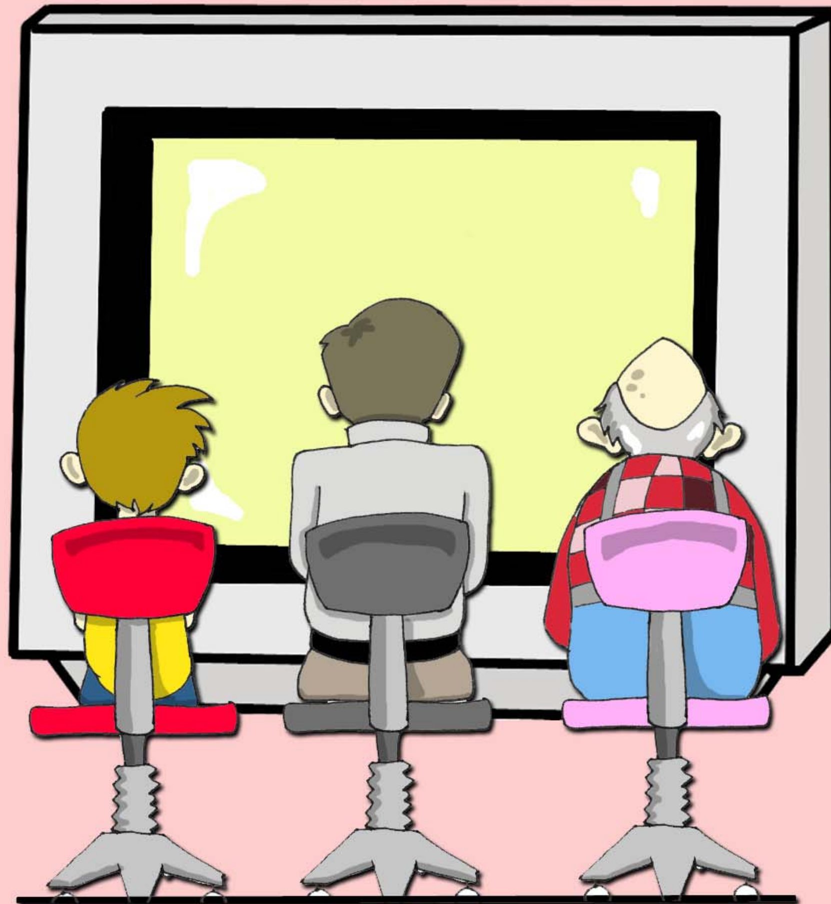
www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

PROGRAMMATION JAVA™

pour les enfants, les parents et les grands-parents



YAKOV FAIN

Programmation Java™ pour les enfants, les parents et les grands-parents

Yakov Fain

Traduit de l'anglais (américain) par Vincent Lataye et Maxime Daniel
(vincent_lataye@xoteam.fr, maxime_daniel@xoteam.fr)

Programmation Java pour les enfants, les parents et les grands-parents

Yakov Fain

Copyright © 2004
Copyright © 2005 pour la traduction française
Smart Data Processing, Inc.
14 Molly Pitcher Dr.
Manalapan, New Jersey, 07726, USA

Tous droits réservés. Toute reproduction, même partielle, par quelque procédé et par qui que ce soit, est interdite sans autorisation écrite préalable de l'éditeur.

Couverture et illustrations : Yuri Fain

Rédacteur technique adulte : Yuri Goncharov

Rédacteur technique enfant : David Fain

Mai 2004 : Première édition électronique
Juin 2005 : Première édition électronique en français

L'information contenue dans ce livre n'est pas sujette à garantie. Ni l'auteur ni l'éditeur ne pourront être tenus responsables des préjudices ou dommages de quelque nature que ce soit pouvant résulter directement ou indirectement des instructions fournies dans ce livre ou de l'utilisation des logiciels ou matériels informatiques qui y sont décrits.

Java et toutes les marques et logos dérivés de Java sont des marques commerciales ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans les autres pays.

Windows 98, Windows NT, Windows 2000 et Windows XP sont des marques commerciales de Microsoft Corporation.
Tous les autres noms de produits et de sociétés sont la propriété de leurs propriétaires respectifs.

L'éditeur met la version française de ce livre électronique à la disposition du public à titre gracieux. Pour plus d'information, écrire à books@smartdataprocessing.com.

ISBN: 0-9718439-4-5

Table des matières

PREFACE IX

REMERCIEMENTS XI

CHAPITRE 1. TON PREMIER PROGRAMME JAVA 12

Comment installer Java sur ton ordinateur 13

Les trois étapes principales de la programmation 17
Etape 1 – Tape le programme 17
Etape 2 – Compile le programme 19
Etape 3 – Exécute le programme 20

Autres lectures 21

CHAPITRE 2. PASSAGE A ECLIPSE 22

Installation d'Eclipse 22

Démarrer avec Eclipse 26

Création de programmes dans Eclipse 28

Exécution de `BonjourMonde` dans Eclipse 30

Comment fonctionne `BonjourMonde` ? 31

Autres lectures 34

Exercices 34

Exercices pour les petits malins 34

CHAPITRE 3. ANIMAUX FAMILIERS ET POISSONS – CLASSES JAVA 35

Classes et Objets 35

Types de données 38

Création d'un animal familier 42

Héritage – un Poisson est aussi un AnimalFamilier 47

Surcharge d'une méthode 50

Autres lectures 52

Exercices 52

Exercices pour les petits malins 53

CHAPITRE 4. BRIQUES DE BASE JAVA 54

Commentaires de programme **54**

Prises de décisions à l'aide de la clause `if` **55**

Opérateurs logiques **57**

Opérateur conditionnel **58**

Utilisation de `else if` **58**

Prises de décisions à l'aide de la clause `switch` **60**

Quelle est la durée de vie des variables ? **61**

Méthodes spéciales : constructeurs **62**

Le mot-clé `this` **63**

Tableaux **63**

Répétition d'actions à l'aide de boucles **65**

Autres lectures **68**

Exercices **68**

Exercices pour les petits malins **69**

CHAPITRE 5. UNE CALCULATRICE GRAPHIQUE **70**

AWT et Swing **70**

Paquetages et déclarations d'importation **70**

Principaux éléments de Swing **71**

Gestionnaires de disposition **74**

`FlowLayout` **74**

`GridLayout` **75**

`BorderLayout` **77**
Combiner les gestionnaires de disposition 77

`BoxLayout` **80**

`GridBagLayout` **81**

`CardLayout` **83**

`SpringLayout` **84**
Puis-je créer des fenêtres sans utiliser de gestionnaire de disposition ? 84

Composants des fenêtres **85**

Autres lectures **88**

Exercices **88**

Exercices pour les petits malins 89

CHAPITRE 6. EVENEMENTS DE LA FENETRE **90**

Interfaces 91

Récepteur d'événements 93
Enregistrement d'un `ActionListener` auprès d'un composant 94
Quelle est la source d'un événement ? 95

Comment passer des données entre classes 97

Fin de la calculatrice 99
Autres récepteurs d'événements 105

Utilisation des adapteurs 106

Autres lectures 107

Exercices 107

Exercices pour les petits malins 107

CHAPITRE 7. L'APPLET MORPION **108**

Apprendre HTML en 15 minutes 109

Choix de la librairie AWT pour écrire des applets 112

Comment écrire des applets AWT 113

Ecriture d'un jeu de morpion 115
Stratégie 115
Code 116

Autres lectures 127

Exercices 127

Exercices pour les petits malins 128

CHAPITRE 8. ERREURS ET EXCEPTIONS **129**

Lecture de la trace de la pile 130

Arbre généalogique des exceptions 131

Bloc `try/catch` 132

Le mot-clé `throws` 135

Le mot-clé `finally` 136

Le mot-clé `throw` 137

Création de nouvelles exceptions 139

Autres lectures **141**

Exercices **141**

Exercices pour les petits malins **141**

CHAPITRE 9. ENREGISTREMENT DU SCORE **142**

Flux d'octets **143**

Flux à tampon **145**

Arguments de la ligne de commande **147**

Lecture de fichiers texte **150**

Classe `File` (*fichier*) **152**

Autres lectures **155**

Exercices **155**

Exercices pour les petits malins **156**

CHAPITRE 10. AUTRES BRIQUES DE BASE JAVA **157**

Utilisation des valeurs de date et d'heure **157**

Surcharge de méthode **159**

Lecture des entrées clavier **161**

Compléments sur les paquetages Java **163**

Niveaux d'accès **166**

Retour sur les tableaux **169**

Classe `ArrayList` **173**

Autres lectures **177**

Exercices **177**

Exercices pour les petits malins **177**

CHAPITRE 11. RETOUR SUR LE GRAPHIQUE – LE JEU DE PING-PONG **178**

Stratégie **178**

Code **179**

Bases des fils d'exécution Java **187**

Fin du jeu de ping-pong **192**

Que lire d'autre sur la programmation de jeux ? **202**

Autres lectures ... 203

Exercices ... 203

Exercices pour les petits malins ... 204

ANNEXE A. ARCHIVES JAVA - JARS ... 205

Autres lectures ... 206

ANNEXE B. ASTUCES ECLIPSE ... 207

Débogueur Eclipse ... 208

ANNEXE C. COMMENT PUBLIER UNE PAGE WEB ... 212

Autres lectures ... 215

Exercices ... 215

INDEX ... 216

Préface

Un jour, mon fils Davey entra dans mon bureau tel un ouragan, mon didacticiel Java pour adultes à la main. Il me demanda de lui apprendre à programmer pour pouvoir créer des jeux informatiques. A cette époque, j'avais déjà écrit une paire de livres sur Java et donné de nombreux cours de programmation, mais c'était pour les grandes personnes ! Une recherche sur Amazon n'a rien donné d'autre que des livres "pour les nuls", mais Davey n'est pas un "nul" ! Après avoir passé des heures sur Google, je ne trouvai que de pauvres tentatives de cours de Java pour les enfants ou des livres très superficiels. Sais-tu ce que j'ai fait ? J'ai décidé d'en écrire un. Pour m'aider à comprendre la mentalité des plus jeunes, je décidai de demander à Davey de devenir mon premier étudiant enfant.

Ce livre sera utile aux personnes suivantes :

- Les jeunes de 11 à 18 ans.
- Les enseignants en informatique.
- Les parents souhaitant apprendre à programmer à leurs enfants.
- Tous ceux qui débutent en programmation (l'âge n'a pas d'importance).

Même si j'emploie un langage simple pour expliquer la programmation, je m'engage à traiter mes lecteurs avec respect – je ne vais pas écrire quelque chose comme "Cher ami ! Tu es sur le point de commencer un nouveau et passionnant voyage…". Bon, d'accord ! Venons-en aux faits.

