assimilées à des méthodes.
- Les calculs sont réalisés par envois de messages ou des opérations primitives.

### Analyse du programme

Le programme de départ respectait certains critères de qualité d'un programme structuré, qui ont une influence sur son évolution (section 1).

- Utilisation de constantes pour mettre en évidence les paramètres du programme et rangement de ces constantes dans un seul endroit. La constante `NBLIEUX` permet de paramétrer les tailles de tableaux et les contrôles associés. Si on avait mis simplement la valeur 20 partout, alors la modification du programme pour passer à 30 est à opérer sur l'ensemble du programme.
  De ce fait, l'évolution numéro 2 se fait simplement en changeant valeur de la constante `NBLIEUX`.
- Le programme principal est concis et construit à partir d'appels de procédures. Chaque procédure correspond bien à une activité indépendante et cohérente : saisie, calcul et affichage. La lisibilité en est accrue.
  De ce fait, l'évolution numéro 1 se fait simplement en ajoutant une procédure `saisie_fichier`, qui lit les fiches dans un fichier. Dans le programme principal, on demande à l'utilisateur le type de saisie à effectuer. Voici sa traduction en Java.

La version fichier peut être une spécialisation de la version clavier. Pour accéder simplement aux variables d'instances, il faut changer les modificateurs de visibilité (d'accès) en `protected` pour toutes les variables (sauf somme). La classe après modification s'appelle `pluvioSimpleMod`. Il suffit alors de changer la méthode de saisie (on pourrait découper celle-ci en plusieurs méthodes pour restreindre la lecture à une sous-méthode redéfinie dans les sous-classes). La lecture dans un fichier consiste simplement à modifier la source du "BufferReader".

Listing 3.2 – Traduction Java du programme de calcul pluviométrique avec fichiers

```
package pluviometrie;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * @author pascal andre
 * traduction du programme pluvio de Pascal en Java
 * avec fichiers
 */

public class PluvioSimpleFic extends PluvioSimpleMod {

    public PluvioSimpleFic() {
        // initialisation par défaut des variables d'instance
        super();
    }
```

```
    public void saisir_releves(String nom) {
        /*
         * saisit les relevés dans un fichier dont on précise le nom (surcharge)
         * et les stocke dans les variables
         * d'instance. On essaie d'être aussi proche que possible du code pascal.
         */
        String line[]; // variables locales
        String l;
        int h, m;
        nbSaisies = 0;
        BufferedReader stdin = new BufferedReader(new InputStreamReader(
                System.in));
        // stdin est utile pour la correction du mois
        try {
            BufferedReader file = new BufferedReader(new FileReader(nom));
            // à placer dans le try car exception peut être levée
            // déclare le tampon de lecture (pour types primitifs)
            line = file.readLine().split("( )*,( )*");
            // ou line = StringTokenizer(stdin.readLine())
            l = line[0];
            while (!l.equals("z") && (nbSaisies < NBLIEUX)) {
                m = Integer.parseInt(line[1]);
                h = Integer.parseInt(line[2]);
                while (m<0 || m>12) {
                    System.out.print("Erreur de mois, redonnez-le :");
                    line = stdin.readLine().split("( )*,( )*");
                    m = Integer.parseInt(line[0]);
                }
                lieux[nbSaisies] = l;
                mois[nbSaisies] = m;
                hauteur[nbSaisies] = h;
                nbSaisies++;
                line = file.readLine().split("( )*,( )*");
                l = line[0];
            }
        } catch (IOException e) {
            System.err.println("IOException thrown " + e.toString());
            // return false;
        }
    }

    public static void main(String args[]) {
        /* programme principal */
        PluvioSimpleFic pluvio = new PluvioSimpleFic();
        pluvio.saisir_releves(System.getProperty("user.dir")+
                System.getProperty("file.separator")+"releves.txt");
        // on peut saisir le nom et ajouter des vérifications
        pluvio.afficher_releves();
        pluvio.exploiter_releves();
        pluvio.afficher_so();
    }
}
```

La lecteur d'information au clavier peut être facilitée en utilisant des fenêtres de dialogue (`JDialog`) du framework `Swing` que nous abordons dans le chapitre 7.

Ce code produit le résultat suivant :

```
Relevés :
Fiche (1) lieu paris - mois 12 - hauteur 10
Fiche (2) lieu nantes - mois 10 - hauteur 234
Fiche (3) lieu paris - mois 12 - hauteur 10
Fiche (4) lieu paris - mois 12 - hauteur 10
Fiche (5) lieu nantes - mois 5 - hauteur 123
Fiche (6) lieu nantes - mois 10 - hauteur 6
Totaux :
somme du mois 1 est 0
somme du mois 2 est 0
somme du mois 3 est 0
somme du mois 4 est 0
somme du mois 5 est 123
somme du mois 6 est 0
somme du mois 7 est 0
somme du mois 8 est 0
somme du mois 9 est 0
somme du mois 10 est 240
somme du mois 11 est 0
somme du mois 12 est 30
```

- Le passage des paramètres, quoique un peu lourd en ce sens que qu'on aurait pu travailler directement sur les variables globales, garantit une bonne utilisation des procédures et une réutilisation de ces procédures dans un contexte différent.
- L'évolution numéro 3 est plus délicate car elle implique des modifications dans les déclarations, dans les procédures et dans le programme principal. On ajoute un nouveau tableau pour les auteurs. La saisie est modifiée (un paramètre supplémentaire) et une procédure pourrait être ajoutée pour le traitement statistique dont on ignore le contenu. Le programme principal comprend deux traitements réutilisés : exploiter les relevés, calculer des statistiques. Leur coordination peut être séquentielle, alternative ou itérative.
- L'évolution numéro 4 induit une itération dans le programme principal sur la saisie des relevés (avec ou pas des exploitation intermédiaires). Il faut déplacer l'initialisation du nombre de saisies de la procédure de saisie vers le programme principal.
- L'évolution numéro 5b est simplement une recopie du programme en changeant des noms (hauteur devient température) et en changeant la procédure d'exploitation qui ne calcule plus une somme mais une moyenne (attention à la division par 0 s'il n'y a pas de relevés saisis).
- Les statistiques sur deux ans doublent les structures de données. On peut utiliser des matrices (année, lieu) et passer en paramètre des procédures l'année considérée. Cela ne pose pas de difficultés mais la modification porte sur tout le code.

Le programme initial pose aussi un certain nombre de problèmes.

- Le résultat est mono-bloc. Son organisation rend la lecture du code ralativement aisée mais c'est moins le cas lorsque le programme figure sur quelques centaines de lignes. La lisibilité du programme est un critère essentiel à son évolution.
- L'utilisation de tableaux bornés, qui est une contrainte dans le langage cible, ajoute des éléments non négligeables dans le code : constante, bornes de tableaux, vérifications de dépassement... En Java, on peut disposer d'une varité de collections d'objets qui vont nous abstraire de ces contingences. On utilisera des tableaux dynamiques (`Vector`) ou des ensembles (`HashSet`). L'évolution numéro 2 est implicitement résolue.
- L'évolution numéro 1 qui utilise les entrées/sorties sur fichiers (ou sur un autre support d'entrée-sortie) se fait par l'intermédiaire de flots de données.

Les autres évolutions sont facilitées lorsqu'on s'abstrait du code en mettant en évidence des concepts de l'application. En particulier, on distingue ici des fiches contenant des informations et un programme de calcul de résultats à partir de fiches. Nous étudions ce point de vue dans

la section suivante.

### 1.1 Conception du programme en Java

Dissocier la notion de fiche de l'exploitation des résultats facilite l'évolution de programme.

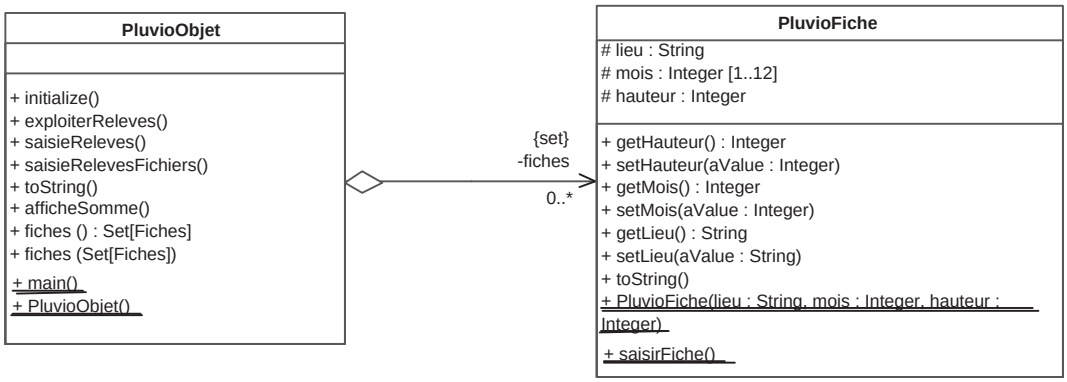



Figure 7 : *Conception objet du programme* `pluvio`

Le programme se présente schématiquement selon le diagramme de la figure 7. On remarque que l'interface des classes est largement simplifiée, ce qui facilite la lecture, le test, l'évolution et la réutilisation du code. L'agrégation est orientée, elle est implantée par une variable d'instance `fiches` contenant un ensemble d'objets `Fiche` dans la classe `PluvioObjet`.

- Une fiche regroupe l'ensemble des informations relative à un relevé. La saisie de la fiche est un traitement propre à cette fiche (une méthode). La prise en compte des évolutions numéro 3 et 5a se fait facilement par héritage (figure 8). Avec une classe abstraite `Fiche`, on peut mettre en œuvre un *pattern* `Fiche-Exploitation`, réutilisable dans différents contextes.
- La saisie fichier reste une opération générale car on souhaite centraliser l'utilisation du fichier.
- Les évolutions numéro 4, 5b et 6a concernent uniquement la classe `OPluvioObjet`, les modifications deviennent "locales" à cette classe.

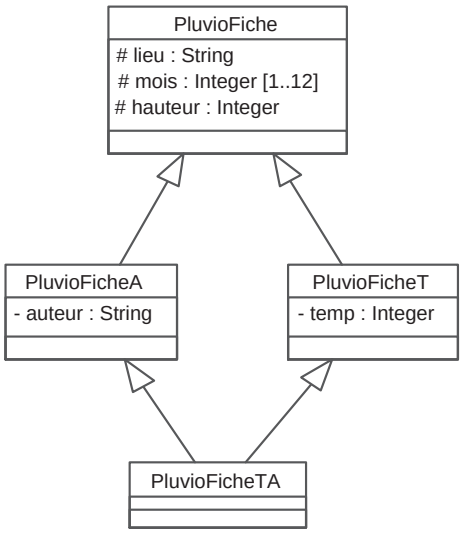



Figure 8 : *Héritage des fiches*

Détaillons maintenant le code des différentes classes.

### Classe PluvioFiche

La fiche contient les 3 informations de base. On peut la saisir au clavier et l'afficher.

Listing 3.3 – Code Java des fiches pluviométriques

```
package pluviometrie;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class PluvioFiche {
    // les variables sont protégées
    protected String lieux;

    protected int mois;

    protected int hauteur;

    public PluvioFiche(String l, int m, int h) {
        // initialisation par défaut des variables d'instance
        super();
        lieux = l;
        mois = m;
        hauteur = h;
    }

    // accesseurs
    public int getMois() {
        return mois;
    }

    public int getHauteur() {
        return hauteur;
    }

    public String getLieux() {
        return lieux;
    }

    // opérations
    static public PluvioFiche saisirFiche() {
        /*
         * saisit un relevé au clavier et rend la fiche.
         */
        // String line[]; // variables locales
        StringTokenizer line;
        String l;
        int h = 0, m = 0;
        BufferedReader stdin = new BufferedReader(new InputStreamReader(
                System.in));
        // déclare le tampon de lecture (pour types primitifs)
        try {
            System.out.print("donner votre fiche: lieu mois hauteur : ");
            //line = stdin.readLine().split("( )*,( )*");
            //lecture et extraction des trois valeurs séparées par un blanc
            line = new StringTokenizer(stdin.readLine());
```

```
            l = line.nextToken();
            if (! l.equals("z")) {
                m = Integer.parseInt(line.nextToken());
                h = Integer.parseInt(line.nextToken());
                while (m < 0 || m > 12) {
                    System.out.print("Erreur de mois, redonnez-le :");
                    line = new StringTokenizer(stdin.readLine());
                    m = Integer.parseInt(line.nextToken());
                }
            }
        } catch (IOException e) {
            System.err.println("IOException thrown " + e.toString());
            return null;
        }
        return new PluvioFiche(l, m, h);
    }

    public String toString() {
        /* affiche le relevé */
        return ("lieu " + lieux + " - mois " + mois + " - hauteur " + hauteur);
    }
}
```

### Classe PluvioObjet

Le code est simplifié pour différentes raisons : utiliser un ensemble permet de s'abstraire du nombre maximum de lieux (illimité) et du nombre de saisies (c'est le cardinal de l'ensemble), la saisie des fiches est simplifiée (une fiche et non trois collections). Nous avons ajouté des tests de fichiers et l'utilisation du `tokenizer` pour varier le code de la saisie clavier. Dans ce dernier cas, le séparateur est le blanc et non la virgule.

Listing 3.4 – Programme Java de calcul pluviométrique

```
package pluviometrie;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;

/**
 * @author pascal andre
 * traduction du programme pluvio de Pascal en Java
 * avec fichiers - version objet
 */

public class PluvioObjet {

    protected HashSet<PluvioFiche> fiches;

    public PluvioObjet() {
        // initialisation par défaut des variables d'instance
        super();
        fiches = new HashSet<PluvioFiche>();
    }
```

```
public void saisir_releves() {
    /*
     * saisit les fiches au clavier et les stocke dans fiches.
     */
    PluvioFiche f = PluvioFiche.saisirFiche();
    while (!f.getLieux().equals("z")) {
            fiches.add(f);
            f = PluvioFiche.saisirFiche();
    }
}

public void saisir_releves_fichier(String nom) {
    /*
     * saisit les relevés dans un fichier dont on précise le nom (surcharge)
     * et les stocke dans la variable fiches.
     */
    String line[]; // variables locales
    String l;
    int h, m;
    BufferedReader stdin = new BufferedReader(new InputStreamReader(
            System.in));
    // stdin est utile pour la correction du mois
    try {
        BufferedReader file = new BufferedReader(new FileReader(nom));
        // à placer dans le try car exception peut être levée
        // déclare le tampon de lecture (pour types primitifs)
        line = file.readLine().split("( )*,( )*");
        // ou line = StringTokenizer(stdin.readLine())
        l = line[0];
        while (!l.equals("z")) {
            m = Integer.parseInt(line[1]);
            h = Integer.parseInt(line[2]);
            while (m<0 || m>12) {
                System.out.print("Erreur de mois, redonnez-le :");
                line = stdin.readLine().split("( )*,( )*");
                m = Integer.parseInt(line[0]);
            }
            fiches.add(new PluvioFiche(l, m, h));
            line = file.readLine().split("( )*,( )*");
            l = line[0];
        }
    } catch (IOException e) {
        System.err.println("IOException thrown " + e.toString());
        // return false;
    }
}

public int[] exploiter_releves() {
    /*
     * calcule et stocke les sommes par mois
     */
    int somme[] = new int[12]; // par défaut initialisé à 0
    for (PluvioFiche f : fiches)
        somme[f.getMois()-1] += f.getHauteur();
    return somme;
}
```

```
    public void afficher_releves() {
        /*
         * affiche les relevés
         */
        System.out.println("Relevés :");
        for (PluvioFiche f : fiches) // itération sur une collection générique
            System.out.println("Fiche :"+f.toString());
    }

    public void afficher_so() {
        /*
         * affiche les résultats annuels
         */
        int somme[] = this.exploiter_releves();
        System.out.println("Totaux :");
        for (int i = 0; i < somme.length; i++) //length évite les débordements
            System.out.println("somme du mois " + (i + 1) + " est " + somme[i]);
    }

    public static void main(String args[]) {
        /* programme principal */
        PluvioObjet pluvio = new PluvioObjet();
        String nom_fic = System.getProperty("user.dir")+System.getProperty("file.separator")+"releves.txt";
        File f = new File(nom_fic);
        if (f.exists() && f.canRead())
            pluvio.saisir_releves_fichier(nom_fic);   // saisie fichier
        else pluvio.saisir_releves() ; // saisie clavier
        pluvio.afficher_releves();
        pluvio.exploiter_releves();
        pluvio.afficher_so();
    }
}
```

Ce code produit le résultat suivant pour la lecture sur fichier :

```
Relevés :
Fiche :lieu nantes - mois 10 - hauteur 6
Fiche :lieu paris - mois 12 - hauteur 10
Fiche :lieu paris - mois 12 - hauteur 10
Fiche :lieu paris - mois 12 - hauteur 10
Fiche :lieu nantes - mois 5 - hauteur 123
Fiche :lieu nantes - mois 10 - hauteur 234
Totaux :
somme du mois 1 est 0
somme du mois 2 est 0
somme du mois 3 est 0
somme du mois 4 est 0
somme du mois 5 est 123
somme du mois 6 est 0
somme du mois 7 est 0
somme du mois 8 est 0
somme du mois 9 est 0
somme du mois 10 est 240
somme du mois 11 est 0
somme du mois 12 est 30
```

Ce code produit le résultat suivant pour la lecture clavier :

```
donner votre fiche: lieu mois hauteur : paris 12 10
```

```
donner votre fiche: lieu mois hauteur : paris 12 10
donner votre fiche: lieu mois hauteur : paris 12 10
donner votre fiche: lieu mois hauteur : nantes 10 234
donner votre fiche: lieu mois hauteur : nantes 10 234
donner votre fiche: lieu mois hauteur : nantes 12 234
donner votre fiche: lieu mois hauteur : z
Relevés :
Fiche :lieu paris - mois 12 - hauteur 10
Fiche :lieu paris - mois 12 - hauteur 10
Fiche :lieu nantes - mois 10 - hauteur 234
Fiche :lieu nantes - mois 10 - hauteur 234
Fiche :lieu nantes - mois 12 - hauteur 234
Fiche :lieu paris - mois 12 - hauteur 10
Totaux :
somme du mois 1 est 0
somme du mois 2 est 0
somme du mois 3 est 0
somme du mois 4 est 0
somme du mois 5 est 0
somme du mois 6 est 0
somme du mois 7 est 0
somme du mois 8 est 0
somme du mois 9 est 0
somme du mois 10 est 468
somme du mois 11 est 0
somme du mois 12 est 264
```

On notera que l'ordre d'itération dans un ensemble de fiches n'est pas l'ordre FIFO comme d ns un vecteur ou un tableau.

### 1.2 Bilan

A travers ce petit exemple, nous avons mis en évidence qu'un découpage en objet permet de définir une meilleure abstraction du code. Ce découpage facilite la lisibilité, la conception, le test des différents modules. La cohésion des objets et l'héritage facilitent l'extension de l'application et la réutilisation de code. Noter aussi que la notation UML est un bon outil de documentation du code.

## 2 Quelques hiérarchies de classes

L'API Java est relativement bien fournie et augmente chaque jour, notamment par l'activité du monde du logiciel libre. La programmation Web avec Java est aussi une source riche d'information. Chaque paquetage est une collection d'interfaces et de classes. Nous présentons très sommairement quelques hiérarchies de classes importantes en Java.

### 2.1 Paquetages

Voici une liste des paquetages de l'API Java 2 SE 1.5
(source `http://java.sun.com/j2se/1.5.0/docs/api/`).
Nous avons omis les paquetages issus de `omg`. Les paquetages `java` correspondent au langage de base et `javax` aux extensions (extended java).

| **paquetage** | **description** |
|---|---|
| java.applet | Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context. |
| java.awt | Contains all of the classes for creating user interfaces and for painting graphics and images. |
| java.awt.color | Provides classes for color spaces. |
| java.awt. datatransfer | Provides interfaces and classes for transferring data between and within applications. |
| java.awt.dnd | Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI. |

| | |
|---|---|
| java.awt.event | Provides interfaces and classes for dealing with different types of events fired by AWT components. |
| java.awt.font | Provides classes and interface relating to fonts. |
| java.awt.geom | Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry. |
| java.awt.im | Provides classes and interfaces for the input method framework. |
| java.awt.im.spi | Provides interfaces that enable the development of input methods that can be used with any Java runtime environment. |
| java.awt.image | Provides classes for creating and modifying images. |
| java.awt.image. renderable | Provides classes and interfaces for producing rendering-independent images. |
| java.awt.print | Provides classes and interfaces for a general printing API. |
| java.beans | Contains classes related to developing beans – components based on the JavaBeansTM architecture. |
| java.beans.beancontext | Provides classes and interfaces relating to bean context. |
| java.io | Provides for system input and output through data streams, serialization and the file system. |
| java.lang | Provides classes that are fundamental to the design of the Java programming language. |
| java.lang.annotation | Provides library support for the Java programming language annotation facility. |
| java.lang.instrument | Provides services that allow Java programming language agents to instrument programs running on the JVM. |
| java.lang.management | Provides the management interface for monitoring and management of the Java virtual machine as well as the operating system on which the Java virtual machine is running. |
| java.lang.ref | Provides reference-object classes, which support a limited degree of interaction with the garbage collector. |
| java.lang.reflect | Provides classes and interfaces for obtaining reflective information about classes and objects. |
| java.math | Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal). |
| java.net | Provides the classes for implementing networking applications. |
| java.nio | Defines buffers, which are containers for data, and provides an overview of the other NIO packages. |
| java.nio.channels | Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations. |
| java.nio.channels.spi | Service-provider classes for the java.nio.channels package. |
| java.nio.charset | Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters. |
| java.nio.charset.spi | Service-provider classes for the java.nio.charset package. |
| 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 garbage-collection (DGC). |
| java.rmi.registry | Provides a class and two interfaces for the RMI registry. |
| java.rmi.server | Provides classes and interfaces for supporting the server side of RMI. |
| java.security | Provides the classes and interfaces for the security framework. |
| java.security.acl | The classes and interfaces in this package have been superseded by classes in the java.security package. |
| java.security.cert | Provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. |
| java.security. interfaces | Provides interfaces for generating RSA (Rivest, Shamir and Adleman AsymmetricCipher algorithm) keys as defined in the RSA Laboratory Technical Note PKCS#1, and DSA (Digital Signature Algorithm) keys as defined in NIST's FIPS-186. |
| java.security.spec | Provides classes and interfaces for key specifications and algorithm parameter specifications. |
| java.sql | Provides the API for accessing and processing data stored in a data source (usually a relational database) using the JavaTM programming language. |
| java.text | Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages. |

| | |
|---|---|
| java.util | Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). |
| java.util.concurrent | Utility classes commonly useful in concurrent programming. |
| java.util.conc-.atomic | A small toolkit of classes that support lock-free thread-safe programming on single variables. |
| java.util.conc-.locks | Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. |
| java.util.jar | Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format with an optional manifest file. |
| java.util.logging | Provides the classes and interfaces of the JavaTM 2 platform's core logging facilities. |
| java.util.prefs | This package allows applications to store and retrieve user and system preference and configuration data. |
| java.util.regex | Classes for matching character sequences against patterns specified by regular expressions. |
| java.util.zip | Provides classes for reading and writing the standard ZIP and GZIP file formats. |
| javax.accessibility | Defines a contract between user-interface components and an assistive technology that provides access to those components. |
| javax.crypto | Provides the classes and interfaces for cryptographic operations. |
| javax.crypto.interfaces | Provides interfaces for Diffie-Hellman keys as defined in RSA Laboratories' PKCS #3. |
| javax.crypto.spec | Provides classes and interfaces for key specifications and algorithm parameter specifications. |
| javax.imageio | The main package of the Java Image I/O API. |
| javax.imageio.event | A package of the Java Image I/O API dealing with synchronous notification of events during the reading and writing of images. |
| javax.imageio.metadata | A package of the Java Image I/O API dealing with reading and writing metadata. |
| javax.imageio.plugins.bmp | Package containing the public classes used by the built-in BMP plug-in. |
| javax.imageio.plugins.jpeg | Classes supporting the built-in JPEG plug-in. |
| javax.imageio.spi | A package of the Java Image I/O API containing the plug-in interfaces for readers, writers, transcoders, and streams, and a runtime registry. |
| javax.imageio.stream | A package of the Java Image I/O API dealing with low-level I/O from files and streams. |
| javax.management | Provides the core classes for the Java Management Extensions. |
| javax.mana-.loading | Provides the classes which implement advanced dynamic loading. |
| javax.mana-.modelmbean | Provides the definition of the ModelMBean classes. |
| javax.mana-.monitor | Provides the definition of the monitor classes. |
| javax.mana-.openmbean | Provides the open data types and Open MBean descriptor classes. |
| javax.mana-.relation | Provides the definition of the Relation Service. |
| javax.mana-.remote | Interfaces for remote access to JMX MBean servers. |
| javax.mana-.remote.rmi | The RMI connector is a connector for the JMX Remote API that uses RMI to transmit client requests to a remote MBean server. |
| javax.mana-.timer | Provides the definition of the Timer MBean. |
| javax.naming | Provides the classes and interfaces for accessing naming services. |
| javax.naming.directory | Extends the javax.naming package to provide functionality for accessing directory services. |
| javax.naming.event | Provides support for event notification when accessing naming and directory services. |
| javax.naming.ldap | Provides support for LDAPv3 extended operations and controls. |
| javax.naming.spi | Provides the means for dynamically plugging in support for accessing naming and directory services through the javax.naming and related packages. |
| javax.net | Provides classes for networking applications. |
| javax.net.ssl | Provides classes for the secure socket package. |
| javax.print | Provides the principal classes and interfaces for the JavaTM Print Service API. |
| javax.print.attribute | Provides classes and interfaces that describe the types of JavaTM Print Service attributes and how they can be collected into attribute sets. |

| | |
|---|---|
| javax.print.attribute.standard | Package javax.print.attribute.standard contains classes for specific printing attributes. |
| javax.print.event | Package javax.print.event contains event classes and listener interfaces. |
| javax.rmi | Contains user APIs for RMI-IIOP. |
| javax.rmi.CORBA | Contains portability APIs for RMI-IIOP. |
| javax.rmi.ssl | Provides implementations of RMIClientSocketFactory and RMIServerSocketFactory over the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols. |
| javax.security.auth | This package provides a framework for authentication and authorization. |
| javax.security.auth.callback | This package provides the classes necessary for services to interact with applications in order to retrieve information (authentication data including usernames or passwords, for example) or to display information (error and warning messages, for example). |
| javax.security.auth.kerberos | This package contains utility classes related to the Kerberos network authentication protocol. |
| javax.security.auth.login | This package provides a pluggable authentication framework. |
| javax.security.auth.spi | This package provides the interface to be used for implementing pluggable authentication modules. |
| javax.security.auth.x500 | This package contains the classes that should be used to store X500 Principal and X500 Private Crendentials in a Subject. |
| javax.security.cert | Provides classes for public key certificates. |
| javax.security.sasl | Contains class and interfaces for supporting SASL. |
| javax.sound.midi | Provides interfaces and classes for I/O, sequencing, and synthesis of MIDI (Musical Instrument Digital Interface) data. |
| javax.sound.midi.spi | Supplies interfaces for service providers to implement when offering new MIDI devices, MIDI file readers and writers, or sound bank readers. |
| javax.sound.sampled | Provides interfaces and classes for capture, processing, and playback of sampled audio data. |
| javax.sound.sampled.spi | Supplies abstract classes for service providers to subclass when offering new audio devices, sound file readers and writers, or audio format converters. |
| javax.sql | Provides the API for server side data source access and processing from the JavaTM programming language. |
| javax.sql.rowset | Standard interfaces and base classes for JDBC RowSet implementations. |
| javax.sql.rowset.serial | Provides utility classes to allow serializable mappings between SQL types and data types in the Java programming language. |
| javax.sql.rowset.spi | The standard classes and interfaces that a third party vendor has to use in its implementation of a synchronization provider. |
| javax.swing | Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms. |
| javax.swing.border | Provides classes and interface for drawing specialized borders around a Swing component. |
| javax.swing.colorchooser | Contains classes and interfaces used by the JColorChooser component. |
| javax.swing.event | Provides for events fired by Swing components. |
| javax.swing.filechooser | Contains classes and interfaces used by the JFileChooser component. |
| javax.swing.plaf | Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities. |
| javax.swing.plaf.basic | Provides user interface objects built according to the Basic look and feel. |
| javax.swing.plaf.metal | Provides user interface objects built according to the Java look and feel (once codenamed Metal), which is the default look and feel. |
| javax.swing.plaf.multi | Provides user interface objects that combine two or more look and feels. |
| javax.swing.plaf.synth | Synth is a skinnable look and feel in which all painting is delegated. |
| javax.swing.table | Provides classes and interfaces for dealing with javax.swing.JTable. |
| javax.swing.text | Provides classes and interfaces that deal with editable and noneditable text components. |
| javax.swing.text.html | Provides the class HTMLEditorKit and supporting classes for creating HTML text editors. |

| | |
|---|---|
| javax.swing.text.html.parser | Provides the default HTML parser, along with support classes. |
| javax.swing.text.rtf | Provides a class (RTFEditorKit) for creating Rich-Text-Format text editors. |
| javax.swing.tree | Provides classes and interfaces for dealing with javax.swing.JTree. |
| javax.swing.undo | Allows developers to provide support for undo/redo in applications such as text editors. |
| javax.transaction | Contains three exceptions thrown by the ORB machinery during unmarshalling. |
| javax.transaction.xa | Provides the API that defines the contract between the transaction manager and the resource manager, which allows the transaction manager to enlist and delist resource objects (supplied by the resource manager driver) in JTA transactions. |
| javax.xml | Defines core XML constants and functionality from the XML specifications. |
| javax.xml.datatype | XML/Java Type Mappings. |
| javax.xml.namespace | XML Namespace processing. |
| javax.xml.parsers | Provides classes allowing the processing of XML documents. |
| javax.xml.transform | This package defines the generic APIs for processing transformation instructions, and performing a transformation from source to result. |
| javax.xml.transform.dom | This package implements DOM-specific transformation APIs. |
| javax.xml.transform.sax | This package implements SAX2-specific transformation APIs. |
| javax.xml.transform.stream | This package implements stream- and URI- specific transformation APIs. |
| javax.xml.validation | This package provides an API for validation of XML documents. |
| javax.xml.xpath | This package provides an object-model neutral API for the evaluation of XPath expressions and access to the evaluation environment. |

TABLEAU IV– *Paquetages de Java*

**Avertissement** : *Certaines des hiérarchies de classes, présentées dans la suite de ce chapitre, peuvent ne pas être complètes ou à jour en fonction de l'API Java utilisée.*

## 2.2 Object

La classe `Object` est fondamentale en Java, elle donne le comportement de base de chaque objet. Voici un aperçu de sa hiérarchie, que nous aborderons en détail par la suite.
(source http://www.oracledbaexpert.com/japplet/d.html )

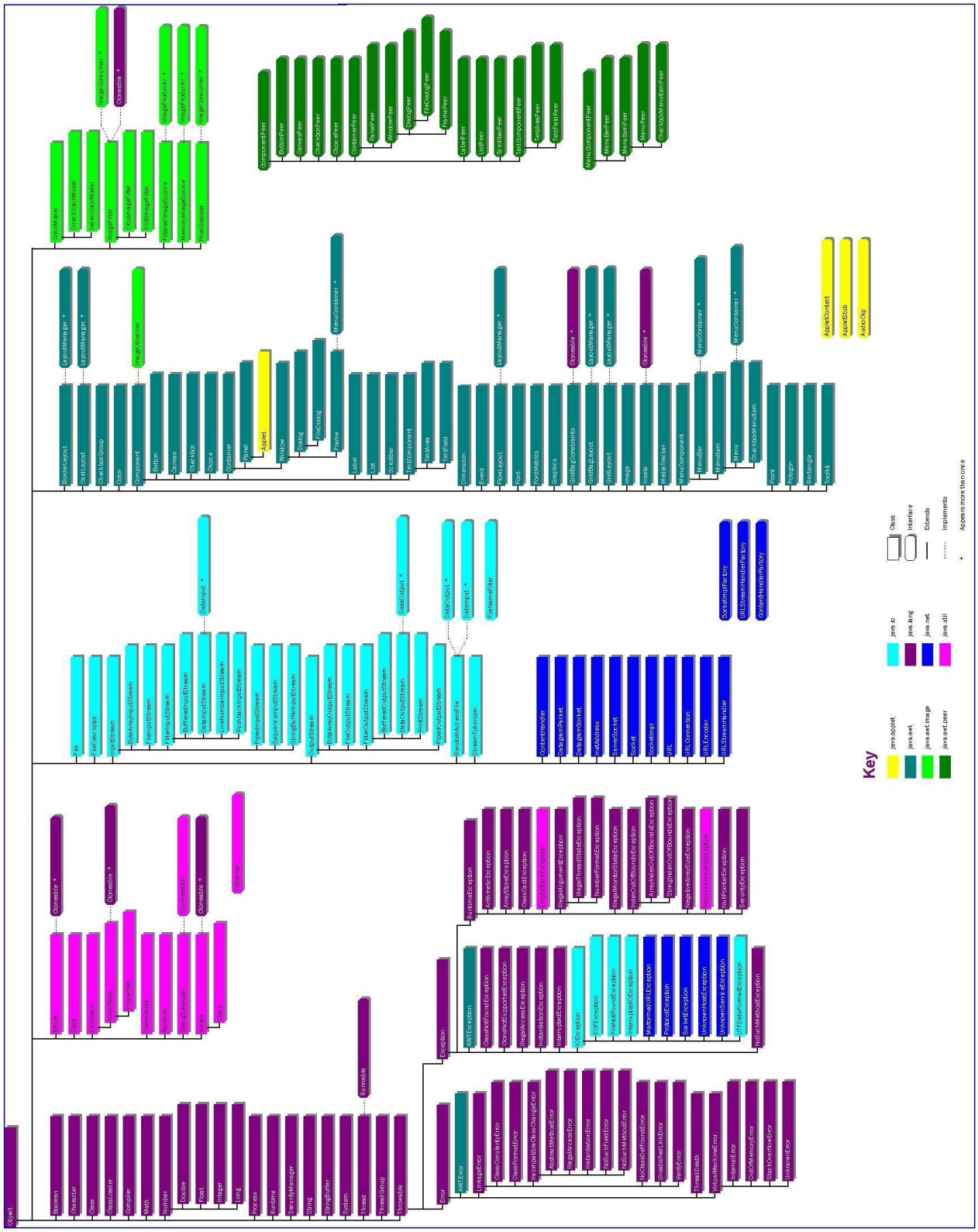



Figure 9 : *Hiérarchie d'héritage de* `Object`

### 2.3 La classe `Object` et les processus

La classe `Object` se trouve dans le paquetage `java.lang` qui contient les éléments de base du langage (types primitifs, classes, exécution, erreurs et exceptions).

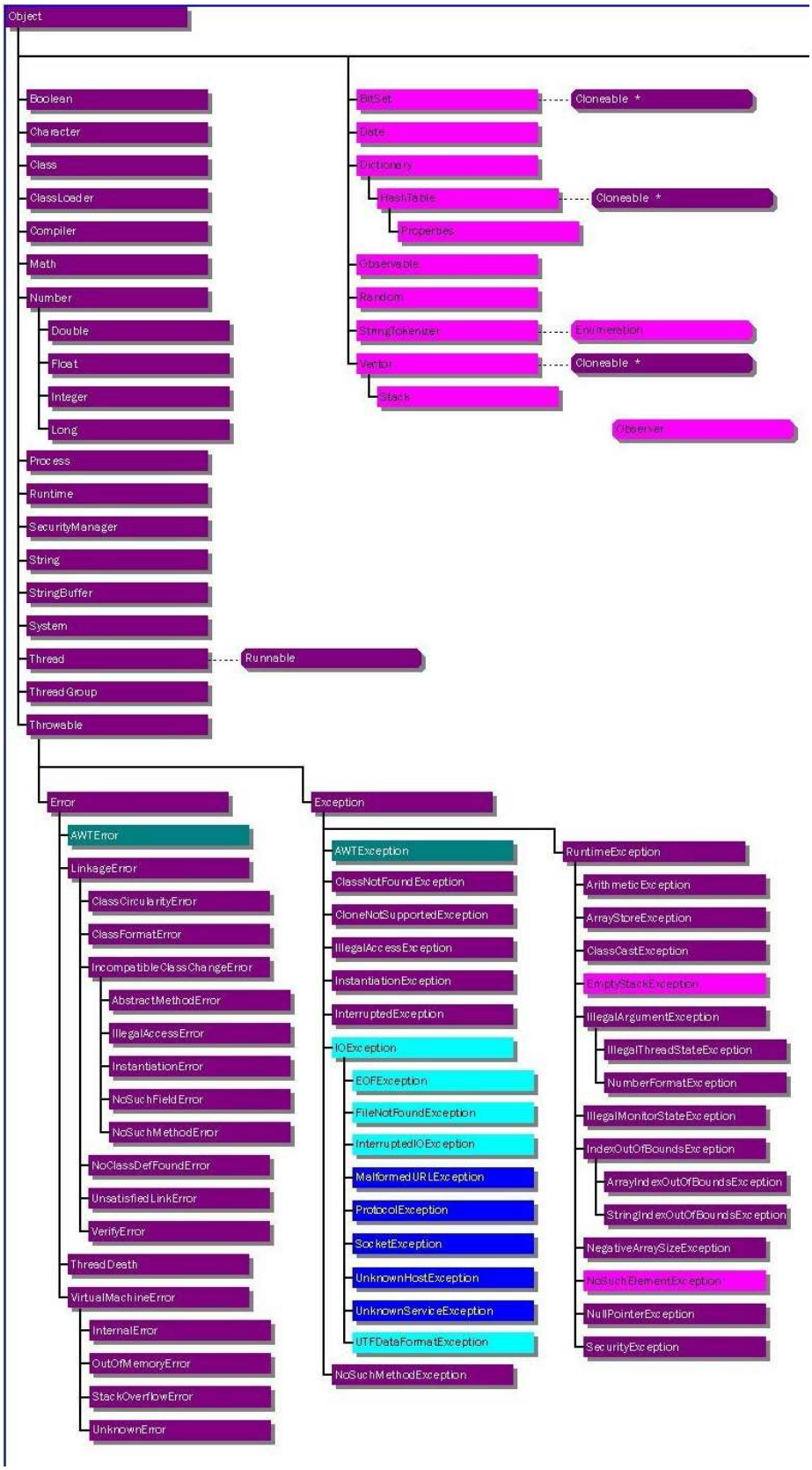



Figure 10 : *Hiérarchie d'héritage de* `Object`

```
package java.lang;
public class Object {
  private static native void registerNatives();
  static {
    registerNatives();
  }
  public final native Class<? extends Object> getClass();
  public native int hashCode();
  public boolean equals(Object obj) {
     return (this == obj);
  }
  protected native Object clone() throws CloneNotSupportedException;
  public String toString() {
     return getClass().getName() + "@" + Integer.toHexString(hashCode());
  }
  public final native void notify();
  public final native void notifyAll();
  public final native void wait(long timeout) throws InterruptedException;
  public final void wait(long timeout, int nanos) throws InterruptedException {
    if (timeout < 0) {
      throw new IllegalArgumentException("timeout value is negative");
    }
    if (nanos < 0 || nanos > 999999) {
      throw new IllegalArgumentException(
        "nanosecond timeout value out of range");
    }
    if (nanos >= 500000 || (nanos != 0 && timeout == 0)) {
       timeout++;
    }
    wait(timeout);
    }
    public final void wait() throws InterruptedException {
    wait(0);
    }
    protected void finalize() throws Throwable { }
}
```

`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/package-tree.html`

Le mot-clé `native` indique une méthode primitive (voir JNI dans la section 3 du chapitre 6).

Les mathématiques sont approfondies avec les classes du paquetage `java.math`.

La classe `Class` représente les classes Java, elle est instanciée à chaque chargement de classe en mémoire. Elle permet un traitement simplifié de niveau métaclasse (instanciation notamment) ou réflexif (voir section 4) du chapitre 5.

Les processus et les *threads* seront étudiés dans la section 2 du chapitre 6. Les exceptions seront étudiés dans la section 1 du chapitre 5.

## 2.4 Collections

La hiérarchie des collections implante les types de données relatifs aux structures de données de haut niveau de la programmation : tableaux, listes, ensembles, multi-ensembles, collections ordonnées ou triées... Elle se trouve dans le paquetage `java.util`.

La hiérarchie des collections est un arbre d'interfaces et d'implantations (classes). La figure 11 en donne une représentation synthétique. La collection est un outil fondamental d'abstraction en modélisation. En programmation à objets, toutes les implantations imaginables de ces types de données sont réalisables par adaptation de la hiérarchie des collections.

On notera que les tableaux de taille fixe ne sont pas des collections. On notera aussi que depuis la version 1.5, les collections sont génériques. En Java, le parcours des collections se fait avec un itérateur (comme en C++).

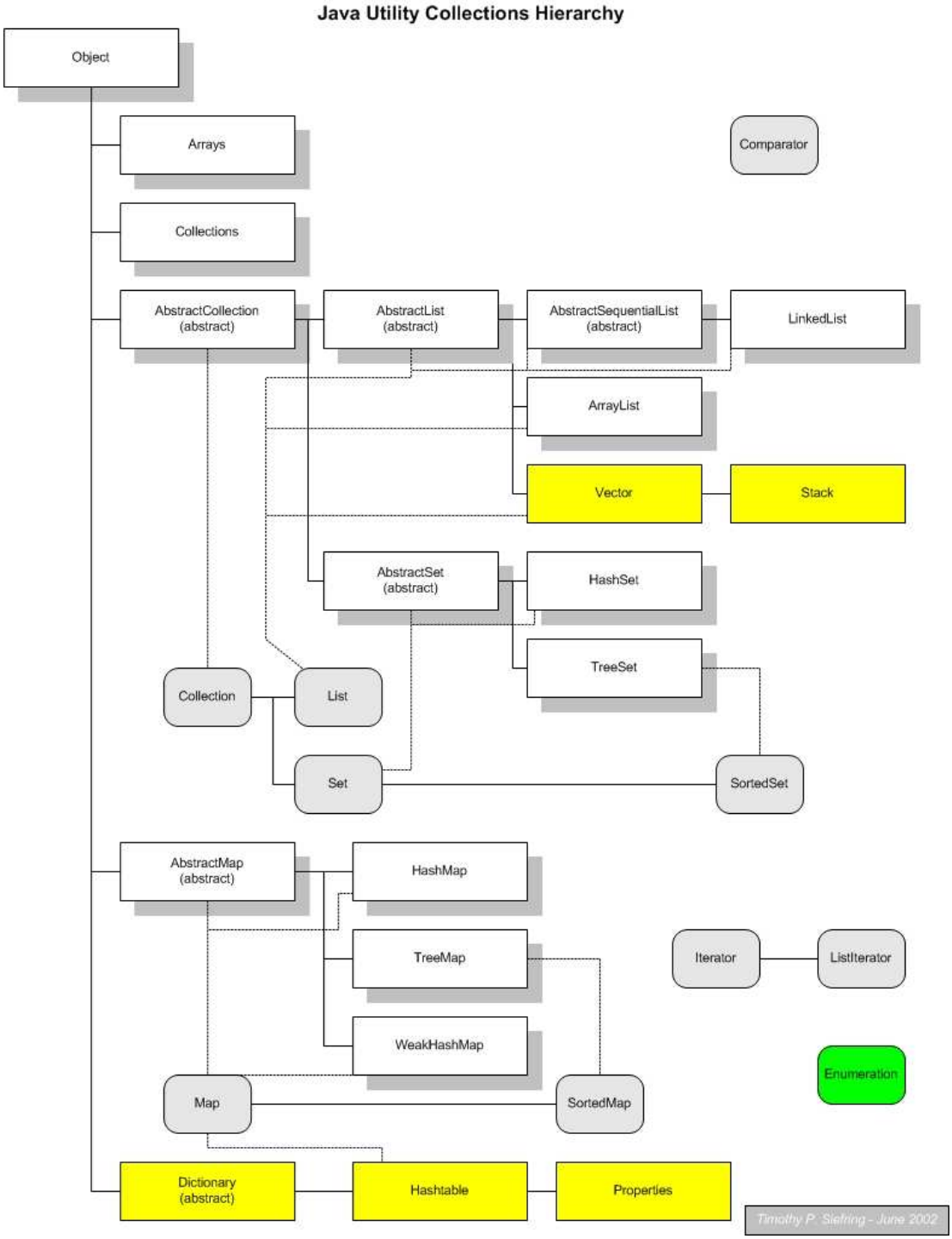



Figure 11 : *Hiérarchie des collections*

(source http://rvcc2.raritanval.edu/ tsiefrin/java-hos-utilcollhier.htm )

```
public interface Collection {
   // Basic Operations
   int size();
   boolean isEmpty();
```

```
   boolean contains(Object element);
   boolean add(Object element); // Optional
   boolean remove(Object element); // Optional
   Iterator iterator();
   // Modification Operations
   boolean containsAll(Collection c);
   boolean addAll(Collection c); // Optional
   boolean removeAll(Collection c); // Optional
   boolean retainAll(Collection c); // Optional
   void clear(); // Optional
   // Array Operations
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```

## 2.5 Les classes utilitaires

Les classes utilitaires sont définies dans le paquetage `java.util`. Elles comprennent les collection (section 2.4), les archives `jar`, les dates et heures, etc.

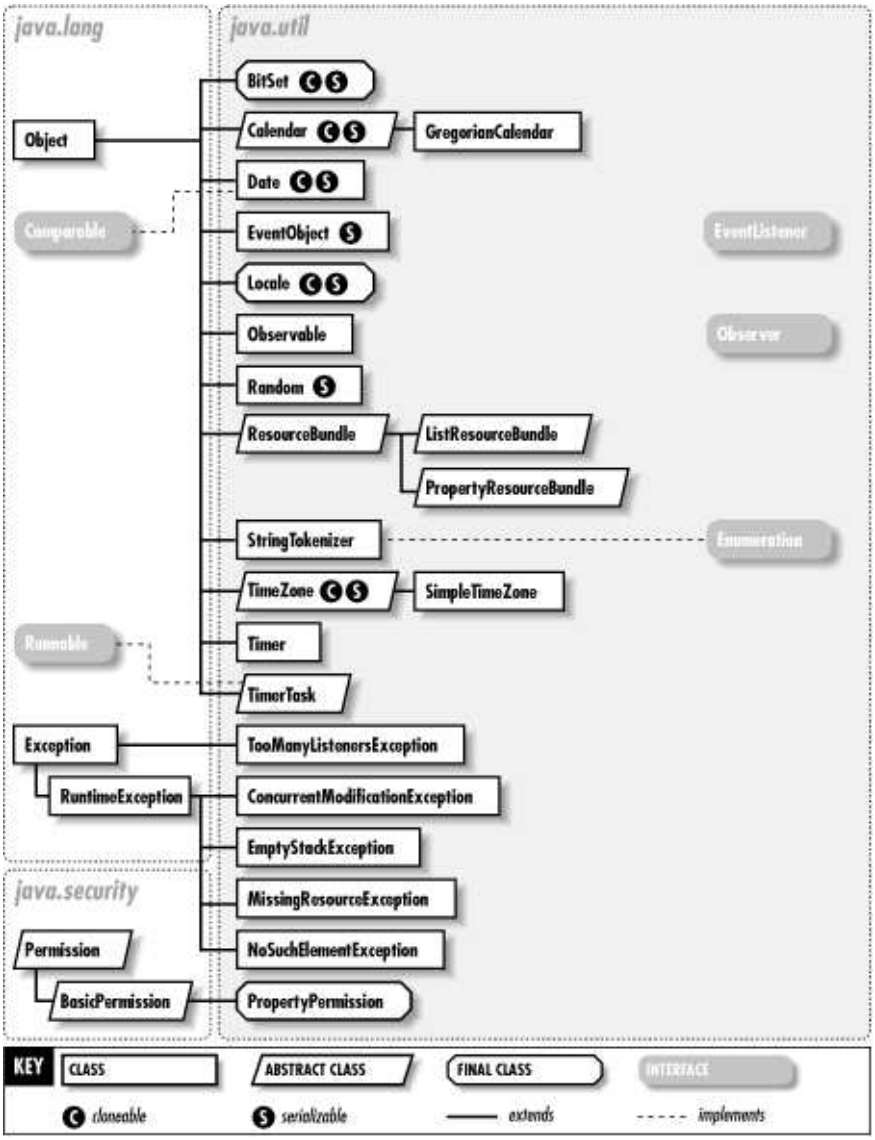



Figure 12 : *Hiérarchie d'héritage des classes utilitaires (hors collections)*

(source http://www.unix.org.ua/orelly/java-ent/jnut/ch24_01.htm )

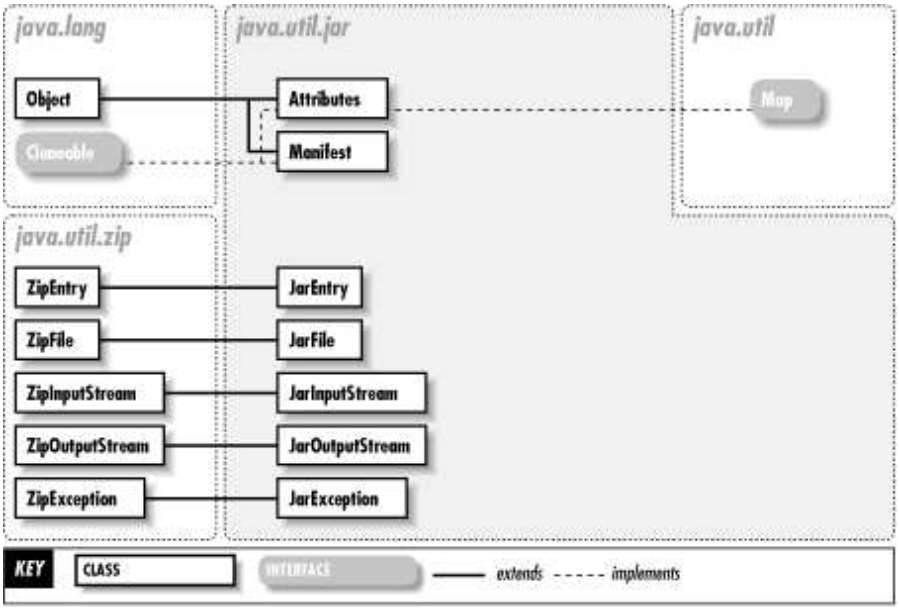



Figure 13 : *Hiérarchie d'héritage des classes utilitaires (jar)*

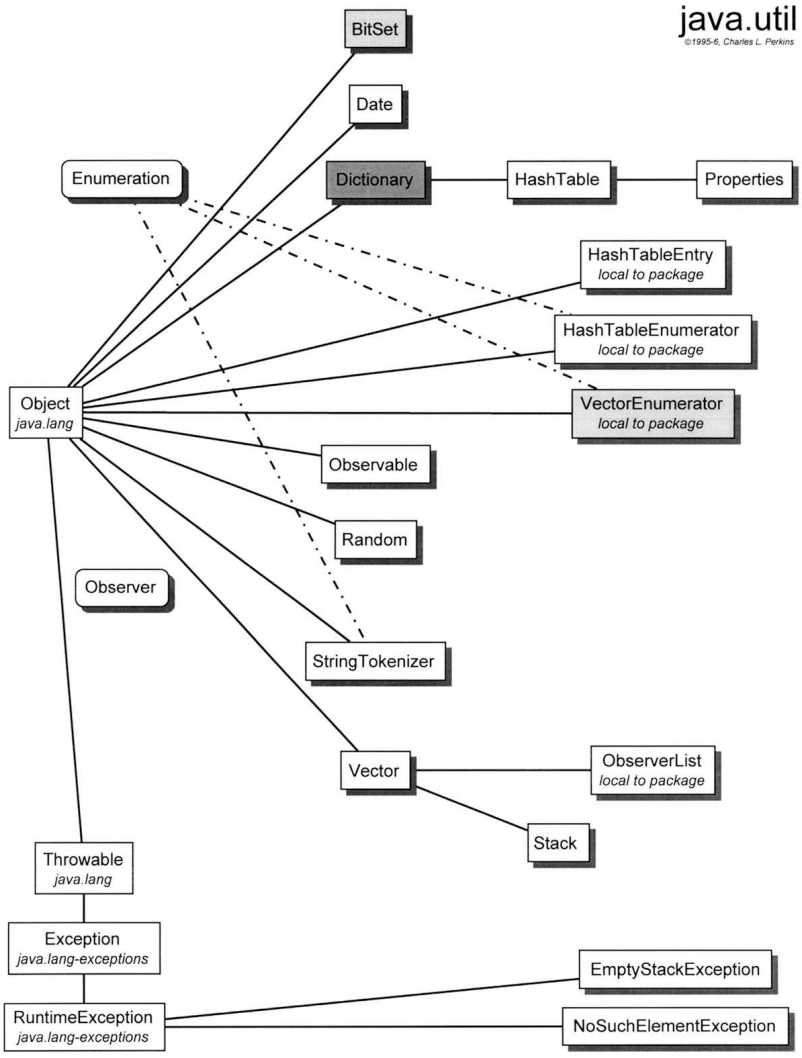



Figure 14 : *Hiérarchie d'héritage des classes utilitaires (autre vue)*

(source http://docs.rinet.ru/J21/ch30.htm )

### 2.6 Flux, Entrées-sorties et réseau

Les entrées-sorties sont définies dans le paquetage `java.io`. Elles sont basées sur des flux (flots) d'information (binaires, octets, caractères...). Nous approfondissons ces classes dans la section 2 du chapitre 5.

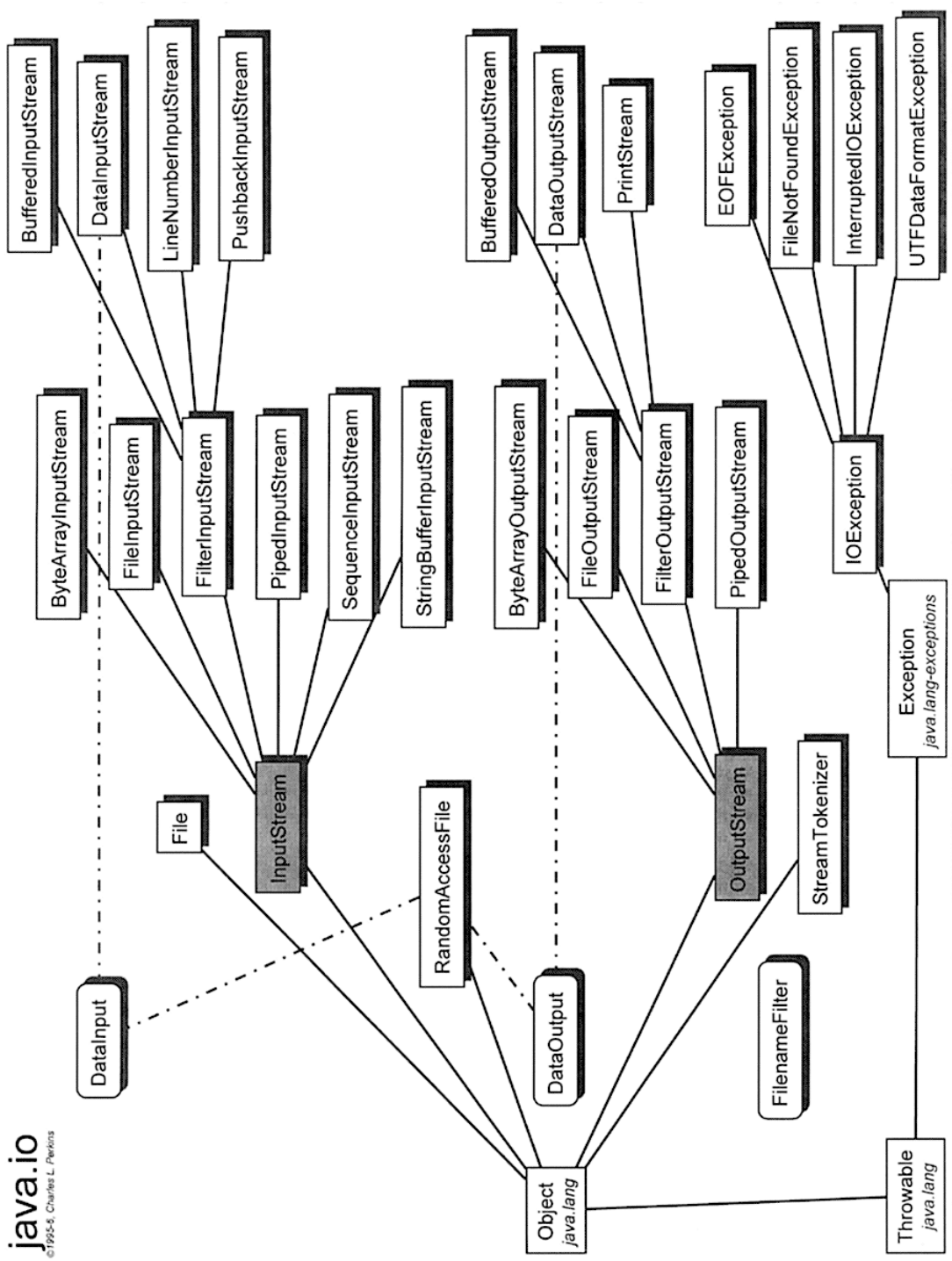



Figure 15 : *Hiérarchie d'héritage des entrées-sorties*

(source http://docs.rinet.ru/J21/ch30.htm )

## 2.7 Composants visuels

Le composant visuel est la brique de base de l'interface graphique. Il comprend des formes, des zones textuelles, des vues imbriquées... et des boutons enficheables. Les composants visuels sont organisés en deux couches : la couche primitive AWT (*Abstract Windowing Toolkit*) du paquetage `java.io` et la couche avancée Swing du paquetage `javax.swing`.

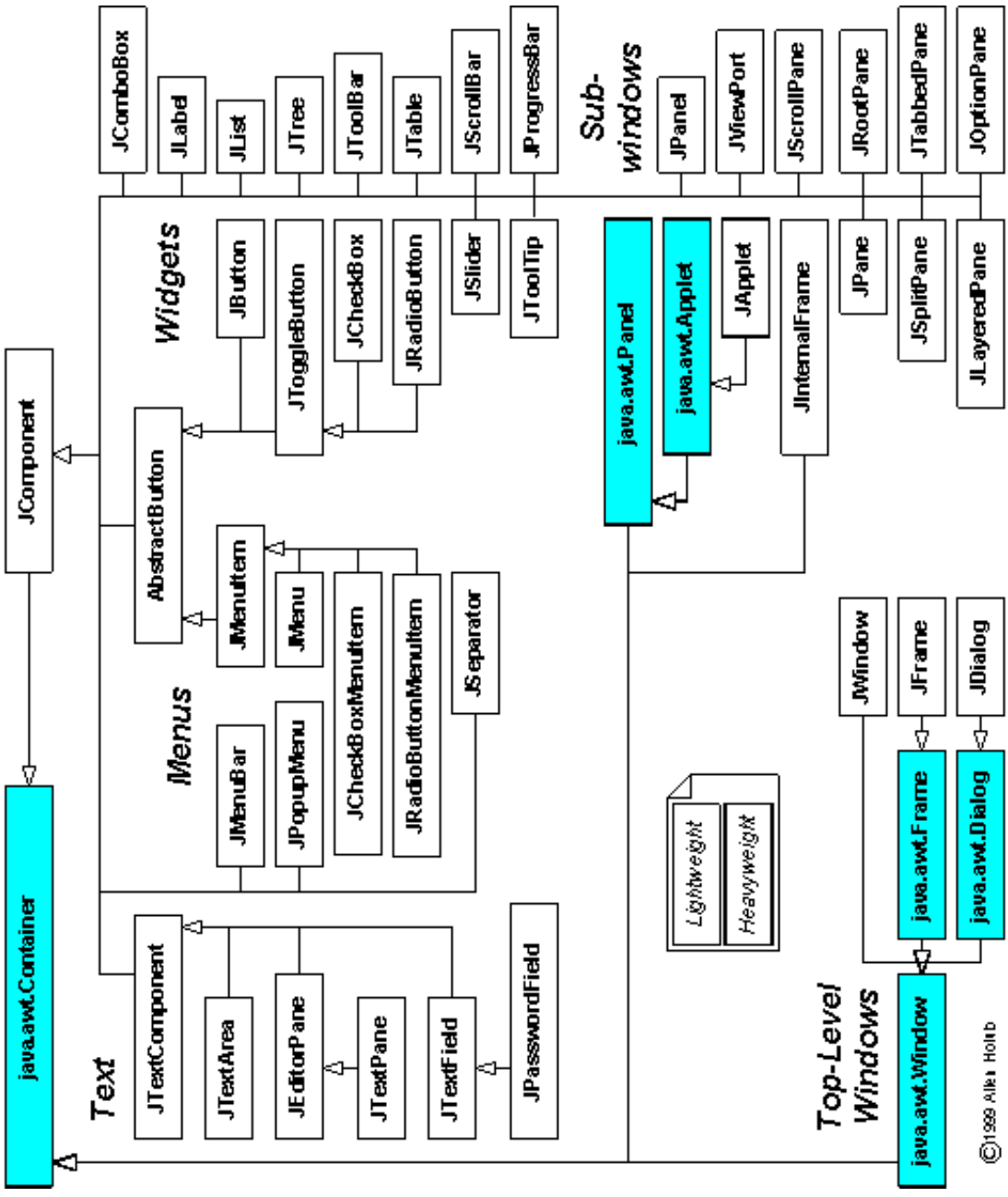



Figure 16 : *Hiérarchie d'héritage des composants* `Swing`

(source http://www.holub.com/goodies/java.swing.html )

La hiérarchie suivante présente les deux couches.

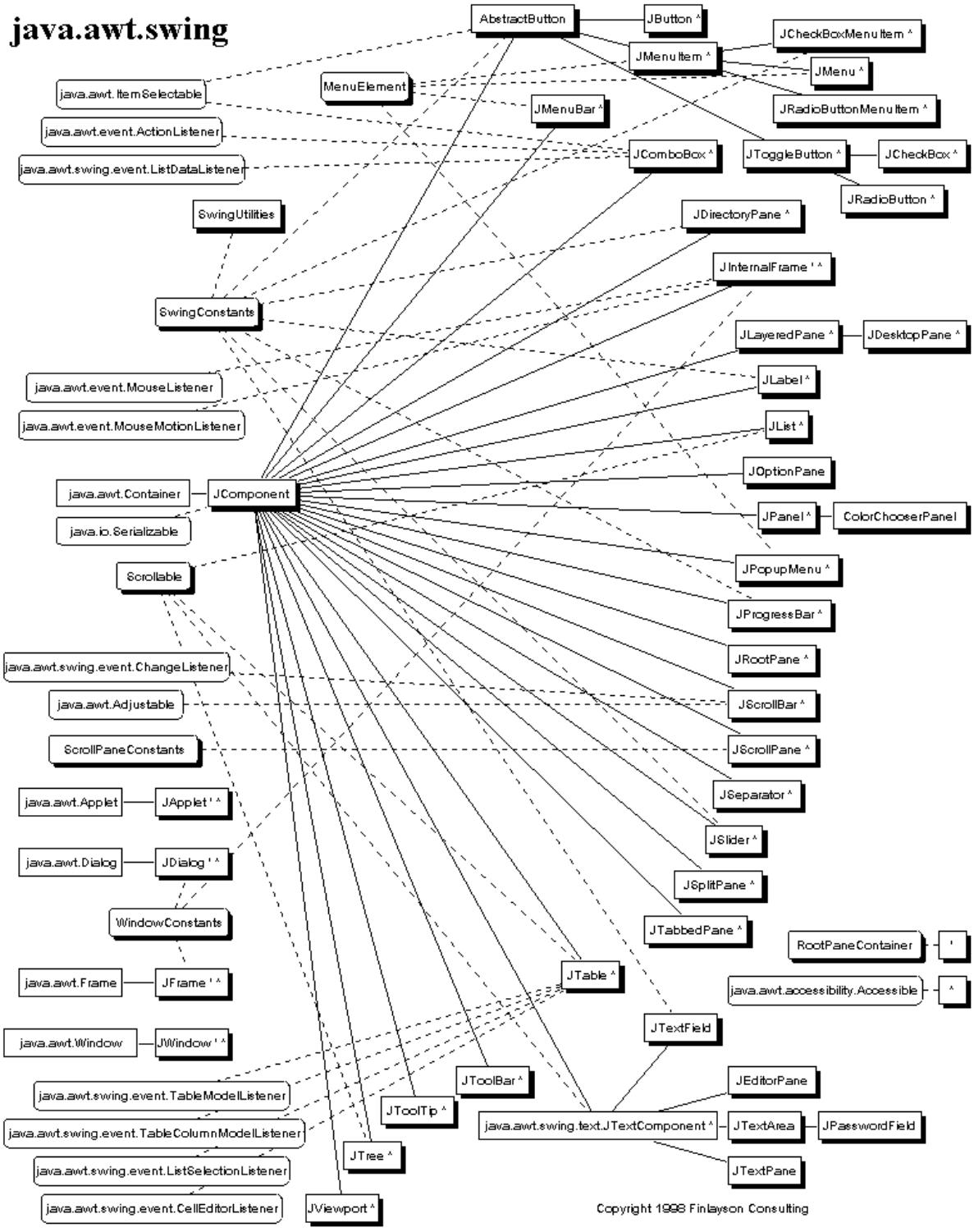



Figure 17 : *Hiérarchie d'héritage des composants visuels*

(source http://tiki-lounge.com/ raf/jfcmanual/jfc.2.html )

### Evénements

La hiérarchie suivante présente la gestion des événements, notamment ceux liés à l'IHM. Elles se trouvent dans le paquetage `java.awt.events` et le paquetage `java.awt.swing.event`.

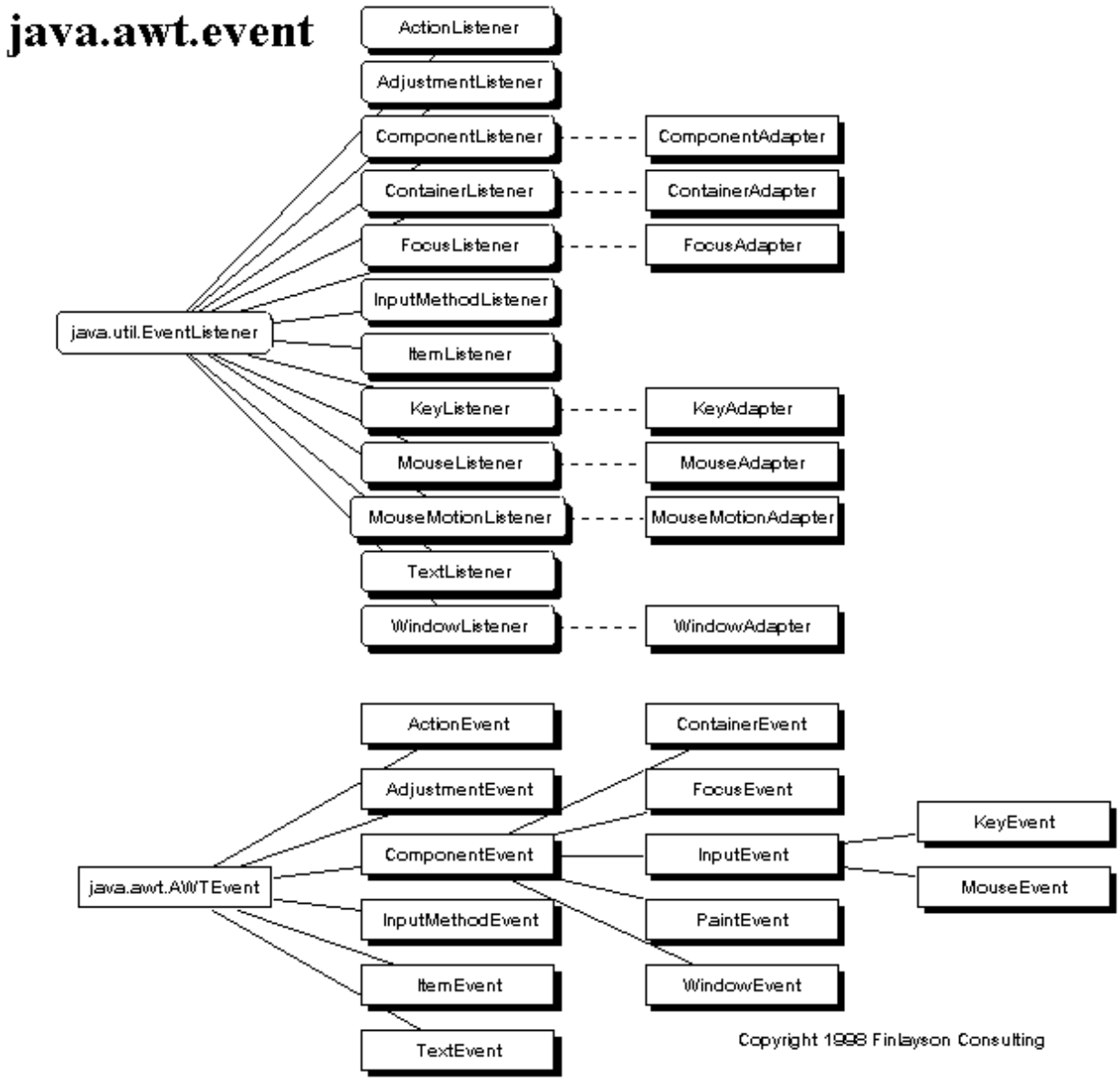



Figure 18 : *Hiérarchie d'héritage des événements*

(source http://tiki-lounge.com/ raf/jfcmanual/jfc.3.html )

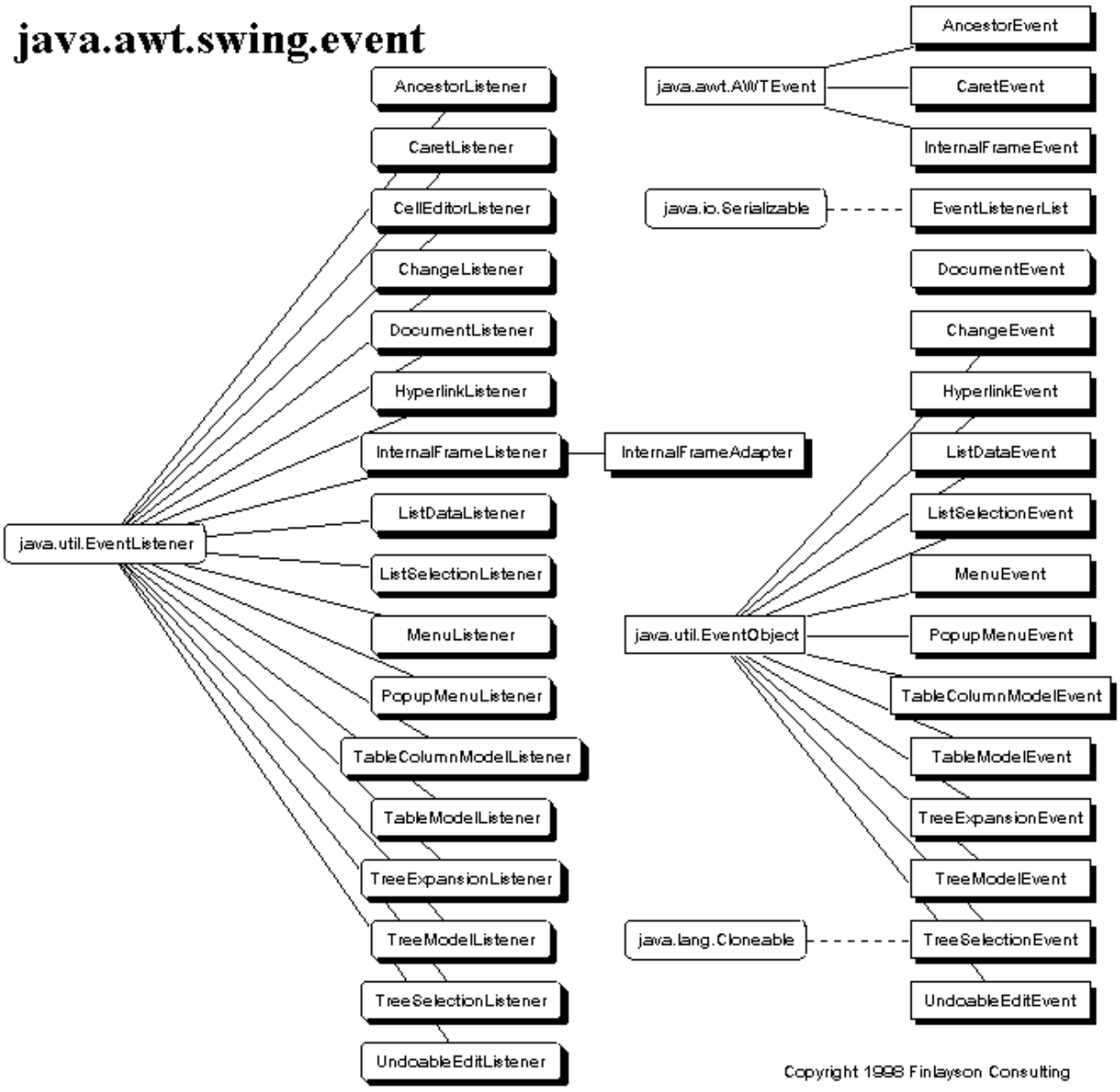



Figure 19 : *Hiérarchie d'héritage des événements Swing*

(source http://tiki-lounge.com/ raf/jfcmanual/jfc.3.html )

### Images

La hiérarchie suivante présente les classes liées au graphisme. Elles se trouvent dans le paquetage `java.awt.image` et le paquetage `java.awt.image.renderable`. Voir aussi

```
java.awt
java.awt.color
java.awt.font
java.awt.event
java.awt.geom
```

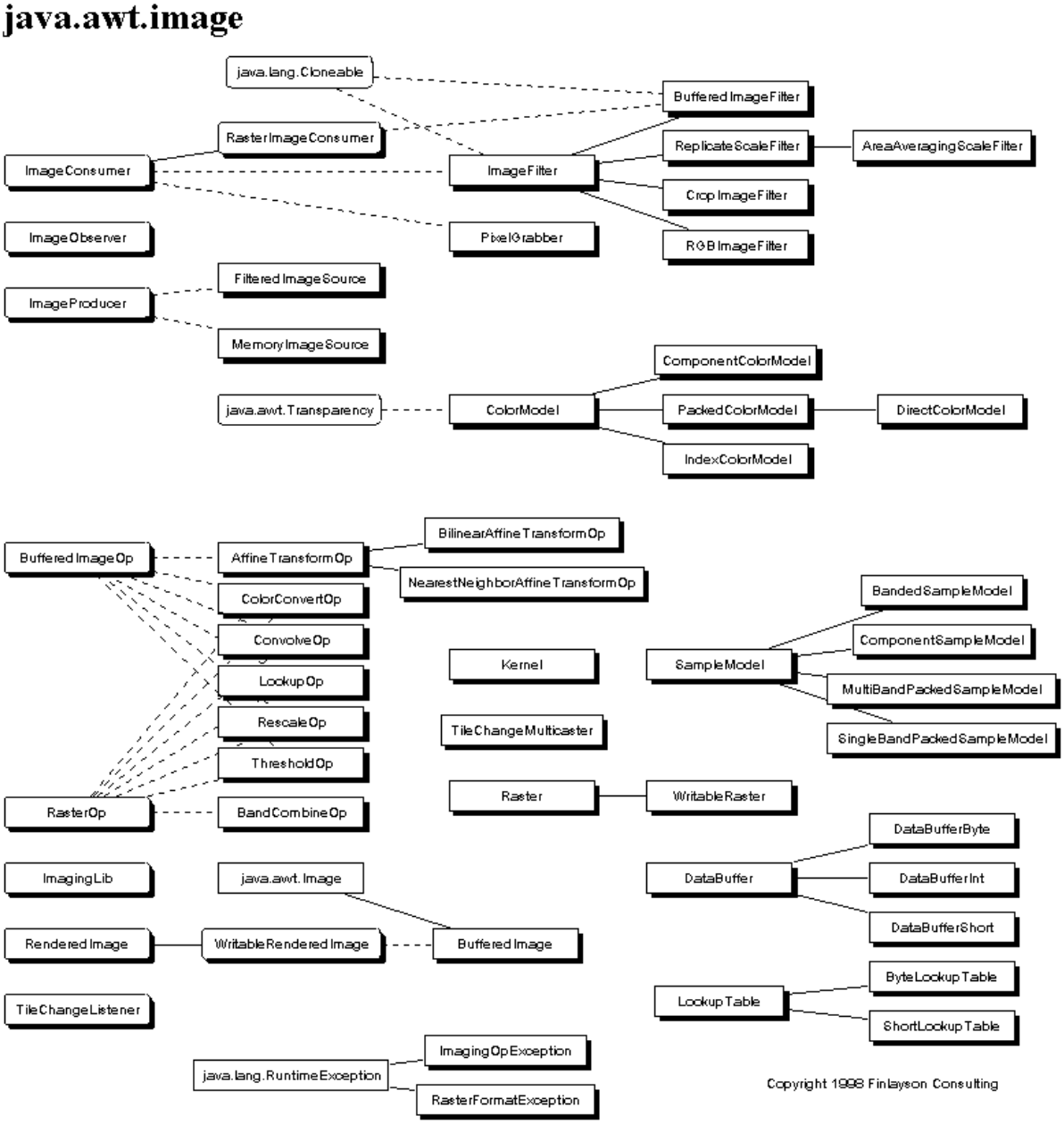



Figure 20 : *Hiérarchie d'héritage des images*

(source http://tiki-lounge.com/ raf/jfcmanual/jfc.4.html )

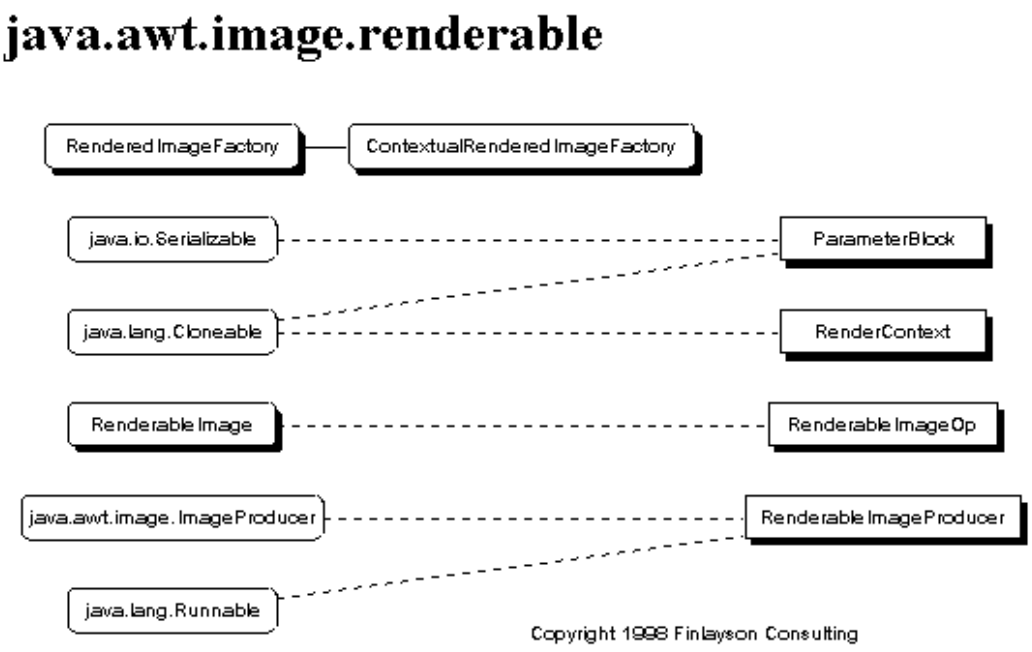



Figure 21 : *Hiérarchie d'héritage des images avec rendu*

(source http://tiki-lounge.com/ raf/jfcmanual/jfc.4.html )

### IHM

Nous reviendrons sur les IHM dans le chapitre 7.

## 2.8 Applets

Les applets du paquetage `java.applet` sont utilisées pour le code chargé dans des pages web.

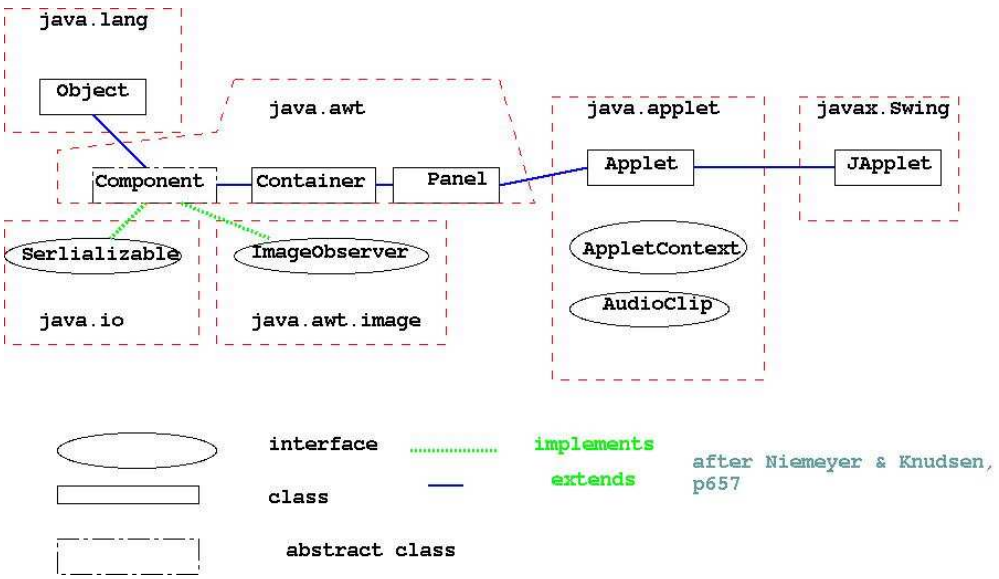



Figure 22 : *Hiérarchie d'héritage des applets*

(source http://www.cs.bath.ac.uk/ jjb/here/CM10135/CM10135_Lecture14_2005.html )

Nous y reviendrons dans la section 1 du chapitre 6.

## 2.9 Réflexion

La réflexion (ou introspection) permet d'ajouter à des Java quelques mécanismes issus des méta-objets et des méta-classes, naturelles dans Smalltalk. Elles se trouvent dans le paquetage `java.lang.reflect`.

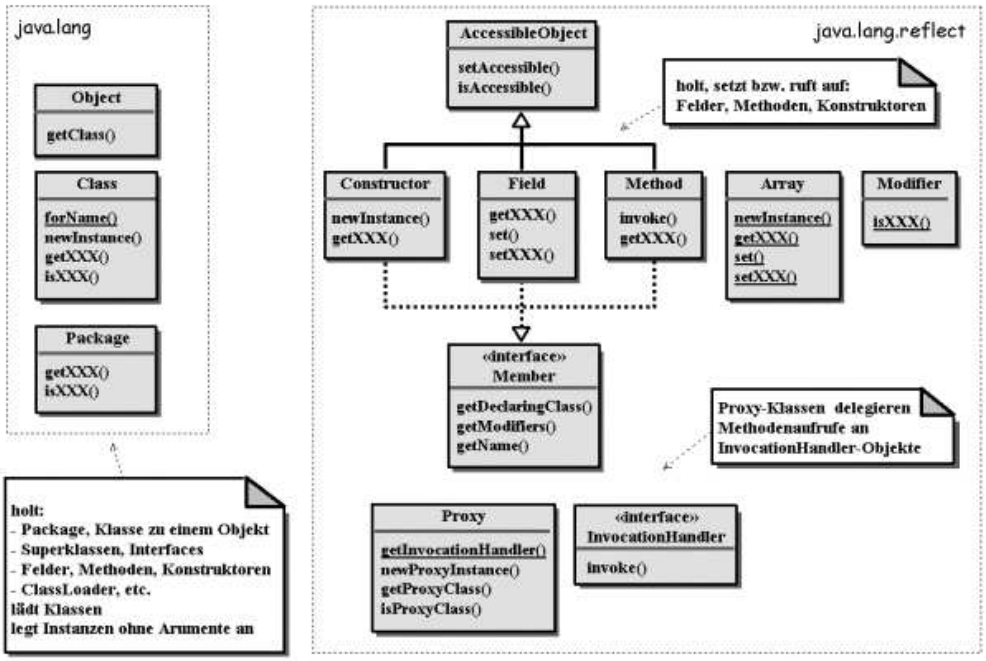



Figure 23 : *Hiérarchie d'héritage de la réflexion*

(source http://.galileocomputing.de/openbook/java2/kap_13.htm )

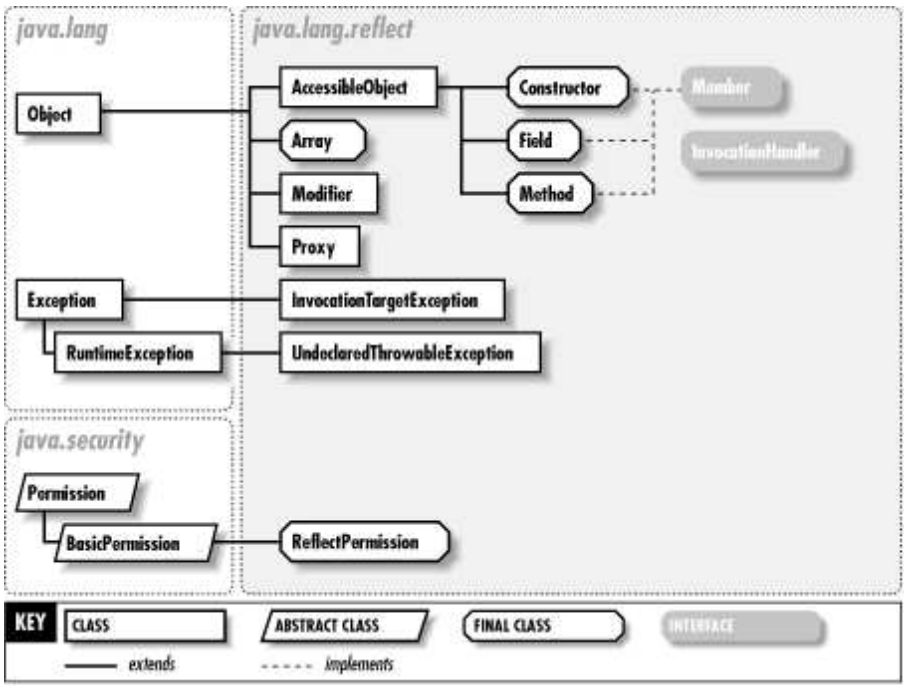



Figure 24 : *Hiérarchie d'héritage de la réflexion*

(source http://www.unix.org.ua/orelly/java-ent/jnut/ch14_01.htm )

Nous y reviendrons dans la section 4 du chapitre 5.

# Chapitre 4

# L'environnement de programmation Java

Dans ce chapitre nous survolons les éléments principaux de la mise en place de Java et du développement avec Java. Pour le développement nous utilisons l'environnement de développement intégré (IDE, *Integrated Development Environnement*) `Eclipse`.

## 1 Installation de java

### 1.1 Exécution de programmes Java

Pour exécuter un programme Java (Applet ou application standard), on doit disposer de la machine virtuelle, qui comprend l'interpréteur Java. C'est ce qu'on appelle l'environnement d'exécution Java (JRE, *Java Runtime Environment*). On procède en trois étapes : téléchargement, installation et mise-à-jour de l'environnement (variables et chemins notamment). Nous ne détaillons pas ces éléments car les sources internet (à jour) sont nombreuses et que les conditions varient d'un système à l'autre et d'une configuration (environnement existant) à l'autre.

- Windows :
  http://www.java.com/fr/download/help/win_offline.xml
- Unix :
  http://www.java.com/fr/download/help/linux_install.xml

Une **variable d'environnement** importante à positionner est `CLASSPATH` qui indique à la machine virtuelle et aux outils Java où trouver les librairies de classes et les librairies utilisateur.

Voici un exemple d'un programme Hello world typique écrit en Java :

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

Le fichier source doit s'appeler `HelloWorld.java`. Vous pouvez le tester avec les commandes suivantes (sous Linux) :