Les premiers chapitres de ce livre aboutiront à des programmes de jeu simples, avec les instructions détaillées pour les faire fonctionner. Nous allons aussi créer une calculatrice qui ressemble à celle que tu as dans ton ordinateur. Dans la seconde partie du livre, nous créerons ensemble les programmes de jeu Morpion et Ping-Pong.

Tu devras t'habituer à l'argot des programmeurs professionnels ; les mots importants sont écrits dans *cette police.*

Les programmes et les éléments du langage Java sont écrits dans une autre police, par exemple `String`.

Ce livre ne couvre pas tous les éléments du langage Java, sinon il serait trop gros et ennuyeux. Mais à la fin de chaque chapitre, il y a une section *Autres lectures* avec des liens vers des sites web contenant des explications plus détaillées sur le sujet.

Tu trouveras aussi des devoirs à la fin de chaque chapitre. Chaque lecteur doit faire les exercices de la section *Exercices*. Si ceux-ci sont trop faciles pour toi, je te mets au défi de faire ceux de la section *Exercices pour les petits malins*. En fait, si tu lis ce livre, tu es une personne intelligente et tu devrais essayer de faire tous les exercices.

Pour retirer le maximum de ce livre, lis-le du début à la fin. Ne change pas de chapitre avant d'avoir compris celui que tu es en train de lire. Les ados, les parents et les grands-parents devraient pouvoir maîtriser ce livre sans aide extérieure, mais les enfants plus jeunes devraient lire ce livre avec l'aide d'un adulte.

Remerciements

Merci à tous les architectes et développeurs qui ont travaillé gratuitement sur Eclipse – l'un des meilleurs environnements de développement intégrés disponibles pour Java.

Remerciements particuliers aux chauffeurs de bus de la société New Jersey Transit pour la souplesse de leur conduite – la moitié de ce livre a été écrite dans le bus n° 139, que je prenais pour aller au travail.

Merci à cette dame adorable, ma femme, Natasha, pour avoir fait tourner avec succès cette affaire qu'on appelle famille.

Remerciements particuliers à Yuri Goncharov – un développeur Java expert de Toronto, au Canada. Il a relu le livre, testé chaque exemple de code et fourni un retour précieux afin d'améliorer un peu ce livre.

Chapitre 1. Ton premier programme Java

Les gens communiquent entre eux à l'aide de différentes langues. De la même façon, on peut écrire des programmes informatiques, tels que des jeux, calculatrices ou éditeurs de texte, à l'aide de différents langages de programmation. Sans programmes, ton ordinateur serait inutile et son écran serait toujours noir. Les éléments de l'ordinateur constituent le *matériel* (*hardware*) [1] et les programmes le *logiciel (software).* Les langages informatiques les plus populaires sont Visual Basic, C++ et Java. Qu'est-ce qui fait de Java un langage différent de beaucoup d'autres ?

Premièrement, le même programme Java peut *tourner* (fonctionner) sur différents ordinateurs, tels que PC, Apple et autres, sans modification. En fait, les programmes Java ne savent même pas où ils s'exécutent, car ils le font à l'intérieur d'une enveloppe logicielle spéciale appelée Machine Virtuelle Java, ou plus simplement Java. Si, par exemple, ton programme Java a besoin d'imprimer des messages, il demande à Java de le faire et Java sait comment se débrouiller avec ton imprimante.

Deuxièmement, Java permet de traduire facilement tes programmes (écrans, menus et messages) en différentes langues.

Troisièmement, Java te permet de créer des composants logiciels (*classes*) [2] qui représentent les objets du monde réel. Par exemple, tu peux créer une *classe* Java nommée `Voiture` et lui donner des attributs tels que portes ou roues similaires à ceux d'une vraie voiture. Ensuite, à partir de cette classe, tu peux créer une autre classe, par

---

[1] NDT : L'anglais constituant la langue de référence dans le monde de l'informatique en général et de la programmation en particulier, nous avons pris le parti de rappeler, entre parenthèses, l'équivalent anglais des principaux termes techniques lors de leur première apparition dans le texte.

[2] NDT : En français : une *classe*, des *classes*. En anglais : a *class*, *classes*.

exemple `Ford`, qui aura toutes les caractéristiques de la classe `Voiture` et d'autres que seules les Ford possèdent.

Quatrièmement, Java est plus puissant que beaucoup d'autres langages.

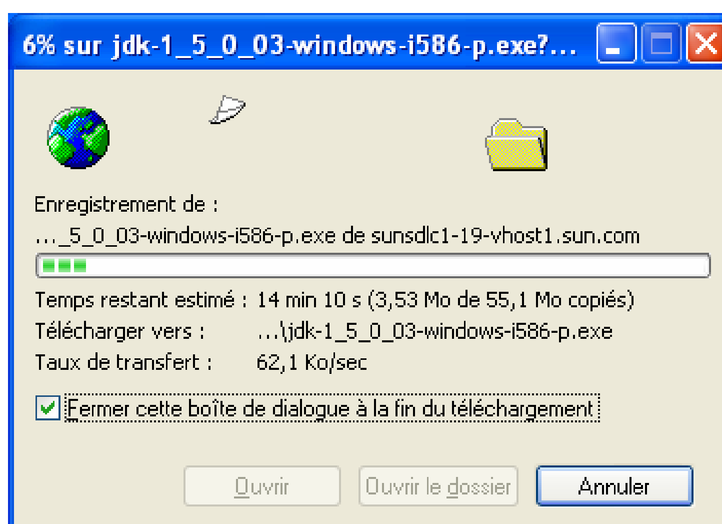
Cinquièmement, Java est gratuit ! Tu peux trouver sur Internet tout ce qu'il faut pour créer tes programmes Java sans débourser un euro !

Comment installer Java sur ton ordinateur

Pour pouvoir programmer en Java, tu as besoin de télécharger un logiciel spécial depuis le site web de la société Sun Microsystems, qui a créé ce langage. Le nom complet de ce logiciel est Java 2 Software Development Kit (J2SDK). A l'heure où j'écris ces lignes, la dernière version disponible, 1.5.0, peut être téléchargée depuis ce site :

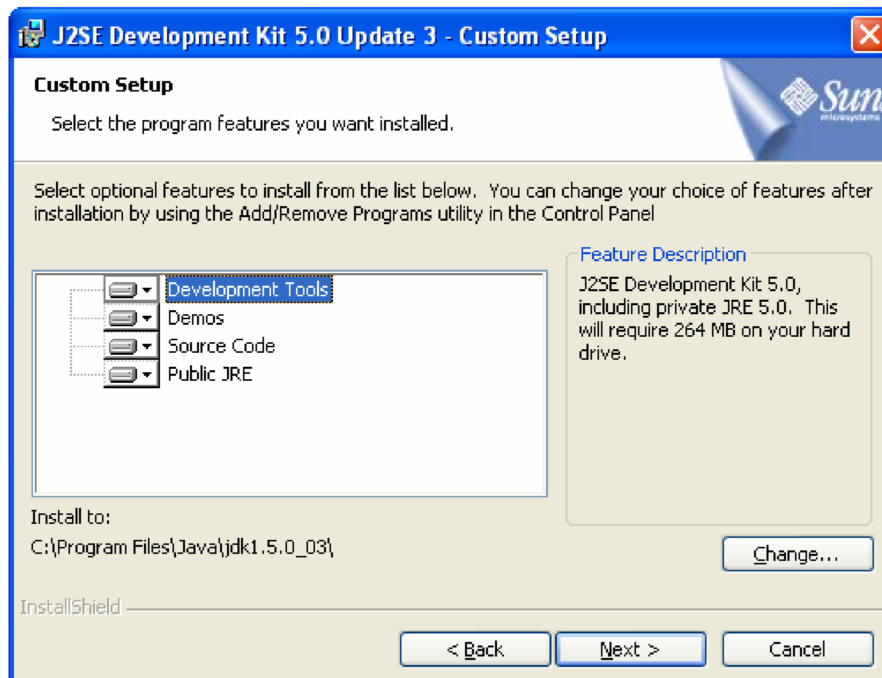
http://java.sun.com/j2se

Sélectionne la version (*release*) J2SE 1.5.0 ou la plus récente dans la rubrique Downloads, puis sur la page de téléchargement clique sur le lien Download JDK. Accepte le contrat de licence et sélectionne *Windows Offline Installation* (à moins que tu aies un ordinateur Macintosh, Linux ou Solaris). Dans l'écran suivant, clique sur le bouton *Enregistrer* puis choisis le répertoire de ton disque dur où tu souhaites enregistrer le fichier d'installation Java.



Une fois le téléchargement terminé, lance l'installation – double-clique simplement sur le fichier que tu as téléchargé pour installer J2SDK sur ton disque. Par exemple, sur une machine Windows, le programme d'installation créera un répertoire comme celui-ci :

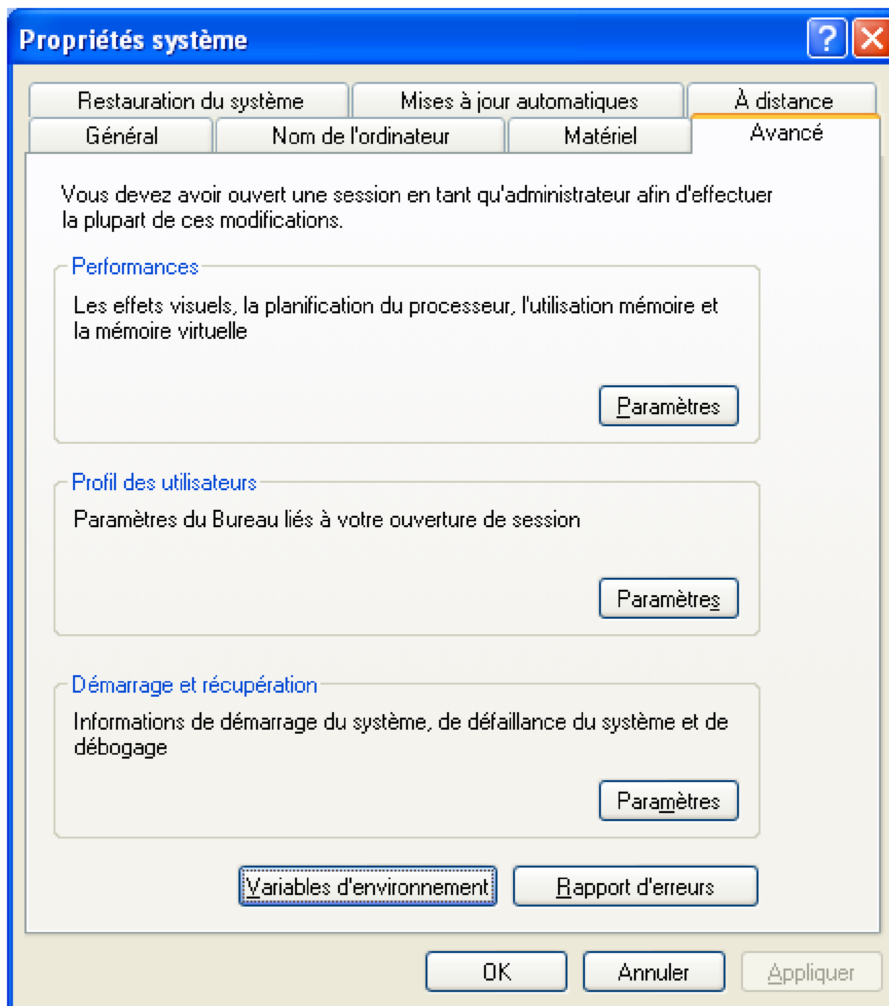
c:\Program Files\java\j2sdk1.5.0_03, où c: est le nom de ton disque dur.