```
javac HelloWorld.java
CLASSPATH=. java HelloWorld
```

Une fois la machine installée, on peut lancer des programmes Java ou exédcuter des applets viaz des pages web.

## 1.2 Programmer en Java

Pour programmer en langage Java, nous devons disposer d'au moins un éditeur, un compilateur, un interpréteur et une documentation. Le paragraphe précédent indiquait comment installer les éléments de l'interpréteur. Dans cette section, nous indiquons comment installer les bases de l'environnement de programmation, à savoir l'environnement JDK (*Java Development toolKit*) ou J2SE Development Kit. JDK inclut un compilateur, un interpréteur, ainsi qu'un environnement de développement comprenant des outils et des utilitaires, tels que (`http://www.labo-sun.com/`) :

- `javac` : Java Compiler, le compilateur Java ; il compile un fichier source .java en un fichier exécutable `.class` contenant du bytecode.
- `java` : Exécute le ou les fichiers compilés par javac. C'est l'interpréteur Java, c'est-à-dire une implémentation de la machine virtuelle Java (JVM).
- `javadoc` : C'est un utilitaire très puissant qui permet de construire, à partir des commentaires insérés dans des sources Java, des fichiers HTML à propos des classes, méthodes, données membres...définies dans ces sources.
- `apt` : Annotation processing tool, nouveauté de Java 2 Standard Edition 5.0 (1.5 ń Tiger ż), permet l'exploitation des annotations (à ne pas confondre avec les commentaires).
- `appletviewer` : Ce programme permet d'exécuter une Applet Java sans nécessiter d'utiliser un navigateur web. Il dispose d'une interface graphique.
- `jdb` : Débogueur Java qui permet de détecter les erreurs de programmation.
- `jar` : Permet la création et la gestion des fichiers JAR (Java Archive).

## 1.3 API J2SE 1.5

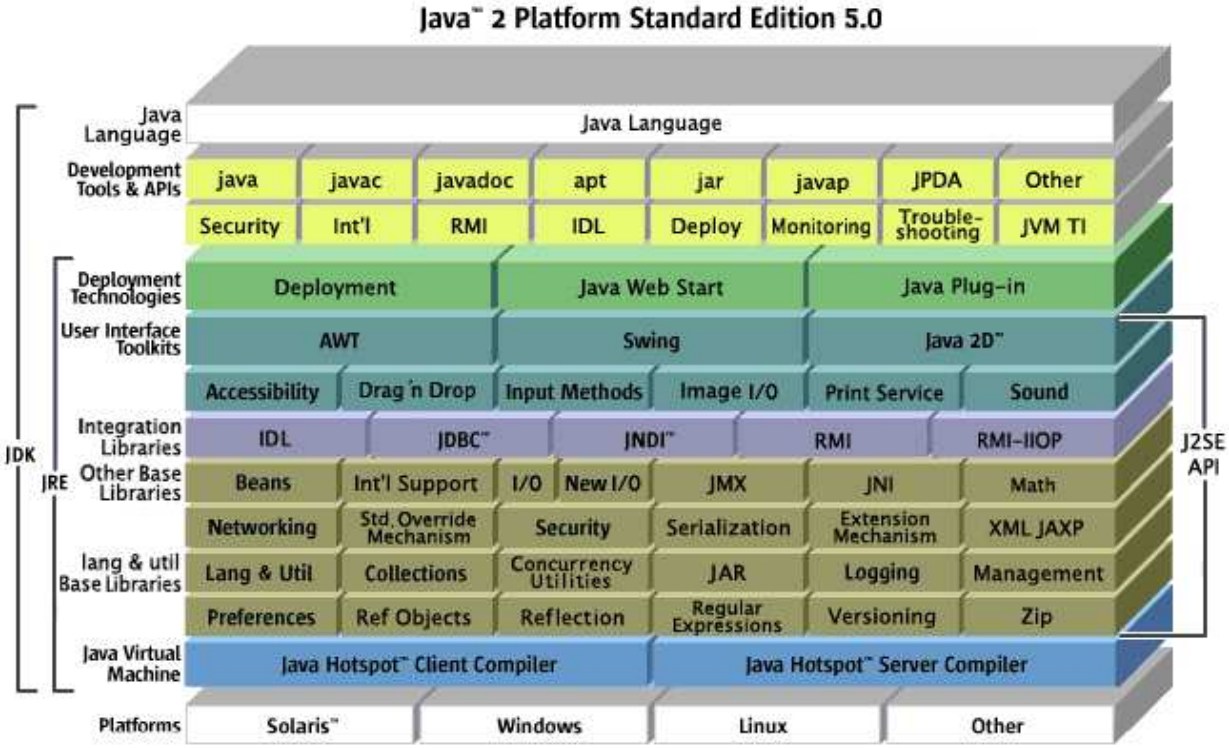



Figure 25 : *Java 2 Platform Standard Edition 5.0*

## 1.4 Installation

Ici aussi, on procède en trois étapes : téléchargement, installation et mise-à-jour de l'environnement (variables et chemins notamment). Nous ne détaillons pas ces éléments car les

sources internet (à jour) sont nombreuses et que les conditions varient d'un système à l'autre et d'une configuration (environnement existant) à l'autre.

- Windows 32 bits :
  http://java.sun.com/j2se/1.5.0/install-windows.html
- Linux 32 bits :
  http://java.sun.com/j2se/1.5.0/install-linux.html
- Autres :
  http://java.sun.com/j2se/1.5.0/install.html

Voir aussi http://wpetrus.developpez.com/java/jdkwin/
http://www.linux-france.org/prj/edu/archinet/DA/install-java/index.html

## 2 Environnement de développement intégré

Ce texte est inspiré de plusieurs sources et entre autres de :
http://fr.wikipedia.org/wiki/Java_(langage)

La programmation peut se faire pour des exemples simples avec le compilateur javac, mais pour avoir plus de confort il est préférable d'utiliser un environnement de développement intégré ou IDE, certains sont gratuits.

Un environnement de développement intégré (EDI ou IDE en anglais pour Integrated Development Environment) est un programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un débogueur. Bien que des EDI pour plusieurs langages existent, bien souvent (surtout dans les produits commerciaux) un EDI est dédié à un seul langage de programmation. On peut également trouver dans un EDI un système de gestion de versions et différents outils pour faciliter la création de l'interface graphique (GUI en anglais pour Graphical User Interface). Voici quelques environnements existants :

- BlueJ http://www.bluej.org
  BlueJ est un IDE Java spécifiquement conçu pour l'enseignement de Java. Il est donc simple et basé sur des techniques visuelles et interactives : environnement entièrement intégré, affichage graphique de structure de classe, édition graphique et textuelle, éditeur intégré, compilateur, machine virtuelle, débogueur, etc... interface interactive (création d'objet, appels de méthodes, test...), développement incrémental d'applications.
- Eclipse http://www.eclipse.org/
  Eclipse est un outil gratuit issu de la communauté *open source* du même nom dont les projets visent à fournir des environnements de développement extensibles (architecture par *plugins*) et des *frameworks* d'application pour développer du logiciel.
  Eclipse n'est pas dédié à Java uniquement.
- Idea http://www.jetbrains.com/idea/
  Fournissant déjà de nombreux outils par défaut (refactoring, Ant, JUnit, CVS...), IntelliJ IDEA dispose également d'une communauté active fournissant nombre de plugins tiers. Il fréquemment cité comme éditeur le mieux conçu pour aider le développeurs, rendant les tâches rébarbatives rapides à concevoir...
- JBuilder http://www.borland.com/jbuilder/
  JBuilder est un environnement Borland (version professionnelle ou personnelle), inter plate-forme pour développer des applications Java dŠentreprise robustes. Grâce à ses fonctions avancées de collaboration dŠégal à égal (édition et débogage conjoints, différentiation active, etc.), JBuilder favorise le fonctionnement en « binôme virtuel »permettant aux développeurs de travailler ensemble. DŠautres outils de collaboration (refactoring distribué, visualisation du code UML, intégration aux outils de gestion du cycle de vie, etc.) permettent aux développeurs et aux autres participants de rester synchronisés, de l'initialisation au déploiement. Il inclue aussi des fonctionnalités de test unitaire et

d'analyse des performances, des assistants de conception EJB (Enterprise JavaBeans), JSF (JavaServer Faces), Struts et de services Web.

- JCreator http://www.jcreator.com
  Jcreator est une puissante IDE gratuite pour les programmes Java. Jcreator fournit à l'utilisateur une large variété de fonctionnalités comme : la gestion de projet, des modèles de projet, un éditeur avec une mise en valeur de la syntaxe, des assistants et une interface entièrement adaptable. Avec Jcreator vous pouvez directement compiler ou mettre en marche votre programme Java sans activer le document principal en premier. Jcreator trouvera automatiquement le fichier avec la fonction principale ou le fichier HTML comportant l'applet Java, ensuite il démarrera l'outil approprié. http://www.lewebdejamy.com/1/Developpement/Java/Jcreator-v-3-10.html
- jEdit http://www.jedit.org/
  jEdit est un éditeur de programme open source (GPL), extensible et configurable, écrit en Java, simple d'utilisation, et qui fonctionne sur différentes plate-formes.
- NetBeans http://www.netbeans.org/
  C'est l'outil par défaut de Sun Microsystems pour Java. Sun Microsystems a fondé le projet open source NetBeans en Juin 2000 et continue d'être le sponsor principal du projet. Aujourd'hui, deux projets existent : L'EDI NetBeans et la Plateforme NetBeans. L'EDI NetBeans est un environnement de développement - un outil pour les programmeurs pour écrire, compiler, déboguer et déployer des programmes. Il est écrit en Java - mais peut supporter n'importe quel langage de programmation. Il y a également un grand nombre de modules pour étendre l'EDI NetBeans. L'EDI NetBeans est un produit gratuit, sans aucune restriction quant à son usage. Également disponible, La Plateforme NetBeans ; une fondation modulable et extensible utilisée comme brique logicielle pour la création d'applications bureautiques.
- jDeveloper http://www.oracle.com/technology/products/jdev/
  Loin de se limiter aux outils Oracle (serveur d'application, base de données...), JDeveloper est un éditeur complet : modélisation UML, gestion d'équipe de développement (CVS, ClearCase...), éditeur XML puissant, support des services Web...

En annexe, dans la section 1 du chapitre A nous proposons quelques discussions autour des IDE Java, issues du Net. Pour la suite, nous utilisons l'IDE `Eclipse` parce qu'il est gratuit et que de nombreuses extensions ont été proposées : Python, les outils Ant et CVS, tests JUnit, les aspects, UML, Struts, Tomcat, JBoss...

## 3 Eclipse

En annexe, dans la section 2 du chapitre A nous proposons un *Tutorial Eclipse 3.1* par Serge Baccou et daté du 26-07-2005, accessible à http://www.baccoubonneville.com. Ce tutoriel propose un aperçu du projet Eclipse (promoteurs, historique, plugins, installation). Nous en reprenons quelques éléments pour alimenter cette section. Une autre source plus complète est bibliographique [Hol04, Dau04, Dja05].

Le principal objectif d'Eclipse est de fournir une platforme pour le développement d'applications. Voici quelques autres règles qui s'appliquent également au projet Eclipse : Eclipse doit être extensible, Eclipse doit tourner sur plusieurs systèmes d'exploitation, Eclipse doit être indépendant des langages de programmation (HTML, Java, C, JSP, XML...).

### 3.1 L'architecture d'Eclipse

Le projet Eclipse est composé de la plateforme Eclipse, les Java Development Tools (JDT) et le plugin Development Environment (PDE). En dehors du projet Eclipse, des éditeurs ou des individus peuvent offrir leurs propres plugins. Eclipse et les plugins Eclipse sont codés en

Java. La figure 26 illustre l'architecture d'Eclipse :

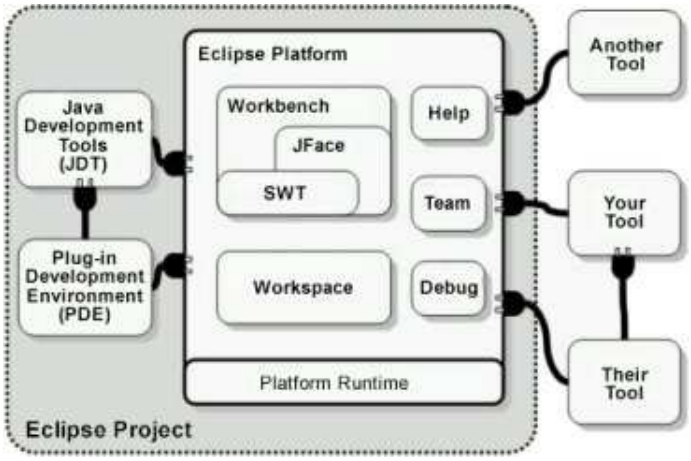



Figure 26 : *L'architecture d'Eclipse*

La plateforme d'Eclipse a elle-même plusieurs composants :

- Le *platform runtime* est le noyau d'Eclipse. Toutes les autres fonctionnalités de la plateforme Eclipse sont localisées dans des plugins. Au démarrage le *platform runtime* découvre tous les plugins disponibles.
- Le workspace consiste en un ou plusieurs projets utilisateur
- Le workbench est l'interface utilisateur d'Eclipse. Il est construit grâce à SWT et à JFace.
  - Le `Standard Widget Toolkit (SWT)` est une librarie de composants intégrée au système d'exploitation natif mais avec une API indépendante. La page d'accueil de SWT est située à l'adresse : `http://www.eclipse.org/swt/`.
  - `JFace` est une boîte à outil de type interface utilisateur implémenté en utilisant SWT et qui facilite les tâches de programmation qui concernent les interfaces utilisateur.

  On notera que JFace et SWT sont une alternative au framework Swing de Java/Sun.

Les éléments *Help*, *Team* et *Debug* sont des blocs de base pour fournir l'aide en ligne, les capacités de travail d'équipe et les capacités de déboguage au programmeur.

Le *Java Development Tools (JDT)* ajoute à la plateforme Eclipse des facilités pour le développement de programmes Java. Cela consiste en un éditeur de code source Java, un compilateur, un débogueur... La page d'accueil de JDT est située à l'adresse : `http://www.eclipse.org/jdt/`.

Le *Plugin Development Environment (PDE)* fournit des vues et des éditeurs supplémentaires pour faciliter la création de plugins Eclipse. Le page d'accueil de PDE est située à l'adresse : `http://www.eclipse.org/pde/`.

## 3.2 Installation d'Eclipse

Vous pouvez télécharger Eclipse 3.1 sur le site de la Fondation Eclipse Fundation à l'URL suivante : `http://www.eclipse.org/downloads`.

### MS Windows

Télécharger une archive ZIP appelée `eclipse-SDK-3.1-win32.zip` qui contient un répertoire `eclipse`. Le fait est que Eclipse ne comprends pas de programme d'installation car cela n'est pas nécessaire. Vous n'avez qu'à décompresser et désarchiver le fichier ZIP sous `C:\`.
**Attention** Vous ne devez pas désarchiver le fichier ZIP sous `C:\ECLIPSE` sinon vous allez obtenir un répertoire double `C:\ECLIPSE\ECLIPSE` ce qui n'est probablement pas le résultat escompté.

Maintenant vous pouvez créer un raccourci sur votre bureau : clic droit sur le bureau et choisissez `Nouveau > Raccourci`. La cible doit être `C:\ECLIPSE\eclipse.exe` et le nom doit être "ECLIPSE". Vous pouvez copier ce raccourci dans la barre de lancement rapide.

Cliquez sur le raccourci pour lancer Eclipse. Vous visualisez l'écran d'accueil suivant.

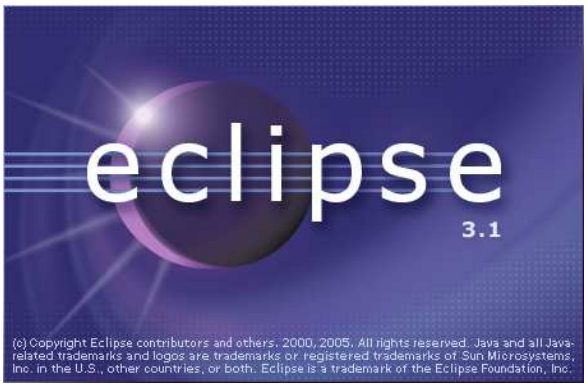



Figure 27 : *Ecran d'accueil d'Eclipse*

Après un instant, Eclipse vous demande un chemin pour votre espace de travail. L'espace de travail est le répertoire où Eclipse va stocker vos données utilisateurs (vos projets avec stockage de fichiers compilés notamment). Vous pouvez entrer `C:\mes documents\workspace` ou `C:\ECLIPSE\WORKSPACE`. Vous pouvez cocher la case "*Use this as the default and do not ask*". Plus tard, vous pouvez changer de workspace en utilisant le menu `File > Switch Workspace...`.

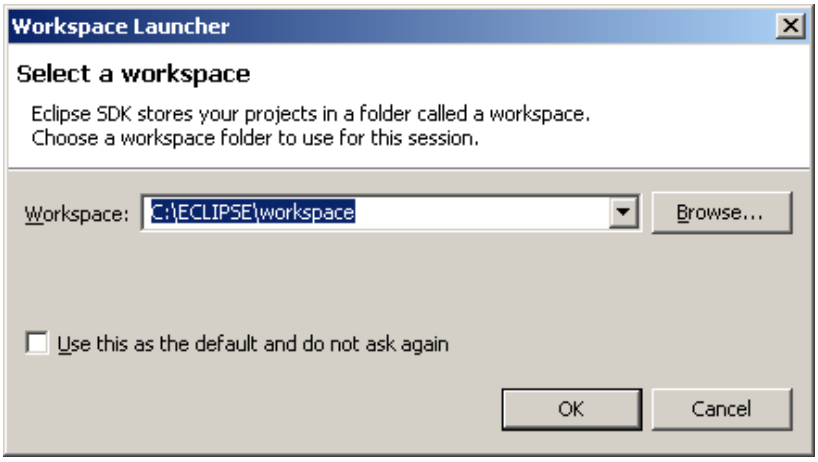



Figure 28 : *Choix du Workspace*

Une page d'accueil est affichée. La page d'accueil contient les rubriques suivantes :

–
– `Overview` : pour comprendre ce qu'est Eclipse : les bases du workbench, le support du travail d'équipe, le développement Java, le développement de plug-ins Eclipse,
– `Tutorials` : pour apprendre comment construire une application simple Java, une application indépendante Standard Widget Tool (SWT), un plug-in Eclipse plug-in ou une application Rich Client Platform (RCP),
– `Samples` : avec des exemples SWT, un éditeur multi-pages (comment créer un éditeur avec plusieurs pages), Property sheet and outline (comment utiliser les property sheets et les vues outline), Readme tool (comment créer ses propres points d'extension), éditeur Java (démontre les fonctionnalités standard disponibles pour personnaliser l'éditeur texte)

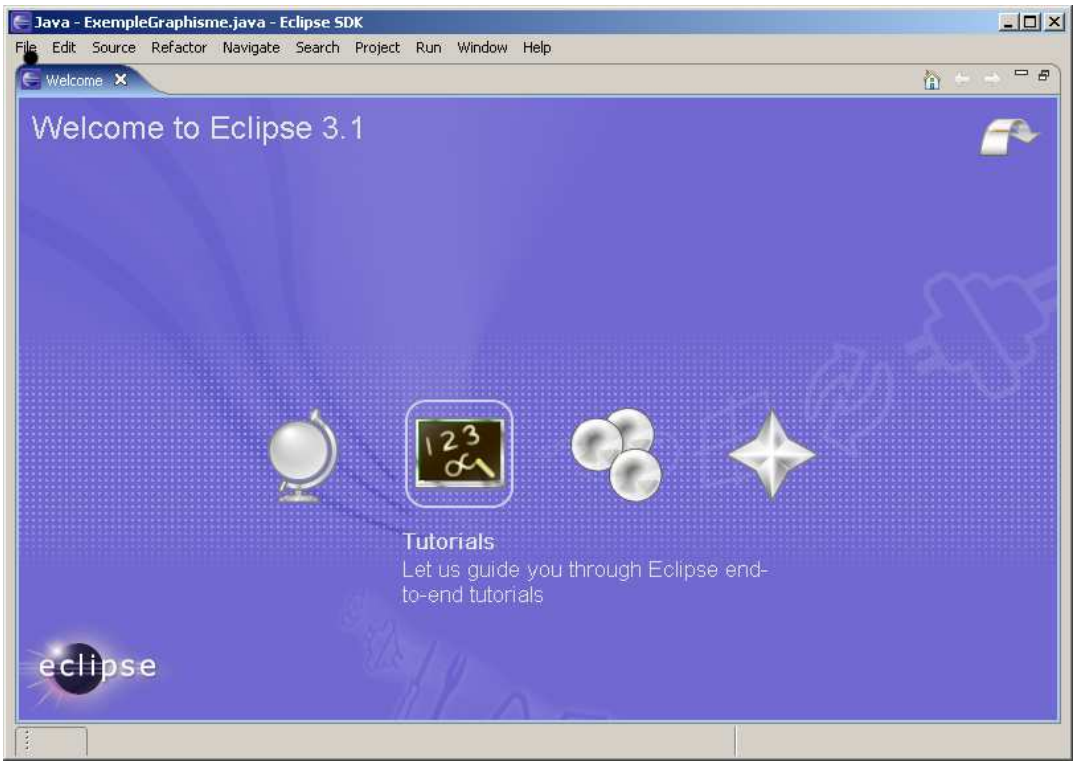



Figure 29 : *Page d'accueil d'Eclipse*

Vous pouvez fermer cette page d'accueil. Vous pouvez revoir cette page d'accueil avec le menu `Help > Welcome`. Une fois la page d'accueil fermée, vous pouvez voir l'écran du SDK Eclipse.

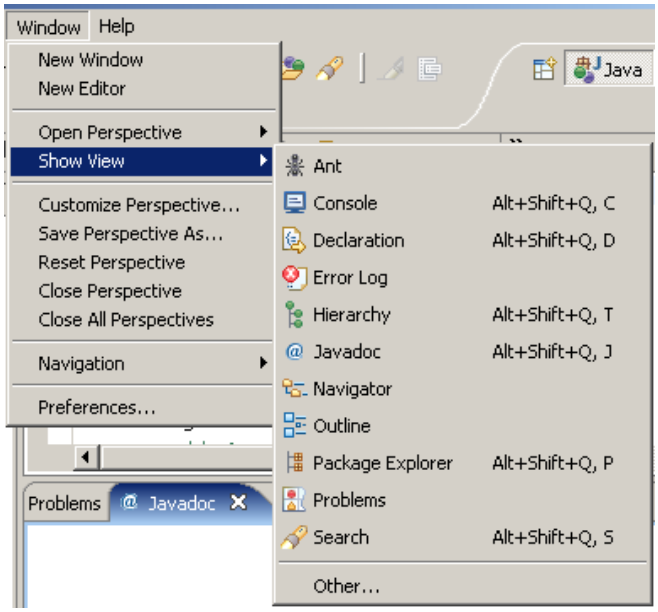



Figure 30 : *Eclipse SDK*

### Linux

Pour installer Eclipse sous Linux, consulter par exemple
http://perso.wanadoo.fr/jm.doudoux/java/dejae/chap002.htm.

## 3.3 Les principaux concepts d'Eclipse

Cette section résume les concepts utilisés par Eclipse : workspace, projet, workbench, perspective, editeur, vue and plugin. La figure 31 illustre la fenêtre principale d'Eclipse.

### Workspace

Un **espace de travail** (`workspace`) est un chemin où Eclipse va stocker des méta-données, vos projets, répertoires et fichiers. Un espace de travail peut contenir plusieurs projets. Vous pouvez avoir plusieurs espaces de travail mais vous ne pouvez activer qu'un seul espace de travail à la fois. Pour passer d'un espace de travail à un autre, on utilise le menu `File > Switch Workspace...`.

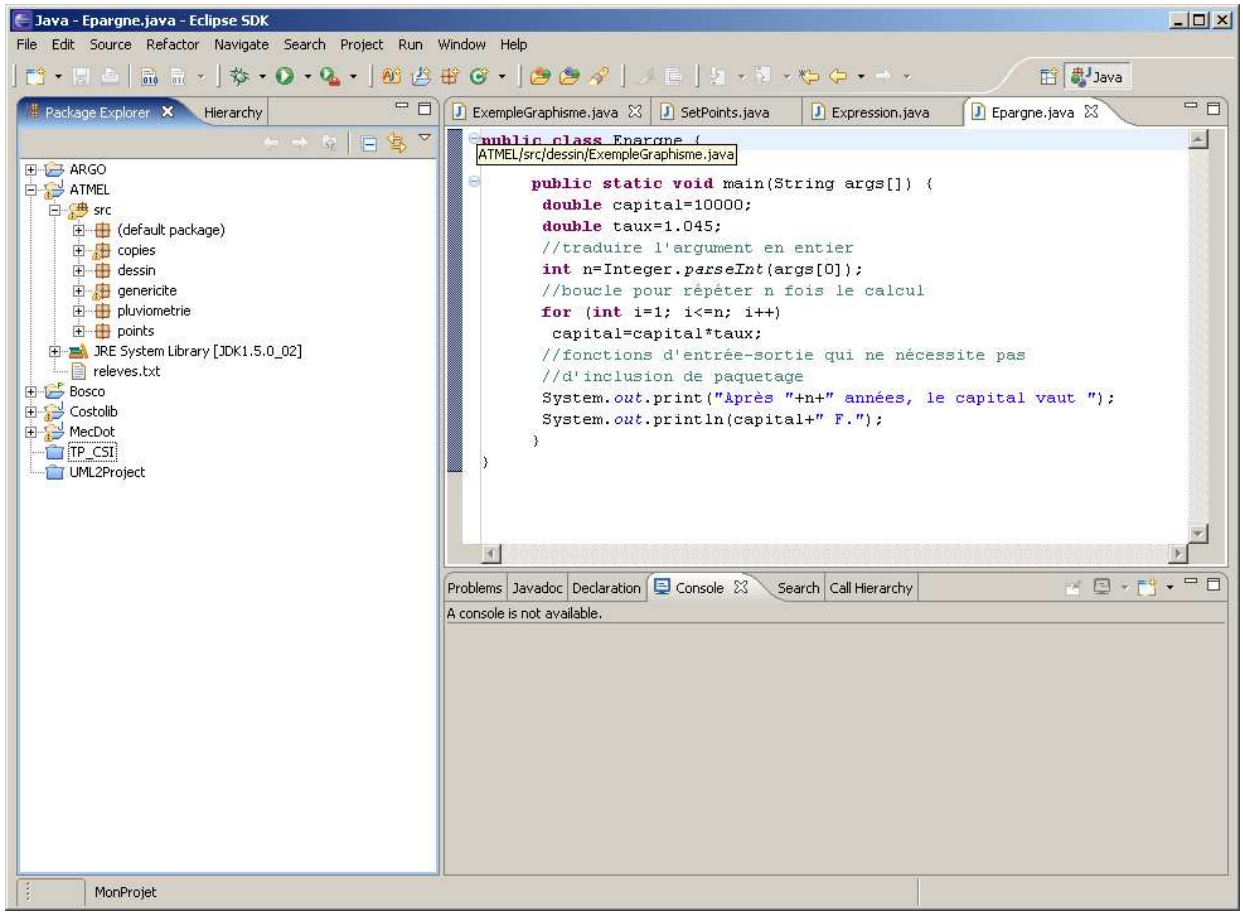



Figure 31 : *L'environnement Eclipse*

### Projet

Un **projet** (`project`) est une collection de répertoires et de fichiers. Pour créer un nouveau projet, vous pouvez utiliser le menu `File > New > Project > JavaProject > MonProjet`. Un projet Java inclue les bibliothèques de base, ce qui n'est pas le cas d'un projet simple. Les autres formes dépendent des plugins installés.

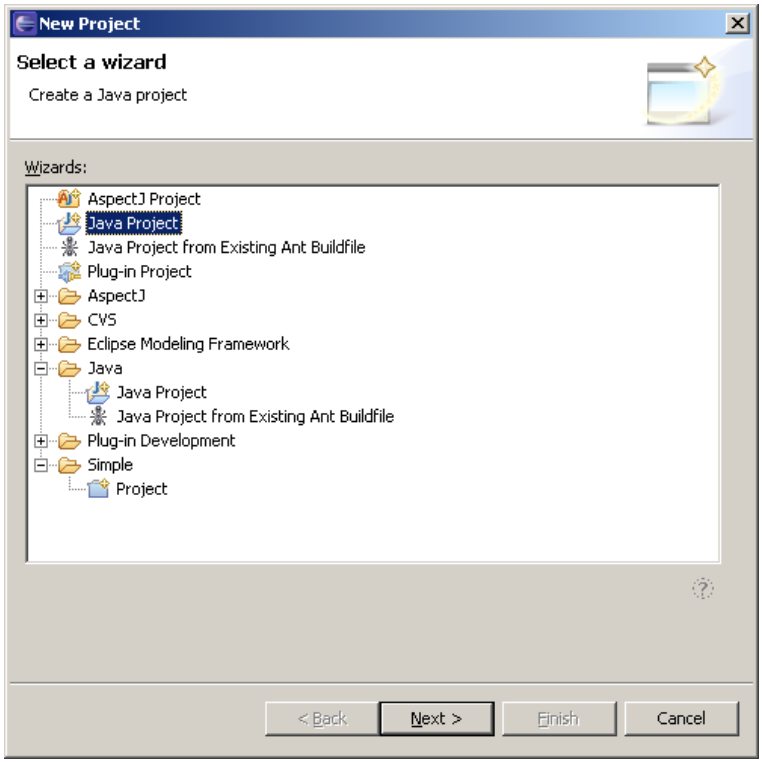



Figure 32 : *Créer un nouveau projet dans Eclipse*

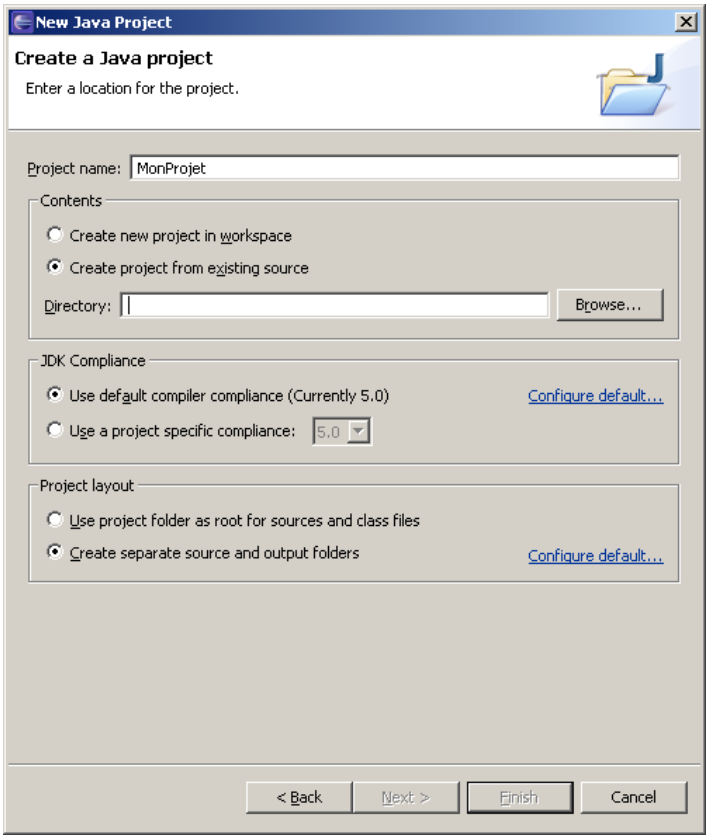



Figure 33 : *Créer un nouveau projet dans Eclipse, les sources*

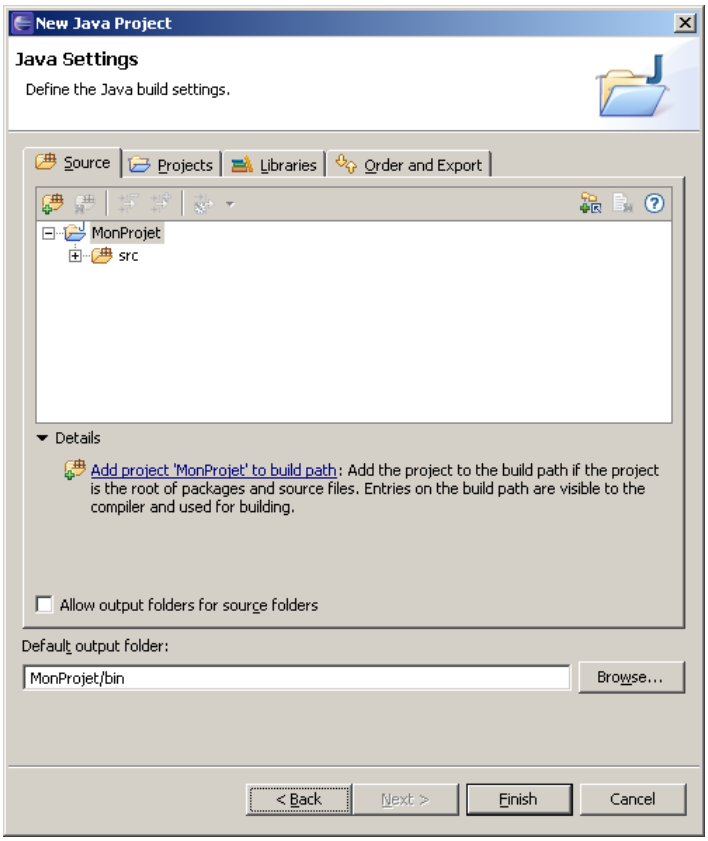



Figure 34 : *Créer un nouveau projet dans Eclipse, l'environnement*

Le projet créé est affiché dans la vue Navigateur. La vue Navigateur est affiché par le menu Eclipse `Window > Show View > Package Explorer`.

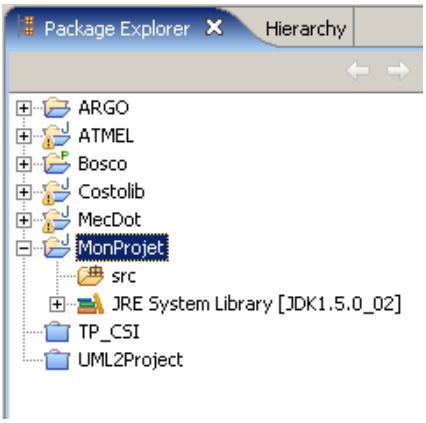



Figure 35 : *Créer un nouveau projet dans Eclipse, paquetage*

### Workbench

Le workbench est l'environnement de développement. Chaque fenêtre du workbench peut contenir une ou plusieurs perspectives. Plus d'une fenêtre du workbench peuvent exister sur le bureau d'Eclipse à un moment donné.

### Perspective

Une perspective est un agencement de plusieurs éditeurs et vues. Par défaut, vous pouvez ouvrir les perspectives suivantes un utilisant le menu `Window > Open Perspective > Others`.

- *CVS Repository Exploring* : pour le travail d'équipe avec Concurrent Version System (CVS), le fameux système de contrôle de versions open-source
- *Debug* : pour visualiser le débogueur d'Eclipse. Cette perspective est habituellement ouverte quand l'application est lancée en mode déboguage. Par défaut, le debogueur d'Eclipse est utilisé comme un débogueur Java mais certains plugins peuvent étendre le débogueur Eclipse pour supporter d'autres langages comme C/C++.
- *Java (Default)* : pour le développement Java
- *Java Browsing* : une autre perspective pour le développement Java qui montre les projets, les packages, les unités de compilation, les types, les fonctions membres et un éditeur dans une seule fenêtre
- *Java Type Hierarchy* : composé de la vue hiérarchique et d'un éditeur
- *Plugin Development* : pour développer de nouveaux plugins Eclipse
- *Resource* : pour visualiser les projets, les répertoires et les fichiers
- *Team Synchronizing* : une autre perspective pour le travail d'équipe avec CVS pour fusionner les changements dans le repository.

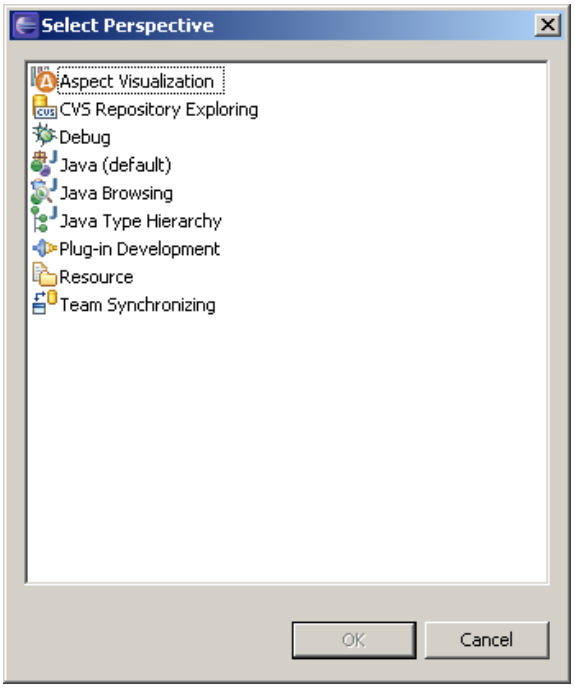



Figure 36 : *Ouvrir une perspective dans Eclipse*

### Editeur

Eclipse comprends par défaut des éditeurs pour certains types de contenus comme Java par exemple. Vous pouvez étendre Eclipse avec de nouveaux éditeurs pour supporter d'autres langages (comme C/C++ par example) en utilisant des plugins Eclipse.

### Vue

Une vue offre une présentation alternative des informations et en cela, une vue est complémentaire aux éditeurs. Vous pouvez ouvrir plusieurs vues en ouvrant une perspective mais vous pouvez également ouvrir une vue particulière grâce au menu "Window | Show view...".

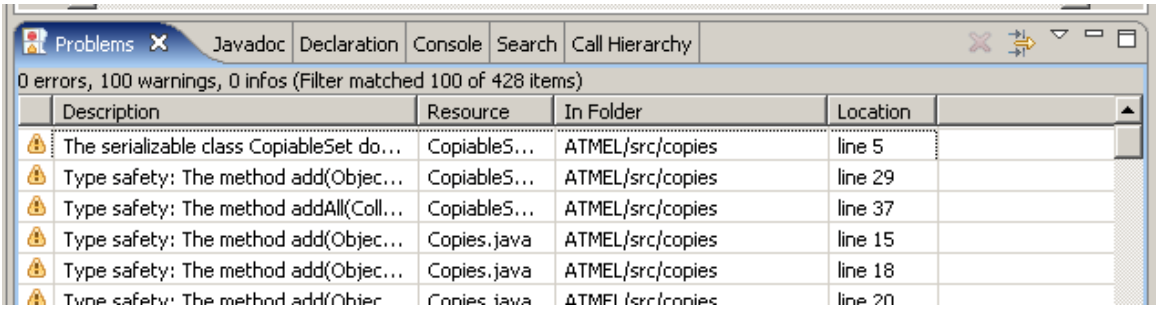



Figure 37 : *Les vues d'Eclipse*

Voici la description de quelques (pas toutes) vues d'Eclipse :

- `Ant` : montre les fichiers de construction (buildfile) Ant et rend simple l'éxécution d'un fichier de construction Ant ou une cible (target) particulière dans un fichier de construction
- `Console` : montre la sortie standard d'un programme et permet d'avoir une interaction clavier avec ce programme. La console peut afficher la sortie standard, la sortie erreur et l'entrée standard avec des couleurs différentes.
- `Declaration` : montre les déclarations Java
- `Error log` : capture tous les avertissements et erreurs internes envoyés par Eclipse et par les plugins Eclipse
- `Hierarchy` : permet de visualiser une hiérarchie complète de types, certains sous-types seulement ou les supertypes uniquement
- `Javadoc` : montre les information Javadoc information pour le code éditer dans l'éditeur Java avec des commentaires se conformant à la syntaxe Javadoc
- `Navigator` : provides a hierarchical view of the resources
- `Outline` : affiche une visualisation structurée de ce qui se trouve dans l'éditeur (dépends de l'éditeur)
- `Package Explorer` : montre la hiérarchie d'éléments Java des projets Java
- `Problems` : montre les erreurs, avertissements et informationq au suejt du code présent dans l'éditeur (dépends de l'éditeur)
- `Search` : montre les résultats d'une recherche (une recherche est lancée grâce au menu "Search | Search...")

### Plugins d'Eclipse

L'architecture d'Eclipse est extensible. Un plugin est un moyen d'étendre Eclipse et de fournir de nouvelles fonctionnalités, perspectives, éditeurs et vues. La section intitulée 'Quelques plugins Eclips" présente quelques uns des plugins les plus populaires et comment installer les plugins. Voici quelques sites Web hébergeant des plugins Eclipse.

- `http://www.eclipse.org/tools/`
  `http://www.eclipse.org/webtools/` avec `Eclipse Modeling Framework (EMF)` pour développer des outils basés sur le méta-modèle d'Eclipse `http://www.eclipse.org/emf/`
  et `Eclipse Graphical Editor Framework (GEF)` pour développer rapidement une IHM SWT autour d'une application existante (je ne l'ai pas testé) `http://www.eclipse.org/gef/`
- `http://eclipse-plugins.2y.net`
- `www.eclipseplugincentral.com`
- `http://sourceforge.net/`
- `http://www.eclipsetotale.com/`
- `http://subclipse.tigris.org/`
- `http://www.phpeclipse.net/`
- `http://eclipse.objectweb.org/`

– http://www.sysdeo.com/eclipse

## 3.4 Utilisation d'Eclipse

Présenter le développement Java avec Eclipse est un cours à part entière, nous ne présenterons que quelques aspects simples. Nous renvoyons donc le lecteur à des sources bibliographiques plus complète pour Eclipse 3 comme [Dau04] ou webographiques. A ce titre et bien que les informations ne soient pas mise à jour pour Eclipse 3.1, une bonne source d'information pour débuter avec Eclipse est
http://perso.wanadoo.fr/jm.doudoux/java/dejae/ disponible aussi à
http://jmdoudoux.developpez.com/java/eclipse/ Nous nous sommes principalement inspirés de la dernière source, en l'adaptant à Eclipse 3.1, pour rédiger les lignes qui suivent.

### Environnement de travail

La première chose à faire est de définir son environnement de travail c'est-à-dire l'organisation du plan de travail (workbench) et les bibliothèques nécessaire au développement à réaliser.

Au lancement d'Eclipse, une seule fenêtre s'ouvre contenant le plan de travail (Workbench). Le plan de travail est composé de perspectives dont plusieurs peuvent être ouvertes mais une seule est affichée en même temps. A l'ouverture, c'est la perspective "Ressource" qui est affichée par défaut. Une perspective contient des sous fenêtres qui peuvent contenir des vues (views) et des éditeurs (editors). Par exemple la figure 38 montre une configuration courante de développement Java.

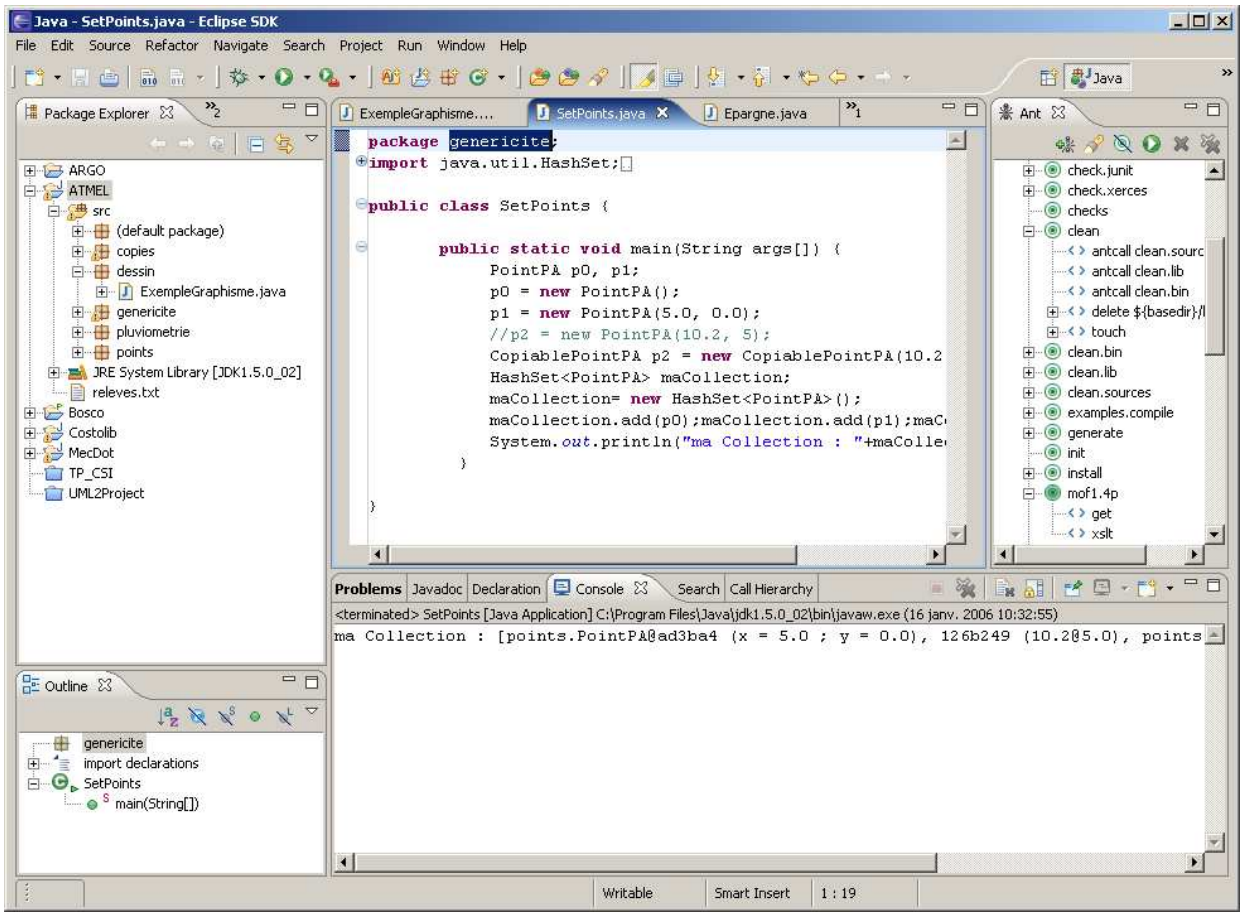



Figure 38 : *Un environnement Eclipse/Java*

La partie droite de la barre d'outils du plan de travail est composée d'un onglet qui contient une icône pour chaque perspective ouverte et une icône pour ouvrir une nouvelle perspective. L'icône enfoncée est celle de la perspective actuellement affichée. Le titre de la fenêtre du plan de travail contient le nom de la perspective courante. Eclipse possède dans le plan de travail une barre de menu et une barre de tâches. Elles sont toutes les deux dynamiques en fonction du type de la sous fenêtre active de la perspective courante. Eclipse propose de nombreux assistants pour faciliter la réalisation de certaines tâches.

Pour programmer en Java, nous ouvrons la perspective `Java (default)`. Les vues de cette perspective sont disponibles dans le menu `Windows`. Habituellement, on dispose

- d'une vue de navigation hiérarchique sur les projets de l'espace de travail `Package Explorer` (ou `Hierarchy` pour naviguer dans les hiérarchies d'héritage, ou `Navigator` pour naviguer dans les hiérarchies de fichiers).
- d'une vue d'édition (onglet avec les fichiers ouverts),
- d'une vue de focus (`Outline`) qui met en évidence des élements relatifs au contenu de la fenêtre d'édition,
- d'une vue utilitaire (console, messages d'erreurs, javadoc...).

On peut y ajouter une vue `Ant` pour lancer des scripts de tâches au format XML e.g. `build.xml`.

La mise en place de l'environnement comprend l'ajout des bibliothèques de travail sous forme de modules enfichables (**plugins**). Le mode d'installation peut varier d'un plugin à l'autre, il faut en général consulter la documentation du plugin. De manière générale, on utilisera le menu `Helps > Software updates`.

Les préférences (par défaut) sont définies dans le menu `Windows > Preferences...`. Les options varient avec les modules (plugins) installés (Javadoc, Ant, CVS, JUnit, AspectJ, UML...). Elles sont personnalisables par projet.

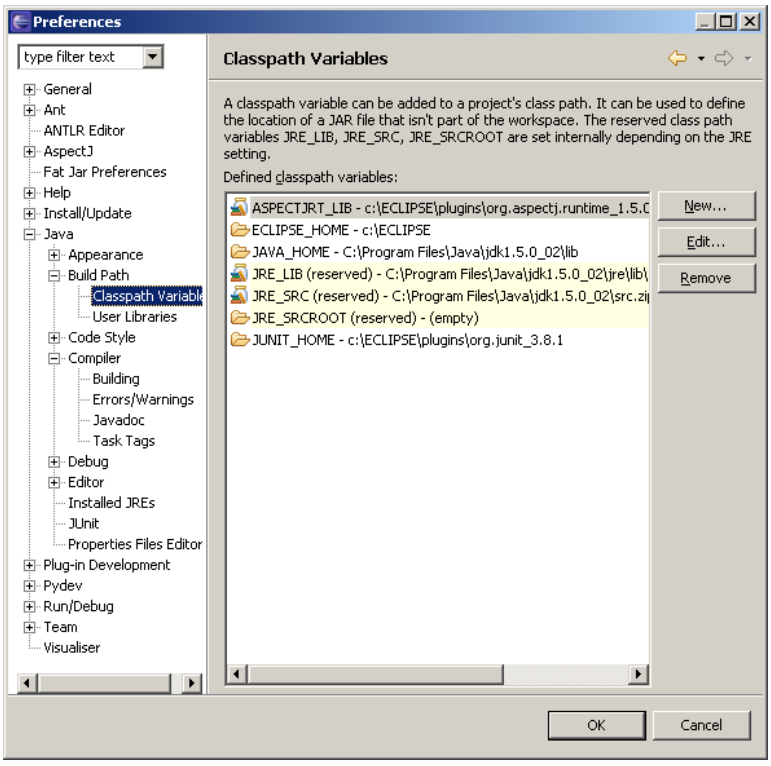
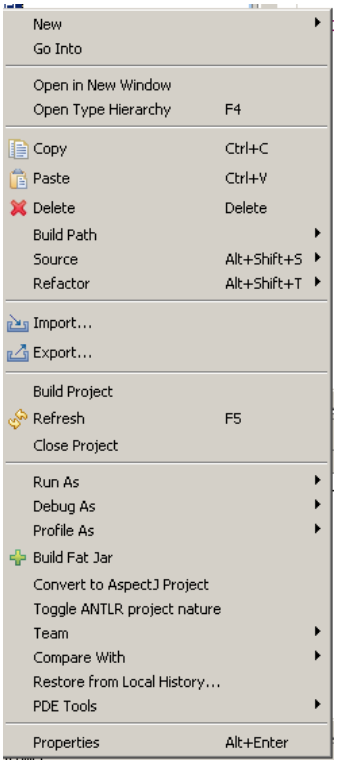



Figure 39 : *La personnalisation de l'environnement Eclipse*

### Projet

L'explorateur de paquetages met en évidence les projets de l'espace de travail. Les fichiers sources des projets ne se trouve pas forcément dans l'espace de travail par contre en général les fichiers compilés s'y trouvent. Les projets peuvent être ouverts ou fermés. Pour un projet Java, les fonctions activables par un menu dynamique ("pop-up") sont principalement : l'édition, la compilation (`build`), l'environnement (`build path`), la restructuration (`refactor`, `source`), l'importation/exportation de fichiers, et l'exécution. Ces fonctions dépendent des éléments sélectionnés. Elles se retrouvent dans différents menus de la fenêtre principale d'Eclipse.




**Créer des éléments** L'ajout d'éléments (projets, classes...) se fait par le menu `File > New` ou le menu dynamique `New` activé sur un projet ou un paquetage. L'éditeur comprend de nombreuses fonctionalités telles que la complétion, l'ajout de paquetages, de déclarations, la coloration, ...

**Compiler un projet** La compilation sous Eclipse, la commande `build` du menu dynamique `BuildProject` activé sur un projet ou par le menu `Project > BuildProject`. Elle peut être automatique ou à la demande selon l'option choisie dans le menu `Project > Build automatically`. Elle peut aussi être lancée par un script Ant.

La compilation dépend de l'environnement du projet fixé par le menu dynamique `BuildPath>` activé sur un projet ou par les menus `Project > Build...`.

**Exécuter un projet** L'exécution sous Eclipse se fait par les commandes du menu `Run` (accessibles aussi par les menus dynamiques). Il existe différents types d'exécution, qui dépendent du type des éléments à lancer (application Java, applet, TestJunit, application SWT...).

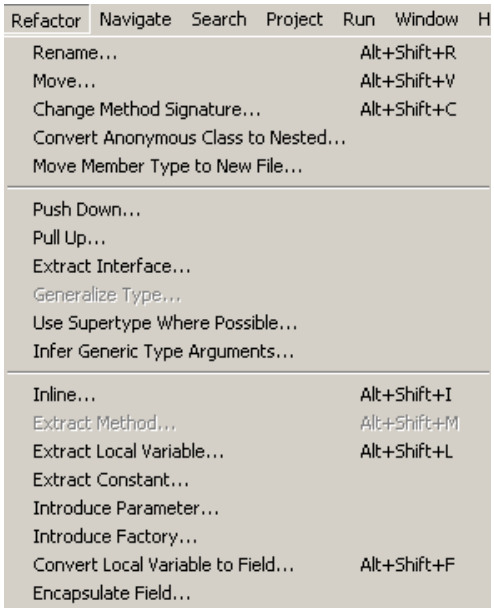
On peut exécuter n'importe quel programme, exécuter le programme dont la fonction `main` est dans l'éditeur, déboguer... Le deboguage se fait en plaçant des points d'arrêt dans le code. L'exécution pas-à-pas est accessible par les commandes du menu `Run`.

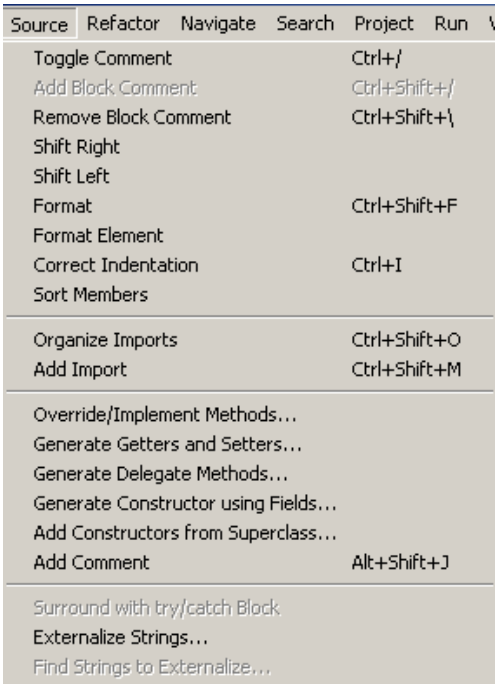
### Documenter

La documentation au format HTML est générée automatiquement par l'outil Javadoc. Il faut que le développeur place les balises dans sont code. Cela lui est facilité par Eclipse par le menu `Source > Add Comment...` si le plugin correspondant est installé. La génération de la documentation se fait via le menu `Project > Generate Javadoc...`.

### Restructurer

La restructuration est un apport fondamental des IDE pour l'écriture, la transformation et la documentation des programmes. Elle comprend un certain nombre de facilités pour générer ou modifier du code. Elle se traduit par deux catégories de fonctions (et aussi deux menus) :




- `Refactor` : modification profonde d'éléments, extractions, conversions...
  - `Rename` : permet de renommer un élément partout où il apparaît (fort utile pour les classes).
  - `Move` : permet de dépacer un élément et le modifier partout où il apparaît (fort utile pour les classes)...
- `Source` : modifie un fichier source par génération de code, formattage automatique, génération de commentaires et documentations, réorganisation du code...
  - `Generate Getters and Setters...` : crée (au choix) des méthodes d'accès en lecture et écriture pour les variables d'instance.
  - `Generates Constructor using Fields` : crée (au choix) des méthodes d'instanciation en fonction des variables d'instance.
  - `Add Constructors from Superclass` : crée (au choix) des méthodes d'instanciation par héritage.
  - `Add comment` : génération de commentaires et documentations JavaDoc...

### Navigation

La navigation permet de flâner (*browsing*) dans le code et la documentation. Eclipse permet entre autre de fouiller dans l'API Java dans passer par le traditionnel mode Web. Elle se traduit par deux catégories de fonctions (et aussi deux menus) :

– `Navigate` : recherche dans les différentes API chargées dans Eclipse, navigation dans les hiérarchies de classes et d'interfaces, de javadocs...
– `Search` : recherche de fichiers, de textes, d'implanteurs ou d'utilisateurs de méthodes.

### 3.5 Pour plus de détails

Consulter [Dau04] ou les documents fournis en dans la section 2 du chapitre A des annexes.

## 4 Exemple

Dans cette section, nous allons monter comment coder un programme en Java avec `Eclipse`. L'exemple support est le jeu de Nim.

### 4.1 Le jeu de Nim

Le jeu de Nim se joue entre deux joueurs et avec un tas d'allumettes. Les joueurs enlèvent alternativement 1, 2 ou 3 allumettes. Le perdant est celui qui épuise le tas.

### 4.2 Conception abstraite à objets

Le diagramme de classes de la figure 40 représente une conception abstraite du programme dans la notation uml [AV01b]. Une conception abstraite à objets plus formelle est présentée dans [AR98], qui explicite la spécification des opérations. Quatre classes composent l'application : `Personne`, `Personne`, `Joueur`, `Arbitre` et `JoueurIntelligent`.

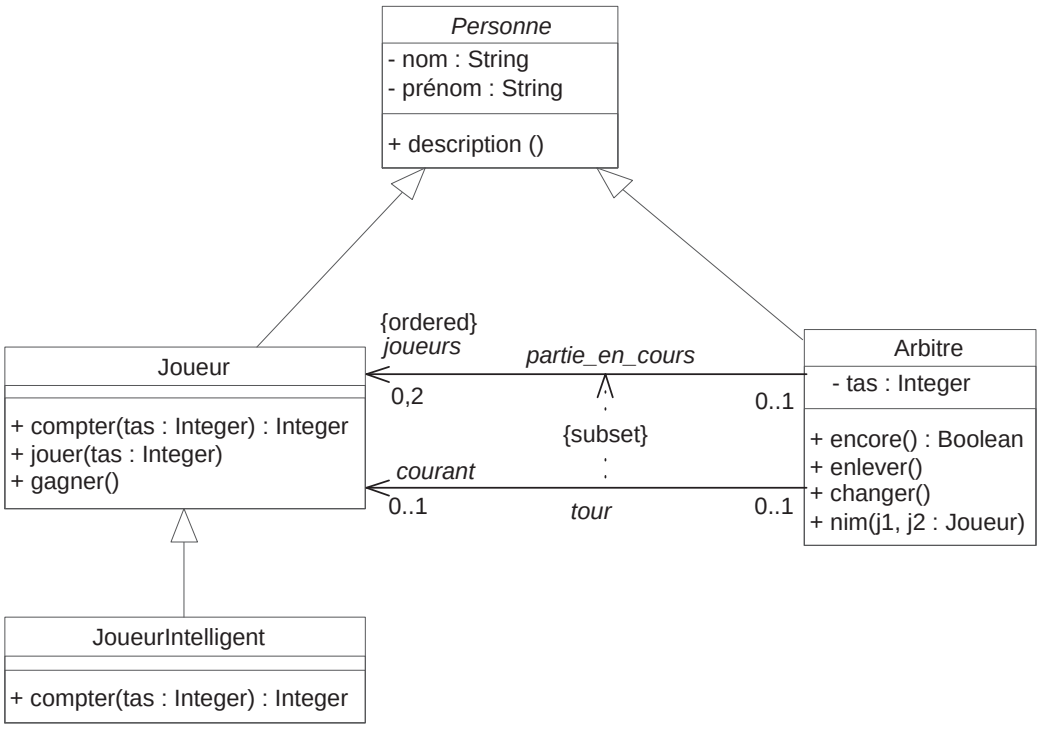



Figure 40 : *Environnement initial*

La classe abstraite `Personne` définit les participants au jeu. Les participants sont caractérisés par deux attributs : le nom et le prénom. Nous avons défini une opération `description()` qui rend une description textuelle de la personne. On distingue les deux joueurs de l'arbitre en utilisant la relation de spécialisation.

La classe `Joueur` définit les opérations (les responsabilités) suivantes :

- `compter` : le joueur détermine le nombre d'allumettes à ôter,
- `jouer` : le joueur joue un coup,
- `gagner` : le joueur se proclame gagnant.

La sous-classe `JoueurIntelligent` décrit des joueurs qui savent "mieux compter". Au lieu de tirer au hasard, le joueur choisi en fonction du nombre restant. L'opération `compter` est redéfinie.

L'arbitre a la maîtrise du jeu, c'est lui qui détient le tas d'allumettes et organise les actions des joueurs. Il définit les opérations (les responsabilités) suivantes :

- `encore` : indique si la partie n'est pas finie,
- `enlever` : modifie le tas d'allumettes après qu'un joueur ait joué un coup,
- `changer` : l'arbitre passe la main à un autre joueur,
- `nim` : l'arbitre démarre une nouvelle partie. Si une partie était en cours, elle est annulée.

La partie n'est possible que lorsque les joueurs sont mis en relation avec l'arbitre. Les deux associations `partie_en_cours` et `tour` symbolisent les liens entre joueurs et l'arbitre. L'association `partie_en_cours` indique les deux joueurs de la partie en cours (ou aucun s'il n'y a pas de partie en cours). La contrainte `{ordered}` signifie qu'ils sont distingués selon un ordre donné. L'association `tour` indique le joueur qui doit jouer un coup. La contrainte `{subset}` met en évidence le fait que le joueur joue un coup dans la partie en cours. Autrement dit, le joueur qui a la main est un des participants du jeu. La navigation est unidirectionnelle pour les deux associations, cela signifie que les joueurs n'ont pas connaissance de l'arbitre ou encore que les communications se font dans le sens arbitre vers joueurs. Ce choix dans la navigation a une conséquence en conception détaillée (ou concrète ). Il n'y a pas non plus de communication directe entre deux joueurs.

Noter qu'on respecte le principe d'encapsulation : les attributs sont privés et les opérations sont publiques. Ainsi seul l'arbitre peut modifier le tas d'allumettes (par l'opération `enlever`), les noms et prénoms ne peuvent être changés car il n'y a pas d'opération associées. Les opérations d'accès en lecture et de création d'instance n'ont pas été explicitement modélisées. Les types des attributs sont ceux du langage Object Constraint Language (OCL), qui fait partie de la notation UML.

## 4.3 Conception détaillée

Ce que nous appelons conception détaillée (ou concrète) est la mise en place de la conception dans l'environnement de développement cible. Il ne s'agit pas encore de coder le programme mais de faire l'adéquation entre les concepts de la conception abstraite et ceux de la conception détaillée. En principe, les concepts d'objets et de classes se retrouvent dans l'environnement cible.

### Principes généraux

En supposant, un seul langage de programmation cible[1], voici les grandes lignes d'une telle transition :

- `Vocabulaire` : on adapte le vocabulaire à celui du langage cible. Par exemple, un attribut (resp. une opération) d'instance sera appelée variable (resp. méthode) d'instance en Java ou donnée (resp. fonction) membre en C++.
- `Typage` : on adapte les règles de typage et les conventions associées à celles du langage cible. Par exemple, en Java, le typage est statique alors qu'en Smalltalk il est dynamique. Le polymorphisme des méthodes est implicite en Java alors que celui des fonctions C++ est explicite par le qualificatif `virtual`.
- `Association` : ce point est détaillé dans la section suivante.

1. Avec CORBA, des objets issus de différents langages peuvent coopérer.

- `Héritage multiple` :
  - Si le langage cible n'autorise pas l'héritage multiple (Java par exemple), il faut mettre en œuvre une politique de traduction adpatée. Par exemple, on choisit un chemin d'héritage principal et on duplique les caractéristiques issues des chemins secondaires. La duplication en Java se fait en déclarant de nouvelles interfaces.
  - Si le langage cible n'autorise pas l'héritage multiple, il faut adapter la stratégie de gestion des conflits à celle du langage cible (renommage et redéfinition en Eiffel, tri topologique en CLOS, implantation multiple d'interfaces en Java).
- `Contrôle des objets` : La plupart des langages à objets manipulent des objets séquentiels passifs. Lorsque la conception définit des objets actifs, il faut mettre en place un environnement d'exécution distribué simulé (processus en Java) ou intégré (*threads* Java). Ce point, qui inclue les variantes d'envois de messages et d'événements, constitue une étape essentielle de la conception pour les systèmes temps-réels.
- `Méta-objets` : certains langages autorisent des protocoles pour méta-objets (Java), pour les autres, il faut simuler en fonction des possiblités du langage (routines en Eiffel).
- `Assertions` : si le langage autorise les assertions (Eiffel par exemple), la traduction de contraintes OCL est simplifiée, sinon il faut programmer explicitement les contraintes.
- `Réutilisation` : une partie de la conception est réécrite en fonction des éléments qui existent dans l'environnement cible. C'est le cas par exemple des collections OCL.

### Conception des associations

En programmation à objets, une association s'implante habituellement avec des pointeurs. On peut s'inspirer des règles de transformation du schéma E-A-P dans le modèle relationnel. Examinons les alternatives de modélisation.

1. L'association ne possède pas de propriétés. Les liens sont représentés par des attributs de navigation, un par classes de la relation. Par exemple, *arbitre.courant* donne le joueur qui a la main. Ces attributs prennent en général les noms des rôles. Il y a quelques cas particuliers :
   (a) Navigation unidirectionnelle : un seul attribut de lien, dans la classe origine de la navigation. C'est le cas des deux associations ici.
   (b) Cardinalité maximale de 1 dans un sens : on peut choisir une navigation unidirectionnelle.
2. L'association possède des propriétés et
   (a) une cardinalité égale à 1. Les propriétés migrent dans la classe correspondant à la cardinalité. Après migration, on se retrouve dans le cas 1.
   (b) aucune cardinalité maximale de 1. L'association donne lieu à une nouvelle classe. Elle est reliée aux classes sous-jacentes par des associations binaires sans propriétés (cas 1).
   (c) une cardinalité dans 0..1. Les deux cas précédents sont applicables. Si les propriétés migrent dans la classe, il se pose le problème des valeurs partiellement définies.

### Application à l'exemple

Les attributs sont représentés par des variables d'instances. Le type des attributs est noté dans le commentaire de la classe. Les types `Integer`, `String` et `Boolean` existent tels quels en Java, mais on peut utiliser `int`, `StringBuffer` et `boolean` selon le besoin ou l'habitude. L'héritage ne pose pas de problèmes dans cet exemple puisqu'il n'y a pas de cas d'héritage multiple.

Dans la classe `Personne`, on définit deux variables d'instance : `nom` et `prenom`. Le commentaire de la classe indique que leur type est `String`. Il n'y a pas de définition de variables d'instance dans les classes `Joueur` et `JoueurIntelligent` car les variables sont implicitement héritées en Java.

Dans la classe `Arbitre`, la situation est plus compliquée pour les raisons suivantes :

1. Le tas d'allumettes est représenté par un entier qui donne le nombre d'allumettes dans le tas. C'est une variable d'instance « conditionnelle ». Le nombre d'alumettes est fixé aléatoirement au départ du jeu et non à la création de l'arbitre. Le nombre initial d'allumettes varie entre `10` et `NbMaxAllu` où la constante `NbMaxAllu` sera représenté par une variable de classe.
2. Les associations sont représentées sous forme de variables d'instances. Le nom de la variable est le rôle associé s'il existe. Il faut modéliser les contraintes portant sur les associations.
   (a) La contrainte d'ordre se modélise en choisissant une collection ordonnée pour ranger les joueurs. La taille de cette collection est au maximum de deux, plus précisément elle contient deux ou aucun objet de la classe `Joueur`. **Toute mise à jour de cette variable devra vérifier cette condition**.
   (b) Le joueur qui a la main est représenté par une variable d'instance. En principe, nous devrions utiliser un objet de la classe `Joueur`. Attention, ici aussi, il peut ne pas y avoir de joueur courant (cardinalité 0..1).

   Ces deux variables sont aussi une variable d'instance "conditionnelle" dont l'existence est conditionnée au fait qu'une partie est en cours.

   Pour prendre en compte les variables conditionnelles, nous ajoutons une nouvelle variable d'instance booléenne `partieEnCours` dont la valeur conditionnera les autres variables d'instance. Ainsi tout accès en lecture ou écriture des variables conditionnées implique de vérifier leur existence. En cas d'échec, une erreur est levée.

Cette traduction "systématique" peut être améliorée. En effet, nous n'avons pas représenté la contrainte d'inclusion du joueur courant dans les joueurs de la partie en cours. Nous n'avons pas non plus représenté le mécanisme de changement de joueur. La structure la plus pratique en termes de manipulation est en fait le tableau. Le joueur courant sera représenté par un indice dans le tableau à deux dimensions. Sachant que les tableaux ont des indices de 1 à n en Java, le changement de joueur s'écrit arithmétiquement pas `courant := ((courant mod 2)+1`.

Les opérations sont traduites en méthodes. Le nommage des méthodes en Java est similaire à celui des opérations en UML. Par exemple, la méthode associée à l'opération `compter(tas : Integer) : Integer` est `public int compter (int tas)`. Les méthodes d'instances d'accès en lecture sont générées de manière automatique avec Eclipse.

La méthode de description par défaut des objets Java est `toString()`. Nous remplaçons donc la méthode `description()` de la classe `Personne` par cette méthode.

La méthode d'instance `compter()` de la classe `Joueur` est redéfinie avec le même type dans la classe `JoueurIntelligent` et masque de ce fait la définition héritée de la classe `Joueur`.

Les méthodes de classes, implicites dans la modélisation abstraite, sont définies, en tenant compte les variables d'instances des classes correspondantes et des contraintes du schéma. La classe `Personne` est abstraite et n'a donc pas de méthode d'instanciation. Dans la classe `Joueur`, la méthode d'instanciation sera le constructeur `Joueur (String nom, String prenom)`. Dans la classe `Arbitre`, la définition tient compte de la représentation choisie : il n'y a pas d'arguments car au départ il n'y a pas de partie en cours. Par contre, le tableau des joueurs est créé avec deux positions et la variable `partieEnCours` est initialisée à `false`. C'est la méthode d'instance `nim`, qui initialise les autres variables pour la partie qui démarre.

## 4.4 Codage

Cette section illustre étape par étape l'écriture du code dans l'environnement `Eclipse`.

### Création d'un projet et/ou paquetage

Ouvrir l'application Eclipse. Créer un nouveau projet `Nim` ou ouvrir un projet existant par le menu `File > New...` ou par un menu contextuel (ou dynamique) dans la vue des paquetages. Puis créer un nouveau paquetage `nim` dans un projet existant par le menu `File > New > Package` ou par un menu contextuel dans la vue des paquetages. Nous conseillons de ne pas utiliser le paquetage par défaut pour mieux maîtriser la structure du code.

Le paquetage joue le rôle de classement modulaire (la catégorie en Smalltalk).

### Création d'une classe

Créer une nouvelle classe `Personne` dans le paquetage `nim` par le menu `File > New > Class` ou par un menu contextuel dans la vue des paquetages. Selon les options choisies, certaines déclarations sont générées automatiquement ou pas.

Tout ce travail peut être réalisé par un gestionnaire de fichier hôte et un éditeur de texte simple. L'importation se fait soit en déplaçant les répertoires sous l'arborescence d'un projet existant dans un répertoire `workspace` d'Eclipse soit importé sous Eclipse par le menu contextuel `Import...` sur un projet ou un paquetage. On obtient :

```
package nim;
public class Personne {
    public Personne() {
        super();
        // TODO Auto-generated constructor stub
    }
}
```

Insérer au clavier la définition des variables d'instance `nom` et `prenom`.

### Création d'une classe abstraite

Rajouter le mot-clé `abstract` pour indiquer que la classe `Personne` est abstraite. Modifier le contenu de son constructeur pour invalider l'instanciation.

```
public abstract class Personne {
    String nom, prenom;
}
```

### Création d'une méthode de classe

Les méthodes de classe indispensables sont au moins les constructeurs. En l'absence de constructeur (pour une classe normale en tout cas), un constructeur par défaut est défini. Comme la classe est abstraite, nous pourrions décider, dans un but pédagogique, d'invalider l'instanciation, en modifiant le constructeur de base.

```
public abstract class Personne {
    String nom, prenom;
    public Personne() {
        // méthode abstaite
    }
}
```

### Création des méthodes d'instance

Nous allons maintenant créer les méthodes d'accès en lecture et écriture des variables d'instance et la méthode de description.

Générer automatiquement les méthodes d'accès publique en lecture et protégée (pour les sous-classes) en écriture par le menu contextuel `Source > Generate Getters and Setters...`.

Pour la méthode de description, on redéfinit la méthode `toString`. Ajouter au clavier la méthode de transformation en texte `toString`. Pour aller plus vite, copier ce code depuis une autre classe.

### Classement des méthodes

Java ne dispose pas comme Smalltalk d'un outil de classement des méthodes (le protocole). Le développeur doit lui-même ordonner ses méthodes. Il peut s'appuyer sur des commentaires pour séparer les éléments de code et utiliser des noms de protocoles comme dans Smalltalk (voir section 7.4 du chapitre 2).

### Création des commentaires Javadoc

Générer automatiquement les commentaires pour la classe, les variables d'instance et les méthodes par le menu contextuel `Source > Add Comment` ou le raccourci clavier.

Indenter automatiquement le code par le menu contextuel `Source > Generate Format`.

Listing 4.1 – Code Java de la classe `Personne`

```
package nim;
/**
 * @author pascal andre
 * @date Janvier 2006
 * @version 1 Classe abstraite définissant des personnes pour le jeu de Nim.<br>
 *          Pas de constructeur par défaut pour une classe abstraite.
 */
public abstract class Personne {
    /**
     * Une personne est définie simplement par un nom et un prénom.
     */
    protected String nom, prenom;
    /**
     * @return nom de la personne : String
     */
    public String getNom() {
        return nom;
    }
    /**
     * @param nom
     *             de la personne : String
     */
    protected void setNom(String nom) {
        this.nom = nom;
    }
    /**
     * @return prénom de la personne : String
     */
    public String getPrenom() {
        return prenom;
    }
    /**
```

```
     * @param prénom
     *              de la personne : String
     */
    protected void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    /*
     * (non-Javadoc) méthode d'affichage par défaut
     *
     * @see java.lang.Object#toString()
     */
    public String toString() {
        // redéfinition de la méthode ((Object)this).toString()+
        return ("Je m'appelle " + nom + " " + prenom);
    }
}
```

### Sauvegarde des travaux

La sauvegarde se fait en enregistrant les fichiers java.

### Héritage par extension

Procéder de même pour écrire les autres classes de l'application (`Joueur`, `Arbitre` et `JoueurIntelligent`) selon le code suivant. Ces classes sont définies par héritage d'extension sur la classe `Personne` dans le même paquetage. En Java, une classe ne peut être créée que si sa superclasse existe. Ce n'est pas le cas sans IDE car les classes sont dans des fichiers séparés.

La classe `Joueur` ajoute uniquement de nouvelles méthodes à la classe `Personne`. La méthode `compter` calcule combien d'allumettes il doit enlever. Lorsque le nombre d'allumettes est supérieur à deux, il choisi aléatoirement. Deux possibilités se présentent en Java : définir une suite aléatoire avec la classe `java.util.Random` ou plus simplement demander un nombre aléatoire entre 0 et 1 à la classe `java.lang.Math`. Nous choisissons cette dernière car c'est une demande ponctuelle.

La classe `JoueurIntelligent` est créée après la classe `Joueur`. Les méthodes correspondant au jeu sont placées dans un protocole `jeu`. Pour l'arbitre, on considère que les variables d'instances sont privées (pas de méthodes d'accès). L'initialisation de la variable de classe `NbMaxAllu` (nombre maximal d'allumettes dans le tas) est réalisée par une valeur par défaut (et non une méthode de classe `initialize` comme en Smalltalk - on voit ici la différence avec un vrai protocole de réflexion.). Les méthodes annexes de la méthode `nim` sont définies en `protected` pour être accessibles dans d'éventuelles sous-classes.

### Evaluer un envoi de message

Pour évaluer du code, le plus simple est de définir une méthode `main` qui évalue des expressions. Il n'y a pas d'outil de type `Workspace` comme en Smalltalk pour évaluer n'importe quelle expression car l'environnement de travail est vide par défaut en Java.

Voici le code de ces différentes classes.

Listing 4.2 – Code Java de la classe `Joueur`

```
package nim;
/**
 * @author pascal andre
 * @date Janvier 2006
 * @version 1 Classe définissant des joueurs pour le jeu de Nim.<br>
```

```
 */
public class Joueur extends Personne {
    /**
     * Méthode d'instanciation : constructeur
     */
    public Joueur(String nom, String prenom) {
        super();
        this.nom = nom;
        this.prenom = prenom;
    }
    /**
     * Le joueur détermine combien d'allumettes il enlève du tas. Le joueur
     * décide aléatoirement ce nombre s'il y a au moins trois allumettes dans le
     * tas. Il retire de 1 à 3 allumettes.
     * @param tas :
     *              nombre d'allumettes du tas
     * @return nombre d'allumettes à enlever
     */
    public int compter(int tas) {
        int combien;
        // variable locale qui détermine le nombre d'allumettes à tirer
        if (tas == 1 || tas == 2) {
            combien = 1;
        } else {
            // on peut aussi utiliser le temps Time
            combien = (((int) (Math.random() * 1000)) % 3) + 1;
        }
        return combien;
    }

    public int jouer(int tas) {
        int combien = this.compter(tas);
        // variable locale qui détermine le nombre d'allumettes à tirer
        System.out.print("Il y a " + tas + " allumette(s)");
        System.out.println("   moi, " + this.toString() + " j'enlève "
                + combien + " allumette(s)");
        return combien;
    }

    /**
     * Le joueur se proclame gagnant.
     */
    public void gagner() {
        System.out.println("→ Moi, " + this.nom + " " + this.prenom
                + " j'ai gagné");
    }
}
```

Listing 4.3 – Code Java de la classe `Arbitre`

```
package nim;
import java.util.ArrayList;
/**
 * @author pascal andre
 * @date Janvier 2006
 * @version 1
 * Classe définissant l'arbitre du jeu de Nim.<br>
 * Certaines méthodes peuvent générer des erreurs.
```

```
 */
public class Arbitre extends Personne {
   /**
    * variable de classe définissant le nombre maximum d'allumettes
    * dans le tas
    */
   public static int NbMaxAllu = 30;
   /**
    * variables d'instance privées
    * les variables sont conditionnées par le fait qu'une partie est en cours.
    * tas : le nombre courant d'allumettes dans le tas
    *   compris entre 0 et NbMaxAllu
    * partie_en_cours : liste de joueurs
    *   de taille 2
    * courant : numéro du joueur courant
    *   compris entre 0 et 1
    * ces trois variables sont valides si une partie est en cours
    * nous le gérons par une variable booléenne
    * partieEnCours : vrai si une partie est en cours
    */
   private boolean partieEnCours;
   private int tas;
   private ArrayList<Joueur> partie_en_cours;
   private int courant;

   /**
    * @param nom
    * @param prenom
    * au départ, in n'y a pas de partie en cours
    */
   public Arbitre(String nom, String prenom) {
      super();
      this.nom = nom;
      this.prenom = prenom;
      partieEnCours = false;
      // ce qui suit n'a pas de pertinence
      // car il n'y a pas de partie en cours
      tas = 0;
      courant = 0;
      partie_en_cours = new ArrayList<Joueur>();
   }

   /**
    * @param j1 un joueur
    * @param j2 un autre joueur
    * Cette méthode démarre une partie de Nim.<br>
    * Le tas est défini aléatoirement.
    * Le jeu est une itération qui se termine quand il
    * n'y a plus d'allumettes.
    */
   public void nim(Joueur j1, Joueur j2) {
      partieEnCours = true;
      partie_en_cours = new ArrayList<Joueur>();
      partie_en_cours.add(j1);
      partie_en_cours.add(j2);
      tas = 10 + (((int) (Math.random() * 1000)) % (NbMaxAllu -9));
      courant = 0; // le choix pourrait être aléatoire
      System.out.println("La partie débute avec " + tas +
```

```
            " allumettes \n et " + partie_en_cours.toString());
        while (this.encore()) {
            this.enlever();
            this.changer();
        }
        partie_en_cours.get(courant).gagner();
    }

    // méthodes utilisées pour implanter le jeu
    /**
     * @return true si une partie se continue
     */
    protected boolean encore () {
        if (partieEnCours) return (tas > 0);
        else throw new Error("Pas de partie en cours");
    }
    /**
     * un joueur prend des allumettes
     */
    protected void enlever () {
        if (partieEnCours) {
            tas = tas - (partie_en_cours.get(courant).jouer(tas));
        }
        else throw new Error("Pas de partie en cours");
    }
    /**
     * on change de joueur
     */
    protected void changer () {
        if (partieEnCours) courant = (courant + 1) %2;
        else throw new Error("Pas de partie en cours");
    }

    // méthodes pour tester le jeu
    /**
     * @param args
     * Crée une partie avec un arbitre et deux joueurs.
     * Lance le jeu et affiche les résultats.
     */
    public static void main(String args[]) {
        /* programme principal */
        Arbitre arbitre;
        Joueur j1, j2, j3;
        arbitre = new Arbitre("Sémoi", "Lechef");
        j1 = new Joueur("Alain", "Térieur");
        j2 = new Joueur("Alex", "Térieur");
        j3 = new JoueurIntelligent("Alex", "LeFort");
        arbitre.nim(j1,j2);
        arbitre.nim(j1,j3);
    }
}
```

Listing 4.4 – Code Java de la classe `JoueurIntelligent`

```
package nim;
/**
 * @author pascal andre
 * @date Janvier 2006
```

```
 * @version 1
 * Classe définissant des joueurs intelligents pour le jeu de Nim.<br>
 * A la base ce sont des joueurs.
 */
public class JoueurIntelligent extends Joueur {
   /**
    * @param nom
    * @param prenom
    */
   public JoueurIntelligent (String nom, String prenom) {
      super(nom, prenom);
      // TODO Auto-generated constructor stub
   }
   /**
    * @param tas : nombre d'allumettes du tas
    * @return nombre d'allumettes à enlever
    * Le joueur intelligent détermine combien d'allumettes il enlève du tas.
    * Le joueur décide directement ce nombre quel que soit le nombre d'allumettes dans le tas.
    * Il retire de 1 à 3 allumettes.
    */
   public int compter (int tas) {
      int combien = tas % 4;
      // variable locale qui détermine le nombre d'allumettes à tirer
      switch (tas) {
         case 0:
              combien = 3;
               break;
         case 1:
         case 2:
             combien = 1;
              break;
         default:
             combien = 2;
              break;
      }
      return combien;
   }
}
```

### Compiler et exécuter des travaux

La compilation du code se fait en ligne par la commande `javac`. Sous Eclipse, on utilise un menu `Project>Build...` ou un script dans la fenêtre `Ant`. La compilation produit des fichiers `.java` archivable par l'utilitaire `jar` pour en faire des bibliothèques utilisables (sans les sources) dans d'autres programmes.

Pour exécuter le code, le plus simple est d'activer le menu `Run As > Java Application` ou le menu contextuel `Run As > Java Application` dans la fenêtre d'édition.

L'évaluation du message `arbitre.nim(j1,j2)` du programme `Arbitre.main()` affiche le résultat suivant dans la fenêtre Eclipse de la console :

```
La partie débute avec 30 allumettes
 et [Je m'appelle Alain Térieur, Je m'appelle Alex Térieur]
Il y a 30 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 2 allumette(s)
Il y a 28 allumette(s)   moi, Je m'appelle Alex Térieur j'enlève 2 allumette(s)
Il y a 26 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 2 allumette(s)
```

```
Il y a 24 allumette(s)   moi, Je m'appelle Alex Térieur j'enlève 1 allumette(s)
Il y a 23 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 2 allumette(s)
Il y a 21 allumette(s)   moi, Je m'appelle Alex Térieur j'enlève 2 allumette(s)
Il y a 19 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 18 allumette(s)   moi, Je m'appelle Alex Térieur j'enlève 3 allumette(s)
Il y a 15 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 14 allumette(s)   moi, Je m'appelle Alex Térieur j'enlève 1 allumette(s)
Il y a 13 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 12 allumette(s)   moi, Je m'appelle Alex Térieur j'enlève 3 allumette(s)
Il y a 9 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 3 allumette(s)
Il y a 6 allumette(s)   moi, Je m'appelle Alex Térieur j'enlève 3 allumette(s)
Il y a 3 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 3 allumette(s)
-> Moi, Alex Térieur j'ai gagné
```

L'évaluation du message `arbitre.nim(j1,j3)` affiche le résultat suivant :

```
La partie débute avec 22 allumettes
 et [Je m'appelle Alain Térieur, Je m'appelle Alex LeFort]
Il y a 22 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 21 allumette(s)   moi, Je m'appelle Alex LeFort j'enlève 2 allumette(s)
Il y a 19 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 3 allumette(s)
Il y a 16 allumette(s)   moi, Je m'appelle Alex LeFort j'enlève 2 allumette(s)
Il y a 14 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 2 allumette(s)
Il y a 12 allumette(s)   moi, Je m'appelle Alex LeFort j'enlève 2 allumette(s)
Il y a 10 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 9 allumette(s)   moi, Je m'appelle Alex LeFort j'enlève 2 allumette(s)
Il y a 7 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 6 allumette(s)   moi, Je m'appelle Alex LeFort j'enlève 2 allumette(s)
Il y a 4 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
Il y a 3 allumette(s)   moi, Je m'appelle Alex LeFort j'enlève 2 allumette(s)
Il y a 1 allumette(s)   moi, Je m'appelle Alain Térieur j'enlève 1 allumette(s)
-> Moi, Alex LeFort j'ai gagné
```

### 4.5 Conclusion

Dans cette section, nous avons mis en œuvre un programme Java à partir d'une description abstraite pour illustrer l'utilisation des outils de Java pour le codage. Nous n'avons pas présenté les outils de mise au point (inspecteurs et débogueurs) bien que nous les ayons utilisé pour tester le programme.

Cet exemple montre qu'il est rapide de programmer en Java, c'est pourquoi le langage est utilisé pour le prototypage rapide. Il montre aussi que la connaissance des classes du système influence fortement le résultat.

# Chapitre 5

# Concepts complémentaires

Dans ce chapitre, nous présentons quelques caractéristiques complémentaires du langage Java, importantes pour la programmation d'applications : les exceptions, les flux et les tampons de chaînes de caractères, la documentation et l'introspection.

## 1 Exceptions

Le traitement d'**exception** fait partie intégrante de Java, dès sa conception. Il est considéré comme une bonne pratique de programmation dans laquelle on sépare le traitement ordinaire des cas d'erreurs. Il permet d'améliorer la fiabilité du code. Sans traitement d'exceptions, on doit placer des tests partout dans le code (sous forme d'alternatives *si... alors ... sinon...* pour vérifier la validité es paramètres des fonctions, des expressions... C'est ce qu'on appelle la programmation défensive. Avec le traitement d'exceptions, on se concentre sur le traitement normal (programation offensive), les cas d'erreur et leur traitement sont vus ailleurs, en général à l'appel de méthodes.

Comme son nom l'indique, une **exception** est un événement anormal ou inattendu[1]. Le mécanisme est simple, si une situation normale est

Même si le développeur Java ne le souhait pas, il doit intégrer ce mécanisme dans sa programmation car le compilateur détecte que si un code appelé lève une exception, elle doit être traitée ou relayée.

### 1.1 Traitement d'exceptions

Le traitement d'exception consiste à exécuter un bloc de code susceptible de lever des exceptions, et lorsqu'une exception est levée, on la traite dans un boc de code approprié.

```
try {
   // bloc de code susceptible de lever une exception
}
catch (XXException e) {
   // traitement de l'exception de type XX
}
finally  {
   // traitement plus général
}
```

Le bloc de code `try` est exécuté. Si une erreur (prévue) se produit, l'exception est levée. Les blocs de code `catch` sont une sorte d'alternative multiple (`case`) qui en fonction du type d'exception exécute un bloc de code. Il y en a un par exception traitée. Le bloc de code

1. C'est l'exception qui confirme la règle...

optionnel `finally` est exécuté systématiquement en cas d'exception. Il permet par exemple de fermer des fichiers ou "tuer" des processus.

La combinaison `try .. finally` est aussi autorisée dans un bloc `try` qui ne lève pas d'exceptions mais contient des instructions `break`, `continue` et `return`.

Les exemples du programme de pluviométrie de la section 1 du chapitre 3 utilisent des traitements d'exceptions pour les entrées-sorties. Voici un exemple avec traitement multiples, extrait d'un outil de GL.

```
try {
   // contenu non précisé
}
catch (FileNotFoundException e) {
   if (errverbose) {
      System.err.println("Preference file Mec.userprefs not found");
      System.err.println("Please create this file for any Mec experimentation");
   }
   //return false;
}
catch (IOException e) {
   if (errverbose) System.err.println("IOException thrown "+e.toString());
   //return false;
}
catch (NullPointerException e) {
   if (errverbose)
   System.err.println("Some of the Mec.userprefs component or services are not found");
   e.printStackTrace();
   //return false;
}
catch (NonExistingLinkException e) {
   if (errverbose)
   System.err.println("The link "+e.getMessage()+" does not exist (in any order).");
   //return false;
}
// the other exceptions are propagated up
catch (Exception e) {
   if (errverbose) System.err.println("Exception thrown "+e.toString());
   //return false;
}
```

### 1.2 La classe `Exception`

Une exception est une instance de la classe `java.lang.Exception` ou de l'une de ses sous-classes (voir section 1.7). Elle peut être paramétrée par un message ou une cause.

```
public class Exception extends Throwable {
    static final long serialVersionUID = -3387516993124229948L;
    public Exception() {
    super();
    }
    public Exception(String message) {
    super(message);
    }
    public Exception(String message, Throwable cause) {
    super(message, cause);
    }
    public Exception(Throwable cause) {
    super(cause);
```

```
    }
}
```

Elles bénéficient aussi par héritage de méthodes de traitement similaires à celles des erreurs.

```
public class Throwable {
    public Throwable() ;
    public Throwable(String message) ;
    public String getMessage()
    public String toString() ;
    public void printStackTrace() ;
    public void printStackTrace(java.io.PrintStream s) ;
    private native void printStackTrace0(java.io.PrintStream s);
    public native Throwable fillInStackTrace();
}
```

### 1.3 Définir une exception

On peut définir sa propre exception en créant une classe qui hérite de `java.lang.Exception`, comme le montre l'exemple suivant.

```
package org.unantes.bosco.utils ;
public class ConfigFormatException extends Exception {
    public ConfigFormatException() {
        super();
    }
    public ConfigFormatException(String message) {
        super(message);
    }
}
```

### 1.4 Lever une exception

Pour lever une exception on utilise l'instruction `throw e` où `e` est une expression qui rend une exception. Dans le profil d'une méthode qui lève une exception, on ajoute la clause `throws E` où `E` est une classe d'exceptions.

```
  public ModelRepository(
    String name,
    String modelname,
    String basedir,
    String prefix,
    String language,
    String jar,
    String basepackage) throws ConfigFormatException {
    super();
    this.visitors=new HashMap();
    this.contexts=new HashMap();
    this.name = name;
    this.modelname = modelname;
    this.basedir = basedir;
    this.prefix = prefix;
    this.language = language;
    this.jar=jar;
    this.packagebasename=basepackage;
    this.setOptDefaultValues();
```

```
    if (!this.isValid()) throw new ConfigFormatException("Invalid or incomplete format");
  }
```

### 1.5 Propager une exception

Si une méthode `m` utilise directement ou indirectement une autre méthode qui lève une exception, elle doit la traiter. Elle peut le faire explicitement par un bloc `try ... catch(E e)` soit implicitement en propageant (ou relayant) l'exception, il suffit pour cela d'ajouter la clause `throws E` au profil de la méthode `m`, avec `E` est une classe d'exceptions.

### 1.6 Retour sur le traitement d'exceptions

Le traitement est le suivant : lorsqu'une exception est levée, le traitement en cours est interrompu et la pile d'exécution est dépilée jusqu'à trouver un bloc qui capture l'exception. S'il y en a un, il exécute le traitement d'exception et le programme continue à partir de là. S'il n'y en a pas, ce qui en principe n'est pas autorisé à la compilation[2], le programme s'arrête sur une erreur.

### 1.7 Type d'exceptions

Les exceptions sont des extensions de la classe `java.lang.Exception`, elles sont liées au traitement d'erreur comme le montre la hiérarchie de la figure 41.

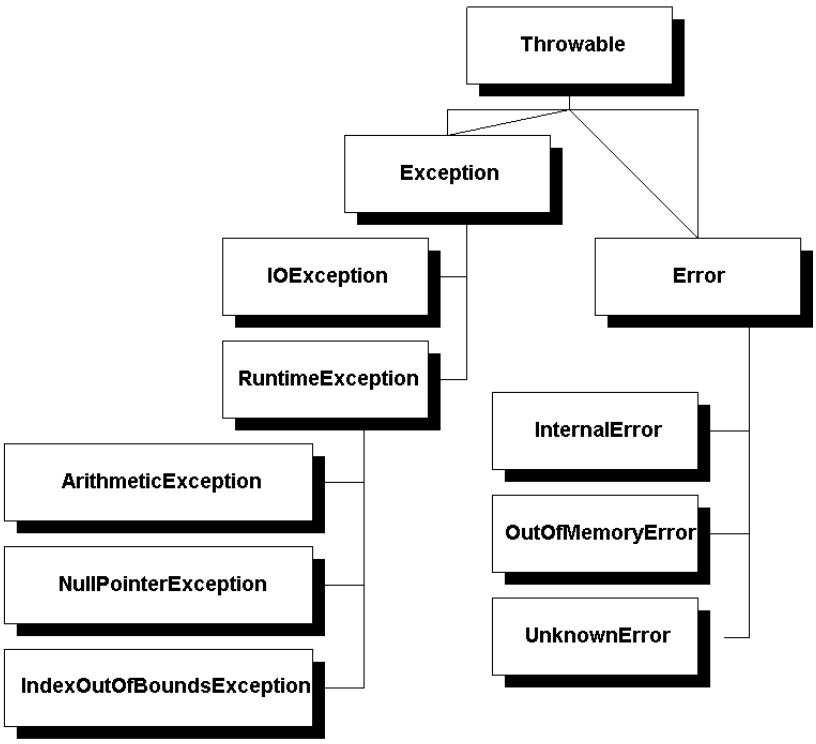



Figure 41 : *Erreurs et exceptions en Java*

(source http://sofia.fhda.edu/gallery/java/unit07/lesson07-3.html )

Les listes et hiérarchies d'erreurs et d'exceptions qui suivent sont extraites de (source http://www.webbasedprogramming.com/Java-Unleashed-Second-Edition/ch10.htm )

2. Plus exactement il existe deux catégories d'exceptions : les exceptions contrôlées et les exceptions non-contrôlées, qui n'ont pas à être déclarées (`RunTimeException` et `Error`).

### Erreurs

Voici une partie des erreurs courantes :

| Error | Cause |
|---|---|
| AbstractMethodError | Attempt to call an abstract method. |
| ClassCircularityError | This error is no longer used. |
| ClassFormatError | Invalid binary class format. |
| Error | Root class of the error hierarchy. |
| IllegalAccessError | Attempt to access an inaccessible object. |
| IncompatibleClassChangeError | Improper use of a class. |
| InstantiationError | Attempt to instantiate an abstract class. |
| InternalError | Error in the interpreter. |
| LinkageError | Error in class dependencies. |
| NoClassDefFoundError | Unable to find the class definition. |
| NoSuchFieldError | Unable to find the requested field. |
| NoSuchMethodError | Unable to find the requested method. |
| OutOfMemoryError | Out of memory. |
| StackOverflowError | Stack overflow. |
| ThreadDeath | Indicates that the thread will terminate. May be caught to perform cleanup. (If caught, must be rethrown.) |
| UnknownError | Unknown virtual machine error. |
| UnsatisfiedLinkError | Unresolved links in the loaded class. |
| VerifyError | Unable to verify bytecode. |
| VirtualMachineError | Root class for virtual machine errors. |

TABLEAU V– *Erreurs de Java*

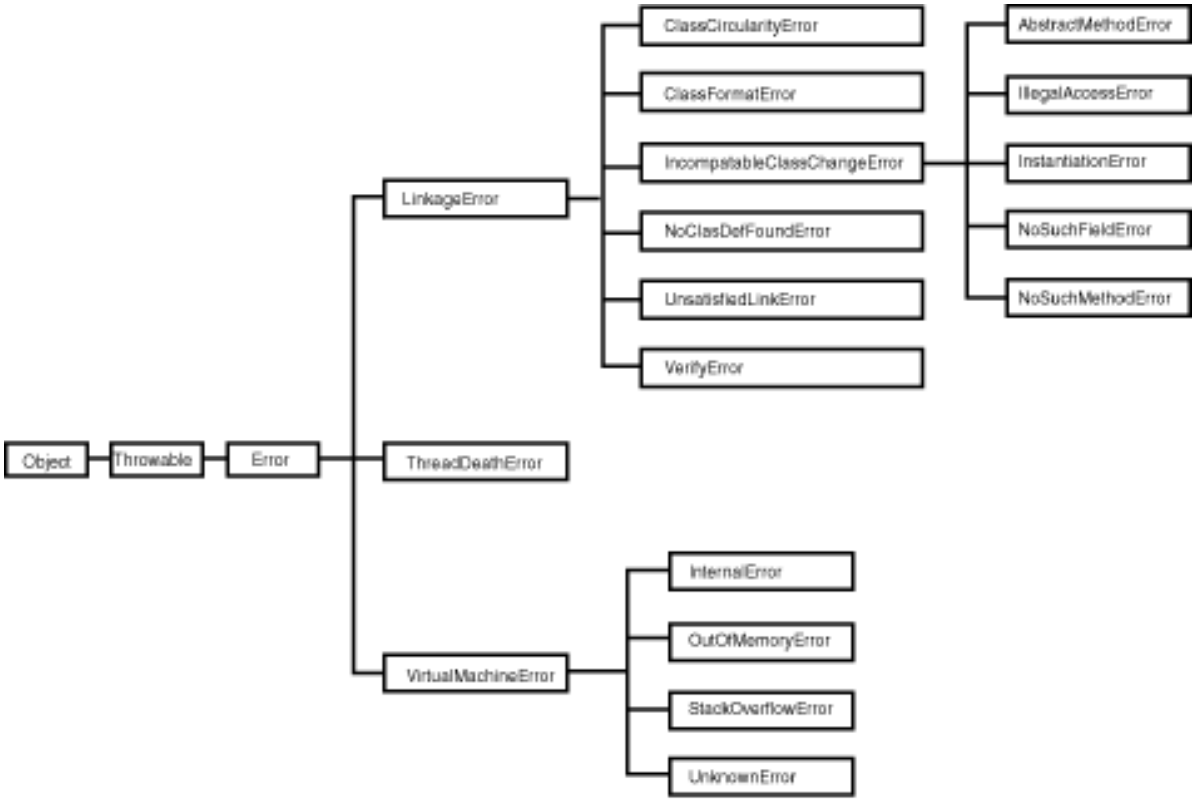



Figure 42 : *Hiérarchie des erreurs de Java*

### Exceptions

Voici quelques exceptions courantes :

| Exception | Cause |
|---|---|
| ArithmeticException | Arithmetic error condition (for example, divide by zero). |
| ArrayIndexOutOfBoundsException | Array index is less than zero or greater than the actual size of the array. |
| ArrayStoreException | Object type mismatch between the array and the object to be stored in the array. |
| ClassCastException | Cast of object to inappropriate type. |
| ClassNotFoundException | Unable to load the requested class. |
| CloneNotSupportedException | Object does not implement the cloneable interface. |
| Exception | Root class of the exception hierarchy. |
| IllegalAccessException | Class is not accessible. |
| IllegalArgumentException | Method receives an illegal argument. |
| IllegalMonitorStateException | Improper monitor state (thread synchronization). |
| IllegalThreadStateException | The thread is in an improper state for the requested operation. |
| IndexOutOfBoundsException | Index is out of bounds. |
| InstantiationException | Attempt to create an instance of the abstract class. |
| InterruptedException | Thread interrupted. |
| NegativeArraySizeException | Array size is less than zero. |
| NoSuchMethodException | Unable to resolve method. |
| NullPointerException | Attempt to access a null object member. |
| NumberFormatException | Unable to convert the string to a number. |
| RuntimeException | Base class for many java.lang exceptions. |
| SecurityException | Security settings do not allow the operation. |
| StringIndexOutOfBoundsException | Index is negative or greater than the size of the string. |

TABLEAU VI– *Exceptions de Java* `java.lang`

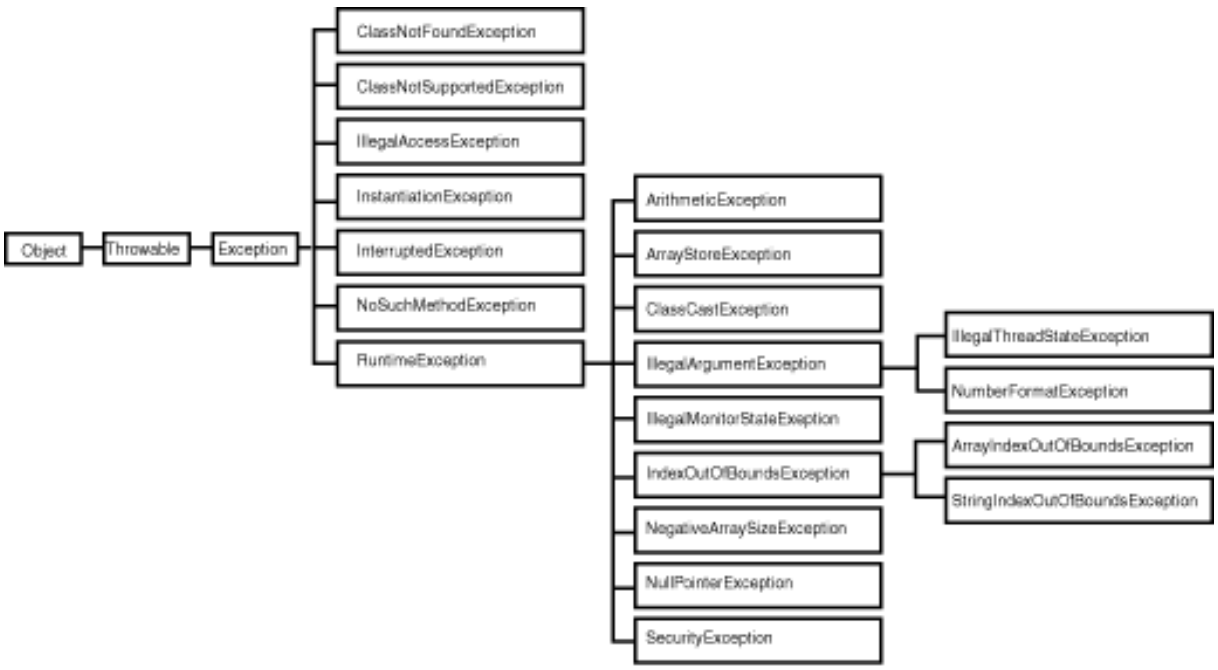



Figure 43 : *Hiérarchie des exceptions de Java*

| Exception | Cause |
|---|---|
| IOException | Root class for I/O exceptions. |
| EOFException | End of file. |
| FileNotFoundException | Unable to locate the file. |
| InterruptedIOException | I/O operation was interrupted. Contains a bytesTransferred member that indicates how many bytes were transferred before the operation was interrupted. |
| UTFDataFormatException | Malformed UTF-8 string. |

TABLEAU VII– *Erreurs d'entrées/sorties de Javajava.io*

## 2 Entrées-sorties, flux et sérialisation

Pour les entrées-sorties, on utilise des flux de données pour les valeurs des types primitifs et la sérialisation pour les objets.

### 2.1 Entrées-sorties et flux

Cette section est fortement inspirée de
http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/Io/

Un **flux de données** représente une suite d'octets reliée à un support qui peut-être une chaîne de caractères, un fichier, un périphérique (clavier, console), une connexion réseau... On distingue les flux entrants (`InputStream`) des flux sortants (`OutputStream`). En plus de cette nature (entrée/sortie), les flux se distinguent par leur suppport (mémoire, périphérique de stockage, réseau...). La hiérarchie d'héritage des flux a été synthétisée dans la figure 15.

Il existe, comme dans la plupart des langages, trois flux prédéfinis pour chaque application : l'entrée standard `System.in`, sa sortie standard `System.out` et sa sortie standard d'erreurs `System.err`. Noter que `System.in` est une instance de la classe `InputStream`, alors que les deux autres flux sont des instances de la classe `PrintStream`.

```
try {
   int c;
   while((c = System.in.read()) != -1) {
   System.out.print(c);   nbc++;
   }
} catch(IOException exc) {
   exc.printStackTrace();   // En fait exc.printStackTrace(System.err);
}
```

Pour ce qui est de la manipulation du flux de sortie, il est préférable de ne pas directement utiliser la classe `InputStream`. En effet, elle ne propose que des méthodes élémentaires de récupération de données. Préférez au contraire la classe `BufferedReader`. Nous reviendrons ultérieurement sur la notion de `Reader` et de `Writer`. L'exemple suivant montre comment générer simplement un objet `BufferedReader` à partir de `System.in`. Cela vous permet de récupérer des chaînes de caractères saisies sur la console par l'utilisateur.

```
Reader reader = new InputStreamReader(System.in);
BufferedReader keyboard = new BufferedReader(reader);

System.out.print("Entrez une ligne de texte : ");
String line = keyboard.readLine();
System.out.println("Vous avez saisi : " + line);
```

Voici un premier exemple :

```
public static void main(String args[]) {
char carac;
int nb1, nb2;
String chaine, message;

BufferedReader stdin = new BufferedReader(new InputStreamReader(
 System.in));
// déclarer le tampon de lecture
try {
   System.out.print("Entrer le 1er nombre : ");
   chaine = stdin.readLine(); // lire les caractères dans une chaine
   nb1 = Integer.parseInt(chaine); // convertir la valeur lue en nombre
   System.out.print("Entrer le 2ème nombre : ");
   chaine = stdin.readLine(); // lire les caractères dans une chaine
   nb2 = Integer.parseInt(chaine); // convertir la valeur lue en nombre
   System.out.print("Entrer un caractère : ");
   chaine = stdin.readLine(); // lire le caractère dans une chaine
   carac = chaine.charAt(0); // récupérer le caractère à la
   // 1ère position de la chaine
   System.out.print("Entrer un message : ");
   message = stdin.readLine();
   System.out.println();
   System.out.println("le nombre lu est : " + nb1);
   System.out.println("le nombre lu est : " + nb2);
   System.out.println("le caractère lu est : " + carac);
   System.out.println("la chaine lue est : " + message);
} catch (IOException e) {
   System.err.println("IOException thrown " + e.toString());
   // return false;
}
}
```

(source http://www.iro.umontreal.ca/ dift1870/A04/Pgms/Lecture.java )

Le tableau VIII montre quelques classes du paquetage `java.io`. Les classes d'E/S octets sont en partie dépréciée depuisla version originale du JDK. En effet, les méthodes permettant l'acquisition de chaînes de caractères présentent l'inconvénient de ne pas être portables d'un système à un autre, ce qui pour Java est inacceptable. La norme ASCII (American Standard Code for Information Interchange) ne spécifie que 128 caractères. Pour pallier à la divversité d'ASCII étendus,Java utilise le système Unicode (16 bits). Mais, attention : dans la majorité des cas, les flux sont des flux 8 bits. De fait, les classes `Reader` et `Writer` permettent de transformer des code ASCII dérivé vers Unicode et réciproquement. Ainsi, par exemple, `System.in` est et restera un flux 8 bits, mais l'utilisation d'un `Reader` permettra de transformer les caractères ASCII (ou dérivé) en code Unicode en tenant compte du système de codage utilisé sur le poste. Ces classes sont apparues à partir du JDK 1.1.

| | **Flux d'entrées** | **Flux de sorties** |
|---|---|---|
| JDK 1.0<br>Flux d'octets<br>(8 bits) | InputStream<br>+- FileInputStream<br>+- DataInputStream<br>+- BufferedInputStream<br>+- . . . | OutputStream<br>+- FileOutputStream<br>+- DataOutputStream<br>+- BufferedOutputStream<br>+- . . . |
| JDK 1.1<br>Flux de caractères<br>(16 bits) | Reader<br>+- FileReader<br>+- StringReader<br>+- . . . | Writer<br>+- BufferedReader<br>+- FileWriter<br>+- BufferedWriter<br>+- StringWriter<br>+- . . . |

TABLEAU VIII– *Flux d'entrées/sorties de Java* `java.io`

Les classes `InputStreamReader` et `OutputStreamWriter` permettent permettent, respectivement, de transformer un `InputStream` en un `Reader` et un `OutputStream` en un `Writer`. Exemple :

```
Reader reader = new InputStreamReader(System.in);
BufferedReader keyboard = new BufferedReader(reader);

System.out.print("Entrez une ligne de texte : ");
String line = keyboard.readLine();
System.out.println("Vous avez saisi : " + line);
```

Pour utiliser des classes de flux adapté à vos besoins, il faut souvent passer par plusieurs constructions intermédiaires. Les exemples de code suivants montrent comment cumuler les constructions de classes de flux pour arriver au résultat escompté. Le but final est d'obtenir un flot sur fichier, bufférisé, permettant de manipuler des données typées.

```
File f = new File("fichier.mp3");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new BufferInputStream(fis);
DataInputStream dis = new DataInputStream(bis);

int a = dis.readInt();
short s = dis.readShort();
boolean b = dis.readBoolean();
```

Il est important de comprendre que l'ordre de construction ne peut en aucun cas être changé. En effet, l'objet final à utiliser se doit d'être de type `DataInputStream`. En effet, c'est sur ce type que les méthodes attendues sont définies. Le second exemple montre le même exemple, mais pour écrire dans un flux.

```
File f = new File("fichier.mp3");
FileOutputStream fos = new FileOutputStream(f);
BufferedOutputStream bos = new BufferedOutputStream(fos);
DataOutputStream dos = new DataOutputStream(bos);

int a = 10;   dos.writeInt(a);
short s = 3;   dos.writeShort(s);
boolean b = true; dos.writeBoolean(b);
```

L'exemple suivant permet de réaliser une copie de fichier. Pour ce faire, le programme attend que le nom du fichier source et le nom du fichier de destination soient renseignés sur la ligne de commande servant à lancer le programme. A titre indicatif, voici un exemple de commande servant à lancer la copie.

```
> java Copy sourceFile.txt destFile.txt
```

Le programme travaille sur des flux binaires et non des flux textuels : il est donc judicieux de choisir les sous-classes de `InputStream` et de `OutputStream`. Par ailleurs, la manipulation de flux peut aboutir à lever des exceptions. Il faut donc spécifier ce que l'on fera d'une éventuelle exception, ce qui permet d'illustrer la section précédente. Ici, on affiche la pile des appels de méthodes (une propagation `throws Exception` sur la signature de la méthode `main` en aurait fait autant).

Dans le but de montrer un maximum de choses, l'auteur récupère la taille du fichier grâce à l'objet de type `File`. Mais il aurait pu coder la boucle de lecture des octets jusqu'à obtenir la valeur -1 (fin de fichier). Pour accélérer le traitement, on place des tampons sur les flux basés sur les fichiers. Dans ce cas, il faut absolument que fermer les flux "bufférisés" et non les flux simples (en fait les derniers objets de flux créés).

```
import java.io.*;

public class Copy {
   public static void main (String[] argv)   {
   // Test sur le nombre de paramètres passés
   if (argv.length != 2) {
      System.out.println("Usage> java Copy sourceFile destinationFile");
      System.exit(0);
   }
   try {
      // Préparation du flux d'entrée
      File sourceFile = new File(argv[0]);
      FileInputStream fis = new FileInputStream(sourceFile);
      BufferedInputStream bis = new BufferedInputStream(fis);
      long l = sourceFile.length();
      // Préparation du flux de sortie
      FileOutputStream fos = new FileOutputStream(argv[1]);
      BufferedOutputStream bos = new BufferedOutputStream(fos);
      // Copie des octets du flux d'entrée vers le flux de sortie
      for(long i=0;i<l;i++) {
         bos.write(bis.read());
      }
      // Fermeture des flux de données
      bos.flush();
      bos.close();
      bis.close();
   } catch (Exception e) {
      System.err.println("File access error !");
      e.printStackTrace();
   }
   System.out.println("Copie terminée");
   }
}
```

La section suivante, qui clôt ce chapitre sur la gestion des entrées/sorties, vous présente le concept de sérialisation avec deux nouvelles classes de flux (`ObjectInputStream` et `ObjectOutputStream`).

## 2.2 Sérialisation

La **sérialisation** est une caractéristique ajoutée dans la version 1.1 de Java qui permet une manipulation plus simple des flux d'objets et des entrées/sorties[3]. Elle permet de s'abstraire

3. BOSS en Smalltalk.

du support physique pour implanter la **persistance** d'objets Java.

La sérialisation consiste à pouvoir prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données binaire (vers un fichier, par exemple). Ce concept permettra aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement. La sérialisation peut donc être considérée comme une forme de persistance des données, le transfert à distance par le protocole RMI (*Remote Method Invocation*[4]).

La plupart des objets de base du langage (int, String, Vector...) sont sérialisables [Cla03]. La classe `ObjectOutputStream` contient en effet plusieurs méthodes de sérialisation des types primitifs : `writeInt`, `writeDouble`, `writeFloat` ... La classe `ObjectInputStream` possède de la même façon des méthodes pour lire des données de type primitifs : `readInt()`, `readDouble()`, `readFloat` ...

Lors de la désérialisation, le constructeur de l'objet n'est jamais utilisé.

La lecture ou l'écriture peut conduire à une erreur de flux sous-jacent ou d'objet sérialisable. Dans ces cas, Java lève une **exception**. Par exemple :

- `InvalidClassException` : fonctionnement incorrect d'une classe sérialisée.
- `NotSerializableException` : l'objet n'implante pas l'interface adéquate.
- `IOException` : erreur du flux sous-jacent.
- `OptionalDataException` : une valeur primitive est rencontrée au lieu d'un objet.
- `StreamCorruptedException` : les informations de contrôle du flux sont incohérentes.
- `ClassNotFoundException` : la classe d'un objet sérialisé n'est pas trouvée.

```
public final class String
extends Object
implements Serializable, Comparable, CharSequence
```

La sérialisation utilise l'interface `Serializable` et les classes `ObjectOutputStream` et `ObjectInputStream`. L'interface `Serializable` ne définit aucune méthode mais permet par sous-typage de marquer une classe comme pouvant être sérialisée. Tout objet qui doit être sérialisé doit implémenter cette interface dans sa classe ou une de ses super-classes. Dans le cas contraire, une exception `NotSerializableException` est levée.

Prenons l'exemple de Pierre-Yves Saumont dans [SM03]. On sérialise des voitures, qui elles mêmes ont des moteurs et des carosseries (à sérialiser). Les classes `Moteur` et `Voiture` sont définies comme suit

```
import java.io.Serializable;

public class Moteur implements Serializable {

  String valeur;
  Moteur (String s) {
    valeur = s;
  }
  String getValeur() {
    return valeur;
  }
}
```

```
import java.io.Serializable;

public class Carrosserie implements Serializable {
  String valeur;
  Carrosserie (String s) {
    valeur = s;
```

4. Le *Remote Procedure Call* de la programmation concurrente.

```
  }
  String getValeur() {
    return valeur;
  }
}
```

Un moteur et une carosserie sont sérialisables (pas de méthodes spécifiques, car `String` est sérialisable). Une voiture est simplement définie comme un "sérialisable", sachant que ses variables d'instance le sont.

```
import java.io.Serializable;

public class Voiture implements Serializable {
  Moteur moteur;
  Carrosserie carrosserie;
  transient int essence;

  Voiture (String m, String c) {
    moteur = new Moteur(m);
    carrosserie = new Carrosserie(c);
  }
  String getMoteur() {
    return moteur.getValeur();
  }
  String getCarrosserie() {
    return carrosserie.getValeur();
  }
  void setCarburant(int e) {
    essence += e;
  }
  int getCarburant() {
    return essence;
  }
}
```

Le modificateur de visibilité `transient` signifie que cet attribut doit être ignoré dans la sérialisation, il est "temporaire". La sérialisation (du même paquetage) s'écrit ainsi :

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class Serialisation {
  public static void main (String[] args) throws IOException {
    Voiture voiture = new Voiture("V6", "Cabriolet");
    voiture.setCarburant(50);
    FileOutputStream f = new FileOutputStream("garage");
    ObjectOutputStream o = new ObjectOutputStream(f);

    o.writeObject(voiture);
    o.close();
  }
}
```

Le fichier 'garage' est peu lisible et non imprimable.

La désérialisation (du même paquetage) s'écrit ainsi :

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

```
public class Deserialisation {
  public static void main (String[] args)
                throws IOException,
                ClassNotFoundException {
    FileInputStream f = new FileInputStream("garage");
    ObjectInputStream o = new ObjectInputStream(f);

    Voiture voiture = (Voiture)o.readObject();
    o.close();

    System.out.println("Carrosserie : " + voiture.getCarrosserie());
    System.out.println("Moteur : " + voiture.getMoteur());
    System.out.println("Carburant : " + voiture.getCarburant());
  }
}
```

Ce code produit le résultat suivant :

```
Carrosserie : Cabriolet
Moteur : V6
Carburant : 0
```

Un autre exemple sur les personnes est donné dans
http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap020.htm

```
public class Personne implements java.io.Serializable {
  private String nom = "";
  private String prenom = "";
  private int taille = 0;

  public Personne(String nom, String prenom, int taille) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
  }
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  public int getTaille() {
    return taille;
  }
  public void setTaille(int taille) {
    this.taille = taille;
  }
  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }
}
```

La classe suivante permet de sérialiser un objet Personne.

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

```
public class SerializerPersonne {
  public static void main(String argv[]) {
    Personne personne = new Personne("Dupond","Jean",175);
    try {
      FileOutputStream fichier = new FileOutputStream("personne.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fichier);
      oos.writeObject(personne);
      oos.flush ();
      oos.close ();
    }
    catch (java.io.IOException e) {
      e.printStackTrace();
    }
  }
}
```

On définit un fichier avec la classe `FileOutputStream`. On instancie un objet de classe `ObjectOutputStream` en lui fournissant en paramètre le fichier : ainsi, le résultat de la sérialisation sera envoyé dans le fichier. On appelle la méthode `writeObject` en lui passant en paramètre l'objet à sérialiser. On appelle la méthode `flush()` pour vider le tampon dans le fichier et la méthode `close()` pour terminer l'opération. Lors de ces opérations une exception de type `IOException` peut être levée si un problème intervient avec le fichier.

Après l'exécution de cet exemple, un fichier nommé « personne.ser »est créé. On peut visualiser son contenu mais surtout pas le modifier car sinon il serait corrompu. En effet, les données contenues dans ce fichier ne sont pas toutes au format caractères. Il est possible dans un même flux d'écrire plusieurs objets les uns à la suite des autres. Ainsi plusieurs objets peuvent être sauvegardés. Dans ce cas, il faut faire attention de relire les objets dans leur ordre d'écriture.

La classe suivante permet de désérialiser un objet Personne.

```
import java.io.*;

public class DeSerializerPersonne {

  public static void main(String argv[]) {
    try {
      FileInputStream fichier  = new FileInputStream("personne.ser");
      ObjectInputStream ois = new ObjectInputStream(fichier);
      Personne personne = (Personne) ois.readObject();
      System.out.println("Personne : ");
      System.out.println("nom : "+personne.getNom());
      System.out.println("prenom : "+personne.getPrenom());
      System.out.println("taille : "+personne.getTaille());
    }
    catch (java.io.IOException e) {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
   }
}
```

Ce code produit le résultat suivant :

```
Personne :
nom : Dupond
```

```
prenom : Jean
taille : 175
```

On créer un objet de la classe `FileInputStream` qui représente le fichier contenant l'objet sérialisé. On créer un objet de type `ObjectInputStream` en lui passant le fichier en paramètre. Un appel à la méthode `readObject()` retourne l'objet avec un type `Object`. Une coercition de type est nécessaire pour obtenir le type de l'objet. La méthode `close()` permet de terminer l'opération.

Si la classe a changé entre le moment où une instance a été sérialisée et le moment où l'instance est désérialisée, une exception est levée. Par exemple si on modifie et recompile la classe `Personne`, on obtient :

```
C:\temp>java DeSerializerPersonne
java.io.InvalidClassException: Personne; Local class not compatible: stream class
desc serialVersionUID=-2739669178469387642 local class serialVersionUID=39870587
36962107851

atjava.io.ObjectStreamClass.validateLocalClass(ObjectStreamClass.java:4
38)
at java.io.ObjectStreamClass.setClass(ObjectStreamClass.java:482)
at java.io.ObjectInputStream.inputClassDescriptor(ObjectInputStream.java
:785)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:353)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:978)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

Une exception de type `StreamCorruptedException` peut être levée si le fichier a été corrompu par exemple en le modifiant avec un éditeur. Modifions les 2 premiers octets du fichier `personne.ser`.

```
C:\temp>java DeSerializerPersonne

java.io.StreamCorruptedException: InputStream does not containa serialized object
at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:731)
at java.io.ObjectInputStream.<init>(ObjectInputStream.java:165)
at DeSerializerPersonne.main(DeSerializerPersonne.java:8)
```

Une exception de type `ClassNotFoundException` peut être levée si l'objet est transtypé vers une classe qui n'existe plus ou pas au moment de l'exécution.

```
C:\temp>rename Personne.class Personne2.class
C:\temp>java DeSerializerPersonne

java.lang.ClassNotFoundException: Personne
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:981)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

Enfin, le *listing 5.1* présente un exemple amusant de sauvegarde de fenêtres Swing.

Listing 5.1 – Code de la classe Serialisation

```
package serialisation ;
```

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

public class Serialisation {
    private final static Reader reader = new InputStreamReader(System.in);
    private final static BufferedReader keyboard = new BufferedReader(reader);

    // Permet de créer une fenêtre et de la sérialiser dans un fichier.
    public void saveWindow() throws IOException {
        JFrame window = new JFrame("Ma fenêtre");
        JPanel pane = (JPanel)window.getContentPane();
        pane.add(new JLabel("Barre de status"), BorderLayout.SOUTH);
        pane.add(new JTree(), BorderLayout.WEST);
        JTextArea textArea = new JTextArea("Ceci est le contenu !!!");
        textArea.setBackground(Color.GRAY);
        pane.add(textArea, BorderLayout.CENTER);

        JPanel toolbar = new JPanel(new FlowLayout());
        toolbar.add(new JButton("Open"));
        toolbar.add(new JButton("Save"));
        toolbar.add(new JButton("Cut"));
        toolbar.add(new JButton("Copy"));
        toolbar.add(new JButton("Paste"));

        pane.add(toolbar, BorderLayout.NORTH);
        window.setSize(400,300);

        FileOutputStream fos = new FileOutputStream("window.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(window);
        oos.flush();
        oos.close();
    }

    // Permet de reconstruire la fenêtre à partir des données du fichier.
    public void loadWindow() throws Exception {
        FileInputStream fis = new FileInputStream("window.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        JFrame window = (JFrame)ois.readObject();
        ois.close();

        window.setVisible(true);
    }

    // Permet de saisir différentes commandes. Testez plusieurs load
    // consécutifs : plusieurs fenêtres doivent apparaîtrent
    public static void main(String[] args) throws Exception {
        Serialisation object = new Serialisation();

        while(true) {
            System.out.print("Saisir le mode d'execution (load ou save) : ");
            String mode = keyboard.readLine();

            if (mode.equalsIgnoreCase("exit")) break;
            if (mode.equalsIgnoreCase("save")) object.saveWindow();
            if (mode.equalsIgnoreCase("load")) object.loadWindow();
        }
```

```
            System.exit(0);
        }
}
```

Source : http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/Io/

### Persistance

La sérialisation est un mécanisme de base qui doit être adapté pour la gestion de versions, la sécurité (cryptage), les propriétés de classes, les objets son sérialisables (le mot-clé transient pas toujours indiqué). Il est possible de personnaliser la sérialisation d'un objet. Dans ce cas, la classe doit implémenter l'interface `Externalizable` qui hérite de l'interface `Serializable`. Cette interface définit deux méthode : `readExternal()` et `writeExternal()`. Par défaut, la serialisation d'un objet qui implémente cette interface ne prend en compte aucun attribut de l'objet. Le mot clé transient est donc inutile avec un classe qui implémente l'interface `Externalisable`. La réflexion, que nous abordons dans la section 4 peut aussi être utile pour l'adaptation de la sérialisation.

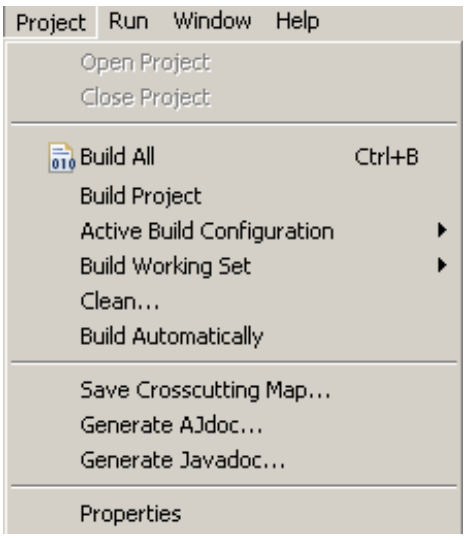
La persistance se gère aussi à plus grande échelle avec les bases de données et l'interface `JDBC` (*Java DataBase Connectivity*) que nous n'étudions pas ici.

## 3 JavaDoc

La documentation est un élément clé dans la maintenance du code. Les méthodes de développement agiles, telles que l'*eXtreme Programming* [Clo03] en font un principe majeur de bonne pratique du développement. Une bonne manière de commenter du code et surtout sa conception est d'utiliser le standard UML [AV01b]. Plus simplement, les concepteurs de Java ont utilisé les techniques récentes du web, et notamment XML (*eXtented Markup Language*) pour concevoir un outil simple mais agréable : `javadoc`.

Cet outil, fourni avec le JDK, permet de lire du code Java dans lequel l'auteur aura inséré des balises de commentaire particulières, non lues par un compilateur Java, et de produire une hiérarchie de fichiers HTML qui documentent chaque élément structurel du code (paquetage, classe, méthodes) et dont le point d'entrée est le fichier index.html. La documentation des API Java est réalisée avec cet outil.

La commande `javadoc` est lancée en ligne sur un paquetage ou par une fonction d'un IDE. Sous Eclipse, un plugin peut être nécessaire selon la version utilisée. La génération se trouve dans le menu `Project > Generate Javadoc`.



Le lancement de la commande permet de sélectionner la commande `javadoc` et ses options (figure 44).

Evidemment, pour préserver l'encapsulation javadoc ne présente que les propriétés publiques des classes. Ces propriétés sont ordonnées : documentation du paquetage, documentation de la classe, des variables, des méthodes, en commençant par les constructeurs. javadoc présente ces éléments par catégorie.

En plus du code et des commentaires, javadoc exploite des commentaires javado, qui se différencient des commentaires traditionnels par une double étoile. Ces commentaires sont parsemées de balises particulières (préfixées par `@`) qui permettent de le structurer (il n'y en a pas assez à notre goût) : auteurs, dates, paramètres, retour, exceptions, référencement... L'auteur peut lui-même insérer des balises HTML et personnaliser la présentation.

Sous Eclipse, l'insertion de commentaires fait partie des facilités de modification du code (raccourcis ALT+SHIFT+J) et il y a complétion pour la balise `@`.

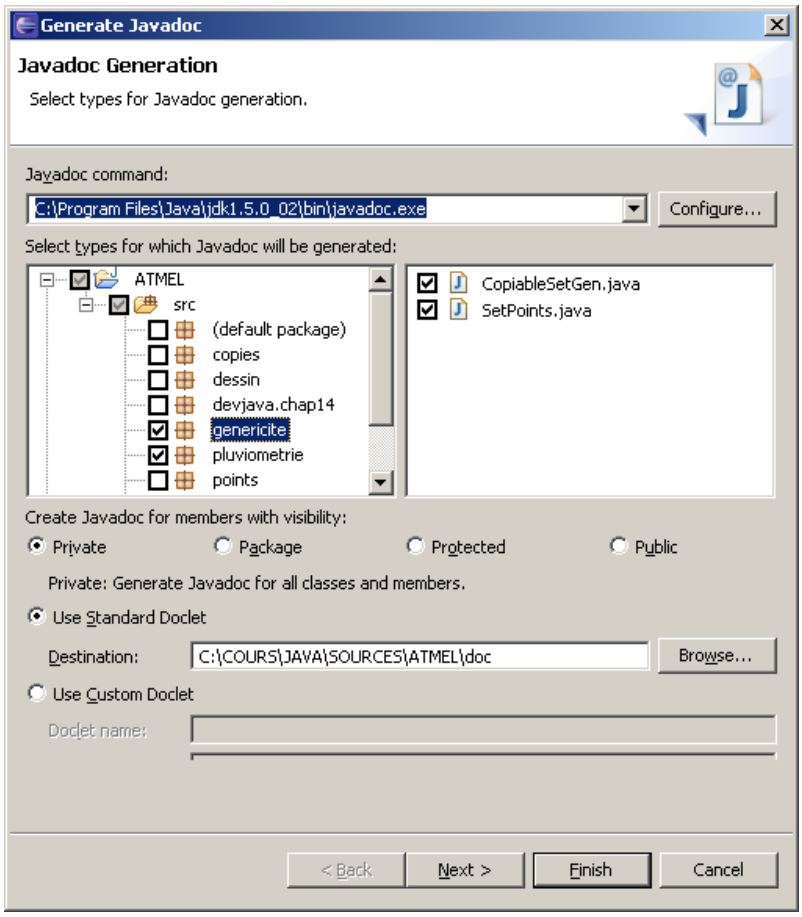



Figure 44 : *Le menu Projet/Javadoc Eclipse*

Reprenons la classe générique `CopiableSetGen` de la page 44 en la commentant. La balise `@author` indique l'auteur. La balise `@version` donne la version de la classe. La balise `@since` permet de faire référence au JDK. La balise `@see` permet de faire des liens hypertextes vers d'autres classes ou méthodes (nom de la classe puis le caractère # suivit du nom d'une méthode et éventuellement la signature en cas de surcharge). La balise `@deprecated` indique que la classe ou la méthode est périmée. La balise `@throws` indique que la classe ou la méthode lève un type d'exception. La balise `@param` d'une méthode décrit un paramètre. La balise `@return` d'une méthode décrit le résultat...

Listing 5.2 – Code commenté de la classe CopiableSetGen

```
package genericite;
import java.util.HashSet;
import java.util.Vector;

import copies.Copiable;
import copies.CopiablePointPA;
import copies.CopiableSet;

/**
 * Cette classe implante des ensembles génériques
 * de <B>Copiables</B>.<br>
 * Les copiables sont des objets qu'on peut cloner.
 * @author andre
 * @version 1
 * @see Copiable
 * @see HashSet
 */
public class CopiableSetGen extends HashSet<Copiable> implements Copiable {
    /**
     * Les classes sérialisables doivent déclarer un numéro de série static final de type long.
     */
    private static final long serialVersionUID = 1L;
    /**
     * variables d'instance
     * name : le nom de l'ensemble
     */
    private String name; // pour l'affichage

    /**
     * Constructeur par défaut
     */
    public CopiableSetGen() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * Constructeur
     * @param n nom de l'ensemble
     */
    public CopiableSetGen(String n) {
        super();
        name = n;
    }

    /**
     * Accesseur en écriture qui modifie le nom de l'ensemble
     * @param name
     */
    public void setName(String name) {
        this.name = name;
    }

    /* (non-Javadoc)
     * @see copies.Copiable#deepCopy()
     */
```

```
public Copiable deepCopy() {
    // redéfinition de la méthode
    CopiableSetGen copie = new CopiableSetGen();
    for (Copiable copiable : this) {
        copie.add(copiable.deepCopy());
    }
     return copie;
}

/* (non-Javadoc)
 * @see copies.Copiable#shallowCopy()
 */
public Copiable shallowCopy() {
    // redéfinition de la méthode
    CopiableSetGen copie = new CopiableSetGen();
    copie.addAll(this);
    return copie;
}

/* (non-Javadoc)
 * @see copies.Copiable#copy()
 */
public Copiable copy() {
    // redéfinition de la méthode
    return this.deepCopy();
}

/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
public String toString() {
    // redéfinition de la méthode ((Object)this).toString()+
    return (name+super.toString());
}

 /**
 * Programme principal classique<br>
 * qui illustre les ensembles de copiables.
 * @param args
 */
public static void main(String args[]) {
     CopiablePointPA p1 = new CopiablePointPA(5.0, 6.0);
     CopiablePointPA p2 = new CopiablePointPA(7.0, 8.0);
     CopiablePointPA p3 = p1;
     CopiableSet s1 = new CopiableSet("s1");
     CopiableSet s2, s3 ;
     HashSet s5 = new HashSet(); // ancien s4
     Vector s4 = new Vector(); // pour mieux distinguer
     s1.add(p1); s1.add(p2); s1.add(p3);
     s2 = (CopiableSet) s1.shallowCopy(); //shallow copy
     s2.setName("s2");
     s3 = (CopiableSet) s2.deepCopy(); //deep copy
     s3.setName("s3");
     s4.add(s1); s4.add(s2); s4.add(s3);
     s5.add(s1); s5.add(s2); s5.add(s3);
     System.out.println("Résultat : "+s4.toString()+ " →" + s4.size());
     System.out.println("Résultat : "+s5.toString()+ " →" + s5.size());
 }
```

}

# 4 Introspection, Réflexion

Dans cette section, nous abordons des mécanismes puissants des langages de programmation, qui permettent dynamiquement d'exploiter des informations des classes et des programmes soit pour faciliter des vérifications soit pour personnaliser l'exécution du programme.

## 4.1 Généralités

En modélisation, on dit qu'un modèle est réflexif s'il est défini à partir de lui-même. Par exemple, le MOF, méta-modèle d'UML, est décrit en utilisant UML.

En programmation informatique, la **réflexion** est la capacité d'un programme à examiner, et éventuellement à modifier, ses structures internes de haut niveau (par exemple ses objets) lors de son exécution. On distingue deux techniques utilisés par les systèmes dotés de réflexion :

- l'**introspection**, qui est la capacité d'un programme à examiner sur son propre état. L'introspection est utilisée pour effectuer des mesures de performance, inspecter des modules ou déboguer un programme. Elle est implémentée dans des langage comme SmallTalk ou Java qui fournissent des outils pour connaître la classe d'un objet, ses attributs, ses méthodes, etc. L'introspection n'existe pas dans des langages comme le C ou le Pascal.
- l'**intercession**, qui est la capacité d'un programme à modifier son propre état d'exécution ou d'altérer sa propre interprétation ou signification. L'intercession permet à un programme d'évoluer automatiquement en fonction des besoins et de l'environnement. Cette propriété apparaît dans des langages comme SmallTalk ou Python, mais elle n'existe pas dans des langages comme Java.

(source `http://fr.wikipedia.org/wiki/Réflexion_(informatique)` )

Parallèlement aux concepts d'introspection et d'intercession, il existe deux types de réflexion : la réflexion structurelle et la réflexion comportementale.

- La `réflexion structurelle` consiste à réifier le code d'un programme et tous les types abstraits accessibles par ce programme. Dans le premier cas, la réification du code d'un programme permet de manipuler ce programme pendant l'exécution. Il est possible ainsi de maintenir un programme même lorsque celui effectue des tâches. Dans le deuxième cas, la réification des types abstraits permet au programme permet d'examiner et de modifier la structure de types complexes. On peut ainsi, par exemple, mettre au point des algorithmes génériques de sérialisation.
- La `réflexion comportementale` (ou réflexion de comportement) concerne plus particulièrement l'exécution du programme et l'environnement du programme. Par ce type de réflexion, un programme a moyen de savoir comment est-ce qu'il est interprété et a la possibilité de modifier sa façon d'être exécuté, en intervenant sur les structures de données de l'évaluateur du programme et sur l'évaluateur lui-même. De cette manière, le programme peut par exemple obtenir des informations sur son implémentation ou même s'auto-réorganiser afin de s'adapter au mieux à un « environnement ».

Le langage OCL [AV01b] dispose aussi de mécanismes réflexifs pour représenter des contraintes sur les modélisations UML.

## 4.2 Réflexion et programmation objet

En programmation orientée objet, l'architecture réflexive est implémentée par le concept des méta-objets. Ceux-ci représentent des éléments des programmes orientés objets comme les classes, les messages et les fonctions génériques. La manipulation ces métaobjets se fait

par un protocole à méta-objets qui permet de décider des comportements du langage. CLOS est le premier langage à avoir implanté un protocole à méta-objets.

Les langages suivant supportent la réflexion : SmallTalk, CLOS, Python, Ruby, Java, Objective-C, PHP, depuis la version 5, certain langages qui fonctionnent sur l'architecture .NET (comme le C#).

Dans les langages ou il n'y a pas de disctinction entre la compilation et l'exécution (Lisp par exemple), il n'y a pas de différence entre l'interprétation du code et la réflexion.

Squeak [BD01] comme tous les Smalltalks [BS96] est un langage réflexif. Etre réflexif pour un langage signifie permettre *l'introspection*, i.e., permettre d'analyser les structures de données qui définissent le langage lui-même et *l'intercession*, i.e., permettre de modifier depuis le langage lui-même sa sémantique et son comportement. Notons que bien que Java définisse une interface réflexive, il n'est pas réflexif et seulement permettant une introspection limitée. (source `http://www.iam.unibe.ch/ ducasse/` )

L'exemple suivant est écrit en Java :

```
// Sans utiliser la réflexion
Foo foo = new Foo ();
foo. hello  ();

// En utilisant la réflexion
Class cl  = Class.forName ("Foo");
Method method = cl.getMethod ("hello", null);
method.invoke (cl.newInstance (),  null );
```

Les deux morceaux de code créent une instance de la classe `Foo` et appellent leur méthode `hello`. Dans le premier programme, le nom des classes et des méthodes est codé en dur, il n'est pas possible d'utiliser le nom d'une autre classe. Dans le second programme, en revanche, le nom des classe et des méthodes peut varier à l'execution.

## 4.3 Méta-objets en Smalltalk

La notion de **méta-objet** apparaît dès qu'une classe est considérée comme un objet. La classe a les droits d'un objet :

1. Une classe est activable par envoi de message. Les opérations liées aux classes deviennent alors des méthodes de classe. Parmi ces méthodes, deux sont particulièrement intéressantes :
   (a) La méthode d'instantiation `new`.
   (b) La méthode de définition d'une méthode d'instance `define-method`.
2. Une classe est instance d'une classe qui définit sa description abstraite et son comportement (notamment sa relation d'héritage). La classe d'une classe est appelée `méta-classe`. Appelons `MétaClasse`, la méta-classe par défaut, du système, définie approximativement dans la définition 4.1. Nous supposons des types `Champ`, `Invariant`, `Méthode`, `Classe` qui peuvent être des classes. L'identificateur `UneClasse`, à gauche du symbole d'affectation ( :=), désigne une variable, qui a pour valeur la nouvelle classe créée. Les `"[, ]"` symbolisent les listes (pas de variables de classes ici). L'invariant par défaut est `vrai`. Les méthodes d'instance seront définies ultérieurement.

**Définition 4.1 (méta-classe)** *Une **méta-classe** est une classe dont les instances sont des classes.*

```
structure     champs : Liste[Champ]
              contrainte : Invariant
```

```
            variablesDeClasse : Liste[Champ]
            méthodes : Dictionnaire[Méthode]
            méthodesDeClasse : Dictionnaire[Méthode]
méthodes    new : String Liste[Champ] Invariant Liste[Champ] → Classe
```

La présence de méta-classes rend le langage souvent plus uniforme, en particulier la création des instances peut se faire par envoi de message. Un autre avantage est de pouvoir paramétrer et redéfinir le comportement du système objet. La présence de méta-classes s'accompagne souvent d'un **protocole des méta-objets**. Celui-ci donne une définition quasi-réflexive des classes et des méta-classes primitives du système. Il permet de plus une reconfiguration partielle du système. Ceci est possible par la définition de nouvelles méta-classes ayant des comportements différents de ceux prédéfinis, par exemple, au niveau de l'héritage ou de la création et de l'initialisation des instances. Les méta-objets n'existent pas dans tout les langages de classes, par exemple il n'y en a pas en Eiffel, en SCOOPS ou en C++. Dans un système avec méta-objets comme ObjVlisp [Coi87] la distinction entre classe et méta-classe disparaît dans la mesure où une classe est aussi une instance. Les variables et méthodes de classes ne seront plus rangées dans la structure de la classe comme dans la définition 4.1 mais dans la méta-classe.

En Smalltalk, chaque classe `Classe` est instance d'une seule méta-classe appelée `Classe class`. La méta-classe contient la structure (variables de classes, variables d'instance de la méta-classe) et le protocole de classe (méthodes de classes). Dans le protocole de la méta-classe, figurent les méthodes d'instanciation, les méthodes d'initialisations des variables de la structure de la classe et des méthodes globales accessibles sans créer d'instances de la classe (par exemple écriture sur la console, lancement du ramasse-miette, etc.).

## 4.4 Réflexion en Java

Pour rédiger cette partie, nous nous appuyons sur les références suivantes ([Cla03] p. 255), ([SM03] p. 613), et `http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap023.htm`

Depuis la version 1.1 de java, il est possible de créer et de gérer dynamiquement des objets. L'introspection est un mécanisme qui permet de connaître le contenu d'une classe dynamiquement. Il permet notamment de savoir ce que contient une classe sans en avoir les sources : classe parente, interface implémentée, champs, méthodes, ... De plus l'introspection traite les conventions de codage définies en Java. Ainsi, à partir des méthodes exposées par une classe on peut aussi en déduire ses propriétés et ses événements. On parle aussi de **RTTI** (*Run-Time Type Information*) c'est-à-dire d'informations de type ajoutées aux objets en mémoire par certains compilateurs de langage de haut niveau (à commencer par C++) et qui permettent par exemple de vérifier le typage des données lors de l'exécution du programme. Ces mécanismes sont largement utilisés dans des outils tels que les debogeurs, les inspecteurs d'objets et les environnement de développement (IDE) qui doivent faire une analyse des objets qu'ils manipulent en utilisant ces mécanismes.

Ces informations descriptives sont stockées dans les méta-classes. Une méta-classe est donc un type qui en identifie un autre. En fait, les concepts de méta-classes et de réflexion sont les clés de voûte de tout le système Java (Java Bean, JNI, RMI, ...). Pour que l'on puisse manipuler les méta-classes, il faut que le compilateur joigne la table des symboles au fichier de byte code générer. C'est ce que ne sait pas faire C++ : les méta-classes en sont donc plus difficiles à fournir (certain compilateurs y arrivent malgré tout en mode debug).

De façon statique (c'est-à-dire pas pendant une exécution), l'utilitaire `javap` participe à l'extraction d'informations. Il est disponible dans le SDK et consiste en un décompilateur élémentaire.

Pour pouvoir utiliser la réflexion dynamiquement, l'environnement Java fournit une API de réflexion avec la classe `java.lang.Class`, ainsi qu'un bon nombre d'autres classes utilitaires

localisées dans le package `java.lang.reflect`. Les classes de l'API de réflexion sont présentées dans la figure 23 et figure 24. A titre d'exemple, le programme suivant affiche l'ensemble des attributs d'une classe en précisant si, pour chaque attribut, son type est scalaire ou s'il s'agit d'un agrégat.