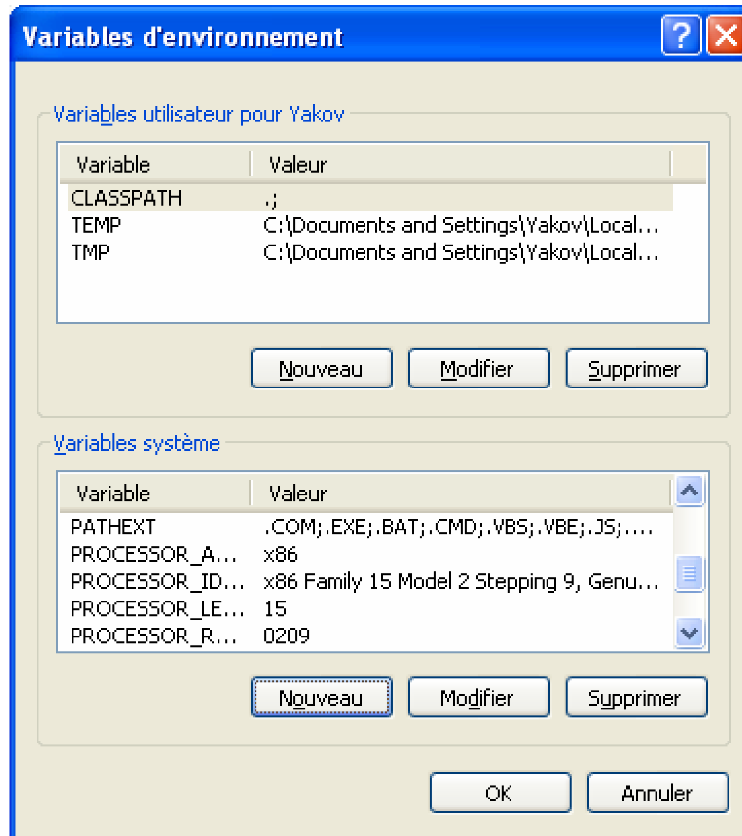
Si tu n'as pas assez de place sur ton disque c:, choisis-en un autre, sinon, contente-toi d'appuyer sur les boutons *Next*, *Install* et *Finish* dans les fenêtres qui s'affichent à l'écran. L'installation de Java sur ton ordinateur ne prendra que quelques minutes.

A l'étape suivante de l'installation, tu dois définir deux *variables système*. Dans Windows, par exemple, clique sur le bouton *démarrer* et ouvre le *Panneau de configuration* (il peut être caché derrière le menu *Paramètres*), puis double-clique sur l'icône *Système*. Sélectionne l'onglet *Avancé* et clique sur le bouton *Variables d'environnement*.

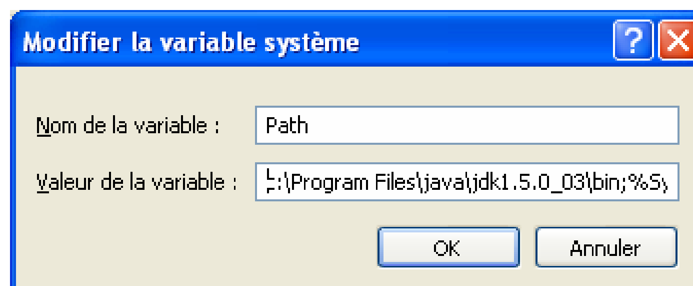
Voici à quoi ressemble cet écran sur mon ordinateur Windows XP :



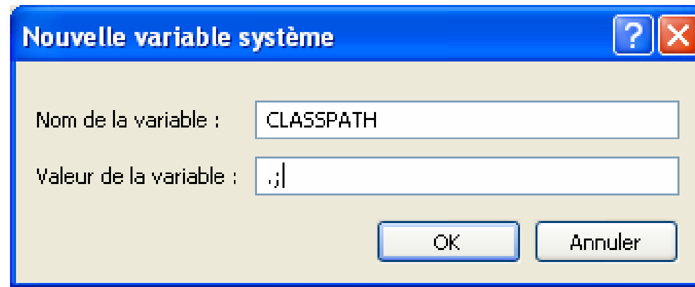
La fenêtre suivante présente toutes les variables système qui existent déjà dans ton système.



Clique sur le bouton *Nouveau* en bas de la fenêtre et déclare la variable `Path` qui permettra à Windows (ou Unix) de trouver J2SDK sur ta machine. Vérifie bien le nom du répertoire où tu as installé Java. Si la variable `Path` existe déjà, clique sur le bouton *Modifier* et ajoute le nom complet du répertoire Java suivi d'un point-virgule au tout début de la boîte *Valeur de la variable* :



De même, déclare la variable `CLASSPATH` en entrant comme valeur un point suivi d'un point-virgule. Cette variable système permettra à Java de trouver tes programmes. Le point signifie que Java doit chercher tes programmes à partir du répertoire courant. Le point-virgule n'est qu'un séparateur :



L'installation de J2SDK est maintenant terminée !

Si tu as un vieil ordinateur Windows 98, tu dois positionner les variables `PATH` et `CLASSPATH` d'une autre manière. Trouve le fichier `autoexec.bat` sur ton disque `c:` et utilise le Bloc-notes ou un autre éditeur de texte pour entrer les bonnes valeurs de ces variables à la fin du fichier, comme ceci :

```
SET CLASSPATH=.;
SET PATH=c:\j2sdk1.5.0_03\bin;%PATH%
```

Après avoir effectué cette modification, il faut redémarrer ton ordinateur.

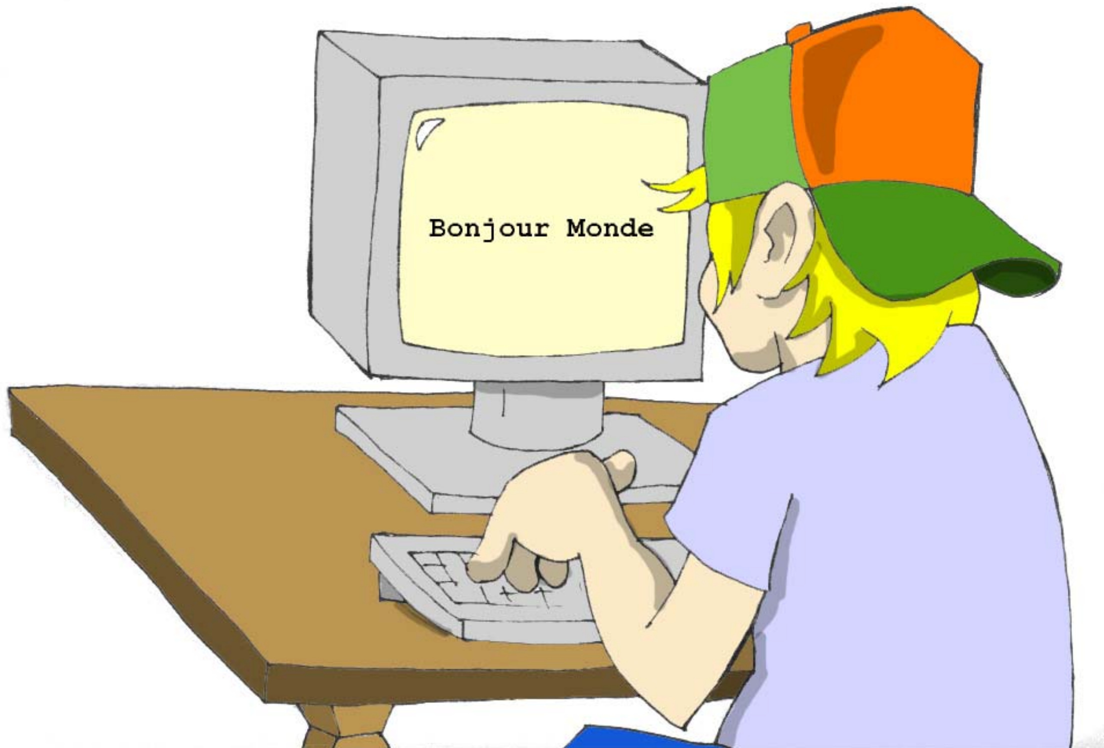
Les trois étapes principales de la programmation

Pour créer un programme Java qui marche, tu dois passer par les étapes suivantes :

- ✓ *Ecrire* le programme en Java et l'enregistrer sur un disque.
- ✓ *Compiler* le programme pour le traduire en un *code binaire* spécial compréhensible par Java.
- ✓ *Exécuter* le programme.

Etape 1 – Tape le programme

Tu peux utiliser n'importe quel éditeur de texte pour écrire des programmes Java, par exemple le Bloc-notes.



Tout d'abord, tu dois taper le programme et l'enregistrer dans un fichier texte dont le nom se termine par *.java*. Par exemple, si tu veux écrire un programme appelé `BonjourMonde`, entre son texte (on appelle ça le *code source*) dans le Bloc-notes et enregistre-le dans un fichier que tu nommeras `BonjourMonde.java`. Ne mets pas de blancs dans les noms de fichiers Java.

Voici le programme qui affiche les mots "Bonjour Monde" à l'écran :

```
public class BonjourMonde {

  public static void main(String[] args) {
    System.out.println("Bonjour Monde");
  }

}
```

J'expliquerai comment fonctionne ce programme un peu plus tard dans ce chapitre, mais pour l'instant fais-moi confiance – ce programme affichera les mots "Bonjour Monde" à l'étape 3.

Etape 2 – Compile le programme

Maintenant, tu dois compiler le programme. Tu vas utiliser le *compilateur* `javac`, qui est une partie de J2SDK.

Disons que tu as enregistré ton programme dans le répertoire `c:\exercices`. Sélectionne les menus *Démarrer*, *Exécuter*, et entre le mot `cmd` pour ouvrir une fenêtre de commande.

Juste pour vérifier que tu as positionné correctement les variables système `PATH` et `CLASSPATH`, entre le mot `set` et jette un œil à leurs valeurs.
Va dans le répertoire `c:\exercices` et compile le programme :

```
cd \exercices

javac BonjourMonde.java
```

Tu n'es pas obligé de nommer le répertoire `exercices` – donne-lui le nom de ton choix.

> Sous Windows 98, sélectionne *Invite de commande MS DOS* depuis le menu *démarrer* pour ouvrir une fenêtre DOS.

Le programme `javac` est le *compilateur* Java. Aucun message ne va te confirmer que ton programme `BonjourMonde` a bien été compilé. C'est le cas typique du "Pas de nouvelles, bonnes nouvelles". Tape la commande `dir` et tu verras la liste de tous les fichiers de ton répertoire. Tu devrais y trouver un nouveau fichier nommé `BonjourMonde.class`. Ca prouve que ton programme a bien été compilé. Ton fichier d'origine, `BonjourMonde.java`, est là aussi, et tu pourras le modifier plus tard pour afficher "Bonjour Maman" ou autre chose.

S'il y a des erreurs de syntaxe dans ton programme, par exemple si tu as oublié la dernière accolade fermante, le compilateur Java affichera

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

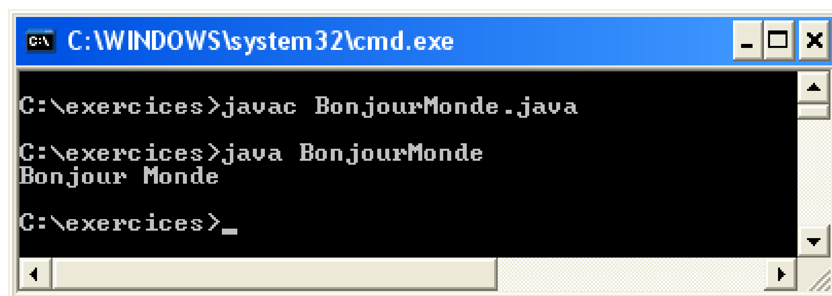
un message d'erreur. Il te faudra alors corriger l'erreur et recompiler le programme. S'il y a plusieurs erreurs, tu peux avoir besoin de répéter ces actions à plusieurs reprises avant d'obtenir le fichier `BonjourMonde.class`.

Etape 3 – Exécute le programme

Maintenant, exécutons le programme. Dans la même fenêtre de commande, tape la commande suivante :

```
java BonjourMonde
```

As-tu remarqué que cette fois tu as utilisé le programme `java` au lieu de `javac` ? Ce programme est appelé *Environnement d'exécution Java* (*Java Run-time Environment* ou *JRE*), ou plus simplement Java comme je l'ai fait jusqu'ici.



N'oublie jamais que Java fait la différence entre les lettres minuscules et majuscules. Par exemple, si tu as nommé le programme `BonjourMonde`, avec un B majuscule et un M majuscule, n'essaie pas de lancer le programme `bonjourmonde` ou `bonjourMonde` – Java se plaindrait.

Maintenant amusons-nous un peu – essaie de deviner comment modifier ce programme. Je t'expliquerai comment fonctionne ce programme dans le prochain chapitre, mais tout de même, essaie de deviner comment le modifier pour qu'il dise bonjour à ton animal familier, ton ami ou qu'il affiche ton adresse. Suis les trois étapes pour voir si le programme fonctionne toujours après tes modifications ☺.

Dans le prochain chapitre, je vais te montrer comment écrire, compiler et exécuter tes programmes dans un environnement plus sympa qu'un éditeur de texte et une fenêtre de commande.

Autres lectures

Pour créer ta première application :
http://java.sun.com/docs/books/tutorial/getStarted/cupojava/win32.html

Instructions d'installation de Java pour Windows :

http://java.sun.com/j2se/1.5.0/install-windows.html

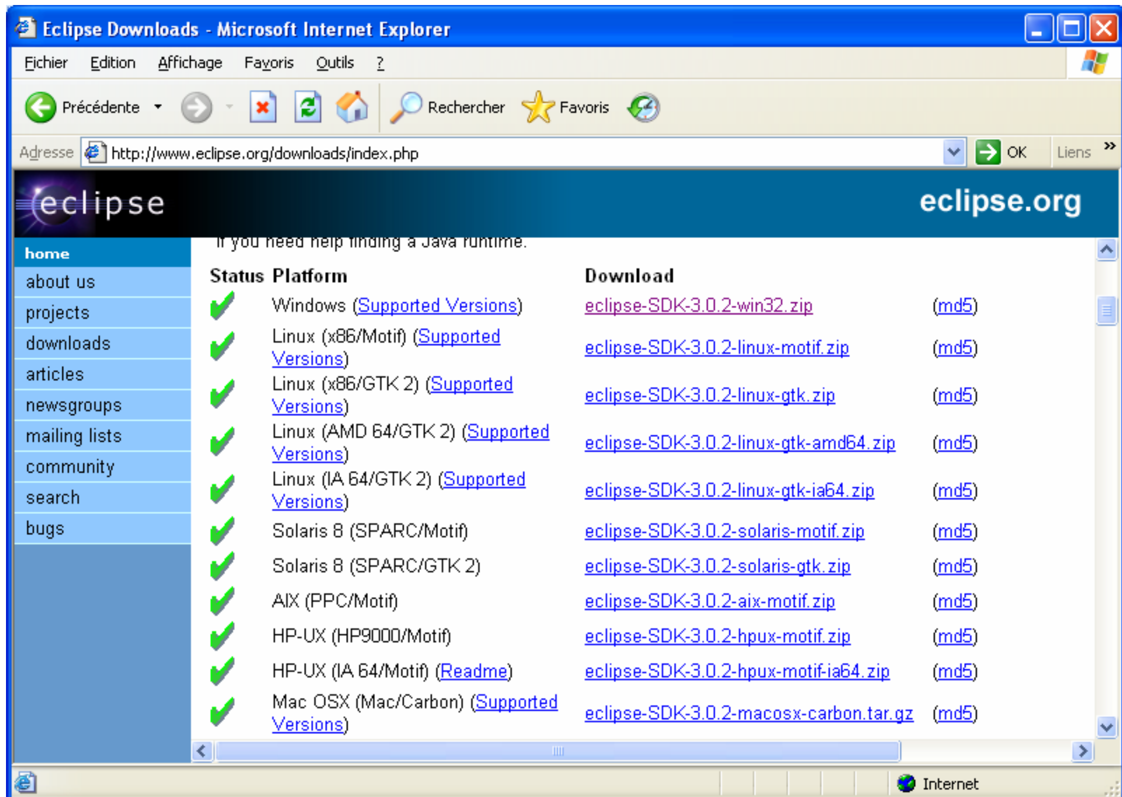
Chapitre 2. Passage à Eclipse

Les programmeurs travaillent généralement dans ce qu'on appelle un *Environnement de développement intégré* (*Integrated Development Environment* ou *IDE*). On peut y écrire, compiler et exécuter les programmes. Un IDE fournit aussi un utilitaire d'*Aide* qui décrit tous les éléments du langage et te permet de trouver et de corriger plus facilement les erreurs dans tes programmes. Alors que la plupart des IDE sont très chers, il en existe un excellent, gratuit : Eclipse. Tu peux le télécharger depuis le site web www.eclipse.org. Dans ce chapitre, je vais t'aider à télécharger et installer l'environnement de développement Eclipse sur ton ordinateur, puis à y créer le projet `Bonjour Monde`. Ensuite, nous créerons tous nos programmes dans cet environnement. Familiarise-toi avec Eclipse – c'est un outil excellent, utilisé par beaucoup de programmeurs Java.

Installation d'Eclipse

Ouvre la page web www.eclipse.org et clique sur le menu *Download* à gauche. Sélectionne ensuite la version d'Eclipse que tu souhaites télécharger. Il y a généralement la version officielle la plus récente (*latest release*) et plusieurs versions intermédiaires stables (*stable builds*). Ces dernières peuvent avoir plus de fonctionnalités, mais elles peuvent encore avoir quelques défauts mineurs. A l'heure où j'écris ces lignes, la dernière version officielle est la 3.0.2. Sélectionne-la [3] pour obtenir la fenêtre suivante :

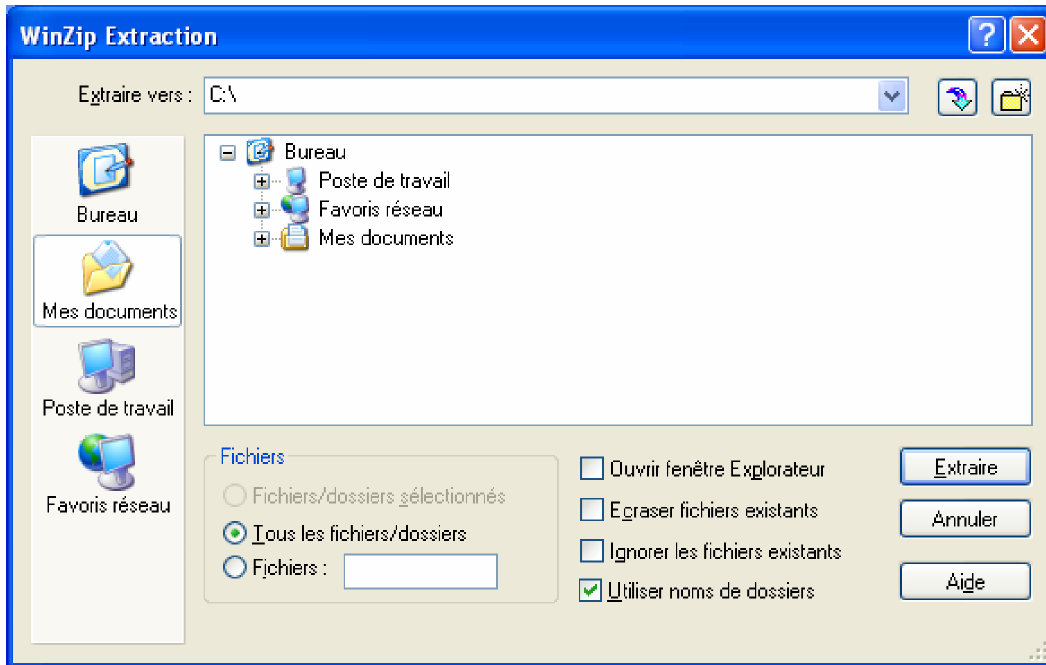
[3] NDT : **Attention** : à moins de préférer travailler dans un environnement en anglais, il est préférable d'utiliser une version officielle, plutôt qu'une version intermédiaire plus récente qui peut ne pas encore être supportée en français. Lis aussi la note d'installation au bas de la page 24.



Clique sur le lien de téléchargement correspondant à ton système et enregistre le fichier avec ce nom si long terminé par *.zip* dans le répertoire de ton choix sur ton disque.

Tu n'as plus qu'à décompresser ce fichier sur ton disque. Si le programme WinZip est déjà installé sur ton ordinateur, clique sur le fichier avec le bouton droit de la souris et sélectionne WinZip dans le menu contextuel, puis l'option *Extraire vers*. Si tu as de la place sur ton disque `c:`, appuie sur le bouton *Extraire*, sinon sélectionne un autre disque avec plus d'espace disponible.