```
import java.lang. reflect .*;

public class MetaDonnees {
    public static void main(String[] args) {
        Class metaClass = javax.swing.JButton.class;
        /* ou Class metaClass = new JButton().getClass(); */

        Field attributes [] = metaClass.getFields();
        for (int i=0; i<attributes.length; i++) {
            boolean scalar = attributes[i].getType().isPrimitive();
            System.out.print(scalar ? "Scalar" : "Object");
            System.out.print(" : " + attributes[i].getName() + " de type ");
            System.out.println(attributes[i].getType());
        }
    }
}
//source : http ://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/Io/
```

### Illustration

Pour illustrer les différents mécanismes, nous prenons un exemple qui va construire une classe qui proposera un ensemble de méthodes pour obtenir des informations sur une classe. (source http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap023.htm )

Voici le début de cette classe qui attend dans son constructeur une chaîne de caractères précisant la classe sur laquelle elle va travailler.

```
import java.util.*;
import java.lang.reflect.*;
public class ClasseInspecteur {
  private Class classe;
  private String nomClasse;
  public ClasseInspecteur(String nomClasse) {
    this.nomClasse = nomClasse;
    try {
      classe = Class.forName(nomClasse);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

### API de réflexion

La classe `Class` est la classe centrale sur laquelle repose la réflexivité. Les instances de la classe Class sont des objets représentant les classes du langage. Il y aura une instance représentant chaque classes utilisées : par exemple la classe `String`, la classe `Frame`, la classe `Class`, etc ... . Ces instances sont créées automatiquement par la machine virtuelle lors du chargement de la classe. Il est ainsi possible de connaître les caractéristiques d'une classe de façon dynamique en utilisant les méthodes de la classe Class. La classe `Class` est définie dans le package `java.lang`. La classe Class permet :

- de décrire une classe ou une interface par introspection : obtenir son nom, sa classe mère, la liste de ces méthodes, de ses variables de classe, de ses constructeurs et variables d'instances, etc ...
- d'agir sur une classe en envoyant, à un objet Class des messages comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée

La classe `Class` ne possède pas de constructeur public mais il existe plusieurs façons d'obtenir un objet de la classe Class. La méthode `getClass()` définit dans la classe `Object` renvoie une instance de la classe `Class`. Par héritage, tout objet java dispose de cette méthode.

```
package introspection;

public class TestGetClass {
   public static void main(java.lang.String[] args) {
      String chaine = "test";
      Class classe = chaine.getClass();
      System.out.println("classe de l'objet chaine = "+classe.getName());
   }
}
```

La classe `Class` possède une méthode statique `forName()` qui permet à partir d'une chaîne de caractères désignant une classe d'instancier un objet de cette classe et de renvoyer un objet de la classe `Class` pour cette classe. Cette méthode peut lever l'exception `ClassNotFoundException`.

```
public class TestForName {
   public static void main(java.lang.String[] args) {
      try {
         Class classe = Class.forName("java.lang.String");
         System.out.println("classe de l'objet chaine = "+classe.getName());
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Il est possible d'avoir un objet de la classe `Class` en écrivant `type.class` ou type est le nom d'une classe.

```
package introspection;
public class TestClass {
   public static void main(java.lang.String[] args) {
      Class c = Object.class;
      System.out.println("classe de Object = "+c.getName());
   }
}
```

Voici les principales méthodes de la classe `Class` :

| **Méthodes** | **Rôle** |
|---|---|
| static Class forName(String) | Instancie un objet de la classe dont le nom est fourni en paramètre et renvoie un objet Class la représentant |
| Class[] getClasses() | Renvoie les classes et interfaces publiques qui sont membres de la classe |
| Constr[] getConstructors() | Renvoie les constructeurs publics de la classe |
| Class[] getDeclaredClasses() | Renvoie un tableau des classes définies comme membre dans la classe |
| Constructor[] getDeclared | Renvoie tous les constructeurs de la classe |
| Constructors()<br>Field[] getDeclaredFields() | Renvoie un tableau de tous les attributs définis dans la classe |
| Method getDeclared<br>Methods() | Renvoie un tableau de toutes les méthodes |
| Field getFields() | Renvoie un tableau des attributs publics |
| Class[] getInterfaces() | Renvoie un tableau des interfaces implémentées par la classe |
| Method getMethod() | Renvoie un tableau des methodes publiques de la classe incluant celles héritées |
| int getModifiers() | Renvoie un entier qu'il faut décoder pour connaître les modificateurs de la classe |
| Package getPackage() | Renvoie le package de la classe |
| Classe getSuperClass() | Renvoie la classe mère de la classe |
| boolean isArray() | Indique si la classe est un tableau |
| boolean IsInterface() | Indique si la classe est une interface |
| Object newInstance() | Permet de créer une nouvelle instance de la classe |

TABLEAU IX– *Protocole de la classe* `java.lang.Class`

Constr = Constructor

La méthode `getSuperClass()` retourne une instance de la classe Class représentant la superclasse si elle existe sinon elle retourne null. Pour obtenir toute la hiérarchie d'une classe il suffit d'appeler successivement cette méthode sur l'objet qu'elle a retourné.

```
public Vector getClassesParentes() {
  Vector cp = new Vector();
  Class sousClasse = classe;
  Class superClasse;

  cp.add(sousClasse.getName());
  superClasse = sousClasse.getSuperclass();
  while (superClasse != null) {
    cp.add(0,superClasse.getName());
    sousClasse = superClasse;
    superClasse = sousClasse.getSuperclass();
  }
  return cp;
}
```

La méthode `getModifiers()` retourne un entier représentant les modificateurs de la classe. Pour décoder cette valeur, la classe `Modifier` possède plusieurs méthodes qui attendent cet entier en paramètre et qui retourne un booléen selon leur fonction : `isPublic`, `isAbstract`, `isFinal` ...

La classe `Modifier` ne contient que des constantes et des méthodes statiques qui permettent de déterminer les modificateurs d'accès :

| Méthodes | Rôle |
|---|---|
| boolean isAbstract(int) | Renvoie true si le paramètre contient le modificateur abstract |
| boolean isFinal(int) | Renvoie true si le paramètre contient le modificateur final |
| boolean isInterface(int) | Renvoie true si le paramètre contient le modificateur interface |
| boolean isNative(int) | Renvoie true si le paramètre contient le modificateur native |
| boolean isPrivate(int) | Renvoie true si le paramètre contient le modificateur private |
| boolean isProtected(int) | Renvoie true si le paramètre contient le modificateur protected |
| boolean isPublic(int) | Renvoie true si le paramètre contient le modificateur public |
| boolean isStatic(int) | Renvoie true si le paramètre contient le modificateur static |
| boolean isSynchronized(int) | Renvoie true si le paramètre contient le modificateur synchronized |
| boolean isTransient(int) | Renvoie true si le paramètre contient le modificateur transient |
| boolean isVolatile(int) | Renvoie true si le paramètre contient le modificateur volatile |

TABLEAU X– *Protocole de la classe* `java.lang.reflect.Modifier`

Ces méthodes étant statiques il est inutile d'instancier un objet de type `Modifier` pour utiliser ces méthodes.

```
public Vector getModificateurs() {
  Vector cp = new Vector();
  int m = classe.getModifiers();
  if (Modifier.isPublic(m))
    cp.add("public");
  if (Modifier.isAbstract(m))
    cp.add("abstract");
  if (Modifier.isFinal(m))
    cp.add("final");
  return cp;
}
```

La méthode `getInterfaces()` retourne un tableau d'instances de `Class` contenant les interfaces implémentées par la classe.

```
public Vector getInterfaces() {
  Vector cp = new Vector();
  Class[] interfaces = classe.getInterfaces();
  for (int i = 0; i < interfaces.length; i++) {
    cp.add(interfaces[i].getName());
  }
  return cp;
}
```

La méthode getFields() retourne les attributs public de la classe sous forme d'un tableau d'objet de type `Field`. La méthode `getField()` attend en paramètre un nom d'attribut et retourne un objet de type `Field` si celui ci est défini dans la classe ou dans une de ses superclasses. Si la classe ne contient pas d'attribut dont le nom correspond au paramètre fourni, la méthode `getField()` lève une exception de la classe `NoSuchFieldException`.

La classe `Field` représente un attribut d'une classe ou d'une interface et permet d'obtenir des informations cet attribut. Elle possède plusieurs méthodes :

| Méthodes | Rôle |
|---|---|
| String getName() | Retourne le nom de l'attribut |
| Class getType() | Retourne un objet de type Class qui représente le type de l'attribut |
| Class get DeclaringClass() | Retourne un objet de type Class qui représente la classe qui définit l'attribut |
| int getModifiers() | Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes static de la classe Modifier. |
| Object get (Object) | Retourne la valeur de l'attribut pour l'instance de l'objet fourni en paramètre. Il existe aussi plusieurs méthodes getXXX() ou XXX représente un type primitf et qui la renvoie la valeur dans ce type. |

TABLEAU XI– *Protocole de la classe `java.lang.reflect.Field`*

```
public Vector getChampsPublics() {
  Vector cp = new Vector();
  Field[] champs = classe.getFields();
  for (int i = 0; i < champs.length; i++)
    cp.add(champs[i].getType().getName()+" "+champs[i].getName());
  return cp;
}
```

L'exemple ci dessous présente une méthode qui permet de formater sous forme de chaîne de caractères les paramètres d'une méthode fournis sous la forme d'un tableau d'objets de type `Class`.

```
private String rechercheParametres(Class[] classes) {
  StringBuffer param = new StringBuffer("(");
  for (int i = 0; i < classes.length; i ++) {
    param.append(formatParametre(classes[i].getName()));
    if (i < classes.length - 1)
      param.append(", ");
  }
  param.append(")");
  return param.toString();
}
```

La méthode `getName()` de la classe `Class` renvoie une chaîne de caractères formatée qui précise le type de la classe. Ce type est représenté par une chaîne de caractères qu'il faut décoder pour obtenir le type. Si le type de la classe est un tableau alors la chaîne commence par un nombre de caractère '[' correspondant à la dimension du tableau. Ensuite la chaîne contient un caractère qui précise un type primitif ou un objet. Dans le cas d'un objet, le nom de la classe de l'objet avec son package complet est contenu dans la chaîne suivi d'un caractère ';'.

| Caractère | Type | Caractère | Type |
|---|---|---|---|
| B | byte | C | char |
| D | double | F | float |
| I | int | J | long |
| Lclassname; | classe ou interface | S | short |
| Z | boolean | | |

Exemple : la méthode `getName()` de la classe Class représentant un objet de type float[10][5] renvoie « [[F ». Pour simplifier les traitements, la méthode `formatParametre()` ci-dessous retourne une chaîne de caractères qui décode le contenu de la chaîne retournée par la méthode `getName()` de la classe `Class`.

```
private String formatParametre(String s) {
  if (s.charAt(0) == '[') {
    StringBuffer param = new StringBuffer("");
    int dimension = 0;
    while (s.charAt(dimension) == '[') dimension++;
    switch(s.charAt(dimension)) {
      case 'B' : param.append("byte");break;
      case 'C' : param.append("char");break;
      case 'D' : param.append("double");break;
      case 'F' : param.append("float");break;
      case 'I' : param.append("int");break;
      case 'J' : param.append("long");break;
      case 'S' : param.append("short");break;
      case 'Z' : param.append("boolean");break;
      case 'L' : param.append(s.substring(dimension+1,s.indexOf(";")));
    }
    for (int i =0; i < dimension; i++)
      param.append("[]");
    return param.toString();
  }
  else return s;
}
```

La méthode `getConstructors()` retourne un tableau d'objet de type `Constructor` contenant les constructeurs de la classe.

La classe `Constructor` représente un constructeur d'une classe. Elle possède plusieurs méthodes :

| **Méthodes** | **Rôle** |
|---|---|
| String getName() | Retourne le nom du constructeur |
| Class[] getExceptionTypes() | Retourne un tableau de type Class qui représente les exceptions qui peuvent être propagées par le constructeur |
| Class[] getParametersType() | Retourne un tableau de type Class qui représente les paramètres du constructeur |
| int getModifiers() | Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes static de la classe Modifier. |
| Object new Instance(Object[]) | Instancie un objet en utilisant le constructeur avec les paramètres fournis à la méthode |

TABLEAU XII– *Protocole de la classe `java.lang.reflect.Constructor`*

```
public Vector getConstructeurs() {
  Vector cp = new Vector();
  Constructor[] constructeurs = classe.getConstructors();
  for (int i = 0; i < constructeurs.length; i++) {
    cp.add(rechercheParametres(constructeurs[i].getParameterTypes()));
  }
  return cp;
}
```

L'exemple ci-dessus utilise la méthode `rechercherParametres()` définie précédemment pour simplifier les traitements.

Pour consulter les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message `getMethod()`, qui renvoie les méthodes publiques qui sont déclarées dans la classe et qui sont héritées des classes mères. Elle renvoie un tableau d'instances de la classe `Method` du package

`java.lang.reflect`. Une méthode est caractérisée par un nom, une valeur de retour, une liste de paramètres, une liste d'exceptions et une classe d'appartenance. La classe `Method` contient plusieurs méthodes :

| **Méthodes** | **Rôle** |
|---|---|
| Class[] getParameterTypes | Renvoie un tableau de classes représentant les paramètres. |
| Class getReturnType | Renvoie le type de la valeur de retour de la méthode. |
| String getName() | Renvoie le nom de la méthode |
| int getModifiers() | Renvoie un entier qui représentent les modificateur d'accès |
| Class[] getExceptionTypes | Renvoie un tableau de classes contenant les exceptions propagées par la méthode |
| Class getDeclaringClass | Renvoie la classe qui définit la méthode |

TABLEAU XIII– *Protocole de la classe `java.lang.reflect.Method`*

L'exemple ci-dessous utilise les méthodes `formatParametre()` et `rechercherParametres()` définies précédemment pour simplifier les traitements.

```
public Vector getMethodesPubliques() {
  Vector cp = new Vector();
  Method[] methodes = classe.getMethods();
  for (int i = 0; i < methodes.length; i++) {
    StringBuffer methode = new StringBuffer();
    methode.append(formatParametre(methodes[i].getReturnType().getName()));
    methode.append(" ");
    methode.append(methodes[i].getName());
    methode.append(rechercheParametres(methodes[i].getParameterTypes()));
    cp.add(methode.toString());
  }
  return cp;
}
```

Pour consulter toutes les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message `getDeclareMethods()`, qui renvoie toutes les méthodes qui sont déclarées dans la classe et qui sont héritées des classes mères quelque soit leur accessibilité. Elle renvoie un tableau d'instances de la classe `Method`.

```
public Vector getMethodes() {
  Vector cp = new Vector();
  Method[] methodes = classe.getDeclareMethods();
  for (int i = 0; i < methodes.length; i++) {
    StringBuffer methode = new StringBuffer();
    methode.append(formatParametre(methodes[i].getReturnType().getName()));
    methode.append(" ");
    methode.append(methodes[i].getName());
    methode.append(rechercheParametres(methodes[i].getParameterTypes()));
    cp.add(methode.toString());
  }
  return cp;
}
```

On peut aussi définir dynamiquement des objets grâce à la classe `Class` ou exécuter dynamiquement une méthode (en utilisant la méthode `invoke`).

Consulter aussi http://ricky81.developpez.com/tutoriel/java/api/reflection/ pour d'autres explications détaillées et d'autres exemples tels que le suivant, qui consulte les informations à visibilité publique d'une classe en affichant les informations sur la console `System.out`.

```
import java.lang. reflect .*;
public class Explorateur
{
    public Explorateur()
    {
    }
    public void explorerChamps(Object o)
    {
        Field [] f = null;
        Class c = null;
        c = o.getClass();
        f = c.getFields ();
        consulterChamps(f,o);
    }
    public void explorerMethodes(Object o)
    {
        Method[] m = null;
        Class c = null;
        c = o.getClass();
        m= c.getMethods();
        consulterMethodes(m);
    }
    private void consulterChamps(Field[] f, Object o)
    {
        for(int i=0;i<f.length;++i)
        {
            System.out.print(Modifier.toString(f[i].getModifiers ()));
            System.out.print(" ");
            System.out.print(f[i].getType().getName());
            System.out.print(" ");
            System.out.print(f[i].getName());
            System.out.print(" = ");
            try
            {
                System.out.println(f[i].get(o));
            }
            catch (IllegalAccessException e)
            {
              System.out.println("Valeur non consultable");
            }
        }
    }
    private void consulterMethodes(Method[] m)
    {
        Class[] params = null;
        for(int i=0;i<m.length;++i)
        {
            System.out.print(Modifier.toString(m[i].getModifiers ()));
            System.out.print(" ");
            System.out.print(m[i].getReturnType().getName());
            System.out.print(" ");
            System.out.print(m[i].getName());
            System.out.print("(");
            params = m[i].getParameterTypes();
            for(int j=0;j<params.length;++j)
            {
              System.out.print(params[j].getName());
```

```
            }
            System.out.println(")");
        }
    }
}
```

# Chapitre 6

# Quelques API importantes

Dans ce chapitre nous abordons quelques bibliothèques de classes utiles pour développer des applications pour le Web (*applets*), la concurrence (*threads*), l'interface native (*JNI*), les bases de données (*JDBC*), et le test (*JUnit*).

## 1 Applets

Dans cette section nous abordons une bibliothèques de classes qui a fait le succès de Java, les (*applets*), le code transportable, très utile pour développer des applications pour le Web. Le fonctionnement des applets nécessite une prise en charge par le navigateur et une machine virtuelle java (JRE). Deux conditions sont nécessaires pour véhiculer du code à travers le réseau : simplicité et sécurité.

**Définition 1.1 (Applet)** *Une applet est une application dont un navigateur peut télécharger le bytecode à travers le réseau pour l'exécuter localement (sur la machine virtuelle du poste client) [Cha03].*

Les *applets* servent à animer des pages web (dans des zones visuelles délimitées) ou rendre des services. Néanmoins les possibilités sont limitées pour préserver la sécurité de l'utilisateur et éviter des intrusions néfastes dans son système. Noter qu'une applet est aussi visualisable dans l'`appletviewer` du JDK. L'API des applets s'organise autour de la classe `java.applet.Applet`, une sous-classe importante pour les IHM est `java.swing.JApplet`. Un aperçu de la hiérarchie est présenté dans la section 2.8 du chapitre 3. Contrairement à un programme classique, et pour pouvoir l'intégrer à un programme classique, le lancement d'une *applet* se fait par la méthode `init` et non la méthode `main`.

La mise en place d'une *applet* se fait en deux parties : écriture du code par héritage de la classe, lancement de l'*applet* dans un environnement d'exécution.

### 1.1 Créer des *applets*

Pour créer une *applet*, il suffit de définir une sous-classe de la classe `Applet` du paquetage `java.applet`.

#### La classe *Applet*

Une classe dérivée de la classe `java.applet.Applet` hérite de méthodes qu'il faut redéfinir en fonction des besoins et doit être déclarée public pour fonctionner. En général, il n'est pas nécessaire de faire un appel explicite aux méthode `init()`, `start()`, `stop()` et `destroy()` : le navigateur se charge d'appeler ces méthodes en fonction de l'état de la page HTML contenant l'applet.

La méthode `init()` permet l'initialisation de l'applet : elle n'est exécutée qu'une seule et unique fois après le chargement de l'applet.

La méthode `start()` est appelée automatiquement après le chargement et l'initialisation (via la méthode `init()`) lors du premier affichage de l'applet.

Le navigateur appelle automatiquement la méthode `stop()` lorsque l'on quitte la page HTML. Elle interrompt les traitements de tous les processus en cours.

La méthode `destroy()` est appelée après l'arrêt de l'applet ou lors de l'arret de la machine virtuelle. Elle libère les ressources et détruit les threads restants

La méthode `update()` est appelée à chaque rafraîchissement de l'écran ou appel de la méthode `repaint()`. Elle efface l'écran et appelle la méthode `paint()`. Ces actions provoquent souvent des scintillements. Il est préférable de redéfinir cette méthode pour qu'elle n'efface plus l'écran : `public void update(Graphics g)  paint (g);`.

La méthode `paint()` permet d'afficher le contenu de l'applet à l'écran. Ce refraîchissement peut être provoqué par le navigateur ou par le système d'exploitation si l'ordre des fenêtres ou leur taille ont été modifiés ou si une fenêtre recouvre l'applet.
`public void paint(Graphics g)`.

La méthode repaint() force l'utilisation de la méthode paint().

Il existe des méthodes dédiées à la gestion de la couleur de fond et de premier plan

La méthode `setBackground(Color)`, héritée de Component, permet de définir la couleur de fond d'une applet. Elle attend en paramètre un objet de la classe `Color`.

La méthode `setForeground(Color)` fixe la couleur d'affichage par défaut. Elle s'applique au texte et aux graphiques. Les couleurs peuvent être spécifiées de trois manières différentes :

1. Utiliser les noms standard prédéfinis `Color.nomDeLaCouleur` Les noms predéfinis de la classe Color sont : black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow
2. utiliser 3 nombres de type entier représentant le RGB (Red,Green,Blue : rouge,vert, bleu) Exemple : `setBackground(150,200,250);`.
3. utiliser 3 nombres de type float utilisant le systeme HSB (Hue, Saturation, Brightness : teinte, saturation, luminance). Ce système est moins répandu que le RGB mais il permet notamment de modifier la luminance sans modifier les autres caractéristiques Exemple : `setBackground(0.0,0.5,1.0);`. Dans ce cas 0.0,0.0,0.0 représente le noir et 1.0,1.0,1.0 représente le blanc.

L'origine des coordonnées en Java est le coin supérieur gauche. Elles s'expriment en pixels avec le type int. La détermination des dimensions d'une applet se fait avec la méthode `getSize()` de la façon suivante :

```
public void paint(Graphics g) {
   super.paint(g);
   Dimension dim = getSize();
   int applargeur = dim.width;
   int apphauteur = dim.height;
   g.drawString("width = "+applargeur,10,15);
   g.drawString("height = "+apphauteur,10,30);
}
```

Les méthodes `getCodeBase()` et `getDocumentBase()` renvoient respectivement l'emplacement de l'applet sous forme d'adresse Web ou de dossier et l'emplacement de la page HTML qui contient l'applet. Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   g.drawString("CodeBase = "+getCodeBase(),10,15);
   g.drawString("DocumentBase = "+getDocumentBase(),10,30);
```

```
}
```

La méthode `showStatus()` affiche une message dans la barre de status de l'applet. Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   showStatus("message à afficher dans la barre d'état");
}
```

La méthode `getAppletInfo()` permet de fournir des informations concernant l'auteur, la version et le copyright de l'applet. Exemple :

```
static final String appletInfo = " test applet : auteur, 1999 \n\nCommentaires";

public String getAppletInfo() {
   return appletInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'`appletviewer`.

La méthode `getParameterInfo()` permet de fournir des informations sur les paramètres reconnus par l'applet. Le format du tableau est le suivant :
`{ {nom du paramètre, valeurs possibles, description} , ... }`
Exemple :

```
static final String [][] parameterInfo =
{ {"texte1", "texte1", " commentaires du texte 1" } ,
  {"texte2", "texte2", " commentaires du texte 2" } };

public String [][] getParameterInfo() {
   return parameterInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'`appletviewer`.

La méthode `getGraphics()` retourne la zone graphique d'une applet : utile pour dessiner dans l'applet avec des méthodes qui ne possèdent pas le contexte graphique en paramètres (ex : mouseDown ou mouseDrag).

La méthode `getGraphics()` permet l'accès à des fonctionnalités du navigateur.

La méthode `setStub()` permet d'attacher l'applet au navigateur.

Parmi les interfaces utiles pour les applets, on trouve 17.3.1. L'interface Runnable l'interface `Runnable` qui fournit le comportement nécessaire à un applet pour devenir un thread (avec les méthodes `start()` et `stop()` pour démarrer et arrêter un thread, notamment pour permettre de limiter l'usage des ressources machines lorsque la page contenant l'applet est inactive) et l'interface `ActionListener` qui permet à l'applet de répondre aux actions de l'utilisateur avec la souris (par exemple la méthode `actionPerformed()`).

### Exemple

Prenons l'exemple de l'horloge tiré de http://chgi.developpez.com/java/applet/

Ce code utilise l'interface `Runnable` pour le *thread* (voir section 2). Le rafraîchissement du dessin se fait toutes les secondes par l'appel de la méthode `repaint()`. Le *thread* est utilisé pour éviter de bloquer l'affichage de l'applet pendant la pause `Thread.sleep(1000)`. Le dessin de l'horloge se fait sur une image en mémoire `imgTmp`. Elle est dessinée en dernier ressort sur la surface de dessin de l'applet à l'aide de la méthode `drawImage`.

Listing 6.1 – Code de l'*applet* `Horloge`

```
package applets;
import java.awt.*;
import java.util.*; //pour Calendar
import java.applet.*;
public class Horloge extends Applet
    implements Runnable {
  Thread tr;
  Image imgTmp;
  Graphics gTmp;

  public Horloge() {
    if (tr == null) {
        tr = new Thread(this);
        tr.start();
        }
  }

  public void run(){
    while (true) {
      repaint();
      try { Thread.sleep(1000); }
      catch(InterruptedException e){}
    }
  }

  public void init() {
    Dimension dim = getSize();
    imgTmp = createImage(dim.width, dim.height);
    gTmp = imgTmp.getGraphics();
    setBackground(Color.white);
  }

  public void update(Graphics g) {
    paint(g);
  }

  public void paint(Graphics gsp) {
    Calendar d = Calendar.getInstance();
    int [] tPolygonX = new int[4];
    int [] tPolygonY = new int[4];
    Color cFond = new Color(232,232,232);
    double n,z,u,x0,y0,x1,y1,x2,y2,x3,y3;
    int h = d.get(Calendar.HOUR);
    int m = d.get(Calendar.MINUTE);
    int s = d.get(Calendar.SECOND);
    gTmp.setPaintMode();
    gTmp.setColor(cFond);
    gTmp.fillOval(2,2,118,118);
    gTmp.setColor(Color.black);
    gTmp.drawOval(2,2,118,118);
    gTmp.drawString("12",55,16);
    gTmp.drawString("6",58,116);
    gTmp.drawString("3",108,66);
    gTmp.drawString("9",8,66);
    //Aiguille des secondes
    n = s*200/60;
    z = n/100*Math.PI;
    u = (n+50)/100*Math.PI;
    x0 = Math.sin(z)*50;
    y0 = -Math.cos(z)*50;
    x1 = -Math.sin(z)*10;
    y1 = Math.cos(z)*10;
    x2 = Math.sin(u)*2;
    y2 = -Math.cos(u)*2;
    x3 = -Math.sin(u)*2;
    y3 = Math.cos(u)*2;
    tPolygonX[0] = (int)x1+60;
    tPolygonX[1] = (int)x2+60;
    tPolygonX[2] = (int)x0+60;
    tPolygonX[3] = (int)x3+60;
    tPolygonY[0] = (int)y1+60;
    tPolygonY[1] = (int)y2+60;
    tPolygonY[2] = (int)y0+60;
    tPolygonY[3] = (int)y3+60;
    gTmp.setColor(Color.red);
    gTmp.fillPolygon(tPolygonX,
            tPolygonY, 4);
    gTmp.setColor(Color.black);
    gTmp.drawPolygon(tPolygonX,
            tPolygonY, 4);
    //Aiguille des minutes
    n = m*200/60;
    z = n/100*Math.PI;
    u = (n+50)/100*Math.PI;
    x0 = Math.sin(z)*50;
    y0 = -Math.cos(z)*50;
    x1 = -Math.sin(z)*10;
    y1 = Math.cos(z)*10;
    x2 = Math.sin(u)*4;
    y2 = -Math.cos(u)*4;
    x3 = -Math.sin(u)*4;
    y3 = Math.cos(u)*4;
    tPolygonX[0] = (int)x1+60;
    tPolygonX[1] = (int)x2+60;
    tPolygonX[2] = (int)x0+60;
    tPolygonX[3] = (int)x3+60;
    tPolygonY[0] = (int)y1+60;
    tPolygonY[1] = (int)y2+60;
    tPolygonY[2] = (int)y0+60;
    tPolygonY[3] = (int)y3+60;
    gTmp.setColor(Color.yellow);
    gTmp.fillPolygon(tPolygonX, tPolygonY, 4);
    gTmp.setColor(Color.black);
    gTmp.drawPolygon(tPolygonX, tPolygonY, 4);
    //Aiguille des heures
    n = h*200/12 + m*200/60/12;
    z = n/100*Math.PI;
    u = (n+50)/100*Math.PI;
    x0 = Math.sin(z)*35;
    y0 = -Math.cos(z)*35;
    x1 = -Math.sin(z)*10;
    y1 = Math.cos(z)*10;
    x2 = Math.sin(u)*4;
    y2 = -Math.cos(u)*4;
```

```
        x3 = -Math.sin(u)*4;
        y3 = Math.cos(u)*4;

        tPolygonX[0] = (int)x1+60;
        tPolygonX[1] = (int)x2+60;
        tPolygonX[2] = (int)x0+60;
        tPolygonX[3] = (int)x3+60;
        tPolygonY[0] = (int)y1+60;
        tPolygonY[1] = (int)y2+60;
        tPolygonY[2] = (int)y0+60;
        tPolygonY[3] = (int)y3+60;
        gTmp.setColor(Color.green);
        gTmp.fillPolygon(tPolygonX, tPolygonY, 4);
        gTmp.setColor(Color.black);
        gTmp.drawPolygon(tPolygonX, tPolygonY, 4);
        gsp.drawImage(imgTmp,0,0,this);
    }
}
```

Le constructeur crée et démarre le thread par sa méthode `start`. La méthode `run` est une boucle infinie qui appelle la méthode `repaint` toutes les secondes afin de rafraîchir l'affichage. Le code de cette méthode s'exécute en parallèle du reste du code, c'est le code du thread, invoqué par la méthode `start` du thread. Dans la méthode `init` on crée l'image en mémoire `imgTmp` et sa surface de dessin `gTmp` et on initialise la couleur de fond avec la méthode `setBackground` (La méthode `init` d'une applet est exécutée une seule fois au démarrage de l'applet.) La méthode `update` est redéfinie pour provoquer uniquement le dessin sans l'effacer au préalable, ceci afin d'améliorer l'affichage (la méthode `update` est appelée par la méthode `repaint`).

Enfin examinons le code de la méthode `paint` (dans une applet cette méthode est appelée à chaque fois que le dessin de l'applet a besoin d'être rafraîchi). On dessine sur la surface de dessin de l'image en mémoire (`gTmp`) en utilisant les méthodes de la classe `java.awt.Graphics` : `drawOval` pour le cercle, `drawString` pour les chiffres, `drawPolygon` et `fillPolygon` pour les aiguilles. Les coordonnées des aiguilles étant calculées à l'aide des fonctions trigonométriques de la classe `java.lang.Math` à partir de l'heure courante qui est donnée par la méthode `get` de la classe `java.util.Calendar`. Pour terminer l'image est affichée avec la méthode `drawImage` de la classe `java.awt.Graphics`.

## 1.2 Lancer des *applets*

Les *applets* ordinaires sont lancées localement par l'`appletviewer` du JDK. ou dans un navigateur web.

Sous Eclipse, pour exécuter le code d'une applet, le plus simple est d'activer le menu `Run As > Java Applet` ou le menu contextuel `Run As > Java Applet` dans la fenêtre d'édition. Voici le résultat :

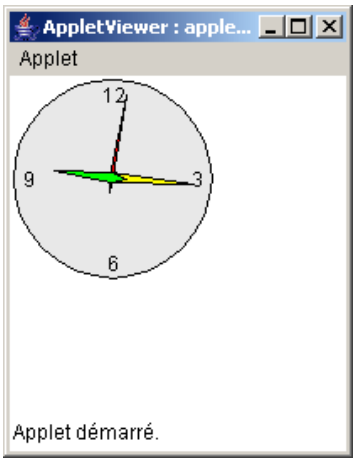



Figure 45 : *Exécution de l'applet `Horloge` par l'`appletviewer`*

Pour lancer l'applet dans un navigateur, il faut insérer la balise suivante dans une page HTML. Elle pointe sur le fichier compilé.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>horloge.html</title>
  <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
  <meta name="author" content="Pascal">
</head>
  <body>
       Une petite <b><i>applet </i>Java</b> qui donne <u>l'heure.</u><br>
       <br>
       <applet code="Horloge.class" name="Horloge" width="125"
 height="125"> Votre  navigateur n'est pas compatible java. </applet> <br>
     <br>
       Pour voir le source, cliquez
<a href="http://chgi.developpez.com/java/applet">ici</a><br>
       <br>
</body>
</html>
```

A l'affichage de la page, la lecture de la balise entraîne :

1. le (télé)chargement du code de l'applet,
2. une création de l'instance de la classe de l'applet,
3. la construction par invocation de la méthode `init` (l'applet étant une sorte de panneau AWT `java.awt.Panel` ou Swing),
4. le lancement par invocation de la méthode `start`.

A la fin, il y a

1. nettoyage par invocation de la méthode `stop`, quand l'applet disparaît de l'écran,
2. suppression par invocation de la méthode `destroy`, quand la page HTML n'est plus affichée.

**Avertissement** : *Il est important de noter qu'il doit y avoir compatibilité entre les différentes versions de Java. Ainsi, nous avons compilé le code sous Java 1.5 et l'applet ne fonctionnait pas alors qu'en prenant une version antérieure du comilateur il n'y a pas de problème. Des problèmes peuvent aussi surgir en fonction du navigateur utilisé.*

Sous *Internet Explorer*, utiliser le menu `Outils > Console Java` pour suivre l'évolution du chargement et de l'exécution de l'applet.
Sous *Netscape*, utiliser le menu `Outils > Développement Web > Console Java` pour suivre l'évolution du chargement et de l'exécution de l'applet.

« *Autant les problèmes de compatibilité entre les différentes versions de Java sont relativement simples à gérer en ce qui concerne les applications, autant c'est un vrai casse-tête [...] quand il s'agit des applets. Ici le slogan "Write once, run everywhere" relève de la pure utopie.* »[SM03].

### 1.3 Balises HTML

Le tableau suivant reprend les différents attributs que supporte le tag <APPLET>. Attention : si vous utilisez la JVM de la société Microsoft, les attributs ne fonctionneront pas tous.

- `CODE` : permet de spécifier la classe de l'applet.
- `WIDTH` et `HEIGHT` : ces deux paramètres fixent la taille qu'occupe l'applet dans le document HTML conteneur.

- `CODEBASE` : précise un répertoire pour les fichiers exécutables, si ce n'est pas le répertoire courant.
- `ARCHIVE` : si l'applet est constituée de nombreuses classes Java, son téléchargement peut alors être plus ou moins long. Afin d'optimiser ce téléchargement, vous pouvez archiver tous vos fichiers dans une archive Java.
- `NAME` : nomme l'applet (utile s'il y en a plusieurs).
- les balise HTML (`ALIGN`, `HSPACE`, `VSPACE`, ...) ont le sens usuel.
- `<PARAM Name="???" Value="???">` : ce tag, qui s'utilise entre les tag `<APPLET>` et `</APPLET>` permet de pouvoir passer des valeurs d'initialisation à l'applet. Cette dernière récupère ces valeurs en utilisant l'instruction `this.getParameter("???")`. Dans cet exemple, `this` référence, bien entendu, l'applet et ? ? ? doit être en accord avec le paramètre `Name` du tag <PARAM>.

### 1.4 Sécurité

En général, une *applet* est une application Java hébergée sur une machine distante (un serveur Web) et qui s'exécute, après chargement, sur la machine client équipée d'un navigateur. Ce navigateur contrôle les accès de l'applet aux ressources locales et ne les autorisent pas systématiquement : chaque navigateur définit sa propre règle.

Le modèle classique de sécurité pour l'exécution des applets, recommandé par Sun, distingue deux types d'applets : les applets non dignes de confiance (*untrusted*) qui n'ont pas accès aux ressources locales et externes, les applets dignes de confiance (*trusted*) qui ont l'accès. Dans ce modèle, une applet est par défaut *untrusted*.

La signature d'une applet permet de désigner son auteur et de garantir que le code chargé par le client est bien celui demandé au serveur. Cependant, une applet signée n'est pas forcement digne de confiance.

### 1.5 Une applet avec interaction

Les applets ont par défaut un menu associé à la fenêtre d'un applet *viewer*. L'exemple suivant montre une interaction entre l'usager et l'applet.

Listing 6.2 – Code de l'*applet* `Grapheur`

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Grapheur extends Applet implements ActionListener {
   Button sinus     = new Button("Sinus");
   Button cosinus   = new Button("Cosinus");
   Button tangente = new Button("Tangente");

   int courbeNumber = 0;

   int sizeX, sizeY;

   double xInf = -Math.PI , yInf = -1;
   double xSup = Math.PI , ySup = 1;

   public void init () {
      this.add(this.sinus);
               this.add(this.cosinus);
```

```
                this.add(this.tangente);

        this.sinus.addActionListener(this);
        this.cosinus.addActionListener(this);
        this.tangente.addActionListener(this);

        Dimension dim = getSize();
        this.sizeX = dim.width;
        this.sizeY = dim.height;
    }

    private int coordToPixX(double x) {
        return (int)(sizeX * (x-xInf)/(xSup-xInf));
    }

    private int coordToPixY(double y) {
        return sizeY-(int)(sizeY * (y-yInf)/(ySup-yInf));
    }

    public void paint(Graphics gc) {
        // Tracé des axes
        gc.drawLine(sizeX/2,0,sizeX/2,sizeY);
        gc.drawLine(0,sizeY/2,sizeX,sizeY/2);

        // Tracé de la courbe
        double x = xInf , y , delta=0.05;
        double oldX=x;
                double oldY= courbeNumber==0?Math.sin(x):
                             courbeNumber==1?Math.cos(x):Math.tan(x);

        while(x <= xSup) {
            y = courbeNumber==0?Math.sin(x):
                            courbeNumber==1?Math.cos(x):Math.tan(x);
            gc.drawLine(coordToPixX(oldX), coordToPixY(oldY),
                        coordToPixX(x), coordToPixY(y));
            oldX = x;   oldY = y;
            x += delta;
        }
    }

    public void actionPerformed(ActionEvent event) {
        Object source = event.getSource();

        if      (source == this.sinus)     courbeNumber=0;
        else if (source == this.cosinus)   courbeNumber=1;
        else if (source == this.tangente) courbeNumber=2;

        repaint();
    }
}
```

(source : http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/applets.html)

### 1.6 Une applet avec champs de saisie

L'exemple suivant montre une applet avec champs de saisie.

Listing 6.3 – Code de l'*applet* `AppletEmprunt`

```
/*
 * Fichier com/eteks/test/AppletEmprunt.java
 * Copyright (C) 2003 Emmanuel PUYBARET / eTeks <info@eteks.com>. All Rights Reserved.
 */
package applets;

import applets.Emprunt;

import java.text.NumberFormat;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Applet de calcul de mensualite d'un emprunt.
 */
public class AppletEmprunt extends JApplet
{
  public void init ()
  {
    // Recuperation des valeurs par defaut passees en parametre
    Double capital = new Double (getParameter ("capital"));
    Double taux    = new Double (getParameter ("taux"));
    Integer duree  = new Integer (getParameter ("duree"));
    // Creation des composants de saisie du capital, du taux et de la duree de l'emprunt
    final JFormattedTextField saisieCapital = new JFormattedTextField(capital);
    final JFormattedTextField saisieTaux   = new JFormattedTextField(taux);
    final JSpinner saisieDuree   = new JSpinner ();
    saisieDuree.setValue (duree);
    final JLabel labelMensualite = new JLabel ();
    final JLabel labelCout = new JLabel ();

    // Ajout des composants a un panneau utilisant une grille de 5 lignes x 2 colonnes
    JPanel panneauEmprunt = new JPanel (new java.awt.GridLayout (5, 2, 0, 2));
    panneauEmprunt.add (new JLabel ("Capital emprunt\u00e9 :"));
    panneauEmprunt.add (saisieCapital);
    panneauEmprunt.add (new JLabel ("Taux d'int\u00e9r\u00eat (en %) :"));
    panneauEmprunt.add (saisieTaux);
    panneauEmprunt.add (new JLabel ("Dur\u00e9e (en ann\u00e9es) :"));
    panneauEmprunt.add (saisieDuree);
    panneauEmprunt.add (new JLabel ("Mensualit\u00e9s :"));
    panneauEmprunt.add (labelMensualite);
    panneauEmprunt.add (new JLabel ("Co\u00fbt :"));
    panneauEmprunt.add (labelCout);

    // Creation du bouton qui provoque un calcul de mensualite
    JButton boutonCalculer = new JButton ("Calculer");
    boutonCalculer.addActionListener(new ActionListener ()
      {
        public void actionPerformed(ActionEvent ev)
        {
          // Recuperation des valeurs saisies
          double capital       = ((Number)saisieCapital.getValue()).doubleValue();
          double taux          = ((Number)saisieTaux.getValue()).doubleValue() / 100. / 12;
          int    nbMensualite = ((Number)saisieDuree.getValue()).intValue() * 12;
          if (nbMensualite <= 0)
```

```
            JOptionPane.showMessageDialog (AppletEmprunt.this,
                        "La dur\u00e9e doit \u00eatre positive.",
                        "Calcul Mensualit\u00e9", JOptionPane.WARNING_MESSAGE);
          else
          {
            // Calcul des mensualites et des interets
            double mensualite = Emprunt.calculerMensualite(taux, nbMensualite, capital);
            double interets   = mensualite * nbMensualite - capital;
            NumberFormat formatNombre = NumberFormat.getInstance();
            labelMensualite.setText (formatNombre.format (mensualite));
            labelCout.setText (formatNombre.format (interets));
          }
        }
      });
    // Ajout du bouton a un panneau utilisant un layout de classe java.awt.FlowLayout
    // (layout par defaut de JPanel) pour qu'il prenne ses dimensions preferees
    JPanel panneauCalcul = new JPanel ();
    panneauCalcul.add(boutonCalculer);
    // Ajout des deux panneaux au contenu de l'applet
    getContentPane().add (panneauEmprunt, BorderLayout.NORTH);
    getContentPane().add (panneauCalcul, BorderLayout.SOUTH);
  }
}
```

(source : [Puy04] p. 163)

Listing 6.4 – Code de la classe `Emprunt`

```
/*
 * Fichier com/eteks/outils/Emprunt.java
 * Copyright (C) 2003 Emmanuel PUYBARET / eTeks <info@eteks.com>. All Rights Reserved.
 */
package applets;

public class Emprunt
{
  /**
   * Calcule le montant des mensualites d'un emprunt.
   */
  public static double calculerMensualite(double taux, int nbMensualite, double capital)
  {
    // Recherche dichotomique de la valeur de la mensualite
    // Mensualite minimum (aucun interet ne sont payes)
    double mensualiteMin = capital / nbMensualite;
    // Mensualite maximum (le capital rembourse a fur et a mesure des mensualites
    // n'est pas pris en compte)
    double mensualiteMax = capital * Math.pow (1 + taux, nbMensualite) / nbMensualite;
    // Calcul du capital restant en utilisant les mensualites min et max
    double capitalRestantMin = calculerCapitalRestant(taux, nbMensualite,
                                                      capital, mensualiteMin);
    double capitalRestantMax = calculerCapitalRestant(taux, nbMensualite,
                                                      capital, mensualiteMax);
    while (Math.abs (capitalRestantMin) > 0.01)
    {
      // Calcul d'une mensualite mediane
      double mensualiteMedian = (mensualiteMin + mensualiteMax) / 2;
      double capitalRestantMedian = calculerCapitalRestant(taux, nbMensualite,
                                                           capital, mensualiteMedian);
      // Affectation de la mensualite mediane au minimum ou maximum
```

```
        if (Math.abs (capitalRestantMin) < Math.abs (capitalRestantMax))
        {
          mensualiteMax = mensualiteMedian;
          capitalRestantMax = capitalRestantMedian;
        }
        else
        {
          mensualiteMin = mensualiteMedian;
          capitalRestantMin = capitalRestantMedian;
        }
      }
      return Math.round(mensualiteMin * 100) / 100.; // Suppression des decimales superflues
    }

    // Calcule le capital restant a rembourser au bout de nbMensualite
    private static double calculerCapitalRestant (double taux, int nbMensualite,
                                                  double capital, double mensualite)
    {
      double capitalRestant = capital;
      for (int i = 0; i < nbMensualite; i++)
        // A chaque mensualite le capital restant est diminue du capital rembourse egal
        // a la mensualite payee moins les interets payes sur le capital restant avant
        capitalRestant -= mensualite - capitalRestant * taux;
      return capitalRestant;
    }
}
```

Le résultat avec appel dans une page HTML est

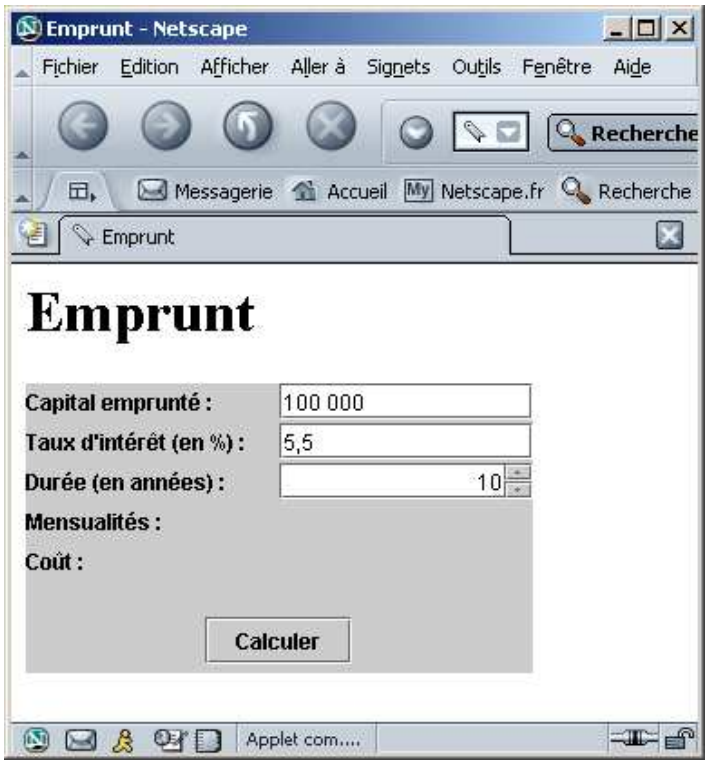



Figure 46 : *Exécution de l'applet* `AppletEmprunt` *par l'*`appletviewer`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
```

```
  <title>emprunt.html</title>
  <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
  <meta name="author" content="Pascal">
</head>
  <body>
     <h1>Emprunt</i> <br><br>
     <applet codebase="./classes" code="com/eteks/test/AppletEmprunt"
             name="emprunt" width="280" height="160">
        <param name="capital" value="100000">
        <param name="taux" value="5.5">
        <param name="duree" value="10">
     Votre navigateur n'est pas compatible java 1.4.
     </applet> <br>
</body>
</html>
```

### 1.7 Compléments

Voici quelques sources d'information utiles sur le sujet, qui nous ont servi de référence : [Cha03] (chapitre 18), [Cla03] (chapitre 5), [SM03] (chapitre 20), [Puy04] (p. 161) et
http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap017.htm
http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/applets.html

## 2 Threads

Dans cette section nous abordons une bibliothèques de classes qui permet une première approche des traitements parallèles et concurrents en Java, les (*threads*). Il est ainsi possible de concevoir des programmes multi-tâches où les tâches se déroulent en parallèle. Ces tâches doivent se synchroniser, être interrompus, attendre, préserver les ressources et communiquer comme le font les processus. Ces tâches peuvent avoir des priorités.

Sans le savoir, l'utilisateur (ou le développeur) de programmes Java utilise des *thread* dans les IHM, les applets... Comme toute programmation parallèle et concurrente, la programmation en *multithread* est à manipuler avec précaution car la mise au point est souvent délicate. Enfin, notons que le parallélisme n'est effectif que si d'une part il y a plusieurs processeurs physiques et si l'implantation de la machine virtuelle prend en compte ces différents processeurs.

### 2.1 Généralités

De manière générale, un *thread* ressemble à un processus, au sens système du terme, car il exécute une suite d'instructions. Cependant, il n'en n'a pas tous les attributs.
(source http://fr.wikipedia.org/wiki/Thread )

**Définition 2.1 (Thread)** *Les processus légers (en anglais, thread), également appelés fils d'exécution, sont similaires aux processus en cela qu'ils représentent tous deux l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur ces exécutions semblent se dérouler en parallèle. Toutefois là où chaque processus possède sa propre mémoire virtuelle, les processus légers appartenant au même processus père partagent un environnement (une même partie de sa mémoire virtuelle).*

Il ne peut y avoir plus de threads en exécution que de processeurs. Le temps est donc partagé entre les threads. Le système (ici la machine virtuelle)

- élit un thread (en général en se basant sur sa priorité et sur un historique),
- alloue un quantum de temps à un thread
- jusqu'à ce que son temps soit écoulé ou que le thread devienne non-Runnable

Les *threads* servent à réaliser en pseudo-parallèle l'implantation de processus logiques (problèmes traditionnels tels que le producteur/consommateur, lecteur/rédacteur, philosophe...), animer des IHM ou des pages web, à définir des applications système ou des échanges réseau... L'API des threads s'organise autour de la classe `java.lang.Thread` et de l'interface `Rannable`.

## 2.2 Utilisation

Comme utilisation typique de processus légers on peut citer une interface graphique d'un programme où les interactions de l'utilisateur avec le processus, par l'intermédiaire des périphériques d'entrée, sont gérées par un processus léger, tandis que les calculs lourds (en terme de temps de calcul) sont gérés par un ou plusieurs autres processus légers. Cette technique de conception de logiciel est avantageuse dans ce cas, car l'utilisateur peut continuer d'interagir avec le programme même lorsque celui-ci sera en train d'exécuter une tâche. Une application pratique se retrouve dans les traitements de texte où la correction orthographique est exécutée tout en permettant à l'utilisateur de continuer à entrer son texte. L'utilisation des processus légers permet donc de rendre l'utilisation d'une application plus fluide, car il n'y a plus de blocage durant les phases de traitements intense.

Les processus légers se distinguent du multitâche plus classique par le fait que deux processus sont typiquement indépendants et peuvent interagir uniquement à travers une API fournie par le système telle que IPC. D'un autre côté les processus légers partagent une information sur l'état du processus, des zones de mémoires ainsi que d'autres ressources. Sur certains systèmes la commutation de contexte entre deux processus légers est moins coûteuse que la commutation de contexte entre deux processus. On peut y voir un avantage de la programmation utilisant des threads multiples ou bien une faiblesse des dits système d'exploitation concernant la commutation de contexte entre deux processus.

## 2.3 Avantages et inconvénients

Dans certains cas les programmes utilisant des processus légers sont plus rapides que des programmes architecturés plus classiquement, en particulier sur les machines comportant plusieurs processeurs. Hormis le problème du coût de la commutation de contexte, qui dépend en grande partie du système d'exploitation utilisé, le principal surcoût à l'utilisation de processus multiples provient de la communication entre processus séparés. En effet le partage de certaines ressources entre processus légers permet une communication plus efficace entre les différents threads d'un processus. Là où deux processus séparés doivent utiliser un mécanisme fourni par le système pour communiquer, les processus légers partagent tout ou partie de l'état du processus.

D'un autre côté la programmation utilisant des processus légers est plus difficile, l'accès à certaines ressources partagées doit être restreint par le programme lui-même, pour éviter que tout ou partie de l'état d'un processus ne devienne temporairement inconsistant, tandis qu'un autre processus léger va avoir besoin de consulter cette portion de l'état du processus. Il est donc obligatoire de mettre en place des mécanismes de synchronisation (à l'aide de sémaphore par exemple). La complexité des programmes utilisant des processus légers est aussi nettement plus grande que celle des programmes déférant le travail à faire à plusieurs processus plus simples. Cette complexité accrue, lorsqu'elle est mal gérée lors de la phase de conception ou d'implémentation d'un programme, peut conduire à de multiples problèmes.

## 2.4 Support des processus légers

Les systèmes d'exploitation implémentent généralement les processus légers soit par le multitâche coopératif, soit par le multitâche préemptif, bien que la première méthode soit de plus en plus souvent considérée comme obsolète.

Certains langages de programmation, tel que Java et C#.NET intègrent un support pour les processus légers dans le langage, tandis que la plupart des autres langages ne le permettent que par des extensions du langage considéré ou par l'intermédiaire de bibliothèques. En programmation orientée objet on parle de classe réentrante lorsque des instances distinctes de cette classe peuvent exister simultanément dans différents processus légers.

## 2.5 Définir des *threads*

Pour créer un *thread*, il suffit de définir une sous-classe de la classe `Thread` du paquetage `java.lang` ou implanter l'interface `Runnable` du paquetage `java.lang`. Le cycle de vie d'un thread est toujours le même qu'il hérite de la classe `Thread` ou qu'il implémente l'interface `Runnable`. L'objet correspondant au *thread* doit être créé, puis la méthode `start()` est appelée qui à son tour invoque la méthode `run()`.

Jusqu'en version 1.1, la méthode `stop()` permettait d'interrompre le *thread*. Elle est obsolète (*deprecated*[1]). Maintenant, on doit arrêter proprement le processus, par exemple en modifant une variable d'état.

Avant que le thread ne s'exécute, il doit être démarré par un appel à la méthode `start()`. On peut créer l'objet qui encapsule le thread dans la méthode `start()` d'une applet, dans sa méthode `init()` ou dans le constructeur d'une classe.

Un thread doit prévoir de rendre la main avant que l'on ne l'y oblige en utilisant la méthode `yield`, qui rend la main au système afin qu'il décide à qui attribuer le prochain quota de temps. Inclure des yields conditionnels dans la boucle de vie d'un thread est un moyen non centralisé d'influer sur la politique d'allocation des threads.

### Un premier exemple

L'exemple suivant implante deux thread, l'un dit "bonjour je suis le thread 1" et l'autre "bonjour je suis le thread 2"

montre une applet avec champs de saisie.

Listing 6.5 – Code du *thread* `BonjourThread`

```
package threads;
class BonjourThread extends Thread
{
   public int numero;
   public BonjourThread (int num)
   {
      this.numero = num;
   }
   public void run ()
   {
      for (int i=0; i<100000; i++)//je sais c'est pas joli ce que j'ai fait
         System.out.println ("bonjour je suis le thread " + this.numero);
   }
   public static void main (String args[])
   {
      BonjourThread bj1 = new BonjourThread (1);
      BonjourThread bj2 = new BonjourThread (2);
```

1. http ://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html#stop()

```
        bj1.start();
        bj2.start();
    }
}
```

(source http://www.mandragor.org/article.php?id=7 )

### La classe `Thread`

(inspiré par http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap008.htm )

La classe `Thread` implante l'interface `Runnable`. Elle possède plusieurs constructeurs : un constructeur par défaut et plusieurs autres qui peuvent avoir un ou plusieurs des paramètres suivants :

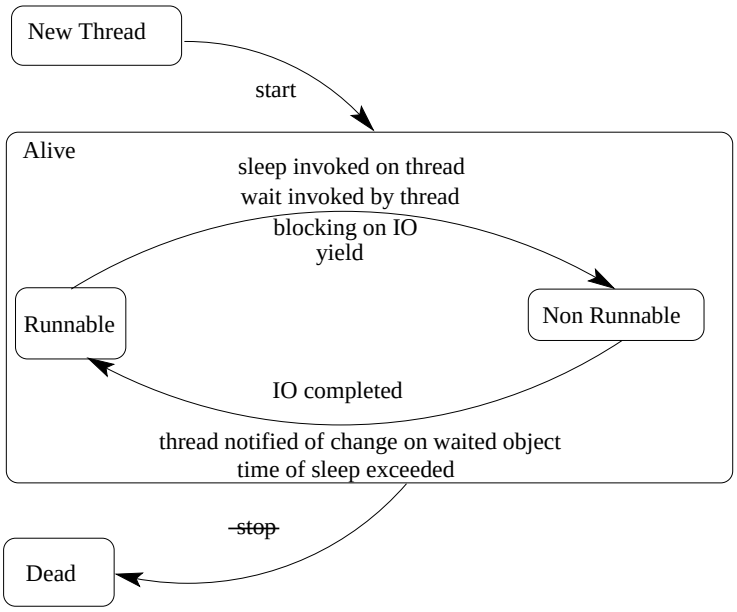
| **Paramètre** | **Rôle** |
|---|---|
| un nom | le nom du thread : si aucun n'est précisé alors le nom sera thread-nnn ou nnn est un numéro séquentiel |
| un objet | l'objet qui l'interface Runnable |
| un groupe | le groupe auquel sera rattaché le thread |

TABLEAU XIV– *Paramètre des constructeurs de la classe `java.lang.Thread`*

Un thread possède une priorité et un nom. Si aucun nom particulier n'est donné dans le constructeur du thread, un nom par défaut composé du suffixe "Thread-" suivi d'un numéro séquentiel incrémenté automatiquement lui est attribué. La classe `Thread` possède plusieurs méthodes pour gérer le cycle de vie du thread.

| **Méthodes** | **Rôle** |
|---|---|
| void destroy() | met fin brutalement au thread : à n'utiliser qu'en dernier recours. |
| int getPriority() | renvoie la priorité du thread |
| ThreadGroup getThreadGroup() | renvoie un objet qui encapsule le groupe auquel appartient le thread |
| boolean isAlive() | renvoie un booléen qui indique si le thread est actif ou non |
| boolean isInterrupted() | renvoie un booléen qui indique si le thread a été interrompu |
| void join() | Le thread en cours attend que le thread joint se termine. Paramétrable en millisecondes. |
| void resume() | reprend l'exécution du thread( ) préalablement suspendu par suspend( ). Cette méthode est dépréciée |
| void run() | contient le code qui sera exécuté par le thread |
| void sleep(long) | mettre le thread en attente durant le temps exprimé en millisecondes fourni en paramètre. Cette méthode peut lever une exception de type Interrupted-Exception si le thread est réactivé avant la fin du temps. |
| void start() | démarrer le thread et exécuter la méthode run() |
| void stop() | arrêter le thread. Cette méthode est dépréciée |
| void suspend() | suspend le thread jusqu'au moment où il sera relancé par la méthode resume( ). Cette méthode est dépréciée |
| void yield() | indique à l'interpréteur que le thread peut être suspendu pour permettre à d'autres threads de s'exécuter. |

TABLEAU XV– *Protocole de la classe `java.lang.Thread`*




(source Gilles Ardourel)

Figure 47 : *Cycle de vie d'un* `thread`

Le comportement de la méthode `start()` de la classe `Thread` dépend de la façon dont l'objet est instancié. Si l'objet qui reçoit le message `start()` est instancié avec un constructeur qui prend en paramètre un objet `Runnable`, c'est la méthode `run()` de cet objet qui est appelée. Si l'objet qui reçoit le message `start()` est instancié avec un constructeur qui ne prend pas en paramètre une référence sur un objet `Runnable`, c'est la méthode `run()` de l'objet qui reçoit le message `start()` qui est appelée.