> Les fichiers avec l'extension `.zip` sont des archives contenant de nombreux fichiers compressés. *Décompresser le fichier* signifie *extraire* le contenu de cette archive sur le disque. Le programme d'archivage le plus populaire est WinZip. Tu peux en télécharger une version d'évaluation à l'adresse suivante : http://www.winzip.com/french.htm
>
> Tu en auras besoin pour terminer l'installation d'Eclipse.

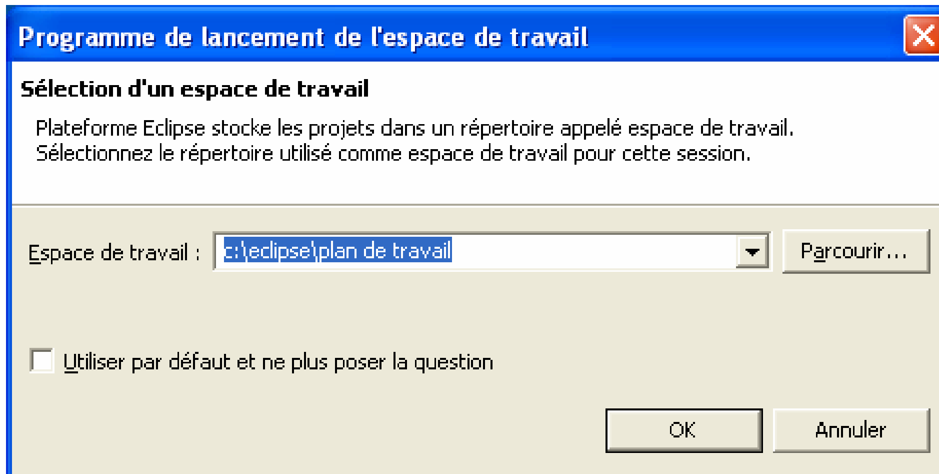


L'installation d'Eclipse est terminée [4] !

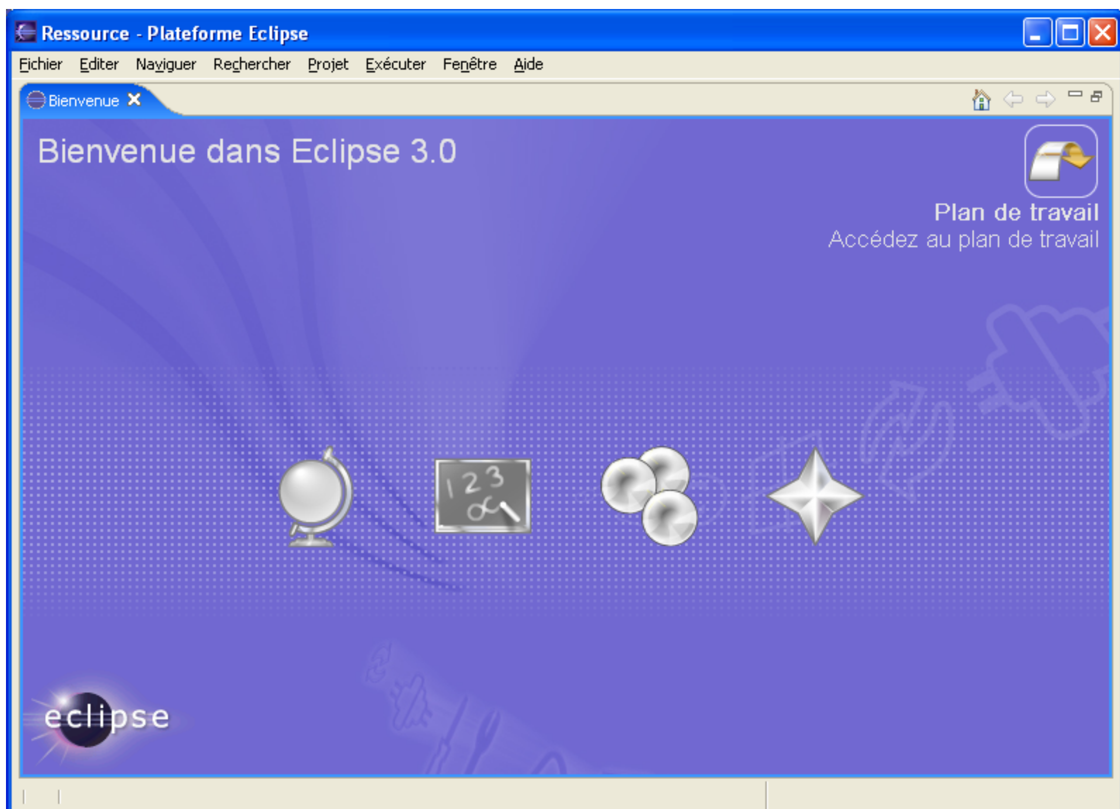
Pour que ce soit plus pratique pour toi, crée un *raccourci* vers Eclipse. Clique avec le bouton droit de la souris sur le bureau, puis clique sur *Nouveau*, *Raccourci*, *Parcourir* et sélectionne le fichier `eclipse.exe` dans le répertoire `c:\eclipse`.

Pour lancer le programme, *double-clique* sur l'icône bleue d'*Eclipse*, et tu verras le premier écran de bienvenue (cet écran peut changer légèrement selon les versions d'Eclipse) :

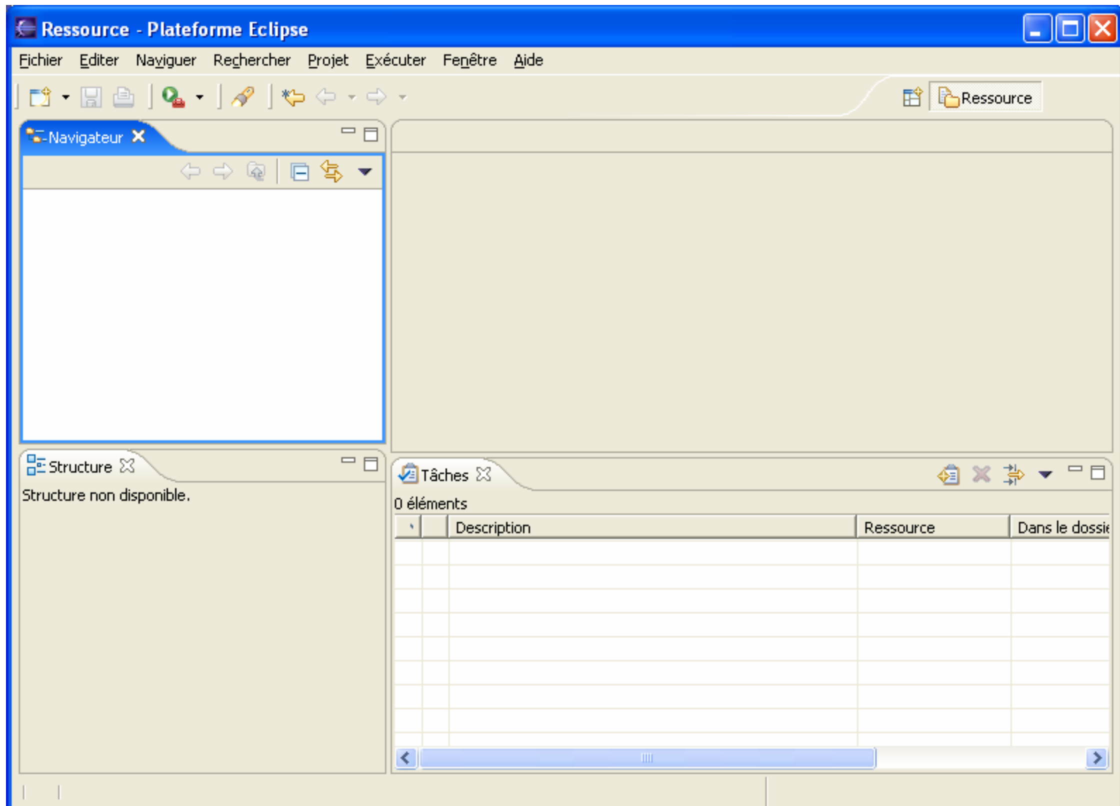
[4] NDT : La version de base d'Eclipse est en anglais. Pour que ton environnement (menus, aide en ligne, messages…) soit en français, il te faut installer quelques fichiers complémentaires. Sur la page web www.eclipse.org, clique sur le menu Download puis sur le lien Platform downloads de la rubrique Eclipse Project, puis sur Language Pack Translations. Dans la section SDK Language Pack, télécharge le fichier .zip correspondant à ton système. Ensuite, décompresse ce fichier à l'endroit où tu as décompressé Eclipse (par exemple `c:\`). **Attention** : cette extension n'est pas forcément compatible avec les versions intermédiaires d'Eclipse.



Complète le nom du plan de travail dans la rubrique *Espace de travail*. Par exemple, `c:\eclipse\plan de travail`. Clique sur le bouton *OK*.



Clique sur la flèche en haut à droite pour afficher ton plan de travail, qui doit ressembler à l'écran suivant :

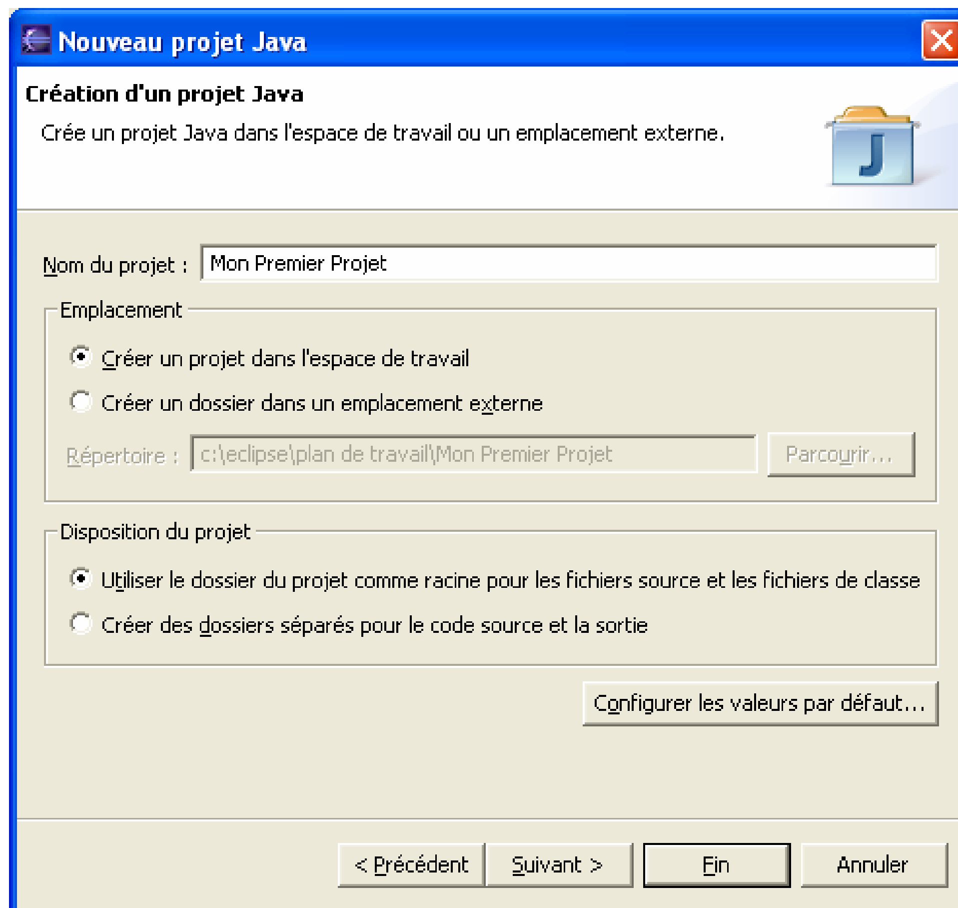


Démarrer avec Eclipse

Dans cette section, je vais te montrer comment créer et exécuter rapidement des programmes Java dans Eclipse. Tu trouveras aussi un didacticiel sympa en passant par le menu *Aide*, *Table des matières de l'aide,* puis *Développement Java – Guide de l'utilisateur*.

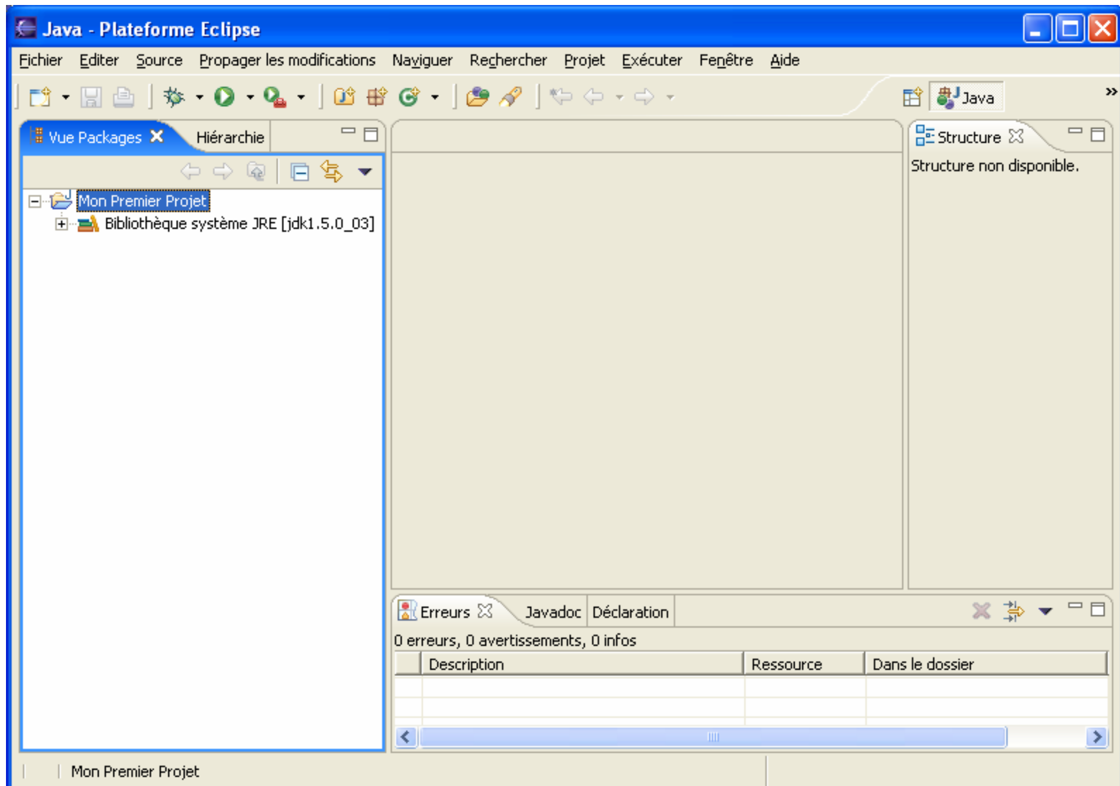
Pour commencer à travailler sur un programme, tu dois créer un nouveau projet. Un projet simple comme notre `BonjourMonde` ne contiendra qu'un fichier – `BonjourMonde.java`. Mais très vite nous allons créer des projets plus élaborés qui contiendront plusieurs fichiers.

Pour créer un nouveau projet dans Eclipse, clique simplement sur les menus *Fichier, Nouveau, Projet, Projet Java*, et clique sur le bouton *Suivant* dans la fenêtre *Nouveau projet*. Tu dois maintenant entrer le nom de ton nouveau projet, par exemple `Mon Premier Projet` :



Regarde la boîte grisée nommée *Répertoire*. Elle t'indique où seront enregistrés les fichiers de ce projet. Eclipse utilise un répertoire spécial appelé `plan de travail` (*workspace*), où seront stockés tous les fichiers de tes projets. Plus tard, tu créeras des projets distincts pour une calculatrice, un jeu de morpion et d'autres programmes. Il y aura plusieurs projets dans le plan de travail avant la fin de ce livre.

Un plan de travail Eclipse contient plusieurs aires plus petites, appelées *perspectives*, qui constituent différentes vues de tes projets. Si Eclipse te le demande, accepte de passer en perspective Java.



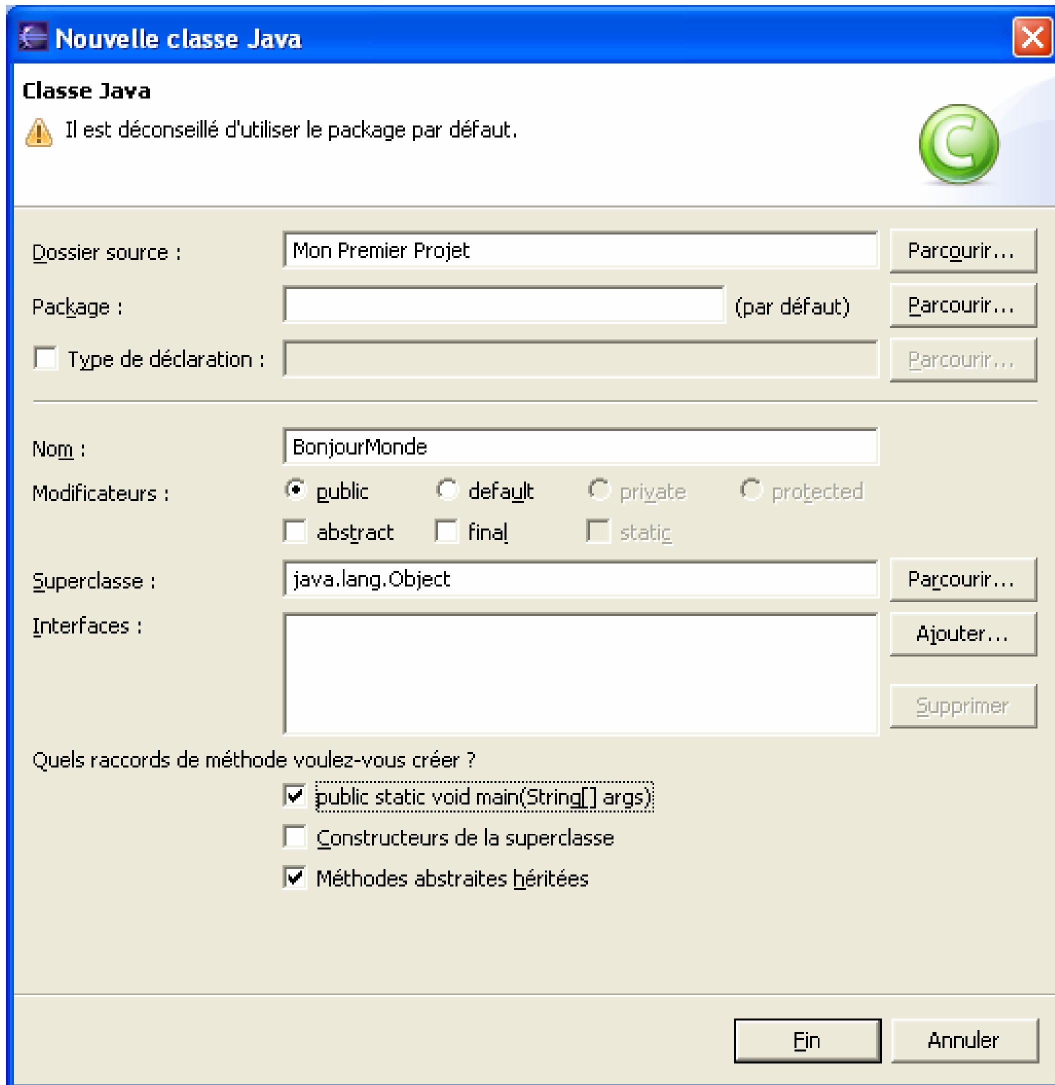
Si tu cliques sur le petit signe plus devant `Mon Premier Projet`, tu verras plus de détails. Pour l'instant, il doit y avoir la ligne `Bibliothèque système JRE (jdk1.5.0_03)`, qui constitue une partie du projet. Si ce n'est pas le cas, clique sur les menus *Fenêtre, Préférences, Java, JRE installés, Ajouter*, puis, à l'aide du bouton *Parcourir*, trouve le répertoire où tu as installé Java, par exemple `c:\Program Files\java\j2sdk1.5.0_03`.

Création de programmes dans Eclipse

Recréons dans Eclipse le programme `BonjourMonde` du chapitre 1. Les programmes Java sont des *classes* qui représentent des objets de la vie réelle. Tu vas en apprendre un peu plus sur les classes dans le prochain chapitre.