A partir du J.D.K. 1.2, les méthodes `stop()`, `suspend()` et `resume()` sont dépréciées. Le plus simple et le plus efficace est de définir un attribut booléen dans la classe du thread initialisé à true. Il faut définir une méthode qui permet de basculer cet attribut à false. Enfin dans la méthode `run()` du thread, il suffit de continuer les traitements tant que l'attribut est à true et que les autres conditions fonctionnelles d'arrêt du thread sont négatives. Exemple : exécution du thread jusqu'à l'appui sur la touche `Entrée`.

```
public class MonThread6 extends Thread {
  private boolean actif = true;
  public static void main(String[] args) {
    try {
      MonThread6 t = new MonThread6();
      t.start();
      System.in.read();
      t.arreter();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public void run() {
    int i = 0;
    while (actif) {
      System.out.println("i = " + i);
      i++;
    }
  }
  public void arreter() {
    actif = false;
  }
```

```
}
```

Si la méthode `start()` est appelée alors que le thread est déjà en cours d'exécution, une exception de type `IllegalThreadStateException` est levée. Exemple :

```
package thread;
public class MonThread5 {
  public static void main(String[] args) {
    Thread t = new Thread(new MonThread3());
    t.start();
    t.start();
  }
}
```

Résultat :

```
java.lang.IllegalThreadStateException
      at java.lang.Thread.start(Native Method)
      at thread.MonThread5.main(MonThread5.java:14)
Exception in thread "main"
```

La méthode `sleep()` permet d'endormir le thread durant le temps en millisecondes fournis en paramètres de la méthode.

La méthode statique `currentThread()` renvoie le thread en cours d'exécution.

La méthode `isAlive()` renvoie un booléen qui indique si le thread est en cours d'exécution.

Le plus simple pour définir un thread est de créer une classe qui hérite de la classe `java.lang.Thread`. Il suffit alors simplement de redéfinir la méthode `run()` pour y inclure les traitements à exécuter par le thread. Exemple :

```
package thread;
public class MonThread2 extends Thread {
  public void run() {
    int i = 0;
    for (i = 0; i < 10; i++) {
      System.out.println("" + i);
    }
  }
}
```

Pour créer et exécuter un tel thread, il faut instancier un objet et appeler sa méthode `start()`. Il est obligatoire d'appeler la méthode `start()` qui va créer le thread et elle-même appeler la méthode `run()`. Exemple :

```
package thread;
public class MonThread2 extends Thread {
  public static void main(String[] args) {
   Thread t = new MonThread2();
   t.start();
  }
  public void run() {
    int i = 0;
    for (i = 0; i < 10; i++) {
      System.out.println("" + i);
    }
  }
}
```

### L'interface `Runnable`

Cette interface doit être implémentée par toute classe qui contiendra des traitements à exécuter dans un thread. Cette interface ne définit qu'une seule méthode : `void run()`.

Dans les classes qui implémentent cette interface, la méthode `run()` doit être redéfinie pour contenir le code des traitements qui seront exécutés dans le thread. Exemple :

```
package threads;
public class MonThread3 implements Runnable {
  public void run() {
    int i = 0;
    for (i = 0; i < 10; i++) {
      System.out.println("" + i);
    }
  }
}
```

Lors du démarrage du thread, la méthode `run()` est invoquée. Pour pouvoir utiliser cette classe dans un thread, il faut l'associer à un objet de la classe Thread. Ceci ce fait en utilisant un des constructeurs de la classe Thread qui accepte un objet implémentant l'interface Runnable en paramètre. Exemple :

```
package thread;
public class LancerDeMonThread3 {
  public static void main(String[] args) {
    Thread t = new Thread(new MonThread3());
    t.start ();
  }
}
```

### Modification de la priorité d'un thread

Lors de la création d'un thread, la priorité du nouveau thread est égale à celle du thread dans lequel il est créé. Si le thread n'est pas créé dans un autre thread, la priorité moyenne est attribué au thread. Il est cependant possible d'attribuer une autre priorité plus ou moins élevée.

En java, la gestion des threads est intimement liée au système d'exploitation dans lequel s'exécute la machine virtuelle. Sur des machines de type Mac ou Unix, le thread qui a la plus grande priorité a systématiquement accès au processeur si il ne se trouve pas en mode « en attente ». Sous Windows 95, le système ne gère pas correctement les priorités et il choisit lui même le thread a exécuter : l'attribution d'un priorité supérieure permet simplement d'augmenter ses chances d'exécution.

La priorité d'un thread varie de 1 à 10 , la valeur 5 étant la valeur par défaut. La classe Thread définit trois constantes :

```
MIN_PRIORITY : priorité inférieure
NORM_PRIORITY : priorité standard
MAX_PRIORITY : priorité supérieure
```

Exemple :

```
package threads;
public class TestThread10 {
  public static void main(String[] args) {
    System.out.println("Thread.MIN_PRIORITY = " + Thread.MIN_PRIORITY);
    System.out.println("Thread.NORM_PRIORITY = " + Thread.NORM_PRIORITY);
    System.out.println("Thread.MAX_PRIORITY = " + Thread.MAX_PRIORITY);
```

```
  }
}
```

Résultat :

```
Thread.MIN_PRIORITY  = 1
Thread.NORM_PRIORITY = 5
Thread.MAX_PRIORITY  = 10
```

Pour déterminer ou modifier la priorité d'un thread, la classe `Thread` contient les méthodes suivantes :

| **Méthodes** | **Rôle** |
|---|---|
| int getPriority() | retourne la priorité d'un thread |
| void setPriority(int) | modifie la priorité d'un thread |

La méthode `setPriority()` peut lever l'exception `IllegalArgumentException` si la priorité fournie en paramètre n'est pas comprise en 1 et 10. Exemple :

```
package threads;
public class TestThread9 {
  public static void main(String[] args) {
    Thread t = new Thread();
    t.setPriority(20);
  }
}
```

Résultat :

```
java.lang.IllegalArgumentException
at java.lang.Thread.setPriority(Unknown Source)
at threads.MonThread9.main(TestThread9.java:8)
Exception in thread "main"
```

### La classe `ThreadGroup`

La classe `ThreadGroup` représente un ensemble de threads. Il est ainsi possible de regrouper des threads selon différents critères. Il suffit de créer un objet de la classe ThreadGroup et de lui affecter les différents threads. Un objet `ThreadGroup` peut contenir des threads mais aussi d'autres objets de type `ThreadGroup`. La notion de groupe permet de limiter l'accès aux autres threads. Chaque thread ne peut manipuler que les threads de son groupe d'appartenance ou des groupes subordonnés. La classe `ThreadGroup` possède deux constructeurs :

| **Méthodes** | **Rôle** |
|---|---|
| ThreadGroup(String nom) | création d'un groupe avec attribution d'un nom |
| ThreadGroup(ThreadGoup groupe_parent, String nom) | création d'un groupe à l'intérieur du groupe spécifié avec l'attribution d'un nom |

Pour ajouter un thread à un groupe, il suffit de préciser le groupe en paramètre du constructeur du thread. Exemple :

```
package thread;
public class LanceurDeThreads {
  public static void main(String[] args) {
    ThreadGroup tg = new ThreadGroup("groupe");
    Thread t1 = new Thread(tg,new MonThread3(), "numero 1");
    Thread t2 = new Thread(tg,new MonThread3(), "numero 2");
  }
}
```

L'un des avantages de la classe `ThreadGroup` est de permettre d'effectuer une action sur tous les threads d'un même groupe.

### Démons

Il existe une catégorie de threads qualifiés de démons : leur exécution peut se poursuivre en tâche de fond même après l'arrêt de l'application qui les a lancés. Une application dans laquelle les seuls threads actifs sont des démons est automatiquement fermée. Le thread doit d'abord être créé comme thread standard puis transformé en demon par un appel à la méthode `setDaemon()` avec le paramètre true. Cet appel se fait avant le lancement du thread, sinon une exception de type `IllegalThreadStateException` est levée.

## 2.6 Concurrence de processus

Lorsque plusieurs processus fonctionnent en parallèle, ils peuvent collaborer, se coordonner ou se gêner. Quelque soit le type de relation, on a besoin de mettre en œuvre des mécanismes de coordination tels que la synchronisation entre processus, l'exclusion mutuelle pour accéder à des ressources partagées, la communication... Nous les abordons dans cette section.

## 2.7 Exclusion mutuelle

Chaque fois que plusieurs threads s'exécutent en même temps, il faut prendre des précautions concernant leur bonne exécution. Par exemple, si deux threads veulent accéder à la même variable, il ne faut pas qu'ils le fassent en même temps. Java offre un système simple et efficace pour réaliser cette tache. Si une méthode déclarée avec le mot clé `synchronized` est déjà en cours d'exécution, alors les threads qui en auraient également besoin doivent attendre leur tour.

Le mécanisme d'exclusion mutuelle en Java est basé sur le principe des moniteurs. Pour définir une méthode protégée, afin de s'assurer de la cohérence des données, il faut utiliser le mot clé `synchronized`. Cela crée à l'exécution, un moniteur associé à l'objet qui empêche les méthodes déclarées `synchronized` d'être utilisées par d'autres objets dès lors qu'un objet utilise déjà une des méthodes synchronisées de cet objet. Dès l'appel d'une méthode synchronisée, le moniteur verrouille tous les autres appels de méthodes synchronisées de l'objet. L'accès est de nouveau automatiquement possible dès la fin de l'exécution de la méthode.

Ce procédé peut bien évidemment dégrader les performances lors de l'exécution mais il garantit, dès lors qu'il est correctement utilisé, la cohérence des données.

**Sécurisation d'une méthode** Lorsque l'on crée une instance d'une classe, on crée également un moniteur qui lui est associé. Le modificateur `synchronized` place la méthode (le bloc de code) dans ce moniteur, ce qui assure l'exclusion mutuelle. Le méthode ainsi déclarée ne peut être exécutée par plusieurs processus simultanement. Si le moniteur est occupé, les autres processus seront mis en attente. L'ordre de réveil des processus pour accéder à la méthode n'est pas prévisible. Si un objet dispose de plusieurs méthodes `synchronized`, ces dernières ne peuvent être appelées que par le thread possédant le verrou sur l'objet.

**Sécurisation d'un bloc** L'utilisation de méthodes synchronisées trop longues à exécuter peut entrainer une baisse d'efficacité lors de l'exécution. Avec java, il est possible de placer n'importe quel bloc de code dans un moniteur pour permettre de réduire la longueur des sections de code sensibles. L'objet dont le moniteur est à utiliser doit être passé en paramètre de l'instruction `synchronized`.

```
synchronized  void methode1() {
   // bloc de code sensible
```

```
    ...
}

 void methode2(Object obj) {
    ...
    synchronized (obj) {
       // bloc de code sensible
       ...
    }
}
```

**Sécurisation de variables de classes** Pour sécuriser une variable de classe, il faut un moniteur commun à toutes les instances de la classe. La méthode `getClass()` retourne la classe de l'instance dans laquelle on l'appelle. Il suffit d'utiliser un moniteur qui utilise le résultat de `getClass()` comme verrou.

## 2.8 Synchronisation de processus

Le mécanisme utilisé ci-dessus correspond à une synchronisation **implicite** des processus autour d'une ressource. On peut aussi procéder par synchronisation **explicite** des processus, c'est-à-dire des signaux qui sont modélisés par les méthodes `wait()` et `notify()`. Lorsque l'exécution d'une méthode dépend d'un état de l'objet, il faut :

- la rendre `synchronized`,
- attendre avec `wait()` dans une boucle testant l'état,
- rendre les methodes de changement d'état `synchronized`,
- faire un `notify` ou un `notifyAll` dans les méthodes de changement d'état.

Un des exemples classiques de l'utilisation des signaux est celui des producteurs-consommateurs, qu'on peut trouver dans

http://cui.unige.ch/java/JAVAF/signaux.html

Soit un tampon borné de n objets, un processus producteur et un processus consommateur. Nous déclarons 2 processus, un producteur et un consommateur. Les données sont produites plus vite que le consommateur ne peux les prélever, à cause de la différence de durée des délais (aléatoires) introduits dans les 2 processus.

```
class tamponCirc {
    private Object tampon[];
    private int  taille ;
    private int en, hors, nMess;
    /** constructeur. crée un tampon de taille éléments */
    public tamponCirc (int taille) {
        tampon = new Object[taille];
        this.taille  = taille;
        en = 0;
        hors = 0;
        nMess = 0;
    }
    public synchronized void depose(Object obj) {
        while (nMess == taille) {     // si plein
            try {
                wait();                    // attends non-plein
            } catch (InterruptedException e) {}
        }
        tampon[en] = obj;
        nMess++;
        en = (en + 1) % taille;
```

```
        notify ();                    // envoie un signal non-vide
    }
    public synchronized Object preleve() {
        while (nMess == 0) {   // si vide
            try {
                wait ();              // attends non-vide
            } catch (InterruptedException e) {}
        }
        Object obj = tampon[hors];
        tampon[hors] = null;     // supprime la ref a l'objet
        nMess--;
        hors = (hors + 1) % taille;
        notify ();                    // envoie un signal non-plein
        return obj;
    }
}
```

Les classes suivantes utilisent le tampon.

```
class producteur extends Thread {
    private tamponCirc tampon;
    private int val = 0;
    public producteur(tamponCirc tampon) {
        this.tampon = tampon;
    }
    public void run() {
        while (true) {
            System.out.println("je depose "+val);
            tampon.depose(new Integer(val++));
            try {
                Thread.sleep((int)(Math.random()*100));   // attend jusqu'a 100 ms
            } catch (InterruptedException e) {}
        }
    }
}

class consommateur extends Thread {
    private tamponCirc tampon;
    public consommateur(tamponCirc tampon) {
        this.tampon = tampon;
    }
    public void run() {
        while (true) {
            System.out.println("je preleve "+((Integer)tampon.preleve()).toString());
            try {
                Thread.sleep((int)(Math.random()*200));   // attends jusqu'a 200 ms
            } catch (InterruptedException e) {}
        }
    }
}

class utiliseTampon {
    public static void main(String args[]) {
        tamponCirc tampon = new tamponCirc(5);
        producteur prod = new producteur(tampon);
        consommateur cons = new consommateur(tampon);
        prod.start ();
        cons.start ();
        try {
```

```
            Thread.sleep(30000);     // s'execute pendant 30 secondes
        } catch (InterruptedException e) {}
    }
}
```

Exécution

```
...
je depose 165
je depose 166
je preleve 161
je depose 167
je preleve 162
je depose 168
je preleve 163
je depose 169
je preleve 164
je depose 170
je preleve 165
je depose 171
je preleve 166
je preleve 167
...
```

Il existe deux variantes de la méthode `wait()` qui permettent de spécifier un temps limite après lequel le processus sera réveillé, sans avoir à être notifié par un processus concurrent. Il s'agit de `wait(long milli)`, qui attend milli millisecondes, et de `wait(long milli,int nano)` qui attend nano nanosecondes en plus du temps milli. Il n'est pas possible de savoir si `wait()` s'est terminé à cause d'un appel à `notify()` par un autre processus, ou de l'épuisement du temps.

La méthode `notifyAll()` réveille tous les processus, et dès que le moniteur sera libre, ils se réveilleront tour à tour. Attention ! Le noyau Java ne fait aucune garantie concernant l'élection des processus lors d'un appel à notify(). En particulier, il ne garantit pas que les processus seront débloqués dans l'ordre ou ils ont été bloqués. C'est à cause de cela que l'on a placé wait() dans une boucle dans l'exemple précédent, car un consommateur pourrait réveiller un autre consommateur alors que le tampon est vide.

Remarquons aussi que les méthodes `wait`, `notify()` et `notifyAll()` ne peuvent être appelées que dans des méthodes synchronisées (synchronized).

### Autre version

Une version plus simple est proposée dans [NP96], que nous avons modifiée.

Listing 6.6 – Code du *thread* `Consumer`

```
package threads.niemeyers;

import java.util.Vector;

class Consumer extends Thread {
    Producer producer;

    Consumer(Producer p) {
      producer = p;
    }
```

```
    public void run() {
      try {
          while ( true ) {
              String message = producer.getMessage();
              System.out.println("Got message: " + message);
              sleep( 3000 );
          }
      } catch( InterruptedException e ) { }
    }

    public static void main(String args[]) {
        final Producer producer = new Producer();
      producer.start();
        new Consumer( producer ).start();
        new Consumer( producer ).start();
    }
}
```

Listing 6.7 – Code du *thread* `Producer`

```
package threads.niemeyers;

import java.util.Vector;

class Producer extends Thread {
    private Vector messages = new Vector();
   static int MAXQUEUE = 5;

    public void run() {
      try {
          while ( true ) {
              System.out.println("Put message");
              putMessage();
              sleep( 1000 );
          }
      } catch( InterruptedException e ) { }
    }

   private synchronized void putMessage() throws InterruptedException {
      while ( messages.size() == MAXQUEUE )
          wait();
      messages.addElement( new java.util.Date().toString() );
      notifyAll();
   }

    public synchronized String getMessage() throws InterruptedException {
      notifyAll();
      while ( messages.size() == 0 )
          wait();
      String message = (String)messages.firstElement();
      messages.removeElement( message );
        return message;
    }
}
```

Exécution

```
Put message
```

```
Got message: Thu Jan 19 18:27:21 CET 2006
Put message
Got message: Thu Jan 19 18:27:22 CET 2006
Put message
Got message: Thu Jan 19 18:27:23 CET 2006
Put message
Got message: Thu Jan 19 18:27:24 CET 2006
Put message
Put message
Got message: Thu Jan 19 18:27:25 CET 2006
Put message
Got message: Thu Jan 19 18:27:26 CET 2006
Put message
Put message
Got message: Thu Jan 19 18:27:27 CET 2006
Put message
Got message: Thu Jan 19 18:27:28 CET 2006
Put message
Put message
...
```

D'autres exemples sont accessibles dans [Cha03] ou à
http://rangiroa.essi.fr/cours/systeme1/thread/

### Rendez-vous en Java

```
Rendez-vous avec des sémaphores

Semaphore write =  0
Semaphore read = 0
Value value

P1 : // émetteur
value := V
V(write)
P(read)

P2 : // récepteur
P(write)
local := Value
V(read)
```

---

```
class RendezVous {
    int value;
    boolean write = false;
    boolean readerwaiting = false; // non nécessaire

    public synchronized int read(String str)  throws InterruptedException {
   int v;

   if (!write) {
      readerwaiting = true;            // non nécessaire
      System.out.println(str+" wait ");
```

```
        wait ();
        System.out.println(str+" active ");
        readerwaiting = false;            // non nécessaire
    }
    v = value;
    write = false;
    System.out.println(str+" read ");

    notify ();
    return v;
     }

     public synchronized void write(String str, int v)  throws InterruptedException {

    value = v;
    System.out.println(str+" write ");
    write = true;

    if (readerwaiting)  // non nécessaire
                    notify ();

    System.out.println(str+" wait ");
    wait ();
    System.out.println(str+" active ");
     }
}

class SendThread extends Thread {
     String name;
     RendezVous rendezVous;

     public SendThread(String str, RendezVous r) {
          name = str;
    rendezVous = r;
     }

     public void run() {
    for (int i = 0; i<10; i++) {
         try {
        Thread.sleep((long)(Math.random() * 1000));
               } catch (InterruptedException e) {}
         System.out.println(name+" send "+i);
               try {
              rendezVous.write(name, i);
         } catch (InterruptedException e) {
        System.out.println ("exception");
         }
    }
    System.out.println(name+" DONE !! ");
     }
}

class ReceiveThread extends Thread {
     String name;
     RendezVous rendezVous;

     public ReceiveThread(String str, RendezVous r) {
          name = str;
```

```
    rendezVous = r;
     }

     public void run() {
    int j = 0;

    for (int i = 0; i<10; i++) {
        try {
       Thread.sleep((long)(Math.random() * 1000));
             } catch (InterruptedException e) {}
             try {
                 j = rendezVous.read(name);
         } catch (InterruptedException e) {
       System.out.println ("exception");
         }
         System.out.println(name+" receive "+j);
    }
    System.out.println(name+" DONE !! ");
     }
}

public class rdv {
     public static void main (String[] args) {
          RendezVous r = new RendezVous();

          new SendThread("Sender", r).start();
          new ReceiveThread("Receiver", r).start();

          System.out.println("DONE! ");
     }
}
```

### 2.9 Compléments

Voici quelques sources d'information utiles sur le sujet, qui nous ont servi de référence : [Cha03] (chapitre 17), [Cla03] (chapitre 5), [SM03] (chapitre 16) et
http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap008.htm
http://rangiroa.essi.fr/cours/systeme1/thread/
http://java.sun.com/docs/books/tutorial/essential/threads/
http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/threads.html

## 3 JNI

JNI est l'acronyme de *Java Native Interface*. C'est une technologie qui permet d'utiliser du code natif, produit dans un autre langage (tel que C, C++), dans une classe Java. On utilise cette facilité soit pour intégrer des algorithmes ou plus généralement des morceaux de code déjà implantés (et testés !) soit pour implanter plus efficacement certaines parties critiques du code (critères de performance). L'inconvénient majeur de cette technologie est d'annuler la portabilité du code Java.

La mise en oeuvre de JNI nécessite plusieurs étapes (selon †) :

- la déclaration et l'utilisation de la ou des méthodes natives dans la classe Java,
- la compilation de la classe Java,
- la génération du fichier d'en-tête avec l'outil `javah`,

- l'écriture du code natif en utilisant entre autre les fichiers d'en-tête fourni par le JDK et celui généré précédemment,
- la compilation du code natif sous la forme d'une bibliothèque.

Le format de la bibliothèque est donc dépendant du système d'exploitation pour lequel elle est développée : `.dll` pour les systèmes de type Windows, `.so` pour les système de type Unix, etc.

Cette API n'est pas plus détaillée pour l'instant. Le lecteur intéressé peut consulter

- http://java.sun.com/j2se/1.4.2/docs/guide/jni/
- http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/jni.html
- (†)http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap030.htm
- http://www.esil.univ-mrs.fr/∼tourai/Java/ (chapitre 43).

## 4 JDBC

JDBC est l'acronyme de *Java DataBase Connectivity* et désigne une API définie par Sun pour permettre un accès aux bases de données avec Java.

Cette API n'est pas au programme. Consulter [Cla03]

http://java.sun.com/products/jdbc/
http://www.esil.univ-mrs.fr/∼tourai/Java/ http://java.developpez.com/IntroJDBC.pdf

## 5 JUnit

Les bonnes pratiques de développement insistent sur un usage poussé des tests, même avant l'implantation.

`JUnit` est un framework de test de non-régression écrit par Erich Gamma et Kent Beck. Il est utilisé par les développeurs pour implanter des tests unitaires en Java. C'est un logiciel *open source* régi par une CGPL (*Common Public License Version*) et hébergé par SourceForge. La définition des tests se fait en créant des sous-classes de test

`JUnit` est aussi implanté par un plugin Eclipse.

Cette API n'est pas au programme. Consulter [Lin03, SM03],

http://junit.sourceforge.net/ http://www-igm.univ-mlv.fr/ dr/XPOSE2003/JUnit_tour/
http://www.junit.org/index.htm

# Chapitre 7

# Les IHM et le modèle Swing

Dans ce chapitre nous présentons et illustrons l'API Java des interfaces homme-machine (IHM). Après un aperçu général, nous décrirons les principaux éléments de l'API, le modèle MVC, les bibliothèques pratiques et la génération automatique d'interfaces.

## 1 Généralités

### 1.1 Interfaces Homme-Machine

Les interfaces homme-machine (IHM) ou *Graphical User Interface* (GUI) ont une place prépondérante dans l'acceptation des logiciels par l'utilisateur, car une IHM prend en charge la communication entre l'utilisateur et l'application.

**Définition 1.1 (IHM)** *L'Interaction Humain Machine, Interaction Homme Machine ou* ***Interface Homme Machine*** *(IHM) étudie la façon dont les humains interagissent avec les ordinateurs ou entre eux à l'aide d'ordinateurs, ainsi que la façon de concevoir des systèmes informatiques qui soient ergonomiques, c'est-à-dire efficaces, faciles à utiliser ou plus généralement adaptés à leur contexte d'utilisation.*

(source `http://fr.wikipedia.org/wiki/IHM`)

Il est maintenant inconcevable de fournir à un client un logiciel sans IHM ergonomique (y compris pour les logiciels embarqués avec commande). Les IHM ont aussi été un facteur important dans l'acceptation de la programmation à objets, notamment avec Smalltalk et plus récemment Java.

Une IHM est composée d'au moins deux éléments : des composants visuels, affichés dans des fenêtres et des composants événementiels permettant l'interaction avec l'utilisateur. Les éléments relatifs au composant visuel sont les attributs graphiques, la structuration des composants, leur disposition (*layout*), leur présentation standard (*look*), etc. Il s'agit de l'aspect graphique (lié aux classes graphiques du langage de programmation). Les éléments relatifs au composant événementiel sont la source d'interaction (clavier, souris), la connexion avec le composant visuel, le lien avec l'application, etc. Il s'agit de l'aspect interaction (lié aux classes d'entrées/sorties du langage de programmation). On parle aussi de **vue** (affichage) et de contrôle. Lorsque le contenu fait l'objet d'une modélisation particulière, connectée aux vues et au contrôleurs, on parle de modèle MVC, *Model/View/Controller* (voir section 1.2).

### 1.2 Principes de l'architecture MVC

Le modèle MVC ou `Model/View/Controller` est la brique de base de construction d'interfaces graphiques en Smalltalk-80. Le principe est le suivant : le modèle contient les données, la vue les affiches et le contrôleur s'occupe des interactions avec l'utilisateur (clavier, souris).

Le principe est simple, mais son application n'est pas triviale. En fait, dans une interface, il y a plusieurs modèles, plusieurs contrôleurs et plusieurs vues qui sont en interaction. Une version très schématique est donnée dans figure 48.

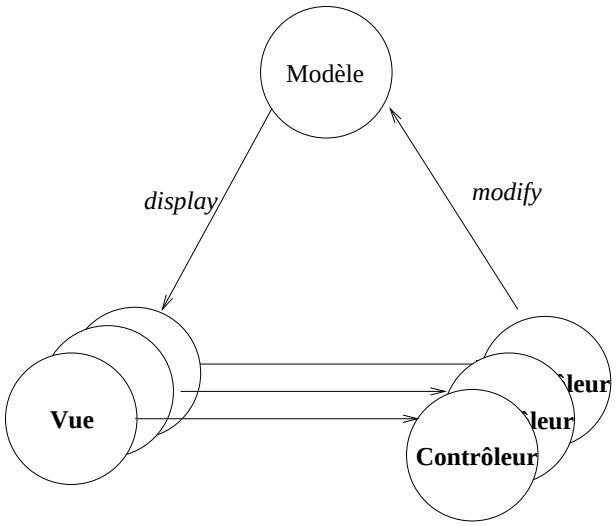



Figure 48 : *Version schématique du modèle MVC*

Contrairement à la programmation événementielle sous Windows, le contrôle est réparti dans plusieurs contrôleurs, programmables et personnalisables. Dans Windows, tous les événements sont captés par un contrôleur unique qui associe à chaque événement une action.

A partir de la version 4.0 de Smalltalk-80, les auteurs du langage ont pris le principe des dépendants pour alléger les liens entre modèles, vues et contrôleurs. Un modèle est à priori une instance d'une sous-classe de la classe `Model`, qui implémente les relations de dépendance plus efficacement que la classe `Object`. Il est illustré par la figure 49.

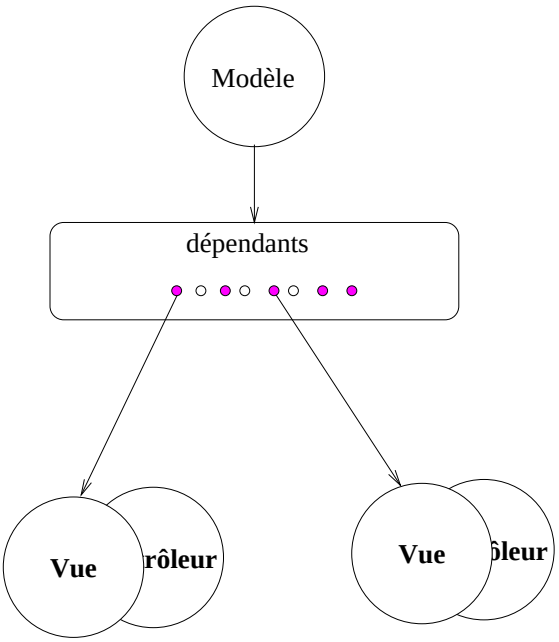



Figure 49 : *Le modèle MVC avec les dépendants*

Son principe est le suivant :

1. L'utilisateur agit sur l'interface par la souris et le clavier.
2. Le **contrôleur** capte cette modification avec une opération.

3. Le **contrôleur** demande éventuellement des informations à l'utilisateur.
4. Muni de ces informations, il informe le **modèle** d'une modification.
5. Le **modèle** effectue la modification sur ses données.
6. Le **modèle** informe ses dépendants qu'il a changé (`self changed`). C'est à ce niveau que des distinctions sont faites entre les vues d'un modèle par l'utilisation de mots-clés.
7. La **vue** modifie (éventuellement) ses informations par une méthode `update` ou l'une de ses variantes. Puis, elle signale à son conteneur qu'elle est modifiée. Il s'en suit une remontée de la hiérarchie des composants visuels qui met à jour l'affichage par `display`.
8. La **vue** informe le contrôleur de sa modification (`ctrl_update`).

Le modèle ne connaît pas ses dépendants. Chaque vue a un contrôleur. La vue connaît son contrôleur et le modèle. Le contrôleur connaît le modèle. Il peut a priori contrôler plusieurs vues. On évite ainsi trop de dépendances circulaires, source de programmes infinis.

La construction se fait sur ce modèle mais aussi sur un modèle simplifié : les **vues enfichables** ou *plugable views*. En Smalltalk, la liaison avec le système de fenêtrage hôte est transparente par la classe `ScheduledWindow` qui va de pair avec la classe `StandardSystemController`. Pour qu'une vue soit affichée dans une fenêtre, il faut qu'elle soit placée dans un **cadre** ou *wrapper* (emballage). Il existe quatre cadres principaux dans le système :

- `TranslatingWrapper` : cadre placé relativement à l'origine de la fenêtre.
- `BoundedWrapper` : cadre délimité par un rectangle extensible.
- `BorderedWrapper` : cadre muni d'un bord.
- `ScrollWrapper` : cadre muni d'ascenseurs.

Lorsque plusieurs vues sur un même modèle ou des modèles différents sont affichées dans une fenêtre, nous utilisons une **vue composite** ou `CompositePart`. Cette vue composite comprend des cadres placés de façon fixe ou proportionnelle dans la vue. Il faut ensuite associer un contrôleur, lorsqu'il ne s'agit pas d'une vue enfichable. Pour cela, on utilise habituellement un **contrôleur avec menu** ou `ControllerWithMenu` ou des boutons (vue enfichable). Le contrôleur par défaut de la fenêtre (vue système) est une instance de la classe `StandardSystemController`. Chaque vue a son contrôleur et réagit donc indépendamment des autres vues. Cependant, lorsque les vues sont organisées hiérarchiquement, le contrôle est lui aussi remonté. Un contrôleur a la structure générique suivante :

```
startUp
   initialisation.
   boucle de controle
   terminaison
```

Cette structure est redéfinie et spécialisée dans les sous-classes. Le lien entre la vue et le contrôleur est souvent décrit dans le cadre `Wrapper`, qui englobe la vue, et plus généralement dans la classe `VisualPart` (et ses sous-classes `DependentPart` (les vues), `CompositePart` et `Wrapper`) qui intercepte les messages vers la vue et le contrôleur.

### 1.3 IHM et Java

Pour les IHM en Java, il existe actuellement deux standards de fait, l'API de Sun (*AWT* (*Abstract Window Toolkit*) + *Swing*) et la bibliothèque *SWT* (*Standard Widget Toolkit*) d'Eclipse. Nous nous intéresserons principalement à la première, qui est en quelque sorte l'API officielle et pour laquelle, en conséquent, une majorité d'outils est compatible.

Pour les IHM en Java, Sun propose le framework *Java Foundation Classes* (JFC) constitué de *AWT*, *Swing* et *Java2D*. La bibliothèque **Astract Window Toolkit** (AWT) est la première bibliothèque graphique pour Java. A partir de Java 2, une nouvelle bibliothèque a

été proposée, **Swing** qui améliore à la fois le panel de composants visuels, la convivialité et la standardisation de ces composants. Swing offre la possibilité de créer des interfaces graphiques identiques quelque soit le système d'exploitation sous-jacent, au prix de performances moindres qu'en utilisant Abstract Window Toolkit (AWT). Il utilise le principe Modèle-Vue-Contrôleur (MVC) et dispose de plusieurs choix d'apparence (de vue) pour chacun des composants standard. AWT sert encore de fondement à Swing, dans la mesure où de nombreuses classes Swing héritent de classes AWT. (source wikipedia)

Pour l'IDE Eclipse, un bibliothèque particulière a été proposée, le **Standard Widget Toolkit** (SWT), qui a l'avantage d'être un source libre. SWT n'est pas un standard Java reconnu. Cette bibliothèque se compose d'une bibliothèque de composants graphiques (texte, label, bouton, panel), des utilitaires nécessaires pour développer une interface graphique en Java, et d'une implémentation native spécifique à chaque système d'exploitation qui sera utilisée à l'exécution du programme. La deuxième partie de SWT n'est en fait qu'une réencapsulation des composants natifs de système (Win32 pour Windows, GTK ou Motif pour Linux). Plusieurs projets travaillent aujourd'hui sur une implémentation utilisant les composants de Swing. L'environnement de développement libre Eclipse repose sur cette architecture. (source wikipedia)

Noter, que pour JFC, les éléments de l'IHM sont appelés **composants** alors qu'il sont appelés gadjets ou **widgets** dans SWT et dans Smalltalk. Pour la suite, nous utiliserons principalement le terme composant puisque nous nous plaçons dans le cadre JFC.

## 1.4 AWT et Swing

Pour rendre portable les IHM Java, AWT utilise directement les ressources système du système de fenêtrage d système d'exploitation cible, en les encapsulant dans des abstractions [1]. Autrement dit, il s'agit de surcharge, au sens objet du terme : on présente une interface unique, mais la réalisation varie d'un intégrateur à l'autre (MSWindows, MacOS, X/Motif, KDE, GNOME...). Ce type d'implantation induit que les composants AWT sont qualifiés de composants **lourds**. Pour les composants (visuels) de base tels que les fenêtres, cette implantation est obligatoire. Par contre pour le contenu des fenêtre on peut en faire abstraction. C'est ainsi qu'a été proposée la bibliothèque Swing, qui propose des composants **légers**. Les composants légers sont écrits entièrement en Java, et donc indépendants de l'intégrateur hôte (système de fenêtrage). L'affichage n'est pas délégué au système hôte. Le programmeur peut aussi plus facilement les adapter par redéfinition. Néanmoins, pour conserver des styles de présentation « typiques », Swing propose un paramétrage des composants par un style d'affichage, le « *look and feel* », largement inspiré de Smalltalk.

Swing ne remplace pas AWT mais le complète. En effet Swing s'appuie sur des composants de base d'AWT, comme le montrent les hiérarchies d'héritage de la section 2.7 du chapitre 3.

L'architecture des IHM est organisée autour des composants visuels (les vues selon la section 1.2) et des écouteurs (*listeners*, les contrôleurs selon la section 1.2) qui permettent de gérer les événements captés par l'IHM.

Le modèle de Swing adapte le modèle MVC en groupant la vue et le contrôleur dans un seul objet appelé `UIDelegate` selon le shéma de la figure 50. Cette adaptation est relativement classique, elle s'appelle vue enfichable en Smalltalk (*Pluggable View*) et permet de simplifier la communication entre la vue et le contrôleur. Elle s'explique par le fait qu'à un type de vue correspond souvent un type de contrôleur. Par exemple, le pilotage d'une liste déroulante est contrôlée d'une certaine façon. Plus exactement, un composant graphique met en œuvre trois objets principaux [Cha03] :

– un modèle ;

1. D'où le terme *Abstract*...

- un objet responsable du dessin et de la gestion des événements (`UIDelegate`) (vue + contrôleur) ;
- un composant graphique (`Jcomponent`) (interface proposée au programmeur pour manipuler les éléments graphiques).

Cette implantation est une variante du modèle MVC original, certains la qualifient d'architecture de type MC, *Model/Component*) [Cha03]. Swing supporte ce type de modélisation explicitement avec ses widgets JList, JTable, JTree, JEditorPane, etc.

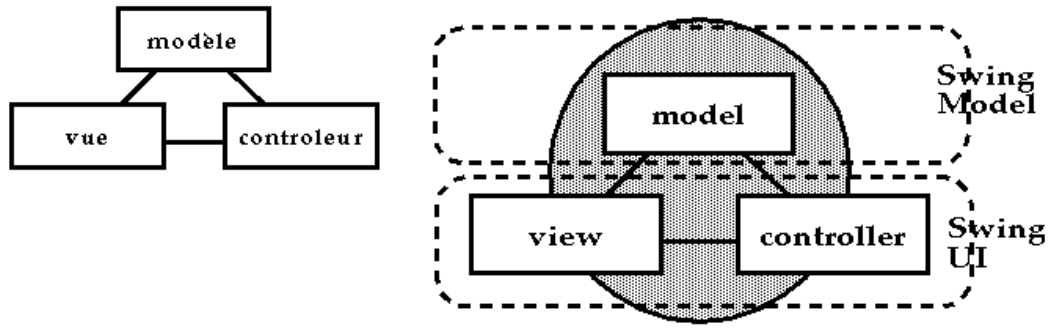



Figure 50 : *Modèle MVC/Java*

(source http://tecfa.unige.ch/guides/tie/html/java-swing/java-swing.html)

Le modèle représente les données de l'application ou autrement dit : l'état. Il ne connaît rien de ses contrôleurs ou de ses vues. Il possède (en règle générale) quatre types de méthodes

- Interrogation de son état interne (lire).
- Manipulation de son état interne (changer, détruire).
- Ajouter et enlever des *event listeners*.
- Exécuter (*fire*) des événements.

L'interface utilisateur comprend la vue, qui est la représentation visuelle des données (dans leur état actuel !), et le contrôleur, qui gère l'interaction utilisateur avec le modèle et intercepte les entrées de l'utilisateur (dans la vue) et les traduit en changements dans le modèle. Elle possède ces trois types de méthodes :

- Dessin.
- Gestion des informations géométriques.
- Gestion d'événements utilisateurs (*click*, *keyboard*, ...).

L'élément de base d'une interface est la **fenêtre** (*window*), qui peut être à l'intérieur d'une autre fenêtre. Le **composant principal**, la fenêtre de base de cette hiérarchie de composition, est un composant lourd (AWT) de type `Window` ou `Applet`.

Lorsqu'on dispose plusieurs zones dans une fenêtre, on doit coordonner l'ensemble. Pour cela on utilise un **conteneur** (*container*).

L'affichage se fait dans une zone d'écran, un **contexte graphique** (*graphic context*).

En résumé, une interface graphique Swing repose sur trois concepts : les fenêtres, les composants et les conteneurs. L'IHM y ajoute les écouteurs pour contrôler les actions de l'utilisateur. Nous y reviendrons dans la suite du chapitre.

## 1.5 Bibliographie et webographie

Voici différentes sources intéressantes sur le sujet, qui nous ont servi, pour rédiger ce chapitre, notamment celle de JM Doudoux.

- La plupart des ouvrages sur Java intègrent au moins un chapitre sur les IHM : ([Cha03], chap. 12, 13, 14), ([Puy04], chap. 9), ([SM03], p. 17), ([Cla03], chap. 4).
- L'ouvrage suivant est entièrement consacré à Swing (et AWT) [BB05] (mais vieillissant !).

- Tutorial "officiel" de Sun :
  http://java.sun.com/docs/books/tutorial/uiswing/
- *Swing : A Quick Tutorial for AWT Programmers* :
  http://www.apl.jhu.edu/ hall/java/Swing-Tutorial/
- Présentations générales :
  http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/

- http://perso.wanadoo.fr/jm.doudoux/java/tutorial/, chapitres 11, 12, 13
  http://gbm.esil.univ-mrs.fr/ tourai/, chapitre 32
- Swing et MVC :
  http://tecfa.unige.ch/guides/tie/html/java-swing/java-swing.html
- Applications pratiques :
  http://www.infres.enst.fr/~domeo/

## 2 Les concepts principaux

### 2.1 Les fenêtres

Le **composant principal** d'une interface graphique est un composant lourd (AWT) de type fenêtre `Window` ou `Applet`. Le premier est utilisé localement dans une application java, le second est employé pour les programmes fonctionnant par l'intermédiaire d'un réseau et auxquels de nombreuses restrictions sont imposées pour des raisons de sécurité.

La figure 51 présente la partie haute de la hiérarchie d'héritage des fenêtres (voir aussi les hiérarchies d'héritage de la section 2.7 du chapitre 3). La classe `Dialog` représente les « petites boîtes de dialogue » à comportement restreint. La classe `Frame` sert à créer des fenêtres sans barre de titre, sans bordures redimensionables et sans les contrôles habituels. A chacune des quatre classes de AWT correspond une classe dans Swing de même nom préfixé par "J" et qui sont les seuls composants "lourds" de Swing. La classe `JInternalFrame` définit des fenêtres à l'intérieur d'autres fenêtres.

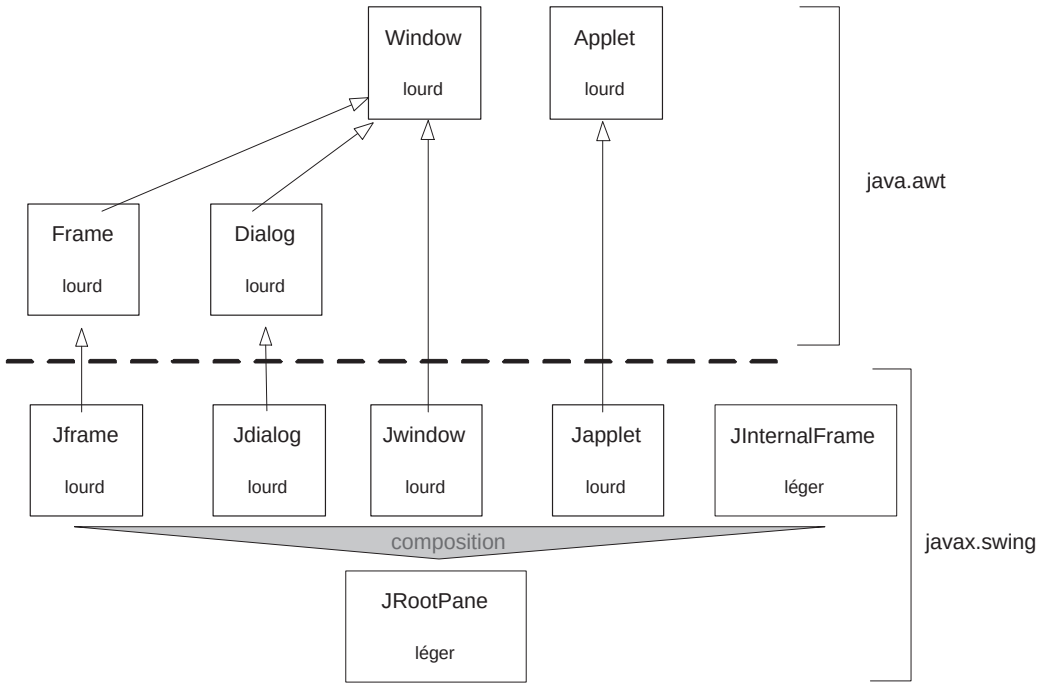



Figure 51 : *Hiérarchie des fenêtres de JFC*

Le composant `JRootPane` contient les éléments constituant le contenu d'une fenêtre. La figure 52 présente sommairement la hiérarchie de composition des fenêtres, qui peut dans ce

cas être interprétée comme une succession de couches (*layers*) avec dans l'ordre de plus en plus profond) : la glace, le menu et le contenu.

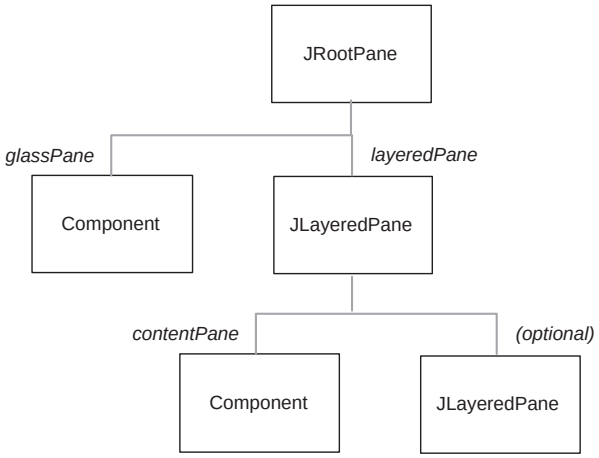



Figure 52 : *Hiérarchie de composition des fenêtres de JFC*

La disposition des éléments du contenu est contrôlée par un gestionnaire d'agencement (*layout manager*).

Exemple : une fenêtre qui affiche bonjour.

```
package swing;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class TestIHM1 {
    public static void main(String[] args) {
        JFrame w = new JFrame("Simple window");
        JLabel msg = new JLabel("That's all folks");
        w.getContentPane().add(msg);
        w.setBounds(100, 200, 200, 300);
        w.setVisible(true);
        //w.show() ; deprecated
    }
}
```

Noter que cet exemple ne permet pas de clore l'application, même si on peut fermer la fenêtre, car il n'y a pas de contrôle des événements.

### Le contexte graphique

Le **contexte graphique** (*graphic context*) est la zone d'affichage des composants visuels. La classe `Graphics` contient les outils nécessaires pour dessiner. Cette classe est abstraite et elle ne possède pas de constructeur `public` : il n'est pas possible de construire des instances de graphics nous même. Les instances nécessaires sont fournies par le système d'exploitation qui instanciera via à la machine virtuelle une sous classe de `Graphics` dépendante de la plateforme utilisée.

On y trouve des méthodes comme `drawRect(x, y, largeur, hauteur)`, `drawPolygon(int[], int[], int)`, `fillArc(x, y, largeur, hauteur, angle1, angle2);`, `drawString(texte, x, y )`, `setColor(c)`, `copyArea(x1, y1, x2, y2, dx, dy)`, etc.

## 2.2 Les composants et les conteneurs

L'API JFC comprend un grand nombre de classes et les imbrications entre AWT et Swing ne sont pas toujours simples. Par exemple, la classe des composants swing `JComponent` hérite

de la classe AWT `Container`, qui elle-même hérite de la classe AWT `Container` selon la figure 53 (et la figure 17 du chapitre 3). Autrement dit, un composant Swing est un conteneur et pas simplement un composant AWT.

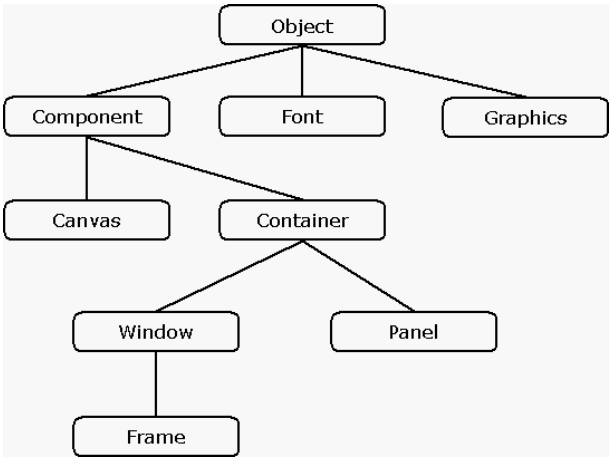



Figure 53 : *Hiérarchie d'héritage des composants visuels AWT*

(source `http://www.scit.wlv.ac.uk/ jphb/naw.index.html` )

### Les composants

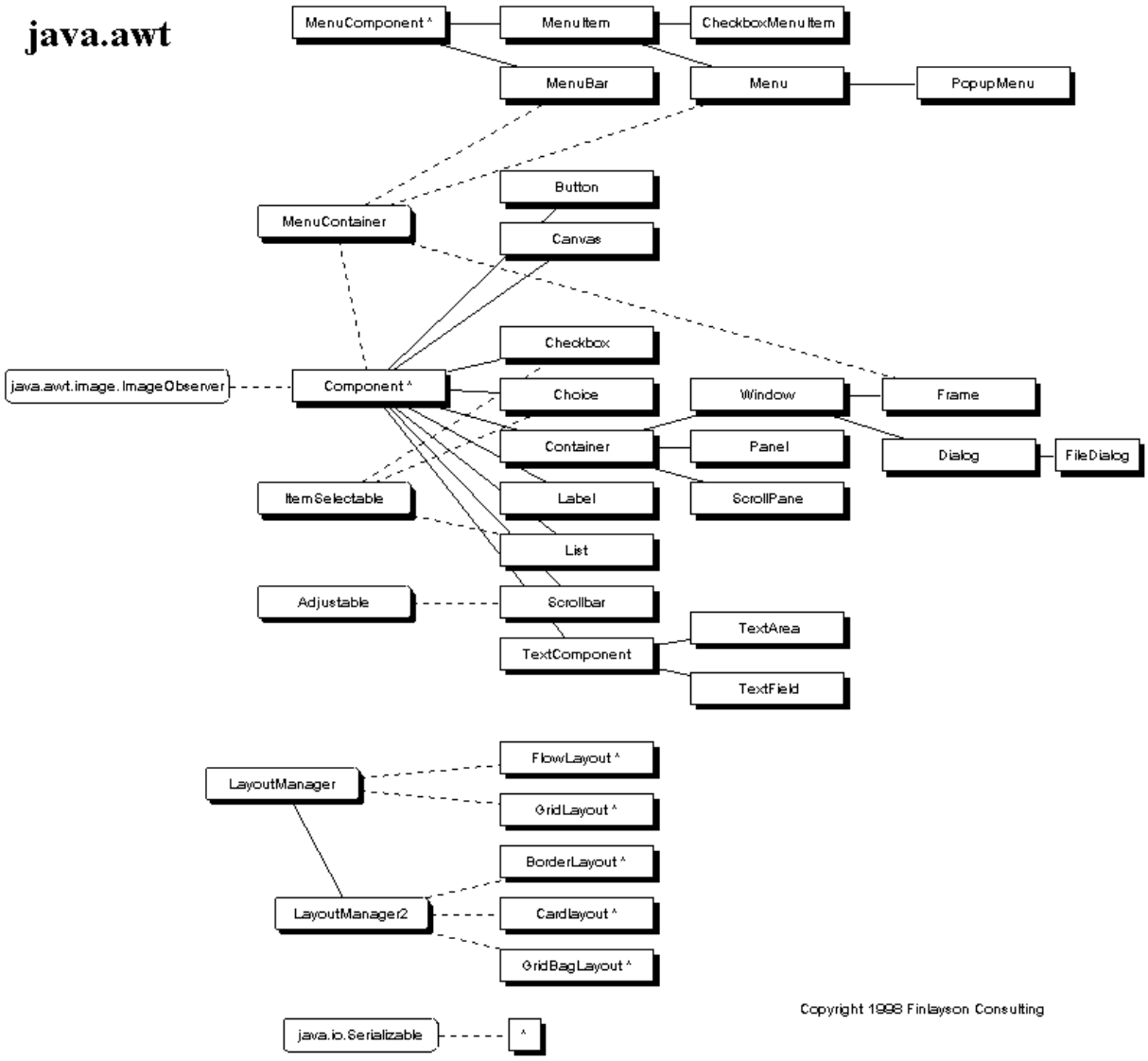



Figure 54 : *Hiérarchie d'héritage des composants visuels AWT*

(source `httphttp://tiki-lounge.com/ raf/jfcmanual/jfc.1.html` )

La figure 53 met en évidence les différents composants AWT. On y trouve des composants de base (labels, boutons, textes, listes, champs de texte, zones de texte, zones de choix, barres de défilement...), des conteneurs (panneaux, fenêtres...), des zones graphiques (cannevas), etc.

Pour utiliser un composant, il suffit de l'ajouter à un conteneur.

La figure 55 met en évidence les différents composants Swing (rappel de la figure 17).

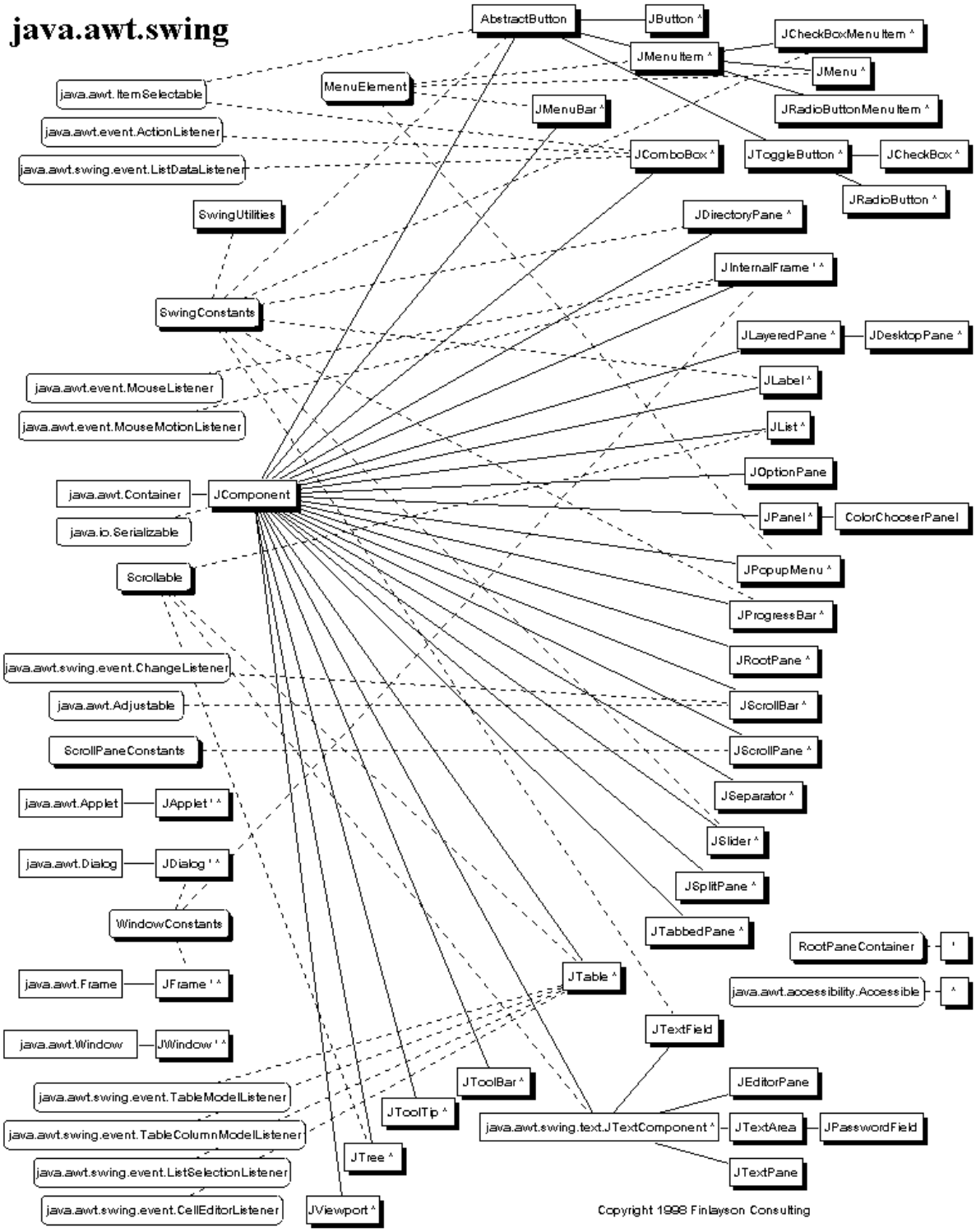



Figure 55 : *Hiérarchie d'héritage des composants visuels*

(source http://tiki-lounge.com/ raf/jfcmanual/jfc.2.html )

La classe `Component` comprend des constantes pour l'alignement des composants. Voici les principales méthodes de la classe `Component` :

| **Méthodes** | **Rôle** |
|---|---|
| Rectangle getBounds() | renvoie la position actuelle et la taille des composants |
| void setEnabled(boolean) | active/désactive les composants |
| Color getBackGround() | renvoie la couleur actuelle d'arrière plan |
| Font getFont() | renvoie la fonte utilisée pour afficher les caractères |
| Color getForeGround() | renvoie la couleur de premier plan |
| Graphics getGraphics() | renvoie le contexte graphique |
| Container getParent() | renvoie le conteneur ( composant de niveau supérieure ) |
| void setVisible(boolean) | affiche/masque l'objet |
| boolean contains(int x, int y) | indique si la coordonnée écran absolue se trouve dans l'objet |
| boolean isEnabled() | indique si l'objet est actif |
| boolean isShowing() | indique si l'objet est visible |
| boolean isVisible() | indique si l'objet est visible lorque sont conteneur est visible |
| boolean isShowing() | indique si une partie de l'objet est visible |
| void doLayout() | repositionne l'objet en fonction du Layout Manager courant |
| Component getComponentAt(Point p) | retourne le composant situé à cet endroit |
| Point getLocation() | retourne l'origine du composant |
| void setLocation(Point p) | déplace les composants vers la position spécifiée |
| void paint(Graphics) ; | dessine le composant |
| void paintAll(Graphics) | dessine le composant et ceux qui sont contenus en lui |
| void repaint() | redessine le composant pat appel à la méthode update() |
| void requestFocus() ; | demande le focus |
| void setBounds(int x, inty, int w, int h) | modifie la position et la taille (unité : points écran) |
| void setSize(int w, int h) | modifie la taille (unité : points écran) |
| void setBackground(Color) | définit la couleur d'arrière plan |
| void setFont(Font) | définit la police |
| void setForeground(Color) | définit la couleur de premier plan |
| void show() | dépréciée : utiliser la méthode setVisible(True). |
| Dimension getSize() | détermine la taille actuelle |

TABLEAU XVI– *Protocole (partiel) de la classe* `java.awt.Component`

### Les conteneurs

Les conteneurs sont des objets graphiques qui peuvent contenir d'autres objets graphiques, incluant éventuellement des conteneurs. Ils héritent de la classe `Container`. Un composant graphique doit toujours être incorporé dans un conteneur. Voici les principaux conteneurs AWT :

| **Conteneur** | **Rôle** |
|---|---|
| Panel | conteneur sans fenêtre propre. Utile pour ordonner les contrôles |
| Window | fenêtre principale sans cadre ni menu. Les objets descendants de cette classe peuvent servir à implémenter des menus |
| Dialog (descendant de Window) | réaliser des boîtes de dialogue simples |
| Frame (descendant de Window) | classe de fenêtre completement fonctionnelle |
| Applet (descendant de Panel) | pas de menu. Pas de boîte de dialogue sans être incorporée dans une classe Frame. |

TABLEAU XVII– Conteneurs AWT

L'insertion de composant dans un conteneur se fait grace à la méthode `add(Component)` de la classe `Container`.

Le conteneur `Panel` est essentiellement un objet de rangement pour d'autres composants. La classe `Panel` possède deux constructeurs : `Panel()` et `Panel(LayoutManager)` qui permet de préciser un layout manager.

La classe `Window` contient entre autres une méthode `void pack()` qui calcule la taille et la position de tous les contrôles de la fenêtre. La méthode `pack()` agit en étroite collaboration avec le layout manager et permet à chaque contrôle de garder, dans un premier temps sa taille optimale. Une fois que tous les contrôles ont leur taille optimale, `pack()` utilise ces informations pour positionner les contrôles. `pack()` calcule ensuite la taille de la fenêtre. L'appel à `pack()` doit se faire à l'intérieur du constructeur de fenêtre après insertion de tous les contrôles. La méthode `void show()` afficher la fenêtre. La méthode `void dispose()` libère les ressources allouée à la fenêtre.

Le conteneur `Frame` permet de créer des fenêtres d'encadrement. Il hérite de la classe `Window` qui ne s'occupe que de l'ouverture de la fenêtre. `Window` ne connait pas les menus ni les bordures qui sont gérés par la classe `Frame`. Dans une applet, elle n'apparaît pas dans le navigateur mais comme une fenêtre indépendante. Le constructeur `Frame(String)` precise le nom de la fenêtre. Les principales méthodes sont :

| **Méthodes** | **Rôle** |
|---|---|
| setCursor(Cursor) | changer le pointeur de la souris dans la fenêtre<br>Exemple : `f.setCursor(Frame.CROSSHAIR_CURSOR);` |
| int getCursor() | déterminer la forme actuelle du curseur |
| Image getIconImage() | déterminer l'icone actuelle de la fenêtre |
| MenuBar getMenuBar() | déterminer la barre de menus actuelle |
| String getTitle() | déterminer le titre de la fenêtre |
| boolean isResizeable() | déteriner si la taille est modifiable |
| void remove(MenuComponent) | supprimer un menu |
| void setIconImage(Image) | définir l'icone de la fenêtre |
| void setMenuBar(MenuBar) | définir la barre de menu |
| void setResizeable(boolean) | définir si la taille peut être modifiée |
| void SetTitle(String) | définir le titre de la fenêtre |

TABLEAU XVIII– *Protocole (partiel) de la classe* `java.awt.Frame`

```
import java.applet.*;
import java.awt.*;
public class AppletFrame extends Applet {
   Frame f;
   public void init () {
      super.init ();
      // insert code to initialize the applet here
      f = new Frame("titre");
      f.add(new Label("hello "));
      setVisible(true); //f.show();
      f.setSize(300, 100);
   }
}
```

Le message « *Warning : Applet window* » est impossible à enlever dans la fenêtre : cela permet d'éviter la création d'une applet qui demande un mot de passe.

Le gestionnaire de mise en page par défaut d'une `Frame` est `BorderLayout` (`FlowLayout` pour une applet).

Exemple : construction d'une fenêtre simple

```
import java.awt.*;
public class MaFrame extends Frame {
```

```
    public MaFrame() {
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        setVisible(true); //show() ; // affiche la fenetre
    }
    public static void main(String[] args) {
        new MaFrame();
    }
}
```

La classe `Dialog` hérite de la classe `Window`. Une boîte de dialogue doit dériver de la Classe `Dialog`. Un objet de la classe `Dialog` doit dépendre d'un objet de la classe `Frame`.

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

public class Apropos extends Dialog {
    public APropos(Frame parent) {
        super(parent, "A propos ", true);
        addWindowListener(new
        AProposListener(this));
        setSize(300, 300);
        setResizable(False);
    }
}

class AProposListener extends WindowAdapter {
    Dialog dialogue;
    public AProposListener(Dialog dialogue) {
        this.dialogue = dialog;
    }
    public void windowClosing(WindowEvent e) {
        dialogue.dispose();
    }
}
```

L'appel du constructeur `Dialog(Frame, String, Boolean)` permet de créer une instance avec comme paramètres : la fenêtre à laquelle appartient la boîte de dialogue, le titre de la boîte, le caractère modale de la boîte.

La méthode `dispose()` de la classe `Dialog` ferme la boîte et libère les ressources associées. Il ne faut pas associer cette action à la méthode `windowClosed()` car dispose provoque l'appel de `windowClosed` ce qui entraînerait un appel récursif infini.

Swing définit d'autres conteneurs. Voici un exemple de conteneurs de conteneurs.

Listing 7.1 – Programmation Swing Java d'un bureau personnel

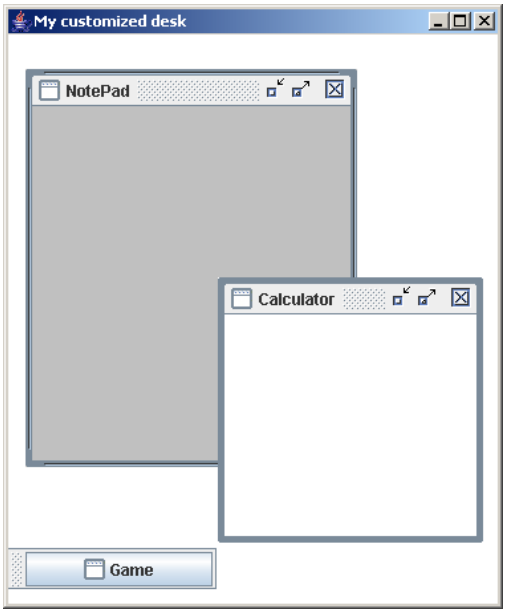
```
package swing;

import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.JInternalFrame;

/**
 * Test des conteneurs JDesktopPane et JinternalFrame
 * source : berthié-briaud
 * @author andre
```

```
 * @date 26/01/06
 */
public class DesktopExample extends JFrame {
    private static final long serialVersionUID = 1L;
    JDesktopPane desk = new JDesktopPane();
    JInternalFrame np = new JInternalFrame("NotePad", true, true, true, true);
    JInternalFrame cal = new JInternalFrame("Calculator", false, true, true,
            true);
    JInternalFrame gam = new JInternalFrame("Game", true, true, true, true);

    public DesktopExample() {
        super();
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(desk, BorderLayout.CENTER);
        this.setSize(450,500);
        this.setTitle("My customized desk");
        desk.add(np);
        desk.add(cal);
        desk.add(gam);
        np.setVisible(true);
        np.setSize(250,300);
        np.setBackground(Color.lightGray);
        cal.setVisible(true);
        cal.setSize(200,200);
        cal.setBackground(Color.white);
        gam.setVisible(true);
        gam.setSize(250,250);
        gam.setBackground(Color.black);
    }
    public static void main(String[] argv)
    {
      DesktopExample de = new DesktopExample();
      de.setVisible(true);
    }
}
```



### Les menus

Il faut insérer les **menus** dans des objets de la classe `Frame` (fenêtre d'encadrement). Il n'est donc pas possible d'insérer directement des menus dans une applet. Il faut créer une barre de menu `MenuBar` et l'affecter à la fenêtre d'encadrement. Il faut créer ensuite créer les menus `Menu` avec des entrées `MenuItem` de chaque sous-menu (et éventuellement des séparateurs `separator`) et les rattacher à la barre. Ajouter ensuite les éléments à chacun des menus.

```
import java.awt.*;

public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        MenuBar mb = new MenuBar();
        setMenuBar(mb);
        Menu m = new Menu(" un menu ");
        mb.add(m);
        m.add(new MenuItem(" 1er element "));
        m.add(new MenuItem(" 2eme element "));
        Menu m2 = new Menu(" sous menu ");
        CheckboxMenuItem cbm1 = new CheckboxMenuItem(" menu item 1.3.1 ");
        m2.add(cbm1);
        cbm1.setState(true);
        CheckboxMenuItem cbm2 = new CheckboxMenuItem(" menu item 1.3.2 ");
        m2.add(cbm2);
        m.add(m2);
        pack();
        setVisible(true); //show();
    // affiche la fenetre
    }
    public static void main(String[] args) {
        new MaFrame();
    }
}
```

Exemple : création d'une classe qui définit un menu

```
import java.awt.*;

public class MenuFenetre extends java.awt.MenuBar {
    public MenuItem menuQuitter, menuNouveau, menuApropos;
    public MenuFenetre() {
        Menu menuFichier = new Menu(" Fichier ");
        menuNouveau = new MenuItem(" Nouveau ");
        menuQuitter = new MenuItem(" Quitter ");
        menuFichier.add(menuNouveau);
        menuFichier.addSeparator();
        menuFichier.add(menuQuitter);
        Menu menuAide = new Menu(" Aide ");
        menuApropos = new MenuItem(" A propos ");
        menuAide.add(menuApropos);
        add(menuFichier);
        setHelpMenu(menuAide);
    }
}
```

La méthode `setHelpMenu()` confère sous certaines plateformes un comportement particulier à ce menu. La méthode `setMenuBar()` de la classe `Frame` prend en paramètre une instance de la classe `MenuBar`. Cette instance peut être directement une instance de la classe `MenuBar` qui aura été modifiée grace aux méthodes `add()` ou alors une classe dérivée de `MenuBar` qui est adaptée aux besoins (voir l'exemple).

```
import java.awt.*;

public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        MenuFenetre mf = new
        MenuFenetre();
        setMenuBar(mf);
        pack();
        setVisible(true); //show(); // affiche la fenetre
    }
    public static void main(String[] args) {
        new MaFrame();
    }
}
```

Le protocole est celui des classes `Menu`, `MenuBar`, `MenuItem` et `CheckboxMenuItem`.

### 2.3 Le placement

Dès qu'il y a plusieurs composants, on doit les positionner les uns par rapport aux autres, soit manuellement en indiquant les coordonnées, soit automatiquement en utilisant un gestionnaire de placement. A chaque conteneur est associé un gestionnaire `LayoutManager`, qui décrit la stratégie de placement. `java.awt.LayoutManager` est une interface avec différentes implantations, décrivant différentes stratégies.

```
BasicComboBoxUI.ComboBoxLayoutManager, BasicInternalFrameTitlePane.TitlePaneLayout,
BasicInternalFrameUI.InternalFrameLayout, BasicOptionPaneUI.ButtonAreaLayout,
BasicScrollBarUI, BasicSplitPaneDivider.DividerLayout, BasicSplitPaneUI.
BasicHorizontalLayoutManager, BasicSplitPaneUI.BasicVerticalLayoutManager,
BasicTabbedPaneUI.TabbedPaneLayout, BorderLayout, BoxLayout, CardLayout,
DefaultMenuLayout, FlowLayout, GridBagLayout, GridLayout, JRootPane.RootLayout,
JSpinner.DateEditor, JSpinner.DefaultEditor, JSpinner.ListEditor, JSpinner.
NumberEditor, MetalComboBoxUI.MetalComboBoxLayoutManager, MetalScrollBarUI,
MetalTabbedPaneUI.TabbedPaneLayout, OverlayLayout, ScrollPaneLayout,
ScrollPaneLayout.UIResource, SpringLayout, ViewportLayout
```

Le placement automatisé a trois avantages : l'aménagement des composants graphiques est délégué aux gestionnaires (il est inutile d'utiliser les coordonnées absolues), en cas de redimensionnement de la fenêtre, les composants sont automatiquement agrandis ou réduits et enfin ils permettent une indépendance vis à vis des plateformes.

Pour supprimer le gestionnaire par défaut d'un conteneur, il faut appeler la méthode `setLayout()` avec comme paramètre `null`. Pour donner à un composant une taille donnée, il faut redéfinir la méthode `getPreferedSize` de la classe `Component`.

```
import java.awt.*;
public class MonBouton extends Button {
    public Dimension getPreferredSize() {
        return new Dimension(800, 250);
```

```
    }
}
```

Le méthode `getPreferedSize()` indique la taille souhaitée mais pas celle imposée. En fonction du gestionnaire, le composant pourra ou non imposer sa taille.

| gestionnaire | Hauteur | Largeur |
|---|---|---|
| Sans Layout | oui | oui |
| FlowLayout | oui | oui |
| BorderLayout(East, West) | non | oui |
| BorderLayout(North, South) | oui | non |
| BorderLayout(Center) | non | non |
| GridLayout | non | non |

(source http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap012.htm)

Lorsqu'on intègre un composant graphique dans un conteneur, il n'est pas nécessaire de préciser son emplacement car il est déterminé de façon automatique par le gestionnaire de mise en page : la mise en forme est dynamique. La position spécifiée est relative par rapport aux autres composants. Chaque gestionnaire implémente l'interface `java.awt.LayoutManager`. Il est possible d'utiliser plusieurs gestionnaires de mise en forme pour définir la présentation des composants. Par défaut, c'est la classe `FlowLayout` qui est utilisée pour la classe `Panel` et la classe `BorderLayout` pour `Frame` et `Dialog`.

Pour affecter une nouvelle mise en page, il faut utiliser la méthode `setLayout()` de la classe `Container`.

```
Panel p = new Panel();
p.setLayout( new GridLayout(5,5));
```

Pour créer un espace entre les composants et le bord de leur conteneur, il faut rédifinir la méthode `getInsets()` d'un conteneur : cette méthode est héritée de la classe `Container`.

```
public Insets getInsets() {
    Insets normal = super.getInsets();
    return new Insets(normal.top + 10, normal.left + 10,
            normal.bottom + 10, normal.right + 10);
}
```

Cet exemple permet de laisser 10 pixels en plus entre chaque bords du conteneur.

La classe `FlowLayout` (mise en page flot) place les composants ligne par ligne de gauche à droite. Chaque ligne est complétée progressivement jusqu'à être remplie, puis passe à la suivante. Chaque ligne est centrée par défaut. C'est la mise en page par défaut des applets. Les constructeurs peuvent être paramétrés par un alignement en utilisant les constantes de classes.

```
import java.awt.*;
public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        setLayout(new FlowLayout());
        add(new Button("Bouton 1"));
        add(new Button("Bouton 2"));
        add(new Button("Bouton 3"));
        pack();
        setVisible(true); //show() ; // affiche la fenetre
    }
```

```
    public static void main(String[] args) {
        new MaFrame();
    }
}
```

Chaque applet possède une mise en page flot implicitement initialisée à `FlowLayout(FlowLayout.CENTER,5,5)`. `FlowLayout` utilise les dimensions de son conteneur comme seul principe de mise en forme des composants. Si les dimensions du conteneurs changent, le positionnement des composants est recalculé.

Avec un `BorderLayout`, la disposition des composants est commandée par une mise en page en bordure qui découpe la surface en cinq zones : North, South, East, West, Center. On peut librement utiliser une ou plusieurs zones. `BorderLayout` consacre tout l'espace du conteneur aux composants. Le composant du milieu dispose de la place inutilisée par les autres composants.

```
import java.awt.*;
public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle("
        Titre de la Fenetre ");
        setSize(300, 150);
        setLayout(new
        BorderLayout());
        add("North", new Button(" bouton haut "));
        add("South", new Button(" bouton bas "));
        add("West", new Button(" bouton gauche "));
        add("East", new Button(" bouton droite "));
        add("Center", new Button(" bouton milieu "));
        pack();
        setVisible(true); //show(); // affiche la fenetre
    }
    public static void
        main(String[] args) {
        new MaFrame();
    }
}
```

Il est possible d'utiliser deux méthodes add surchargées de la classe `Container` : `add(String, Component)` ou le premier paramètre précise l'orientation du composants ou `add(Component, Objet)` ou le second paramètre précise la position sous forme de constante définie dans la classe `BorderLayout`.

```
import java.awt.*;

public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        setLayout(new BorderLayout());
        add(new Button("North"), BorderLayout.NORTH);
        add(new Button("South"), BorderLayout.SOUTH);
        pack();
        setVisible(true); //show(); // affiche la fenetre
    }
    public static void main(String[] args) {
        new MaFrame();
```

```
    }
}
```

La mise en page de type carte `CardLayout` permet de construire des boîtes de dialogue composées de plusieurs onglets. Un onglet se compose généralement de plusieurs contrôles : on insère des panneaux dans la fenêtre utilisée par le `CardLayout` manager. Chaque panneau correspond à un onglet de boîte de dialogue et contient plusieurs contrôles. Par défaut, c'est le premier onglet qui est affiché.

```
import java.awt.*;
public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle("Titre de la Fenetre ");
        setSize(300,150);
        CardLayout cl = new CardLayout();
        setLayout(cl);
        //création d'un panneau contenant les contrôles d'un onglet
        Panel p = new Panel();
        //ajouter les composants au panel
        p.add(new Button("Bouton 1 panneau 1"));
        p.add(new Button("Bouton 2 panneau 1"));
        //inclure le panneau dans la fentre sous le nom "Page1"
        // ce nom est utilisé par show()
        add("Page1",p);
        //déclaration et insertion de l'onglet suivant
        p = new Panel();
        p.add(new Button("Bouton 1 panneau 2"));
        add("Page2", p);
        // affiche la fenetre
        pack();
        setVisible(true); //show();
    }
    public static void main(String[] args) {
        new MaFrame();
    }
}
```

Lors de l'insertion d'un onglet, un nom doit lui être attribué. Les fonctions nécessaires pour afficher un onglet de boîte de dialogue ne sont pas fournies par les méthodes du conteneur, mais seulement par le gestionnaire. Il est nécessaire de sauvegarder temporairement le Layout Manager dans une variable ou déterminer le gestionnaire en cours par un appel à `getLayout()`. Pour appeler un onglet donné, il faut utiliser la méthode `setVisible()` du `CardLayout` Manager.

Les méthodes `first()`, `last()`, `next()` et `previous()` servent à parcourir les onglets de boîte de dialogue :

Le gestionnaire `GridLayout` établit un réseau de cellules identiques qui forment une sorte de quadrillage invisible : les composants sont organisés en lignes et en colonnes. Les éléments insérés dans la grille ont tous la même taille. Les cellules du quadrillage se remplissent de droite à gauche ou de haut en bas. Il existe plusieurs constructeurs : `GridLayout( int, int )` (les deux premiers entiers spécifient le nombre de lignes ou de colonnes de la grille), `GridLayout( int, int, int, int)` (permet de préciser en plus l'espacement horizontal et vertical des composants).

```
import java.awt.*;
public class MaFrame extends Frame {
    public MaFrame() {
```

```
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        setLayout(new GridLayout(2, 3));
        add(new Button("bouton 1"));
        add(new Button("bouton 2"));
        add(new Button("bouton 3"));
        add(new Button("bouton 4"));
        add(new Button("bouton 5 tres long"));
        add(new Button("bouton 6"));
        pack();
        setVisible(true); //show();
    // affiche la fenetre
    }
    public static void main(String[] args) {
        new MaFrame();
    }
}
```

Attention : lorsque le nombre de ligne et de colonne est spécifié alors le nombre de colonne est ignoré. Ainsi par exemple `GridLayout(5,4)` est équivalent à `GridLayout(5,0)`.

```
import java.awt.*;
public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        setLayout(new GridLayout(2, 3));
        add(new Button("bouton 1"));
        add(new Button("bouton 2"));
        add(new Button("bouton 3"));
        add(new Button("bouton 4"));
        add(new Button("bouton 5 tres long"));
        add(new Button("bouton 6"));
        add(new Button("bouton 7"));
        add(new Button("bouton 8"));
        add(new Button("bouton 9"));
        pack();
        setVisible(true); //show();
    // affiche la fenetre
    }
    public static void
        main(String[] args) {
        new MaFrame();
    }
}
```

Le gestionnaire `GridBagLayout` (grille étendue) est le plus riche en fonctionnalités d'AWT : le conteneur est divisé en cellules égales mais un composant peut occuper plusieurs cellules de la grille et il est possible de faire une distribution dans des cellules distinctes. Un objet de la classe `GridBagConstraints` permet de donner les indications de positionnement et de dimension à l'objet `GridBagLayout`.

Les lignes et les colonnes prennent naissance au moment ou les contrôles sont ajoutés. Chaque contrôle est associé à un objet de la classe `GridBagConstraints` qui indique l'emplacement voulu pour le contrôle.

```
GridBagLayout gb1 = new GridBagLayout( );
```

GridBagConstraints gbc = new GridBagConstraints( );

Les variables d'instances pour manipuler l'objet `GridBagLayoutConstrainst` sont :

- `gridx` et `gridy` Ces variables contiennent les coordonnées de l'origine de la grille. Elles permettent un positionnement précis à une certaine position d'un composant. Par défaut elles ont la valeur `GrigBagConstraint.RELATIVE` qui indique qu'un composant se range à droite du précédent
- `gridwidth`, `gridheight` Définissent combien de cellules va occuper le composant (en hauteur et largeur). Par défaut la valeur est 1. L'indication est relative aux autres composants de la ligne ou de la colonne. La valeur `GridBagConstraints.REMAINDER` spécifie que le prochain composant inséré sera le dernier de la ligne ou de la colonne courante. La valeur `GridBagConstraints.RELATIVE` place le composant après le dernier composant d'une ligne ou d'une colonne.
- `fill` Définit le sort d'un composant plus petit que la cellule de la grille.
  `GridBagConstraints.NONE` conserve la taille d'origine : valeur par défaut
  `GridBagConstraints.HORIZONTAL` dilaté horizontalement
  `GridBagConstraints.VERTICAL` dilaté verticalement
  `GridBagConstraints.BOTH` dilatés aux dimensions de la cellule
- `ipadx`, `ipady` Permettent de définir l'agrandissement horizontal et vertical des composants. Ne fonctionne que si une dilatation est demandée par fill. La valeur par défaut est (0,0).
- `anchor` Lorsqu'un composant est plus petit que la cellule dans laquelle il est inséré, il peut être positionné à l'aide de cette variable pour définir le côté par lequel le contrôle doit être aligné dans la cellule. Les variables possibles sont NORTH, NORTHWEST, NORTHEAST, SOUTH, SOUTHWEST, SOUTHEAST, WEST et EAST
- `weightx`, `weighty` Permettent de définir la répartition de l'espace en cas de changement de dimension

```
import java.awt.*;
public class MaFrame extends Frame {
   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      Button b1 = new Button(" bouton 1 ");
      Button b2 = new Button(" bouton 2 ");
      Button b3 = new Button(" bouton 3 ");
      GridBagLayout gb = new GridBagLayout();
      GridBagConstraints gbc = new GridBagConstraints();
      setLayout(gb);
      gbc. fill  = GridBagConstraints.BOTH;
      gbc.weightx = 1;
      gbc.weighty = 1;
      gb.setConstraints(b1, gbc); // mise en forme des objets
      gb.setConstraints(b2, gbc);
      gb.setConstraints(b3, gbc);
      add(b1);
      add(b2);
      add(b3);
      pack();
      setVisible(true); //show() ;
   // affiche la fenetre
   }
   public static void main(String[] args) {
      new MaFrame();
```

```
    }
}
```

Cet exemple place trois boutons l'un à côté de l'autre. Ceci permet en cas de changement de dimension du conteneur de conserver la mise en page : la taille des composants est automatiquement ajustée. Pour placer les trois boutons l'un au dessus de l'autre, il faut affecter la valeur 1 à la variable `gbc.gridx`.

Il est possible de définir de nouveau composant qui hérite directement de `Panel`

```
class PanneauClavier extends Panel {
   PanneauClavier()
   {
      setLayout(new GridLayout(4,3));
      for (int num=1; num ≤9 ; num++) add(new Button(Integer.toString(num)));
      add(new Button("*");
      add(new Button("0");
      add(new Button("#");
   }
}
public class demo exteds Applet {
   public void init () { add(new PanneauClavier()); }
}
```

L'activation ou la désactivation d'un composant se fait par la méthode `setEnabled(boolean)`. La valeur booléenne passée en paramètres indique l'état du composant (`false` : interdit l'usage du composant). Cette méthode est un moyen d'interdire à un composant d'envoyer des événements utilisateurs.

### 2.4 Les événements

La plupart des exemples précédents ne définissent pas d'interaction avec l'utilisateur, notamment pour fermer la fenêtre. L'*applet viewer* permet, par défaut d'avoir un tel menu. Mais pour que le I de IHM soit valide, il faut capter les intentions de l'utilisateur, sous forme d'événement. On a donc besoin de définir les intentions, ce sont les **événements**, et un système de capteurs et d'actionneurs, ce sont les **écouteurs** (ou auditeurs ou *listeners*).

Les événements sont définis par la hiérarchie de `java.util.EventObject`. Ils sont souvent associés à des types de composants (boutons, listes, onglets...). Les écouteurs sont définis par la hiérarchie de `java.util.EventListener`. Ces deux hiérarchies sont illustrées dans la section 2.7 du chapitre 3. Elles sont relativement complexes.

Les événements utilisateurs sont gérés par plusieurs interfaces `EventListener`. Les interfaces `EventListener` permettent à un composant de générer des événements utilisateurs. Une classe doit contenir une interface écouteur pour chaque type de composant :

- `ActionListener` : clic de souris ou enfoncement de la touche `Enter`
- `ItemListener` : utilisation d'une liste ou d'une case à cocher
- `MouseMotionListener` : événement de souris
- `WindowListener` : événement de fenêtre

L'ajout d'une interface `EventListener` impose plusieurs ajouts dans le code :

1. Importer le groupe de classe par
   `import java.awt.event.*;`
2. La classe doit déclarer qu'elle utilisera une ou plusieurs interfaces d'écoute
   `implements ActionListener, MouseListener`...
3. Appel à la méthode `addXXX()` pour enregistrer l'objet qui gère les événements `XXX` du composant. Il faut configurer le composant pour qu'il possède un écouteur adapté à l'événement utilisateur concerné. Par exemple

`Button b = new Button(ńbouttonż);`
`b.addActionListener(this);`
Ce code crée l'objet de la classe `Button` et appelle sa méthode `addActionListener()`. Cette méthode permet de préciser qu'elle sera la classe qui va gérer l'événement utilisateur de type `ActionListener` du bouton. Cette classe doit impérativement implanter l'interface de type `EventListener` correspondante soit `ActionListener` ici. La pseudo-variable `this` indique que la classe elle-même recevra et gérera l'événement utilisateur.

4. Implanter les méthodes déclarées dans les interfaces. Chaque écouteur possède des méthodes pour traiter différents types d'événements. Par exemple, l'interface `ActionListener` envoie des événements à une classe par
`public void actionPerformed(ActionEvent evt).`
Pour identifier le composant qui a généré l'événement il faut utiliser la méthode `getActionCommand()` de l'objet `ActionEvent` fourni en paramètre de la méthode :
`String composant = evt.getActionCommand();`
`getActionCommand` renvoie une chaîne de caractères. Si le composant est un bouton, alors il renvoie le texte du bouton, si le composant est une zone de saisie, c'est le texte saisie qui sera renvoyé (il faut appuyer sur "Entrer" pour générer l'événement), etc ...
La méthode `getSource()` renvoie l'objet qui a généré l'événement. Cette méthode est plus sûre que la précédente

```
Button b = new Button("bouton");
...
void public actionPerformed(actionEvent evt) {
   Object source = evt.getSource();
   if (source == b) // action a effectuer
}
```

La méthode `getSource()` peut être utilisé avec tous les événements utilisateur. L'exemple suivant affiche le composant qui a généré l'événement AWT.

```
package swing;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletAction extends Applet implements ActionListener{

   public void actionPerformed(ActionEvent evt) {
      String composant = evt.getActionCommand();
      showStatus("Action sur le composant : " + composant);
   }
   public void init () {
      super.init ();

      Button b1 = new Button("boutton 1");
      b1.addActionListener(this);
      add(b1);

      Button b2 = new Button("boutton 2");
      b2.addActionListener(this);
      add(b2);

      Button b3 = new Button("boutton 3");
      b3.addActionListener(this);
      add(b3);
```

```
    }
}
```

L'interface `ItemListener` permet de réagir à la sélection de cases à cocher et de liste d'options. Pour qu'un composant genère des événements, il faut utiliser la méthode `addItemListener()`.

```
Checkbox cb = new Checkbox(ń choix ż,true);
cb.addItemListener(this);
```

Ces événements sont reçus par la méthode `itemStateChanged()` qui attend un objet de type `ItemEvent` en argument. Pour déterminer si une case à cocher est sélectionnée ou inactive, utiliser la méthode `getStateChange()` avec les constantes `ItemEvent.SELECTED` ou `ItemEvent.DESELECTED`. Pour connaître l'objet qui a généré l'événement, il faut utiliser la méthode `getItem()`. Pour déterminer la valeur sélectionnée dans une combo box, il faut utiliser la méthode `getItem()` et convertir la valeur en chaîne de caractères.

L'interface `TextListener` permet de réagir aux modifications de la zone de saisie ou du texte. La méthode `addTextListener()` permet à un composant de texte de générer des événements utilisateur. La méthode `TextValueChanged()` reçoit les événements. Exemple ( code Java 1.1 ) :

L'interface `MouseMotionListener` gère les événements liés au déplacement de la souris. La méthode `addMouseMotionListener()` permet de gérer les événements liés à des mouvements de souris. La méthode `mouseDragged()` et `mouseMoved()` reçoivent les événements.

```
package swing;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMotion extends Applet implements MouseMotionListener {
    private int x;
    private int y;
    public void init() {
        super.init();
        this.addMouseMotionListener(this);
    }
    public void mouseDragged(java.awt.event.MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {
        x = e.getX();
        y = e.getY();
        repaint();
        showStatus("x = "+x+" ; y = "+y);
    }
    public void paint(Graphics g) {
        super.paint(g);
        g.drawString("x = "+x+" ; y = "+y,20,20);
    }
}
```

L'interface `MouseMotionListener` gère les événements liés aux commandes de la souris. Les méthodes de cette interface sont :

- `public void mouseClicked(MouseEvent e);`
- `public void mousePressed(MouseEvent e);`
- `public void mouseReleased(MouseEvent e);`
- `public void mouseEntered(MouseEvent e);`
- `public void mouseExited(MouseEvent e);`

```
package swing;
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMouse extends Applet implements MouseListener {
    int nbClick = 0;
    public void init () {
        super.init ();
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e) {
        nbClick++;
        repaint ();
    }
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void paint(Graphics g) {
        super.paint(g);
        g.drawString("Nombre de clics : "+nbClick,10,10);
    }
}
```

Une classe qui implante cette interface doit définir ces cinq méthodes. Si toutes les méthodes ne doivent pas être utilisées, il est possible de définir une classe qui hérite de `MouseAdapter`. Cette classe fournit une implémentation par défaut de l'interface `MouseListener`.

Dans le cas d'une classe qui hérite d'une classe `Adapter`, il suffit de redéfinir la ou les méthodes qui contiendront du code pour traiter les événements concernés. Par défaut, les différentes méthodes définies d'un adaptateur sont vides.

Cette nouvelle classe ainsi définie doit être passée en paramètre à la méthode `addMouseListener()` au lieu de `this` qui indiquait que la classe répondait elle-même à l'événement.

Dans l'interface `WindowListener`, la méthode `addWindwowListener()` permet à un objet `Frame` de générer des événements. Les méthodes de cette interface sont :

- `public void windowOpened(WindowEvent e)`
- `public void windowClosing(WindowEvent e)`
  appelée lorque l'on clique sur la case système de fermeture de la fenêtre.
- `public void windowClosed(WindowEvent e)`
  appelée après la fermeture de la fenêtre : cette méthode n'est utile que si la fermeture de la fenêtre n'entraîne pas la fin de l'application.
- `public void windowIconified(WindowEvent e)`
- `public void windowDeinconified(WindowEvent e)`
- `public void windowActivated(WindowEvent e)`
- `public void windowDeactivated(WindowEvent e)`

```
package swing;
import java.awt.event.*;
class GestionnaireFenetre extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

```
package swing;
import java.awt.*;
```

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

public class TestFrame extends Frame {
    private GestionnaireFenetre gf = new GestionnaireFenetre();
    public TestFrame(String title) {
      super( title );
      addWindowListener(gf);
    }
    public static void main(java.lang.String[] args) {
       try {
          TestFrame tf = new TestFrame("TestFrame");
          tf.setVisible(true);
       } catch (Throwable e) {
          System.err.println("Erreur");
          e.printStackTrace(System.out);
       }
    }
}
```

### Implantation et adaptateurs

La mise en oeuvre des écouteurs peut se faire selon différentes formes : la classe implémentant elle même l'interface, une classe indépendante, une classe interne, une classe interne anonyme.

Revenons sur la notion d'adaptateur(classe `Adapter`). Un adaptateur est une implantation par défaut d'un écouteur.

### 2.5 Bilan

Dans cette section nous avons présenté les principaux éléments d'une IHM Java, à savoir les composants, les fenêtres et le contexte graphique, les événements et les écouteurs. On retrouve tous ces éléments dans AWT. Dans la section suivante, nous présentons quelques aspects spécifiques à Swing.

## 3 Les concepts spécifiques à Swing

Comme nous l'avons vu dans les sections précédentes, les composants `Swing` forment un hiérarchie similaire à celle d'AWT dont les composants lourds sont rattachés à ceux d'AWT et les composants légers sont rattachés au composant `JComponent`. Tous les éléments de Swing font partie d'un package qui a changé plusieurs fois de nom : le nom du package dépend de la version du JDK utilisée :

- `com.sun.java.swing` : jusqu'à la version 1.1 beta 2 de Swing, de la version 1.1 des JFC et de la version 1.2 beta 4 du J.D.K.
- `java.awt.swing` : utilisé par le J.D.K. 1.2 beta 2 et 3
- `javax.swing` : à partir des versions de Swing 1.1 beta 3 et J.D.K. 1.2 RC1

Tous les composants `Swing` possèdent les caractéristiques suivantes :

- ce sont des beans,
- ce sont des composants légers (pas de partie native), hormis quelques exceptions,
- leurs bords peuvent être changés.

La procédure à suivre pour utiliser un composant Swing est identique à celle des composants de la bibliothèque AWT, présentée dans la section précédente : créer le composant en appelant son constructeur, appeler les méthodes du composant si nécessaire pour le personnaliser et l'ajouter dans un conteneur. Swing utilise la même infrastructure de classes que

AWT, ce qui permet de mélanger des composants Swing et AWT dans la même interface. Sun recommande toutefois d'éviter de les mélanger car certains peuvent ne pas être restitués correctement.

Les composants Swing utilisent des modèles pour contenir leurs états ou leur données. Ces modèles sont des classes particulières qui possèdent toutes un comportement par défaut. On retrouve le modèle MVC ou MC Java.

| **Paquetage** | **Contenu** |
|---|---|
| javax.swing | package principal : il contient les interfaces, les principaux composants, les modèles par défaut |
| javax.swing.border | Classes représentant les bordures |
| javax.swing.colorchooser | Classes définissant un composant pour la sélection de couleurs |
| javax.swing.event | Classes et interfaces pour les événements spécifiques à Swing. Les autres événements sont ceux de AWT (java.awt.event) |
| javax.swing.filechooser | Classes définissant un composant pour la sélection de fichiers |
| javax.swing.plaf | Classes et interfaces génériques pour gérer l'apparence |
| javax.swing.plaf.basic | Classes et interfaces de base pour gérer l'apparence |
| javax.swing.plaf.metal | Classes et interfaces pour définir l'apparence Metal qui est l'apparence par défaut |
| javax.swing.table | Classes définissant un composant pour la présentation de données sous forme de tableau |
| javax.swing.text | Classes et interfaces de bases pour les composants manipulant du texte |
| javax.swing.text.html | Classes permettant le support du format HTML |
| javax.swing.text.html .parser | Classes permettant d'analyser des données au format HTML |
| javax.swing.text.rtf | Classes permettant le support du format RTF |
| javax.swing.tree | Classes définissant un composant pour la présentation de données sous forme d'arbre |
| javax.swing.undo | Classes permettant d'implémenter les fonctions annuler/refaire |

La classe de base d'une application est la classe `JFrame`. Elle s'utilise comme la classe `Frame` de l'AWT.

```
package swing;
import javax.swing.*;
import java.awt.event.*;
public class TestJFrame extends JFrame {
    public TestJFrame() {
        super("TestJFrame");
        WindowListener l = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        };
        addWindowListener(l);
        setSize(200, 100);
        setVisible(true);
    }
    public static void main(String[] args) {
        JFrame frame = new TestJFrame();
    }
}
```

Le lecteur notera que cette fois l'application se termine à la fermeture de la fenêtre.

### 3.1 Les composants Swing

Il existe des composants Swing équivalents pour chacun des composants AWT avec des constructeurs semblables. De nombreux constructeurs acceptent comme argument un objet de type `Icon`, qui représente une petite image généralement stockée au format Gif. Le constructeur d'un objet `Icon` admet comme seul paramètre le nom ou l'URL d'un fichier graphique

```
package swing;
import javax.swing.*;
import java.awt.event.*;
public class TestJFrame1 extends JFrame {
   public TestJFrame1() {
      super("TestJFrame 1");
         WindowListener l = new WindowAdapter() {
             public void windowClosing(WindowEvent e){
                System.exit(0);
             }
          };
          addWindowListener(l);
          ImageIcon img = new ImageIcon("j2se5.gif");
          JButton bouton = new JButton("J2SE5",img);
          JPanel panneau = new JPanel();
          panneau.add(bouton);
          setContentPane(panneau);
          setSize(750,500);
          setVisible(true);
   }
   public static void main(String[] args) {
      JFrame frame = new TestJFrame1();
   }
}
```

Retour sur la classe `JFrame`, qui est l'équivalent de la classe Frame de l'AWT : les principales différences sont l'utilisation du double buffering qui améliore les rafraichissements et l'utilisation d'un panneau de contenu (contentPane) pour insérer des composants (ils ne sont plus insérés directement au JFrame mais à l'objet contentPane qui lui est associé). Elle représente une fenêtre principale qui possède un titre, une taille modifiable et éventuellement un menu. La gestion des événements est identique à celle utilisée dans l'AWT depuis le J.D.K. 1.1.

```
package swing;
import javax.swing.*;
import java.awt.event.*;
public class TestJFrame2 extends JFrame {
   public TestJFrame2() {
      super("TestJFrame 2");
         WindowListener l = new WindowAdapter() {
             public void windowClosing(WindowEvent e){
               System.exit(0);
             }
          };
         addWindowListener(l);
         JButton bouton = new JButton("Cliquer ici");
         JPanel panneau = new JPanel();
         panneau.add(bouton);
         setContentPane(panneau);
         setSize(200,100);
         setVisible(true);
```

```
    }
    public static void main(String[] args) {
        JFrame frame = new TestJFrame2();
    }
}
```

Comme l'indique la figure 52, tous les composants associés à un objet `JFrame` sont gérés par un objet de la classe `JRootPane`. Un objet `JRootPane` contient plusieurs « carreaux », avec au moins une « vitre » (`glassPane`, un `JPanel` par défaut) et un empilement (« vitre feuilletée » `layeredPane`).
Le *glassPane* est un `JPanel` transparent qui se situe au dessus du *layeredPane*. Le *glassPane* peut être n'importe quel composant : pour le modifier il faut utiliser la méthode `setGlassPane()` en fournissant le composant en paramètre.
L'empilement se compose d'un contenu *contentPane* (`JPanel` par défaut) et d'une barre de menus *menuBar* (`JMenuBar` par défaut). Le *contentPane* est par défaut un `JPanel` opaque dont le gestionnaire de présentation est un `BorderLayout`. Ce carreau peut être remplacé par n'importe quel composant grâce à la méthode `setContentPane()`.

Tous les composants ajoutés au `JFrame` doivent être ajoutés à un des carreaux du `JRootPane` et non au `JFrame` directement. C'est aussi à une de ces carreaux qu'il faut associer un gestionnaire de placement si nécessaire. La classe de *panes* la plus utilisée est `ContentPane`. Le gestionnaire de placement par défaut est `BorderLayout`.
Attention : il ne faut pas utiliser directement la méthode `setLayout()` d'un objet `JFrame` sinon une exception est levée.

La barre de menu permet d'attacher un menu à la `JFrame`. Par défaut, elle est vide. La méthode `setJMenuBar()` permet d'affecter un menu à la `JFrame`.

```
public static void main(String argv[]) {
    JFrame f = new JFrame("FileManager");
    f.setSize(300,100);
    JButton b =new JButton("Open");
    f.getContentPane().add(b);
    JMenuBar menuBar = new JMenuBar();
    f.setJMenuBar(menuBar);
    JMenu menu = new JMenu("File");
    JMenuItem menuItem = new JMenuItem("Open");
    menu.add(menuItem);
    menuBar.add(menu);
    f.setVisible(true);
```

Il est possible de préciser comment un objet `JFrame`, `JInternalFrame`, ou `JDialog` réagit à sa fermeture grâce à la méthode `setDefaultCloseOperation()`. Cette méthode attend en paramètre une valeur qui peut être :

- `WindowConstants.DISPOSE_ON_CLOSE` : détruit la fenêtre
- `WindowConstants.DO_NOTHING_ON_CLOSE` : rend le bouton de fermeture inactif
- `WindowConstants.HIDE_ON_CLOSE` : cache la fenêtre

Cette méthode ne permet pas d'associer d'autres traitements. Dans ce cas, il faut intercepter l'événement et lui associer les traitements. Par exemple, la fenêtre disparaît lors de sa fermeture mais l'application ne se termine pas.

```
public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);
    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    f.setVisible(true);
```

```
}
```

La méthode `setIconImage()` permet de modifier l'icône de la `JFrame`.

```
public static void main(String argv[]) {
  JFrame f = new JFrame("ma fenetre");
  f.setSize(300,100);
  JButton b =new JButton("Mon bouton");
  f.getContentPane().add(b);
  f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
  ImageIcon image = new ImageIcon("book.gif");
  f.setIconImage(image.getImage());
  f.setVisible(true);
}
```

Si l'image n'est pas trouvée, alors l'icône est vide. Si l'image est trop grande, elle est redimensionnée.

Par défaut, une `JFrame` est affichée dans le coin supérieur gauche de l'écran. Pour la centrer dans l'écran, il faut procéder comme pour une `Frame` : déterminer la position de la `Frame` en fonction de sa dimension et de celle de l'écran et utiliser la méthode `setLocation()` pour affecter cette position.

```
public static void main(String argv[]) {
  JFrame f = new JFrame("ma fenetre");
  f.setSize(300,100);
  JButton b =new JButton("Mon bouton");
  f.getContentPane().add(b);
  f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
  Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
  f.setLocation(dim.width/2 - f.getWidth()/2, dim.height/2 - f.getHeight()/2);
  f.setVisible(true);
}
```

La gestion des événements associés à un objet `JFrame` est identique à celle utilisée pour un objet de type `Frame` de AWT.

```
public static void main(String argv[]) {
  JFrame f = new JFrame("ma fenetre");
  f.setSize(300,100);
  f.setVisible(true);
  f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
  );
}
```

Le composant `JLabel` propose les mêmes fonctionnalités que les intitulés AWT mais ils peuvent en plus contenir des icônes en précisant leur alignement. L'icône doit être au format GIF et peut être une animation dans ce format.

```
public static void main(String argv[]) {

  JFrame f = new JFrame("ma fenetre");
  f.setSize(100,200);

  JPanel pannel = new JPanel();
  JLabel jLabel1 =new JLabel("Mon texte dans JLabel");
```

```
    pannel.add(jLabel1);

    ImageIcon icone = new ImageIcon("book.gif");
    JLabel jLabel2 =new JLabel(icone);
    pannel.add(jLabel2);

    JLabel jLabel3 =new JLabel("Mon texte",icone,SwingConstants.LEFT);
    pannel.add(jLabel3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
```

La classe `JLabel` définit plusieurs méthodes pour modifier l'apparence du composant :

| **Méthodes** | **Rôle** |
|---|---|
| setText() | Permet d'initialiser ou de modifier le texte affiché |
| setOpaque() | Indique si le composant est transparent (paramètre false) ou opaque (true) |
| setBackground() | Indique la couleur de fond du composant (setOpaque doit être à true) |
| setFont() | Permet de préciser la police du texte |
| setForeGround() | Permet de préciser la couleur du texte |
| setHorizontalAlignment() | Permet de modifier l'alignement horizontal du texte et de l'icône |
| setVerticalAlignment() | Permet de modifier l'alignement vertical du texte et de l'icône |
| setHorizontalTextAlignment() | Permet de modifier l'alignement horizontal du texte uniquement |
| setVerticalTextAlignment() | Permet de modifier l'alignement vertical du texte uniquement Exemple : jLabel.setVerticalTextPosition(SwingConstants.TOP) ; |
| setIcon() | Permet d'assigner une icône |
| setDisabledIcon() | Permet d'assigner une icône dans un état désactivée |

L'alignement vertical par défaut d'un `JLabel` est centré. L'alignement horizontal par défaut est soit à droite si il ne contient que du texte, soit centré si il contient un image avec ou sans texte. Pour modifier cet alignement, il suffit d'utiliser les méthodes ci dessus en utilisant des constantes de la classe `SwingConstants` en paramètres.

Par défaut, un label swing est transparent : son fond n'est pas dessiné. Pour le dessiner, il faut utiliser la méthode `setOpaque()` :

```
  public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(100,200);
    JPanel pannel = new JPanel();
    JLabel jLabel1 =new JLabel("Mon texte dans JLabel 1");
    jLabel1.setBackground(Color.red);
    pannel.add(jLabel1);
    JLabel jLabel2 =new JLabel("Mon texte dans JLabel 2");
    jLabel2.setBackground(Color.red);
    jLabel2.setOpaque(true);
    pannel.add(jLabel2);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
```

Dans cet exemple, les deux labels ont le fond rouge demandé par la méthode `setBackground()`. Seul le deuxième affiche un fond rouge car il est rendu opaque avec la méthode `setOpaque()`.

Il est possible d'associer un raccourci clavier au label qui permet de donner le focus à un autre composant. La méthode `setDisplayedMnemonic()` permet de définir le raccourci

clavier. Celui-ci sera activé en utilisant la touche Alt avec le caractère fourni en paramètre. La méthode `setLabelFor()` permet d'associer le composant fourni en paramètre au raccourci.

```
public static void main(String argv[]) {
  JFrame f = new JFrame("ma fenetre");
  f.setSize(300,100);
  JPanel pannel = new JPanel();
  JButton bouton = new JButton("saisir");
  pannel.add(bouton);
  JTextField jEdit = new JTextField("votre nom");
  JLabel jLabel1 =new JLabel("Nom : ");
  jLabel1.setBackground(Color.red);
  jLabel1.setDisplayedMnemonic('n');
  jLabel1.setLabelFor(jEdit);
  pannel.add(jLabel1);
  pannel.add(jEdit);
  f.getContentPane().add(pannel);
  f.setVisible(true);
}
```

Dans cet exemple, à l'ouverture de la fenêtre, le focus est sur le bouton. Un appui sur Alt+'n' donne le focus au champ de saisie.

La classe `JPanel` est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation. Le gestionnaire par défaut d'un panneau Swing est une instance de la classe `FlowLayout`.

### 3.2 Les boutons

Il existe plusieurs boutons définis par Swing (figure 17). La classe abstraite `AbstractButton` définit le comportement des boutons Swing `JButton`, `JMenuItem`, `JToggleButton`...

| **Méthodes** | **Rôle** |
|---|---|
| AddActionListener | Associer un écouteur sur un évenement de type ActionEvent |
| AddChangeListener | Associer un écouteur sur un évenement de type ChangeEvent |
| AddItemListener | Associer un écouteur sur un évenement de type ItemEvent |
| doClick() | Déclencher un clic par programmation |
| getText() | Obtenir le texte affiché par le composant |
| setDisabledIcon() | Associer une icône affichée lorsque le composant à l'état désélectionné |
| setDisabledSelectedIcon() | Associer une icône affichée lors du passage de la souris sur le composant à l'état désélectionné |
| setEnabled() | Activer/désactiver le composant |
| setMnemonic() | Associer un raccourci clavier |
| setPressedIcon() | Associer une icône affichée lorsque le composant est cliqué |
| setRolloverIcon() | Associer une icône affichée lors du passage de la souris sur le composant |
| setRolloverSelectedIcon() | Associer une icône affichée lors du passage de la souris sur le composant à l'état sélectionné |
| setSelectedIcon() | Associer une icône affichée lorsque le composant à l'état sélectionné |
| setText() | Mettre à jour le texte du composant |
| isSelected() | Indiquer si le composant est dans l'état sélectionné |
| setSelected() | Mettre à jour l'état sélectionné du composant |

Tous les boutons peuvent afficher du texte et/ou une image. Il est possible de préciser une image différente lors du passage de la souris sur le composant et lors de l'enfoncement du bouton : dans ce cas, il faut créer trois images pour chacun des états (normal, enfoncé et survolé).

L'image normale est associée au bouton grâce au constructeur, l'image enfoncée grâce à la méthode `setPressedIcon()` et l'image lors d'un survole grâce à la méthode `setRolloverIcon()`. Il suffit enfin d'appeler la méthode `setRolloverEnable()` avec en paramètre la valeur true.

```
package swing;
import javax.swing.*;
import java.awt.event.*;
public class TestSwing1 extends JFrame {
   public TestSwing1() {
     super("Tests de boutons");
     WindowListener l = new WindowAdapter() {
       public void windowClosing(WindowEvent e){
         System.exit(0);
       }
     };
     addWindowListener(l);
     ImageIcon imageNormale = new ImageIcon("arrow.gif");
     ImageIcon imagePassage = new ImageIcon("arrowr.gif");
     ImageIcon imageEnfoncee = new ImageIcon("arrowy.gif");
     JButton bouton = new JButton("Mon bouton",imageNormale);
     bouton.setPressedIcon(imageEnfoncee);
     bouton.setRolloverIcon(imagePassage);
     bouton.setRolloverEnabled(true);
     getContentPane().add(bouton, "Center");
     JPanel panneau = new JPanel();
     panneau.add(bouton);
     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }
   public static void main(String [] args){
      JFrame frame = new TestSwing1();
   }
}
```

Un bouton peut recevoir des évenements de type `ActionEvents` (le bouton a été activé), `ChangeEvents`, et `ItemEvents`. Par exemple, prenons la fermeture de l'application lors de l'activation du bouton.

```
  public static void main(String argv[]) {
    JFrame f = new JFrame("Test bouton");
    f.setSize(300,100);
    JPanel panneau = new JPanel();
    JButton bouton1 = new JButton("Bouton No 1");
    bouton1.addActionListener( new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         System.exit(0);
       }
    }
    );
    panneau.add(bouton1);
    f.getContentPane().add(panneau);
    f.setVisible(true);
  }
```

`JButton` est un composant qui représente un bouton : il peut contenir un texte et/ou une icône. Dans un conteneur de type `JRootPane`, il est possible de définir un bouton par défaut grâce à sa méthode `setDefaultButton()`. Le bouton par défaut est activé par un appui sur

la touche Entrée alors que le bouton actif est activé par un appui sur la barre d'espace. La méthode `isDefaultButton()` de `JButton` permet de savoir si le composant est le bouton par défaut.

La classe `JToggleButton` définit un bouton à deux états : c'est la classe mère des composants `JCheckBox` et `JRadioButton`. La méthode `setSelected()` héritée de `AbstractButton` permet de mettre à jour l'état du bouton. La méthode `isSelected()` permet de connaître cet état.

La classe `ButtonGroup` permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné. Pour utiliser la classe `ButtonGroup`, il suffit d'instancier un objet et d'ajouter des boutons (objets héritant de la classe `AbstractButton`) grâce à la méthode `add()`. Il est préférable d'utiliser des objets de la classe `JToggleButton` ou d'une de ces classes filles car elles sont capables de gérer leurs états.

```
public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();
    ButtonGroup groupe = new ButtonGroup();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    groupe.add(bouton1);
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    groupe.add(bouton2);
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    groupe.add(bouton3);
    pannel.add(bouton3);
    f.getContentPane().add(pannel);
    f.setVisible(true);
}
```

Un groupe de cases à cocher peut être défini avec la classe `JCheckBox`. Dans ce cas, un seul composant du groupe peut être sélectionné. Pour l'utiliser, il faut créer un objet de la classe `JCheckBox` et utiliser la méthode `add()` pour ajouter un composant au groupe. On fait de même avec la classe `JRadioButton`.

```
public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JCheckBox bouton1 = new JCheckBox("Bouton 1");
    pannel.add(bouton1);
    JCheckBox bouton2 = new JCheckBox("Bouton 2");
    pannel.add(bouton2);
    JCheckBox bouton3 = new JCheckBox("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
}
```

### 3.3 Les composants textuels

Swing possède plusieurs composants pour permettre la saisie de textes (figure 17). La classe abstraite `JTextComponent` est la classe racine de tout les composants de texte. Les

données du composant (le modèle dans le motif de conception MVC) sont encapsulées dans un objet qui implémente l'interface `Document`. Deux classes implémentant cette interface sont fournies en standard : `PlainDocument` pour du texte simple et `StyledDocument` pour du texte riche pouvant contenir entre autre plusieurs polices de caractères, des couleurs, des images, ...

La classe JTextComponent possède de nombreuses méthodes dont les principales sont :

| **Méthodes** | **Rôle** |
|---|---|
| void copy() | Copier le contenu du texte et le mettre dans le presse papier système |
| void cut() | Couper le contenu du texte et le mettre dans le presse papier système |
| Document getDocument() | Renvoyer l'objet de type Document qui encapsule le texte saisi |
| String getSelectectedText() | Renvoyer le texte sélectionné dans le composant |
| int getSelectionEnd() | Renvoyer la position de la fin de la sélection |
| int getSelectionStart() | Renvoyer la position du début de la sélection |
| String getText() | Renvoyer le texte saisie |
| String getText(int, int) | Renvoyer une portion du texte incluse entre 2 positions |
| bool isEditable() | Renvoyer un booléen qui précise si le texte est éditable ou non |
| void paste() | Coller le contenu du presse papier système dans le composant |
| void select(int,int) | Sélectioner une portion du texte dont incluse entre 2 positions |
| void setCaretPosition(int) | Déplacer la curseur à la position dans le texte précisé en paramètre |
| void setEditable(boolean) | Permet de préciser si les données du composant sont éditables ou non |
| void setSelectionEnd(int) | Modifier la position de la fin de la sélection |
| void setSelectionStart(int) | Modifier la position du début de la sélection |
| void setText(String) | Modifier le contenu du texte |

La classe `javax.Swing.JTextField` est un composant qui permet la saisie d'une seule ligne de texte simple. Son modèle utilise un objet de type `PlainDocument`. La propriété horizontalAligment permet de préciser l'alignement du texte dans le composant en utilisant les valeurs `JTextField.LEFT`, `JTextField.CENTER` ou `JTextField.RIGHT`.

La classe `JPasswordField` permet la saisie d'un texte dont tous les caractères saisis seront affichés sous la forme d'un caractère particulier ('*' par défaut). Cette classe hérite de la classe `JTextField`. La méthode `setEchoChar(char)` permet de préciser le caractère qui sera utilisé pour afficher la saisie d'un caractère. Il ne faut pas utiliser la méthode `getText()` qui est déclarée *deprecated* mais la méthode `getPassword()` pour obtenir la valeur du texte saisi.