Pour créer une classe dans Eclipse, sélectionne les menus *Fichier, Nouveau, Classe* et saisis `BonjourMonde` dans le champ *Nom*. Ensuite, dans la section *Quels raccords de méthodes voulez-vous créer ?*, coche la case :

```
public static void main(String[] args)
```



Clique sur le bouton *Fin* et tu verras que Eclipse a créé pour toi la classe `BonjourMonde`. Des *commentaires de programme* (le texte entre /* et */) ont été générés et placés en haut - tu devrais les modifier pour décrire ta classe. A la suite des commentaires, tu trouveras le code de la classe `BonjourMonde` avec une *méthode (method)* `main()` vide. Le mot *méthode* signifie *action*. Pour exécuter une classe Java comme un programme, cette classe doit posséder une méthode nommée `main()`.

```
public class BonjourMonde {

  public static void main(String[] args) {
  }
}
```

Pour terminer notre programme, place le curseur après l'accolade de la ligne qui contient `main`, appuie sur la touche *Entrée* et tape la ligne suivante :

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com


```
System.out.println("Bonjour Monde");
```

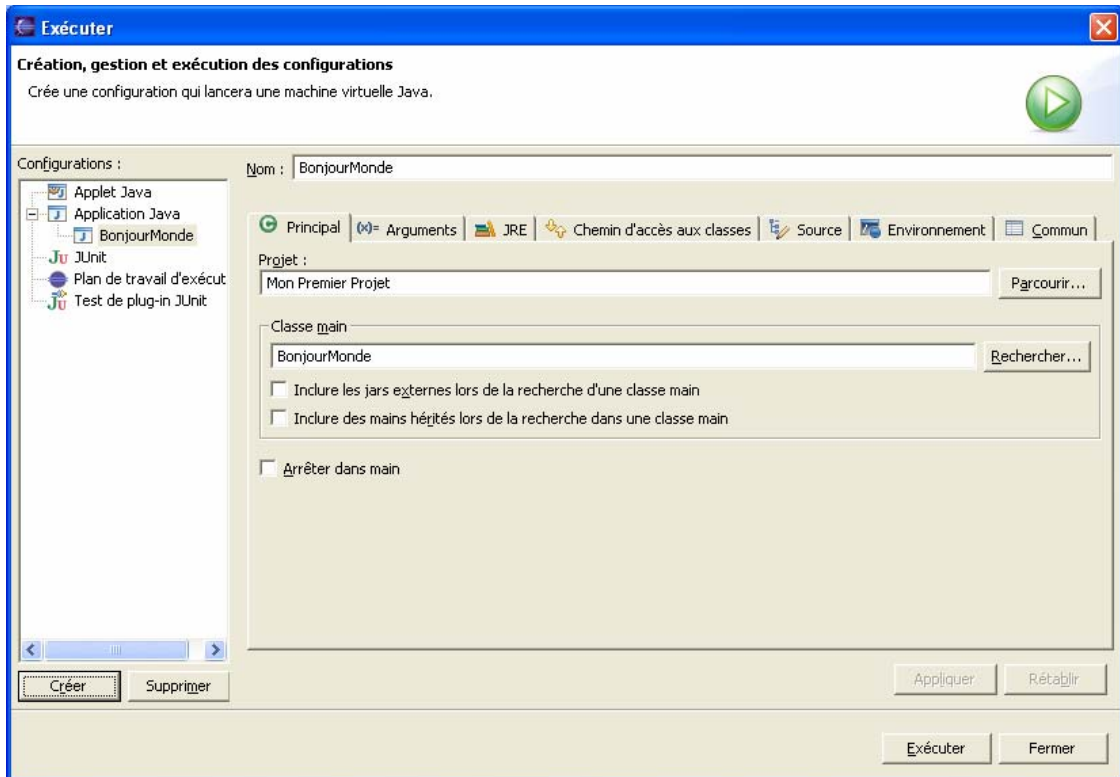
Pour enregistrer le programme sur le disque et le compiler, appuie simultanément sur deux touches de ton clavier : *Ctrl-S*. Si tu n'as pas fait d'erreur de syntaxe, tu ne verras aucun message – le programme est compilé. Mais faisons une erreur volontairement pour voir ce qui va arriver. Supprime la dernière accolade et appuie à nouveau sur *Ctrl-S*. Eclipse affiche alors `Erreur de syntaxe` dans la vue des *Erreurs* et affiche une marque rouge au début de la ligne qui pose problème.

Lorsque tes projets sont plus importants, ils contiennent plusieurs fichiers et le compilateur peut générer plus d'une erreur. Tu peux rapidement trouver (mais pas corriger) les lignes problématiques en double-cliquant sur le message d'erreur dans la vue des erreurs. Remettons l'accolade et appuyons à nouveau sur *Ctrl-S* – voilà, le message d'erreur a disparu !

Exécution de `BonjourMonde` dans Eclipse

Notre petit programme est un projet constitué d'une seule classe. Mais bientôt tes projets contiendront plusieurs classes Java. C'est pourquoi, avant d'exécuter un projet pour la première fois, tu dois dire à Eclipse quelle est la classe principale de ce projet.

Sélectionne le menu *Exécuter*, puis *Exécuter*.... Assure-toi que la ligne *Application Java* est bien sélectionnée dans le coin en haut à gauche, puis clique sur *Créer*. Eclipse initialise pour toi les noms du projet et de la classe principale :



Appuie maintenant sur le bouton *Exécuter* pour lancer le programme. Les mots "Bonjour Monde" seront affichés dans la *vue Console* comme ils l'étaient au chapitre 1.

Tu peux désormais exécuter ce projet en passant par les menus *Exécuter, Exécuter le dernier lancement* ou en appuyant sur les touches *Ctrl-F11* du clavier.

Comment fonctionne `BonjourMonde` ?

Voyons ce qui se passe réellement dans le programme `BonjourMonde`.

La classe `BonjourMonde` ne possède qu'une méthode `main`(), qui est le point d'entrée d'une *application* (programme) Java. On peut dire que `main` est une méthode parce qu'il y a des parenthèses après le mot `main`. Les méthodes peuvent *appeler* (*call*), c'est-à-dire utiliser, d'autres méthodes. Par exemple, notre méthode `main()` *appelle la méthode* `println()` pour afficher le texte "Bonjour Monde" à l'écran.

Toutes les méthodes commencent par une *ligne de déclaration* appelée *signature de la méthode* :

```
public static void main(String[] args)
```

Cette signature nous donne les informations suivantes :

- Qui a accès à la méthode - `public`. Le mot-clé `public` signifie que la méthode `main()` peut être utilisée par n'importe quelle autre classe Java ou par Java lui-même.

- Comment utiliser la méthode - `static`. Le mot-clé `static` signifie qu'il n'est pas nécessaire de créer une *instance* (une copie) de l'objet `BonjourMonde` en mémoire pour pouvoir utiliser cette méthode. Nous reviendrons sur les instances de classe dans le prochain chapitre.

- La méthode *retourne*-t-elle des données ? Le mot-clé `void` signifie que la méthode `main()` ne retourne aucune donnée au programme appelant, qui en l'occurrence est Eclipse. Mais si on prenait l'exemple d'une méthode effectuant des calculs, celle-ci pourrait retourner un résultat à son appelant.

- Le nom de la méthode est `main`.

- La liste des arguments – des données qui peuvent être fournies à la méthode – `String[] args`. Dans la méthode `main()`, `String[] args` signifie que la méthode peut recevoir un *tableau* (*array*) de chaînes de caractères (`String`) qui représentent du texte. Les valeurs qui sont passées à la méthode sont appelées *arguments*.

Comme je l'ai déjà dit, un programme peut être constitué de plusieurs classes, mais l'une d'entre elles possède la méthode `main()`. Les classes Java ont en général plusieurs méthodes. Par exemple, une classe `Jeu` pourrait avoir les méthodes `démarrerJeu()`, `arrêterJeu()`, `lireScore()` et ainsi de suite.

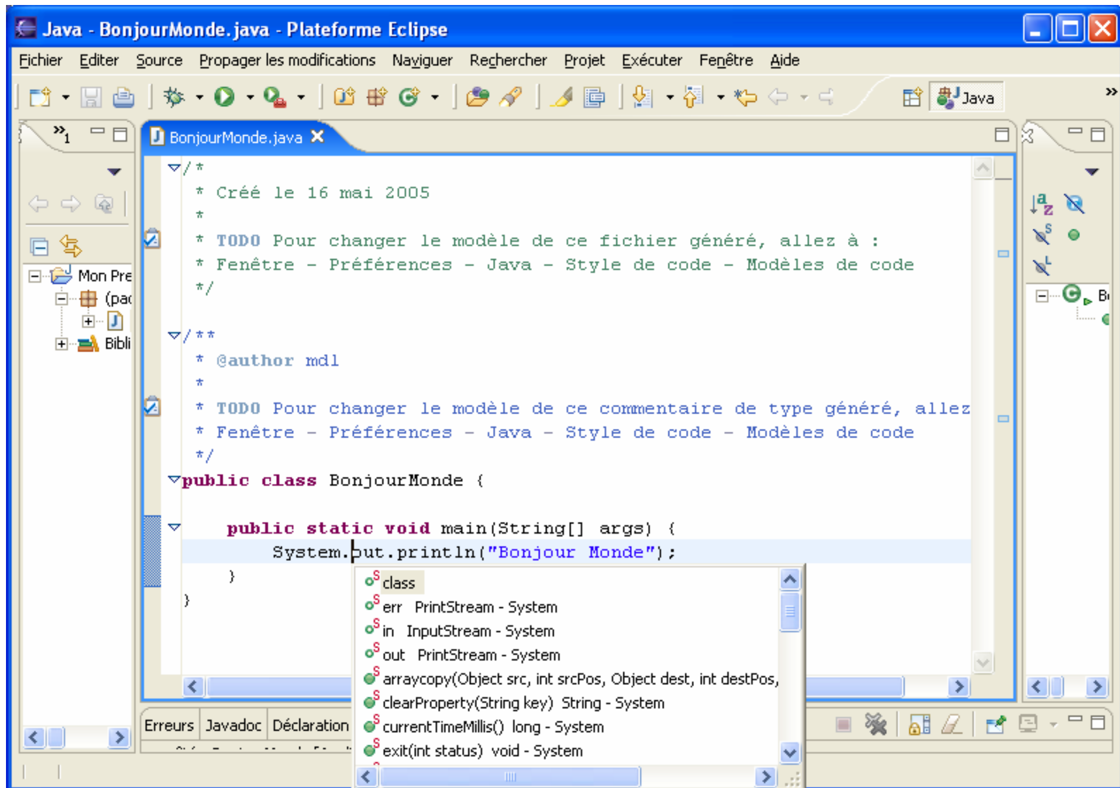
Le corps de notre méthode `main()` ne contient qu'une ligne :

```
System.out.println("Bonjour Monde");
```

Chaque instruction ou appel de méthode doit se terminer par un point-virgule `;`. La méthode `println()` sait comment afficher des données sur la *console* système (fenêtre de commande). Les noms de méthodes Java sont toujours suivis par des parenthèses. S'il n'y a rien entre les parenthèses, cela signifie que la méthode ne prend pas d'arguments.

L'expression `System.out` signifie que la variable `out` est définie à l'intérieur de la classe `System` fournie avec Java. Comment es-tu censé savoir qu'il existe quelque chose nommé `out` dans la classe `System` ?