```
package swing;
import java.awt.Dimension;
import java.awt.event.*;
import javax.swing.*;
public class TestText1 implements ActionListener {
  JPasswordField passwordField1 = null;
  public static void main(String argv[]) {
    TestText1 jpf2 = new TestText1();
    jpf2.init();
  }
  public void init() {
    JFrame f = new JFrame("PassControl");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();
    passwordField1 = new JPasswordField("");
    passwordField1.setPreferredSize(new Dimension(100, 20));
    pannel.add(passwordField1);
    JButton bouton1 = new JButton("Display on console");
    bouton1.addActionListener(this);
    pannel.add(bouton1);
    f.getContentPane().add(pannel);
```

```
    f.setVisible(true);
  }
  public void actionPerformed(ActionEvent e) {
    System.out.println("texte saisie = " + String.copyValueOf(passwordField1.getPassword()));
  }
}
```

Les méthodes `copy()` et `cut()` sont redéfinies pour empêcher l'envoi du contenu dans le composant et émettre simplement un *beep*.

Le JDK 1.4 propose la classe `JFormattedTextField` pour faciliter la création d'un composant de saisie personnalisé. Cette classe hérite de la classe `JTextField`.

Le composant `JEditorPane` permet de saise de texte riche multi-lignes. Ce type de texte peut contenir des informations de mise en pages et de formatage. En standard, Swing propose le support des formats RTF et HTML.

```
package swing;
import java.net.URL;
import javax.swing.*;
import javax.swing.event.*;
public class TestText2 {
  public static void main(String[] args) {
    final JEditorPane editeur;
    JPanel panneau = new JPanel();
    try {
      editeur = new JEditorPane(new URL("http://www.editions-ellipses.fr/"));
      editeur.setEditable(false);
      editeur.addHyperlinkListener(new HyperlinkListener() {
        public void hyperlinkUpdate(HyperlinkEvent e) {
          if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
            URL url = e.getURL();
            if (url == null)
              return;
            try {
              editeur.setPage(e.getURL());
            } catch (Exception ex) {
              ex.printStackTrace();
            }
          }
        }
      });
      panneau.add(editeur);
    } catch (Exception e1) {
      e1.printStackTrace();
    }
    JFrame f = new JFrame("ma fenetre");
    f.setSize(1000, 700);
    f.getContentPane().add(panneau);
    f.setVisible(true);
  }
}
```

La classe `JTextArea` est un composant qui permet la saisie de texte simple en mode multi-lignes. Le modèle utilisé par ce composant est le `PlainDocument` : il ne peut donc contenir que du texte brut sans éléments multiples de formatage. `JTexteArea` propose plusieurs méthodes pour ajouter du texte dans son modèle :

- soit fournir le texte en pramètre du constructeur utilisé
- soit utiliser la méthode setText() qui permet d'initialiser le texte du composant

- soit utiliser la méthode append() qui permet d'ajouter du texte à la fin de celui contenu dans le texte du composant
- soit utiliser la méthode insert() permet d'insérer un texte à une position données en caractères dans dans le texte du composant

La méthode `replaceRange()` permet de remplacer une partie du texte désignée par la position du caractère de début et la position de son caractère de fin par le texte fourni en paramètre.

La propriété `rows` permet de définir le nombre de ligne affichée par le composant : cette propriété peut donc être modifié lors d'un redimensionnement du composant. La propriété `lineCount` en lecture seule permet de savoir le nombre de lignes dont le texte est composé. Il ne faut pas confondre ces deux propriétés.

La classe `JTextPane` est similaire à `JTextArea` avec plus de possibilités visuelles.

Listing 7.2 – Code Java d'un éditeur

```
package swing;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;
import java.io.*;

public class TestText3 {

    // read the file into the pane
    // http://users.cs.cf.ac.uk/O.F.Rana/jdc/swing-nov7-01.txt
    static void readin(String fn, JTextComponent pane) {
        try {
            FileReader fr = new FileReader(fn);
            pane.read(fr, null);
            fr.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }

    public static void main(String args[]) {
        final JFrame frame = new JFrame("Editeur de texte");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        // set up the text pane, either a JTextArea or JTextPane

        final JTextComponent textpane = new JTextArea();
        // final JTextComponent textpane = new JTextPane();

        // set up a scroll pane for the text pane

        final JScrollPane pane = new JScrollPane(textpane);
        pane.setPreferredSize(new Dimension(600, 600));

        // set up the file chooser
```

```
        String cwd = System.getProperty("user.dir");
        final JFileChooser jfc = new JFileChooser(cwd);
        final JLabel elapsed = new JLabel("Elapsed time: ");

        JButton filebutton = new JButton("Choose File");
        filebutton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (jfc.showOpenDialog(frame) != JFileChooser.APPROVE_OPTION)
                    return;
                File f = jfc.getSelectedFile();

                // record the current time and read the file

                final long s_time = System.currentTimeMillis();
                frame.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
                readin(f.toString(), textpane);

                // wait for read to complete and update time

                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        frame.setCursor(Cursor
                                .getPredefinedCursor(Cursor.DEFAULT_CURSOR));
                        long t = System.currentTimeMillis() - s_time;
                        elapsed.setText("Elapsed time: " + t);
                    }
                });
            }
        });

        JPanel buttonpanel = new JPanel();
        buttonpanel.add(filebutton);
        buttonpanel.add(elapsed);

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.add("North", buttonpanel);
        panel.add("Center", pane);

        frame.getContentPane().add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

### 3.4 Les onglets

La classe `javax.swing.JTabbedPane` encapsule un ensemble d'onglets. Chaque onglet est constitué d'un titre, d'un composant et éventuellement d'une image. Pour utiliser ce composant, il faut :

- instancier un objet de type `JTabbedPane`
- créer le composant de chaque onglet
- ajouter chaque onglet à l'objet `JTabbedPane` en utilisant la méthode `addTab()`

```
  public static void main(String[] args) {
    JFrame f = new JFrame("Test de JTabbedPane");
    f.setSize(320, 150);
```

```
    JPanel pannel = new JPanel();
    JTabbedPane onglets = new JTabbedPane(SwingConstants.TOP);
    JPanel onglet1 = new JPanel();
    JLabel titreOnglet1 = new JLabel("Onglet 1");
    onglet1.add(titreOnglet1);
    onglet1.setPreferredSize(new Dimension(300, 80));
    onglets.addTab("onglet1", onglet1);
    JPanel onglet2 = new JPanel();
    JLabel titreOnglet2 = new JLabel("Onglet 2");
    onglet2.add(titreOnglet2);
    onglets.addTab("onglet2", onglet2);
    onglets.setOpaque(true);
    pannel.add(onglets);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
```

A partir du JDK 1.4, il est possible d'ajouter un raccourci clavier sur chacun des onglets en utilisant la méthode `setMnemonicAt()`. Cette méthode attend deux paramètres : l'index de l'onglet concerné (le premier commence à 0) et la touche du clavier associée sous la forme d'une constance `KeyEvent.VK_xxx`. Pour utiliser ce raccourci, il suffit d'utiliser la touche désignée en paramètre de la méthode avec la touche Alt.

La classe JTabbedPane possède plusieurs méthodes qui permettent de définir le contenu des l'onglets : `addTab(String, Component)`, `addTab(String, Component) Permet d'ajouter un nouvel onglet dont le titre et le composant sont fournis en paramètres. Cette méthode possède plusieurs surcharges qui permettent de préciser une icône et une bulle d'aide insertTab(String, Icon, Component, String, index)`, `remove(int)`, `setTabPlacement` (permet de préciser le positionnement des onglets dans le composant `JTabbedPane`. Les valeurs possibles sont les constantes TOP, BOTTOM, LEFT et RIGHT définies dans la classe `JTabbedPane`.). La méthode `getSelectedIndex()` permet d'obtenir l'index de l'onglet courant. La méthode `setSelectedIndex()` permet de définir l'onglet courant.

### 3.5 Le composant JTree

Le composant `JTree` permet de présenter des données sous une forme hiérarchique arborescente. Aux premiers abords, le composant `JTree` peut sembler compliqué à mettre en oeuvre mais la compréhension de son mode de fonctionnement peut grandement failiter son utilisation. Il utilise le modèle MVC en proposant une séparation des données (*data models*) et du rendu de ces données (*cell renderers*). Dans l'arbre, les éléments qui ne possèdent pas d'élément fils sont des feuilles (*leaf*). Chaque élément est associé à un objet (*user object*) qui va permettre de déterminé le libellé affiché dans l'arbre en utilisant la méthode `toString()`.

La classe `JTree` possède 7 constructeurs dont tous ceux qui attendent au moins un paramètre acceptent une collection pour initialiser tout ou partie du modèle de données de l'arbre :

```
public JTree();
public JTree(Hashtable value);
public JTree(Vector value);
public JTree(Object[] value);
public JTree(TreeModel model);
public JTree(TreeNode rootNode);
public JTree(TreeNode rootNode, boolean askAllowsChildren);
```

Lorsqu'une instance de `JTree` est créée avec le constructeur par défaut, l'arbre obtenu contient des données par défaut.

```
package swing;
import javax.swing.JFrame;
import javax.swing.JTree;
public class TestJTree extends JFrame {
  private javax.swing.JPanel jContentPane = null;
  private JTree jTree = null;
  private JTree getJTree() {
    if (jTree == null) {
      jTree = new JTree();
    }
    return jTree;
  }
  public static void main(String[] args) {
    TestJTree testJTree = new TestJTree();
    testJTree.setVisible(true);
  }
  public TestJTree() {
    super();
     initialize ();
  }
  private void initialize () {
    this.setSize(300, 200);
    this.setContentPane(getJContentPane());
    this.setTitle("TestJTree");
  }
  private javax.swing.JPanel getJContentPane() {
    if (jContentPane == null) {
      jContentPane = new javax.swing.JPanel();
      jContentPane.setLayout(new java.awt.BorderLayout());
      jContentPane.add(getJTree(), java.awt.BorderLayout.CENTER);
    }
    return jContentPane;
  }
}
```

L'utilisation de l'une ou l'autre des collections n'est pas équivalente. Par exemple, l'utilisation d'une (hash)table ne garantit pas l'ordre des nœuds puisque cette collection ne gère pas par définition un ordre précis. Généralement, la construction d'un arbre utilise un des constructeurs qui attend en paramètre un objet de type `TreeModel` ou `TreeNode` car ces deux objets permettent d'avoir un contrôle sur l'ensemble des données de l'arbre. Leur utilisation sera détaillée dans la section consacrée à la gestion des données de l'arbre. En fonction du nombre d'éléments et de l'état étendue ou non d'un ou plusieurs éléments, la taille de l'arbre peut varier : il est donc nécesssaire d'inclure le composant `JTree` dans un composant `JScrollPane`.

Pour plus de détails sur cette classe complexe, consulter
http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap014.htm

### 3.6 Bilan

Nous avons donné un aperçu des composants Swing. Le framework est complexe et nécessite une étude approfondie.

## 4 Exemples

Pour comprendre et maîtriser l'écriture d'IHM avec JFC, rien ne vaut la lecture de nombreux exemples. Voir l'archive jointe [SM03, Cha03].

## 5 MVC et Web

L'architecture MVC (Modèle - Vue - Contrôleur) a révolutionné le développement d'applications web. Elle permet en effet de modulariser ce développement et d'intégrer des technologies aussi diverses et variées que JSP (JavaServer Pages), Java Servlets, XML, XSLT, EJB, etc. Néanmoins, le coût associé à ce type de réalisation reste généralement élevé. Jakarta Struts [Cav04] constitue la solution idéale pour établir à un coût raisonnable un cadre de développement MVC robuste, évolutif et performant. Il autorise en outre la gestion des erreurs, la validation des saisies et l'internationalisation. Ce petit ouvrage expose les étapes clés de l'installation et de la configuration du cadre de développement Jakarta Struts, et couvre en détail les actions intégrées et les bibliothèques de balises.

## 6 IHM Builders

Un certain nombre de générateurs d'interface sont disponibles, en particulier dans les environnements de développement comme AnyJ (Net Computing), Netbeans (Sun), JBuilder (Borland), Visual Age (IBM), Simplicity for Java (Data Representations), Sun ONE Studio (Sun microsystems), Visaj (IST). Ils ne sont cependant utilisable que pour des applications se contentant exclusivement de widgets standards. Si vous devez sortir un peu de Swing (zone de dessin, zone d'édition, etc), vous perdrez moins de temps à coder en Swing à la main l'ensemble de votre programme plutôt que devoir vous y retrouver dans le code généré par l'Interface Builder.
(sources `http://www.emn.fr/x-info/cdumas/ihm/dev.html`)
`http://www.eteks.com/liens.html`

Cet aspect reste à creuser.

## 7 SWT

Consulter [Dau04] ou `http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap015.htm`

# Chapitre 8

# Le développement d'applications

Dans ce chapitre nous abordons le développement de logiciel à objets avec Java. Nous l'inscrivons dans le contexte du développement avec UML, largement exposé dans le chapitre 3 de [AV01b].

## 1 Introduction

Le développement avec UML est une utilisation rationnelle de la notation pour la documentation des produits du développement. Toute démarche (cycle de développement) peut s'inscrire dans ce cadre. Néanmoins nous préconisons une inscription dans les principes retenus par l'OMG et mis en pratique dans le processus unifié de Rational : à savoir une démarche itérative et incrémentale, basée sur l'architecture et les cas d'utilisation. Nous y ajoutons la séparation logique applicative et logique technique (au cœur de la méthode 2TUP de Valtech) et les éléments d'environnement de projet (gestion de projet, gestion de configuration, gestion des risques et des ressources [RJB99, Roy98]). Un exposé détaillé est illustré par un exemple dans le chapitre 3 de [AV01b].

## 2 L'architecture logicielle

Le terme **architecture** est comme le terme modèle (voir chapitre 1 du tome 1) un terme générique, employé dans des contextes différents. Dans la littérature sur UML, nous avons parfois du mal à discerner le sens de ce terme. L'architecture de l'application n'est pas la structure des spécifications. Nous tentons dans cette section de clarifier cette notion pour illustrer son utilisation dans le contexte d'UML.

L'**architecture d'un système** est relative à la structure ou l'organisation de ce système. Elle décrit les éléments du système et les liens entre ces éléments. Souvent, par architecture, on entend modèle d'architecture ou architecture de référence, c'est-à-dire une abstraction de la structure, qui s'applique à plusieurs systèmes concrets. Ainsi, une **architecture du logiciel** (ou applicative [GG96]) définit une organisation des éléments du logiciel. L'architecture d'un système informatique décrit les composants du logiciel et du matériel (architectures logique et physique [KMPRS98]). Par exemple, "l'architecture client/serveur est un modèle d'architecture applicative où les programmes sont répartis entre les processus clients et serveurs communiquant par des requêtes avec réponses." [GG96]. Une telle architecture comprendra par exemple les applications clients (sous Windows, OS/2 ou Unix), la base de données et son serveur, et le *middleware*[1]. Cette architecture logicielle sous-entend un réseau "logique", mais une implantation avec trois programmes sur une même machine peut très bien respecter

1. Ensemble des services logiciels construits au dessus d'un protocole de transport afin de permettre l'échange des requêtes et des réponses associées entre client et serveur de manière transparente [GG96].

l'architecture client/serveur. Le réseau est lui-même défini par une architecture en couches (les 7 couches OSI). Dans une architecture distribuée, le *middleware* assure la coordination des objets serveurs et des objets clients. Une bonne architecture est garante de la pérennité et de l'évolutivité d'un système informatique.

Dans un système à objets, la structure de base est la coopération entre les objets. L'architecture permet en autre d'organiser cette coopération. Plusieurs modèles d'architecture sont proposés par KETTANI [KMPRS98]. En particulier, retenons les architectures en couche qui séparent interface (IHM), application (regroupe parfois interface, contrôle et dispositifs physiques), objets du domaine (métier) et objets techniques (ou d'implantation). Afin de permettre à chacun de définir son architecture, un modèle général a été proposé pour structurer les spécifications en UML. Ainsi, l'approche en 4+1 vues de KRUCHTEN exprime cinq perspectives de l'architecture d'un système informatique [KMPRS98]. La figure 56 illustre les différentes perspectives [RJB98].

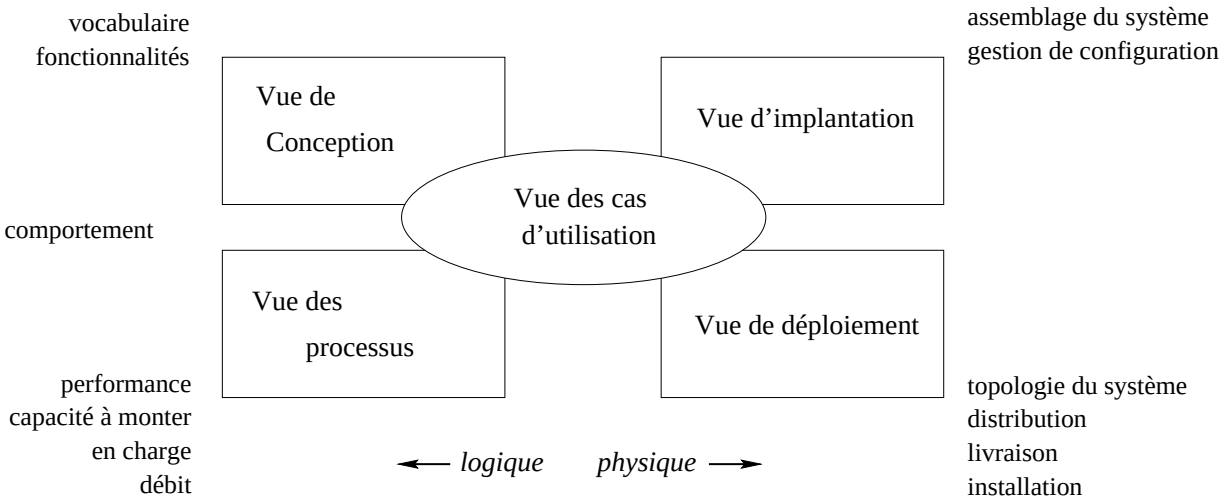



Figure 56 : *Notation UML : les vues*

Ces vues sont relativement indépendantes. Les cas d'utilisation sont centraux car ils servent tant à l'analyse, la conception qu'au test du logiciel. La vue de conception représente le domaine du problème (classes, interfaces et collaborations). La vue des processus est une projection sur la gestion de la concurrence. La vue d'implantation est l'image de la conception logicielle (composants, conception des objets). Enfin, la vue de déploiement représente la distribution de l'application (topologie matérielle).

L'architecture logicielle est fortement marquée par le type d'application, la technologie employée voire même les traditions de l'entreprise. Le modèle précédent permet d'appréhender la plupart des architectures logicielles : architecture en couche, architecture en peau d'oignon, architecture n-tiers, client-serveur, architecture distribuée.

Le tableau XIX, extrait de [Mul97], résume l'emploi des éléments de modélisation dans les diagrammes selon les vues. La coordination des éléments de modélisation et des diagrammes dans les vues donne une idée de la complexité de l'architecture proposée.

## 3 Le processus de développement

UML est normalise la notation standardisée mais pas le processus, même si des études sur ce sujet sont en cours. La normalisation du processus implique de rationaliser des activités qui varient d'une entreprise à l'autre. Actuellement, chacun adapte la notation à son besoin. Par exemple, certains suivent le processus habituel et général de Merise sous l'angle de la notation UML. Dans la suite, nous reprenons les phases habituelles du processus développement.

| | **Vue des cas d'utilisation** | **Vue logique** | **Vue des composants** | **Vue des processus** | **Vue de déploiement** |
|---|---|---|---|---|---|
| **Diagramme de cas d'utilisation** | acteurs<br>cas d'utilisation | | | | |
| **Diagramme d'objets** | objets<br>liens | objets<br>liens<br>classe | | | |
| **Diagramme de séquence** | acteurs<br>objets<br>messages | acteurs<br>objets<br>messages | | objets<br>messages | |
| **Diagramme de collaboration** | acteurs<br>objets<br>liens<br>messages | acteurs<br>objets<br>liens<br>messages | | objets<br>liens<br>messages | |
| **Diagramme de classes** | | classes<br>relations | | | |
| **Diagramme d'états-transitions** | états<br>transitions | états<br>transitions | | états<br>transitions | |
| **Diagramme d'activités** | activités<br>transitions | activités<br>transitions | | activités<br>transitions | |
| **Diagramme de composants** | | | composants | composants | composants |
| **Diagramme de déploiement** | | | | | nœuds<br>liens |

TABLEAU XIX– Croisement des diagrammes et des vues

1. l'**analyse des besoins**
2. l'**analyse**
3. la **conception**
4. la **réalisation** et le **test**
5. l'**installation**
6. la **maintenance**

A chaque transition d'étape, une validation est effectuée. Validation et vérification sont distinguées : la validation est la mise en adéquation avec le besoin initial, la vérification est la conformité du logiciel avec sa spécification (elle peut s'obtenir par preuve ou par tests).

Il y a actuellement convergence d'idée sur les grandes lignes d'un processus dans la communauté des développeurs :

- Le cycle processus est itératif pour les grands projets car les logiciels développés sont souvent très complexes et il est intéressant d'en étudier les grandes parties puis rapidement de valider cette étude par un premier prototype. Les phases itérées sont l'étude du besoin, l'analyse, la conception et le prototypage (réalisation allégée). Une validation du besoin est possible via le prototype : le client affine ainsi sa demande.
- Le processus doit être incrémental pour ne pas remettre en question les modèles déjà écrits : ceci implique une grande modularité des produits. La cohérence est une propriété implicite du modèle objet, mais le couplage, deuxième composante de la modularité, doit être surveillé.
- Le processus est basé sur les cas d'utilisation. Le cas d'utilisation est l'identification parcellaire d'une partie cohérente du fonctionnement du système modélisé. Ceci permet

une approche par morceaux d'un problème complexe. Le regroupement des morceaux donne une vue globale du système. Travailler sur des parties facilite la compréhension, la modélisation et la vérification de ces parties. Cela induit aussi assez naturellement un partage simplifié du travail, dans l'hypothèse où l'architecture globale du système aura été bien définie à un moment donné.
- L'architecture devient centrale dans la cohérence globale du projet. Plusieurs modèles existent : par couche, distribué, par activités (ex. MVC), etc.
- La réutilisation est un autre élément capital dans le développement et plus encore le développement à objets. Elle sous-entend l'intégration de parties logicielles existantes dans un projet :
  - réutilisation horizontale : délégation de services à d'autres objets existants (ex. bibliothèques d'objets spécialisés),
  - réutilisation verticale : définition de comportement spécifiques de certains objets à partir d'objets existants (héritage),
  - réutilisation complexe combinant les deux, par exemple par instanciation de micro-architectures ou *pattern*,
  - etc.

Par exemple, un processus comprend dans ses grandes lignes les éléments suivants :

- Modélisation du besoin avec les cas d'utilisation : définition des acteurs concernés (acteur primaire qui sont à l'origine du déclenchement des cas d'utilisation et acteurs secondaires), définition des cas d'utilisation (des sous-cas liés par `uses` ou `extends`), définition de scénarios illustrant ces cas d'utilisation (cas normaux et exceptions).
- Analyse avec les diagrammes d'objets, de classe, d'états/transitions, d'activités, de collaboration et des diagrammes de séquence. A partir des cas d'utilisation, des diagrammes de collaboration ou de séquence (scénarios enrichis) apportent plus de précision quant à la prise en charge des besoins par le système. De nombreux objets sont identifiés qui prennent en charge une partie de la réalisation des besoins. Certains préconisent dès ce niveau la distinction entre objets interfaces (faisant le lien entre le système et son environnement) et objets métiers. Cette distinction convient aux applications de type gestion (une fenêtre de menu prend en charge une communication) mais elle devient plus subtile pour la modélisation de processus réactifs (processus industriel par exemple). Cette partie est relativement complexe dans la mesure où l'intérieur du système commence à être abordé. Des dessins d'interface ou des spécification de mécanismes physiques documentent les modèles UML produits. Des modèles d'objets et de classes/relations partiels sont établis. Les classifications d'objets sont étudiées. Un modèle statique global regroupe et synthétise les différents modèles partiels de classes.
- Conception système (ou préliminaire). Une architecture du système est définie, qui précise l'organisation logique des éléments logiciels et matériels. On peut utiliser des diagrammes de classes, de composants et de déploiement.
- Conception détaillée qui définit précisément les composants logiciels.
- Réalisation ou prototypage.
- Tests, validation et itération.

KETTANI [KMPRS98] définit l'analyse et la conception comme des activités et non des phases. Ces activités recouvrent trois aspects :

- Architecture. Une première vue logique est donnée, qui reprend les objets métiers. Elle est ensuite raffinée pour faire apparaître les sous-systèmes de bas niveau et les paquetages de service.
- Cas d'utilisation. Pour chaque cas, les objets sont mis en évidence.
- Objet. Pour chaque classe, les responsabilités, relations et attributs sont mis en évidence.

### 3.1 Cycle de développement

Le cycle présente l'organisation temporelle des activités du processus. Le développement avec UML et Java n'accorde avec différentes démarches

- Méthodes classiques
  - de l'analyse aux tests d'intégration
  - cycle linéaire, en cascade, en V
  - restriction ou pas des diagrammes à chaque niveau
  - exemple simple : [AV01b, AV03]
- Processus unifié
  - élabore le modèle final par enrichissement progressifs du modèle d'analyse,
  - basée sur une notation unique (UML),
  - support d'un processus itératif et incrémental, centré sur l'architecture et les cas d'utilisation,
  - concepteurs d'UML
- MDA - *Model Driven Approach*
  - élabore le modèle final par transformations successives de modèles
  - les modèles indépendants des plates-formes (PIM) sont transformés des modèles dépendants des plates-formes (PSM)
  - proposé par l'OMG
- Méthodes "agiles", représentées par XP (*eXtreme Programming*)
  - donne la part belle aux programmeurs
  - basée sur des principes de bonne pratique de la programmation à objets
  - adapté aux petite applications

D'autres sociétés proposent d'autres méthodes : OPEN, Objecteering/Softeam, Rhapsody/I-Logix, Catalysis/ICON Computing, Together/Borland, etc.

### 3.2 Processus unifié

Le Processus unifié de Rational (IBM) ou RUP est

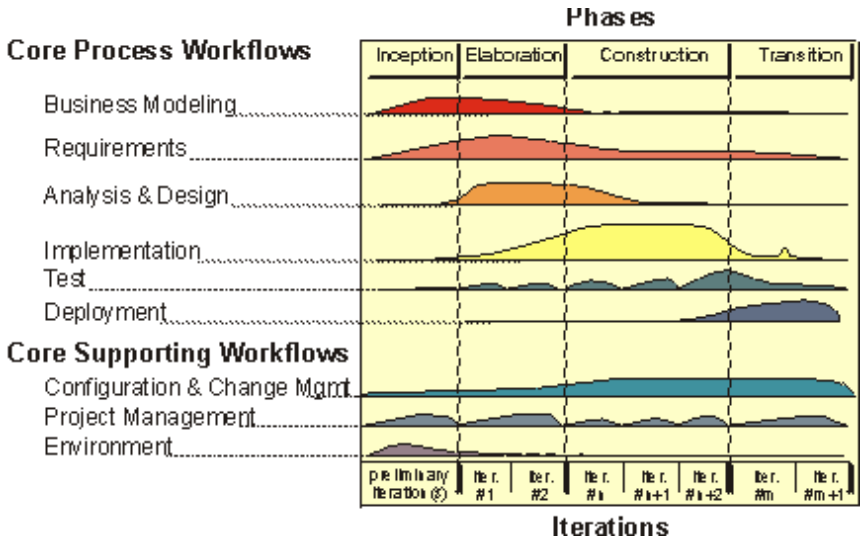
- Itératif : plusieurs exécutions consécutives de la démarche linéaire.
- Incrémental : priorités sont définies dans le développement.
- Architecture : stabilisation des principes de l'applicatif et des techniques.
- Cas d'utilisation : le besoin guide le développement et l'intégration (test).

Il intègre des préoccupations du développement et de la gestion de projet.

Le RUP définit deux axes :

- Activités
  - développement (analyse des besoins ⇒ test)
    ⇒ développement du logiciel ([AV01a], p. 13)
    un modèle produit par activité (cf les domaines)
  - support
    - Gestion de configuration & versions
    - Gestion de projet (organisation, risques, planification)
    - Environnement (support et méthode)
- Itérations
  - Grain fin : itération
  - Gros grain : phases

avec une coordination des deux axes




Noter la variation de l'effort de développement et l'entrelacement des activités de développement et de support dans chaque itération.

**Itérations et phases**

- chaque itération produit une version du système : un jalon mineur
- les phases définissent les grandes étapes du développement : les jalons majeurs, qui contrôlent ainsi le nombre d'itérations
  - préparation (*inception*) :
    Etablir la faisabilité et le contexte du projet
    Résultat : *Lifecycle objectives*
  - élaboration :
    Etablir l'architecture et planification contrôlée du projet
    Résultat : *Lifecycle architecture*
  - construction :
    Construire un système testable
    Résultat : *Initial Operational Capability*
  - transition :
    Mettre le système en production pour l'utilisateur
    Résultat : *Product release*
- Tests et qualité sont pris en compte par le processus

## 4 La conception en Java

Dans cette section nous insistons sur la traduction dans le langage de programmation à objets, Java en l'occurrence.

Parmi les activités détaillées dans [AV01b], la conception détaillée (ou concrète) est la mise en situation de la conception dans l'environnement de développement cible. La conception détaillée précise le contenu des composants (classes, autres composants, bibliothèques, programmes). La description est enrichie à partir des composants techniques issus de l'environnement de développement cible pour réutiliser ses classes. Il ne s'agit pas encore de coder le programme mais de faire l'adéquation entre les concepts de la conception abstraite et ceux de la conception détaillée.

En supposant, un seul langage de programmation à objets[2], voici les grandes lignes d'une

2. Avec CORBA, des objets issus de différents langages peuvent coopérer.

telle transition :

- `Vocabulaire` : on adapte le vocabulaire à celui du langage cible. Par exemple, un attribut (resp. une opération) d'instance sera appelée variable (resp. méthode) d'instance en Smalltalk ou donnée (resp. fonction) membre en C++.
- `Typage` : on adapte les règles de typage et les conventions associées à celles du langage cible. Par exemple, en Smalltalk, le typage est dynamique alors qu'en C++ il est statique. Le polymorphisme des méthodes est implicite en Smalltalk alors que celui des fonctions C++ est explicite par le qualificatif `virtual`.
- `Association` : ce point est détaillé dans la suite.
- `Héritage multiple` :
  - Si le langage cible n'autorise pas l'héritage multiple (c'est le cas de Smalltalk), il faut mettre en œuvre une politique de traduction adpatée : on choisit un chemin d'héritage principal et on duplique les caractéristiques issues des chemins secondaires.
  - Si le langage cible autorise l'héritage multiple, il Il faut adapter la stratégie de gestion des conflits d'héritage à celle du langage cible (renommage et redéfinition en Eiffel, tri topologique en CLOS).
- `Contrôle des objets` : La plupart des langages à objets manipulent des objets séquentiels passifs. Lorsque la conception définit des objets actifs, il faut mettre en place un environnement d'exécution distribué simulé (processus en Smalltalk) ou intégré (*threads* Java). Ce point, qui inclue les variantes d'envois de messages et d'événements, constitue une étape essentielle de la conception pour les systèmes temps-réels.
- `Méta-objets` : certains langages autorisent des protocoles pour méta-objets (Smalltalk), pour les autres, il faut simuler en fonction des possiblités du langage (routines en Eiffel).
- `Assertions` : si le langage autorise les assertions (Eiffel par exemple), la traduction de contraintes OCL est simplifiée, sinon il faut programmer explicitement les contraintes.
- `Réutilisation` : une partie de la conception est réécrite en fonction des éléments qui existent dans l'environnement cible. Par exemple on fera correspondre les collections OCL aux templates de C++. Les patterns peuvent être utilisés à ce niveau.

En programmation à objets, une **association** s'implante habituellement avec des pointeurs. On peut s'inspirer des règles de transformation du schéma E-A-P dans le modèle relationnel [AV01a]. Examinons les alternatives de modélisation.

1. L'association ne possède pas de propriétés. Les liens sont représentés par des attributs de navigation, un par classes de la relation. Par exemple, *film.classe* donne le genre et *genre.classé_en* donne les films. Ces attributs prennent en général les noms des rôles. Il y a quelques cas particuliers :
   (a) Navigation unidirectionnelle : un seul attribut de lien, dans la classe origine de la navigation.
   (b) Cardinalité maximale de 1 dans un sens : on peut choisir une navigation unidirectionnelle.
2. L'association possède des propriétés et
   (a) une cardinalité égale à 1. Les propriétés migrent dans la classe correspondant à la cardinalité. Après migration, on se retrouve dans le cas 1.
   (b) aucune cardinalité maximale de 1. L'association donne lieu à une nouvelle classe. Elle est reliée aux classes sous-jacentes par des associations binaires sans propriétés (cas 1). Implicitement, c'était le cas de la classe *Statistique*.
   (c) une cardinalité dans 0..1. Les deux cas précédents sont applicables. Si les propriétés migrent dans la classe, il se pose le problème des valeurs partiellement définies. Par exemple, la date de l'association *disponible* migre dans la classe cassette *Cassette* mais sa valeur n'a pas de sens si la cassette est empruntée.

Ces éléments seront mis en œuvre dans des exercices spécifiques avec l'IDE Eclipse et Java 5.0.

Notons pour terminer que l'outil AGL utilisé influence fortement la conception de l'application. Un AGL complet inclue en effet des éditeurs de documents, des générateurs d'interface, des éditeurs et générateurs de code, des outils de vérification, de test et de prototypage.

# Chapitre 9

# Conclusion

Nous avons développé dans ce cours les éléments essentiel de la programmation à objets avec le langage Java dans l'environnement `Eclipse`.

Java est un environnement complet de programmation à objets. Le modèle objet est riche et très représentatif de la programmation avec des objets et des classes. Le langage est typé statiquement, ce qui facilite la réutilisation de code et sa documentation. La méthode de programmation doit être rigoureuse pour que le code produit soit de qualité (réutilisable, adaptable, fiable...). A l'inverse, le prototypage et la mise au point sont souvent plus laborieux. Java est un bon compromis entre Smalltalk et C++.

Le langage est basé sur un nombre conséquent de concepts (plus qu'en Smalltalk et moins qu'en C++). La connaissance du langage est donc à la fois la connaissance des concepts de base, des concepts objets, des mécanismes de typage et aussi celle des classes existantes de l'API. En ce sens, comme pour la plupart des langages à objets, la connaissance de l'environnement de développement et des bibliothèques de classes est fondamentale pour le développement d'applications.

L'environnement de développement `Eclipse` pour Java répond à cette attente. L'environnement est complet et personnalisable par plugins.

Toutes les classes sont accessibles par les outils de développement. Cette "aide en ligne" favorise l'apprentissage du langage et la réutilisation. Les classes sont adaptables aux besoins propres des programmeurs.

`Eclipse` propose aussi des outils pour le développement avec (CVS, Ant, ANTLR, JUnit) pour le développement Web (JBoss, Tomcat), et le développement d'applications. Nous avons illustré dans ce cours la création de programmes divers et d'applications graphiques. Des outils pour la gestion de bases de données (JDBC) sont aussi fournis dans l'environnement. Enfin, on trouvera dans l'environnement les classes nécessaires à l'implantation de prototypes de langages à objets ou de systèmes d'exploitation.

Ce cours contient toujours des erreurs. L'auteur remercie d'avance les lecteurs pour toute suggestion visant à améliorer ce cours.

# Bibliographie


[ACR94] Pascal André, Dan Chiorean, and Jean-Claude Royer. The formal class model. In *Joint Modular Languages Conference*, pages 59–78, Ulm, Germany, 28-30 september 1994. GI, SIG and BCS.

[AMC01] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns : Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[And01] Pascal André. Introduction au génie logiciel à objets et à uml. Polycopié de cours, INP-HB, Janvier 2001. (170 pages).

[And05] Pascal André. Introduction à la programmation à objets avec smalltalk-80. Support de formation, ATMEL, Octobre 2005. (117 pages).

[AR98] Pascal André and Jean-Claude Royer. Modélisation par objets. Rapport de Recherche RR-179, IRIN, Octobre 1998. (54 pages).

[AV01a] Pascal André and Alain Vailly. *Conception de systèmes d'information ; Panorama des méthodes et des techniques*, volume 1 of *Collection Technosup*. Editions Ellipses, 2001. ISBN 2-7298-0479-X.

[AV01b] Pascal André and Alain Vailly. *Spécification des logiciels ; Deux exemples de pratiques récentes : Z et UML*, volume 2 of *Collection Technosup*. Editions Ellipses, 2001. ISBN 2-7298-0774-8.

[AV03] Pascal André and Alain Vailly. *Exercices corrigés en UML ; Passeport pour une maîtrise de la notation.*, volume 5 of *Collection Technosup*. Editions Ellipses, 2003. ISBN 2-7298-1725-5.

[BB05] Valérie Berthié and Jean-Baptiste Briaud. *Swing, la synthèse*. Dunod, 1 edition, 2005. ISBN 2-10-049219-5.

[BD01] Xavier Briffault and Stéphane Ducasse. *Squeak Programmation*. Editions Eyrolles, 2001. ISBN 2-212-11023-5.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R, June 1988.

[BM02] Darren Broemmer and Freddie Mac. *J2EE Best Practices : Java Design Patterns, Automation, and Performance*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[BS96] Xavier Briffault and Gérard Sabah. *Smalltalk, Programmation orientée objet et développement d'applications*. Editions Eyrolles, 1996. ISBN 2-212-08914-7.

[Car88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76 :138–164, 1988.

[Cav04] Chuck Cavaness. *Programming Jakarta Struts*. O'Reilly & Associates, Inc., second edition, 2004.

[CFS03] Christophe Calandreau, Alain Fauré, and Nader Soukouti. *EJB 2.0 - Mise en oeuvre*. Dunod, 2003. ISBN 2-10-004729-9.

[Cha96] Jacquelin Charbonnel. *Langage C++ : la proposition de standard ANSI/ISO expliquée*. Masson, 1996. ISBN 2225852197.

[Cha03] Irène Charon. *Le langage Java 2 - Concepts et pratique*. Hermès, 2 edition, 2003. ISBN 2-7462-0629-3.

[Che05] Robert Chevallier. *Java 5 - Informatique - Synthèse de cours & exercices corrigés*. Pearson Education, 1 edition, 2005. ISBN 2-7440-7096-3.

[Cla03] Gilles et al. Clavel. *Java, la synthèse - Concepts, architectures, frameworks*. Dunod, 4 edition, 2003. ISBN 2-10-007102-5.

[Clo03] Pierre-Yves Cloux. *RUP, XP, Architectures et outils*. Dunod, 2003. ISBN 2-100-06430-4.

[Coi87] Pierre Cointe. Metaclasses Are First Classes : the ObjVlisp Model. In *ACM OOPSLA'87 proceedings*, pages 156–167. ACM, October 1987. October 4-8.

[Dau04] Berthold Daum. *Eclipse - Développement d'applications Java*. Dunod, 1 edition, 2004. ISBN 2-10-048733-7.

[DCM03] Alur Deepak, John Crupi, and Dan Malks. *J2EE et les Design Patterns*. CampusPress, 2003. ISBN 2-7440-1523-7.

[Dja05] Karim Djaafar. *Eclipse et JBoss - Développement d'applications J2EE professionnelles, de la conception au déploiement*. Eyrolles, 1 edition, 2005. ISBN 2-212-11406-0.

[Eck02] Bruce Eckel. *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002. ISBN 0131002872, version française http ://penserenjava.free.fr ou http ://www.planetpdf.com/codecuts/pdfs/eckel/TIJ2.zip.

[Fla97a] David Flanagan. *Java Examples in a Nutshell*. O'REILLY, 1st edition edition, 1997. ISBN 1-56592-371-5, A Companion Volume to Java in a Nutshell.

[Fla97b] David Flanagan. *Java in a Nutshell, A Desktop Quick Reference for Java Programmers*. O'REILLY, 2nd edition edition, 1997. ISBN 1-56592-262-X.

[GG96] George Gardarin and Olivier Gardarin. *Le Client-Serveur*. Eyrolles, 1996. ISBN 2-212-08876-0.

[GR05] Vincent Granet and Jean-Pierre Regourd. *Aide-mémoire de Java*. Dunod, 1 edition, 2005. ISBN 2-10-049145-8.

[Hol04] Steve Holzner. *Eclipse - Développement d'applications Java*. O'Reilly, 1 edition, 2004. ISBN 2-84177-264-0.

[How95] Timothy G. Howard. *The Smalltalk Developer's Guide to VisualWorks*. Cambridge University Press, New York, NY, USA, 1995. ISBN 1-884842-11-9.

[KMPRS98] Nasser Kettani, Dominique Mignet, Pascal Paré, and Camille Rosenthal-Sabroux. *De Merise à UML*. Eyrolles, 1998. ISBN 2-212-08997-X.

[Lin03] Johannes Link. *Tests unitaires en Java - Les tests au coeur du développement*. Dunod, 1 edition, 2003. ISBN 22-10-008152-7.

[Mey89] Bertrand Meyer. The New Culture of Software Development : Reflections on the Practice of OOD. In Jean Bezivin and Bertrand Meyer, editors, *Proceedings of TOOLS'89*, Paris, November 1989.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. International Series in Computer Science. Prentice Hall, 2 edition, 1997. ISBN 0-13-629155-4.

[Mic98] Alain Michard. *XML, Langage et applications*. Eyrolles, 1998. ISBN 2-212-09052-8.

[MNC+90] Gérald Masini, Amedeo Napoli, D. Colnet, D. Léonard, and Karl Tombre. *Les langages à objets*. Collection IIA. InterEditions, 1990.

[Mul97] Pierre-Alain Muller. *Modélisation objet avec UML*. Eyrolles, 1997. ISBN 2-212-08966-X.

[Ner90] Jean-Marc Nerson. Case Studies in Object-Oriented Analysis. In Jean Bezivin and Bertrand Meyer, editors, *TOOLS'90*, June 1990. Tutorial.

[NP96] Patrick Niemeyer and Joshua Peck. *Java par la pratique*. O'REILLY, edition française, itp edition, 1996. ISBN 2-84177-022-2.

[Puy04] Emmanuel Puybaret. *Java 1.4 et 5.0*. Eyrolles, 2 edition, 2004. ISBN 2-212-11478-8.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language User Guide*. Object-Oriented Series. Addison-Wesley, 1998. ISBN 0-201-57168-4.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Software Development Process*. Object-Oriented Series. Addison-Wesley, 1999. ISBN 0-201-57169-2.

[Roy94] Jean-Claude Royer. La programmaton à objets. Polycopié de cours, Université de Nantes, 24 septembre 1994. (121 pages).

[Roy98] Walker Royce. *Software Project Management, A Unified Framework*. Object-Oriented Series. Addison-Wesley, 1998. ISBN 0-201-30958-0.

[Ses05] Peter Sestoft. *Java Precisely*. The MIT Press, 2 edition, 2005. ISBN 0-262-69325-9.

[SM03] Pierre-Yves Saumon and Antoine Mirecourt. *Le guide du développeur Java 2 - Meilleures pratiques avec Ant, Junit et les design patterns*. Eyrolles, 1 edition, 2003. ISBN 2-212-11275-0.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, New York, NY, 1987.

[Weg90] Peter Wegner. Concepts and paradigms of oop. *OOP Messenger*, 1(1), August 1990. ACM.

# Annexe A

# IDE

## 1 Comparatifs d'IDE

Documents joints au cours (format adobe pdf).

- Trois IDE Java pour l'entreprise
  De plus en plus intégrés, les ateliers de développement J2EE prennent en charge toutes les étapes liées à la conception d'applications. Ils affichent une richesse fonctionnelle très étendue, au prix parfois d'une mise en oeuvre complexe.
  Guy Faure , Décision Micro, le 28/04/2004
  http://www.01net.com/article/240229_a.html
- Outils J2EE : Rad or not Rad ?
  Plusieurs développeurs ont testé pour DreamSoft les outils Sun Java Studio Creator, IBM Websphere Studio Application Developper, JBuilder 2005 et BEA Workshop. Le résultat de leurs expériences. (09/11/2004)
  http://solutions.journaldunet.com/0411/041109_dreamsoft.shtml
- Choix d'un IDE Java
  Dans cette page ont été repertoriés un certain nombre de caractéristiques de quelques IDE java sur le marché, collectés à l'occasion d'une étude que j'ai faite pour proposer un produit.. 12-12-2002 - Copyright ĺ 1997-2002 Etienne COCQUEBERT
  http://www.chez.com/cocquebert/java.htm

## 2 Tutorial Eclipse 3.1

Documents joints au cours.

Un *Tutorial Eclipse 3.1* (format MS powerpoint) par Serge Baccou et daté du 26/07/2005, accessible à
http://www.baccoubonneville.com/downloads/Eclipse.ppt

Développons en Java avec Eclipse (archive au format HTML) par Jean-Michel Doudoux et daté du 13/07/2003, accessible à
http://jmdoudoux.developpez.com/java/eclipse/ et en version plus complète au format PDF à
http://perso.wanadoo.fr/jm.doudoux/java/

# Annexe B

# Exercices

Ce chapitre a pour but d'illustrer les différents points du cours par des exemples. Les exercices proposés le sont actuellement à titre exérimental.

## 1 Langage

Dans cette section l'objectif est d'appréhender les structures de base de Java.

**Exercice 1.1**
*Ecrire une fonction qui calcule la suite de Fibonacci sur les entiers. Tester cette fonction.*

**Exercice 1.2**
*Définir des ensembles ayant un nombre pair d'objets. Définir des ensembles de couples d'objets.*

**Exercice 1.3**
*Définir des pixels et des points en trois dimensions.*

**Exercice 1.4**
*Ecrire une méthode* `opposé` *qui rend l'opposé d'un nombre. Par exemple, l'opposé de -12 est +12.*

**Exercice 1.5**
*Soit la méthode booléenne* `diviseurDe:`*, qui appliquée à un entier renvoie* `true` *si cet entier divise le paramètre de la méthode.*

*a) Ecrire le code de cette méthode.*

*b) Comment peut-on implanter cette méthode sans modifier la classe* `Integer` *?*

*c) Ecrire une méthode qui génère, sous forme de bloc la fonction* `diviseurDe:` *à partir d'un entier donné en paramètre.*
*C'est une fonction de première classe car une fonction rend en résultat une autre fonction.*

**Exercice 1.6**

*a) Ecrire une méthode qui calcule la somme de deux vecteurs et une méthode qui calcule la produit scalaire d'un vecteur par un entier.*

*b) Coder ces méthodes en Java, sans modifier les classes de base du système.*

## 2 Classes

Dans cette section l'objectif est d'appréhender les classes essentielles de Java.

**Exercice 2.1**
*Etudier la hiérarchie des sous-classes du paquetage* `java.lang` *: la modélisation des nombres et des éléments comparables (relation d'ordre).*

**Exercice 2.2**
*Etudier la hiérarchie des sous-classes de la classe* `Collection`*, les différentes représentations des groupes d'objets.*

**Exercice 2.3**
*Etudier la classe* `Objects`*, à savoir le comportement de base des objets.*

## 3 Programmation à objets

**Exercice 3.1**
*Programmer le jeu de Tetris avec* `VisualWorks`*.*

**Exercice 3.2**
*On souhaite écrire un programme qui simule une partie de tennis entre deux joueurs, selon les règles classiques du tennnis (inspiré librement de [Ner90]. Un match se joue en 3 sets gagnants (soit au maximum 3 ou 5 jeux).*

*Un joueur gagne un set s'il a au moins 6 jeux gagnés et deux jeux d'écart avec son adversaire. Si le score est de 6 jeux à 5 alors plusieurs cas sont possibles : soit le premier joueur gagne le jeu et il remporte le set 7-5, soit le second joueur gagne le jeu et un tie-break est joué, le gagnant du tie-break remporte le set par le score de 7 à 6.*

*Chaque jeu se joue en 4 points maximum pour le gagnant et deux points d'écart avec son adversaire. les points sont notés 0, 15, 30, 40, jeu. En cas d'égalité 40-40, les points sont notés : égalité, avantage au serveur ou avantage au receveur.*

*Le tie-break se joue en 7 points minimum pour le gagnant et deux points d'écart avec son adversaire (par exemple, 7-4, 9-7...).*

*C'est le même joueur qui sert pour tout un set, hormis le tie-break pour lequel, le serveur du set sert une fois puis les deux joueurs servent alternativement Le joueur qui sert en premier est choisi au hasard. On ne tiendra pas compte des fautes ou des deux balles de service. Pour chaque balle engagée, le point sera simplement gagnant ou perdant.*

*On ne tient pas compte pour l'instant des impondérables : blessures des joueurs, intempéries, abandons.*

*a) Ecrire un programme impératif qui modélise un match de tennis. On pourra utiliser des tableaux et une numérotation numérique des joueurs. On pourra aussi coder le codage des points pour faciliter le calcul des scores.*

*b) On souhaite pouvoir adapter le programme aux évolutions suivantes.*

1. *Dans un tournoi certains matchs se jouent en deux sets gagnants (début du tournoi, tournoi féminin...).*
2. *On souhaite inclure ce programme dans le cadre d'un tournoi complet et conserver tous les scores jusqu'à la finale.*
3. *On souhaite modifier le programme pour des matchs de doubles.*
4. *On souhaite écrire un programme similaire pour un tournoi de tennis de table.*
5. *Un joueur peut abandonner la partie.*

*Commenter l'adaptabilité du programme aux modifications souhaitées.*

*c) Proposer une conception abstraite à objets. On proposera par exemple une modélisation avec la notation UML.*

*d) Reprendre la question b) avec la conception à objets.*

*e) Implanter la conception à objets en Java. Discuter des alternatives de codage.*

## 4 Applications

**Exercice 4.1**
*Développer un agenda (nom, téléphone) en utilisant le générateur d'interface de* `Visualworks`.

**Exercice 4.2**
*Développer un browser d'objets géométriques, qui affiche les formes choisies, en utilisant le générateur d'interface de* `Visualworks`.

# Annexe C

# Classes

Liste de classes
http ://www.eteks.com/coursjava/java10.html#Hierarchie

```
*  Classe java.lang.Object
      o Interface java.applet.AppletContext
      o Interface java.applet.AppletStub
      o Interface java.applet.AudioClip
      o Classe java.util.BitSet (implémente java.lang.Cloneable)
      o Classe java.lang.Boolean
      o Classe java.awt.BorderLayout (implémente java.awt.LayoutManager)
      o Classe java.awt.CardLayout (implémente java.awt.LayoutManager)
      o Classe java.lang.Character
      o Classe java.awt.CheckboxGroup
      o Classe java.lang.Class
      o Classe java.lang.ClassLoader
      o Interface java.lang.Cloneable
      o Classe java.awt.Color
      o Classe java.awt.image.ColorModel
            + Classe java.awt.image.DirectColorModel
      + Classe java.awt.image.IndexColorModel
      o Classe java.lang.Compiler
      o Classe java.awt.Component (implémente java.awt.image.ImageObserver)
            + Classe java.awt.Button
            + Classe java.awt.Canvas
            + Classe java.awt.Checkbox
            + Classe java.awt.Choice
            + Classe java.awt.Container
                  # Classe java.awt.Panel
                  * Classe java.applet.Applet
                  # Classe java.awt.Window
                        * Classe java.awt.Dialog
                        o Classe java.awt.FileDialog
                  * Classe java.awt.Frame (implémente java.awt.MenuContainer)
            + Classe java.awt.Label
            + Classe java.awt.List
            + Classe java.awt.Scrollbar
            + Classe java.awt.TextComponent
                  # Classe java.awt.TextArea
            # Classe java.awt.TextField
      o Classe java.net.ContentHandler
      o Interface java.net.ContentHandlerFactory
      o Interface java.io.DataInput
      o Interface java.io.DataOutput
```

```
o Classe java.net.DatagramPacket
o Classe java.net.DatagramSocket
o Classe java.util.Date
o Classe java.util.Dictionary
      + Classe java.util.Hashtable (implémente java.lang.Cloneable)
      # Classe java.util.Properties
o Classe java.awt.Dimension
o Interface java.util.Enumeration
o Classe java.awt.Event
o Classe java.io.File
o Classe java.io.FileDescriptor
o Interface java.io.FilenameFilter
o Classe java.awt.image.FilteredImageSource (implémente java.awt.image.ImageProducer)
o Classe java.awt.FlowLayout (implémente java.awt.LayoutManager)
o Classe java.awt.Font
o Classe java.awt.FontMetrics
o Classe java.awt.Graphics
o Classe java.awt.GridBagConstraints (implémente java.lang.Cloneable)
o Classe java.awt.GridBagLayout (implémente java.awt.LayoutManager)
o Classe java.awt.GridLayout (implémente java.awt.LayoutManager)
o Classe java.awt.Image
o Interface java.awt.image.ImageConsumer
o Classe java.awt.image.ImageFilter (implémente java.awt.image.ImageConsumer, java.lang.Clone
      + Classe java.awt.image.CropImageFilter
+ Classe java.awt.image.RGBImageFilter
o Interface java.awt.image.ImageObserver
o Interface java.awt.image.ImageProducer
o Classe java.net.InetAddress
o Classe java.io.InputStream
      + Classe java.io.ByteArrayInputStream
      + Classe java.io.FileInputStream
      + Classe java.io.FilterInputStream
            # Classe java.io.BufferedInputStream
            # Classe java.io.DataInputStream (implémente java.io.DataInput)
            # Classe java.io.LineNumberInputStream
      # Classe java.io.PushbackInputStream
      + Classe java.io.PipedInputStream
      + Classe java.io.SequenceInputStream
+ Classe java.io.StringBufferInputStream
o Classe java.awt.Insets (implémente java.lang.Cloneable)
o Interface java.awt.LayoutManager
o Classe java.lang.Math
o Classe java.awt.MediaTracker
o Classe java.awt.image.MemoryImageSource (implémente java.awt.image.ImageProducer)
o Classe java.awt.MenuComponent
      + Classe java.awt.MenuBar (implémente java.awt.MenuContainer)
      + Classe java.awt.MenuItem
            # Classe java.awt.CheckboxMenuItem
      # Classe java.awt.Menu (implémente java.awt.MenuContainer)
o Interface java.awt.MenuContainer
o Classe java.lang.Number
      + Classe java.lang.Double
      + Classe java.lang.Float
      + Classe java.lang.Integer
+ Classe java.lang.Long
o Classe java.util.Observable
o Interface java.util.Observer
o Classe java.io.OutputStream
```

```
      + Classe java.io.ByteArrayOutputStream
      + Classe java.io.FileOutputStream
      + Classe java.io.FilterOutputStream
            # Classe java.io.BufferedOutputStream
            # Classe java.io.DataOutputStream (implémente java.io.DataOutput)
      # Classe java.io.PrintStream
+ Classe java.io.PipedOutputStream
o Classe java.awt.image.PixelGrabber (implémente java.awt.image.ImageConsumer)
o Classe java.awt.Point
o Classe java.awt.Polygon
o Classe java.lang.Process
o Classe java.util.Random
o Classe java.io.RandomAccessFile (implémente java.io.DataOutput, java.io.DataInput)
o Classe java.awt.Rectangle
o Interface java.lang.Runnable
o Classe java.lang.Runtime
o Classe java.lang.SecurityManager
o Classe java.net.ServerSocket
o Classe java.net.Socket
o Classe java.net.SocketImpl
o Interface java.net.SocketImplFactory
o Classe java.io.StreamTokenizer
o Classe java.lang.String
o Classe java.lang.StringBuffer
o Classe java.util.StringTokenizer (implémente java.util.Enumeration)
o Classe java.lang.System
o Classe java.lang.Thread (implémente java.lang.Runnable)
o Classe java.lang.ThreadGroup
o Classe java.lang.Throwable
      + Classe java.lang.Error
            # Classe java.awt.AWTError
            # Classe java.lang.LinkageError
                  * Classe java.lang.ClassCircularityError
                  * Classe java.lang.ClassFormatError
                  * Classe java.lang.IncompatibleClassChangeError
                        o Classe java.lang.AbstractMethodError
                        o Classe java.lang.IllegalAccessError
                        o Classe java.lang.InstantiationError
                        o Classe java.lang.NoSuchFieldError
                  o Classe java.lang.NoSuchMethodError
                  * Classe java.lang.NoClassDefFoundError
                  * Classe java.lang.UnsatisfiedLinkError
            * Classe java.lang.VerifyError
            # Classe java.lang.ThreadDeath
            # Classe java.lang.VirtualMachineError
                  * Classe java.lang.InternalError
                  * Classe java.lang.OutOfMemoryError
                  * Classe java.lang.StackOverflowError
            * Classe java.lang.UnknownError
      + Classe java.lang.Exception
            # Classe java.awt.AWTException
            # Classe java.lang.ClassNotFoundException
            # Classe java.lang.CloneNotSupportedException
            # Classe java.lang.IllegalAccessException
            # Classe java.lang.InstantiationException
            # Classe java.lang.InterruptedException
            # Classe java.io.IOException
                  * Classe java.io.EOFException
```

```
                        * Classe java.io.FileNotFoundException
                        * Classe java.io.InterruptedIOException
                        * Classe java.net.MalformedURLException
                        * Classe java.net.ProtocolException
                        * Classe java.net.SocketException
                        * Classe java.io.UTFDataFormatException
                        * Classe java.net.UnknownHostException
                    * Classe java.net.UnknownServiceException
                    # Classe java.lang.RuntimeException
                        * Classe java.lang.ArithmeticException
                        * Classe java.lang.ArrayStoreException
                        * Classe java.lang.ClassCastException
                        * Classe java.util.EmptyStackException
                        * Classe java.lang.IllegalArgumentException
                            o Classe java.lang.IllegalThreadStateException
                        o Classe java.lang.NumberFormatException
                        * Classe java.lang.IllegalMonitorStateException
                        * Classe java.lang.IndexOutOfBoundsException
                            o Classe java.lang.ArrayIndexOutOfBoundsException
                        o Classe java.lang.StringIndexOutOfBoundsException
                        * Classe java.lang.NegativeArraySizeException
                        * Classe java.util.NoSuchElementException
                        * Classe java.lang.NullPointerException
                    * Classe java.lang.SecurityException
            o Classe java.awt.Toolkit
            o Classe java.net.URL
            o Classe java.net.URLConnection
            o Classe java.net.URLEncoder
            o Classe java.net.URLStreamHandler
            o Interface java.net.URLStreamHandlerFactory
o Classe java.util.Vector (implémente java.lang.Cloneable)
```

# Table des figures

# Liste des tableaux

# Listings