Eclipse va t'y aider. Dès que tu as tapé le mot `System` suivi d'un point, Eclipse te propose automatiquement tout ce qui est disponible dans cette classe. Et à tout moment tu peux placer le curseur après le point et appuyer sur *Ctrl-Espace* pour afficher une fenêtre d'aide de ce genre :



L'expression `out.println()` indique que la *variable* `out` représente un objet et que ce "quelque chose nommé `out`" possède une méthode nommée `println()`. Le point entre la classe et le nom de la méthode signifie que la méthode existe à l'intérieur de la classe. Mettons que tu aies une classe `JeuPingPong` qui possède la méthode `sauverScore()`. Voilà comment tu peux *appeler* cette méthode pour Dave qui a gagné trois parties :

```
JeuPingPong.sauverScore("Dave", 3);
```

Encore une fois, les données entre les parenthèses sont appelées *arguments*, ou *paramètres*. Ces paramètres sont fournis à une méthode pour être utilisés dans un traitement, par exemple pour enregistrer des données sur le disque. La méthode `sauverScore()` a deux arguments – la chaîne de caractères `"Dave"` et le nombre `3`.

Eclipse rend amusante l'écriture des programmes Java. L'annexe B présente quelques trucs et astuces utiles pour booster ta programmation Java dans cet excellent environnement de développement.

Autres lectures

Site web Eclipse :

http://www.eclipse.org

Exercices

Modifie la classe `BonjourMonde` pour qu'elle affiche ton adresse, en utilisant plusieurs appels à la méthode `println()`.

Exercices pour les petits malins

Modifie la classe `BonjourMonde` pour afficher le mot *Bonjour* comme ceci :

Java - BonjourMonde.java - Plateforme Eclipse

Fichier Editer Source Propager les modifications Naviguer

Erreurs | Javadoc | Déclaration | Console

<arrêté> BonjourMonde [Application Java] D:\jdk1.5.0\bin\javaw

```
******    ***   **  ***   ****    ***
 *    *  *   *   *   *      *    *   *
 *    * *     *  **  *      *   *     *
 *    * *     *  **  *      *   *     *
 *****  *     *  * * *      *   *     *
 *    * *     *  *  **      *   *     *
 *    * *     *  *  **  *   *   *     *
 *    *  *   *   *   *  *   *    *   *
******    ***   ***  *   ***      ***
```

Chapitre 3. Animaux familiers et poissons – Classes Java

Les programmes Java sont constitués de *classes* qui représentent des objets du monde réel. Même si chaque personne écrit des programmes à sa façon, presque tout le monde est d'accord pour reconnaître que la meilleure façon de le faire est d'employer le style dit *orienté objet*. Cela signifie que les bons programmeurs commencent par décider quels objets doivent être inclus dans le programme et quelles classes Java vont les représenter. Ce n'est qu'une fois cette étape menée à bien qu'ils commencent à écrire du code Java.

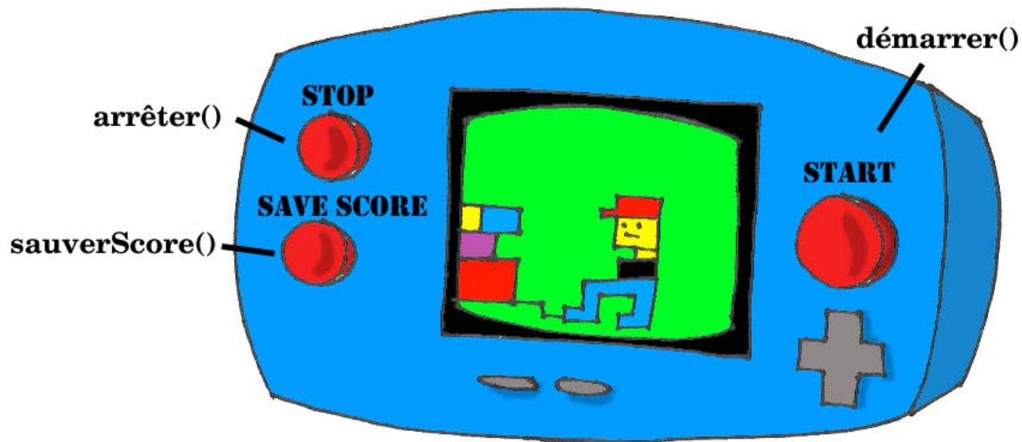
Classes et Objets

Les classes Java peuvent posséder des *méthodes* et des *attributs*.

Les méthodes définissent les actions qu'une classe peut effectuer.

Les attributs décrivent la classe.

Créons et discutons une classe nommée `JeuVidéo`. Cette classe peut avoir plusieurs méthodes, qui représentent les *actions que les objets de cette classe peuvent effectuer* : démarrer le jeu, l'arrêter, enregistrer le score, etc. Cette classe peut aussi posséder des attributs ou propriétés : prix, couleur de l'écran, nombre de télécommandes et autres.



En langage Java, cette classe pourrait ressembler à ceci :

```
class JeuVidéo {
  String couleur;
  int prix;

  void démarrer () {
  }
  void arrêter () {
  }
  void sauverScore(String nomJoueur, int score) {
  }
}
```

Notre classe `JeuVidéo` est sans doute similaire aux autres classes qui représentent des jeux vidéo – ceux-ci ont tous des écrans de taille et de couleur différentes, effectuent des actions similaires et coûtent de l'argent.

Nous pouvons être plus spécifiques et créer une autre classe Java, nommée `GameBoyAdvance`. Elle appartient aussi à la famille des jeux vidéo, mais possède certaines propriétés spécifiques au modèle GameBoy Advance, par exemple un type de cartouche.

```
class GameBoyAdvance {
  String typeCartouche;
  int largeurEcran;

  void démarrerJeu() {
  }
  void arrêterJeu() {
  }
}
```

Dans cet exemple, la classe `GameBoyAdvance` définit deux attributs – `typeCartouche` et `largeurEcran` – et deux méthodes – `démarrerJeu` et `arrêterJeu`. Mais, pour l'instant, ces méthodes ne peuvent effectuer aucune action, parce qu'il n'y a pas de code Java entre leurs accolades.

Après le mot *classe*, tu dois maintenant t'habituer à la nouvelle signification du mot *objet*.

La phrase "créer une instance d'un objet" signifie créer une copie de cet objet dans la mémoire de l'ordinateur en respectant la définition de sa classe.

Un plan de fabrication de la GameBoy Advance correspond à une console réelle de la même façon qu'une classe Java correspond à l'une de ses instances en mémoire. Le processus de fabrication des consoles réelles d'après ce plan est similaire au processus de création d'instances d'objets `GameBoyAdvance` en Java.

La plupart du temps, un programme ne pourra utiliser une classe Java qu'après qu'une instance en ait été créée. De même, les fabricants

créent des milliers de copies d'une console de jeu à partir d'une même description. Bien que ces copies représentent la même classe, leurs attributs peuvent avoir des *valeurs* différentes – certaines sont bleues alors que d'autres sont argentées, et ainsi de suite. En d'autres termes, un programme peut créer de *multiples instances* d'objets `GameBoyAdvance`.

Types de données

Les *variables* Java peuvent représenter les attributs d'une classe, les arguments d'une méthode ou être utilisées à l'intérieur d'une méthode pour stocker temporairement certaines données. Les variables doivent être déclarées avant de pouvoir être utilisées.

Te souviens-tu d'équations telles que `y = x + 2` ? En Java, tu dois déclarer les variables `x` et `y` en précisant qu'elles appartiennent à un *type de données* numérique tel que `int` ou `double` :

```
int x;
int y;
```

Les deux lignes suivantes montrent comment tu peux affecter une *valeur* à ces variables. Si ton programme donne la valeur 5 à la variable `x`, la variable `y` sera égale à 7 :

```
x = 5;
y = x + 2;
```

En Java, tu peux aussi changer la valeur d'une variable d'une façon assez inhabituelle. Les deux lignes suivantes modifient la valeur de la variable `y`, qui passe de 5 à 6 :

```
int y = 5;
y++;
```

Bien qu'il y ait deux signes plus, Java va tout de même incrémenter la valeur de la variable `y` de 1.

Après le fragment de code suivant, la valeur de la variable `monScore` est aussi 6 :

```
int monScore = 5;
monScore = monScore + 1;
```

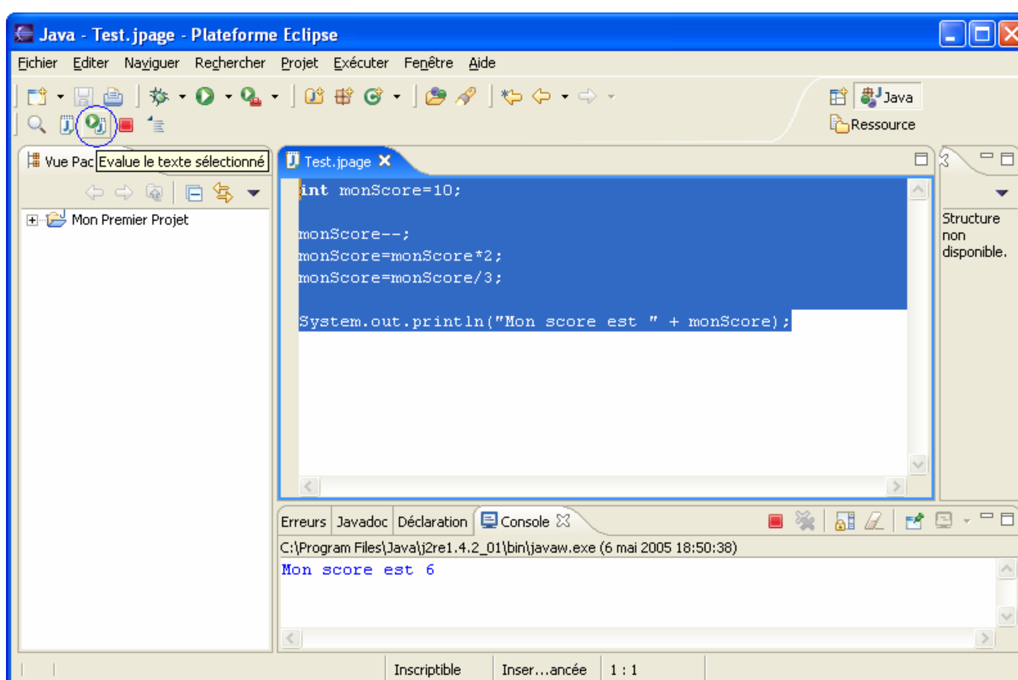
On peut utiliser de la même manière la multiplication, la division et la soustraction. Regarde l'extrait de code suivant :

```
int monScore = 10;

monScore--;
monScore = monScore * 2;
```

```
monScore = monScore / 3;

System.out.println("Mon score est " + monScore);
```

Qu'imprime ce code ? Eclipse a une fonctionnalité bien pratique appelée page de testeur de code, qui permet de tester rapidement n'importe quel bout de code (comme le précédent) sans même créer une classe. Sélectionne le menu *Fichier*, *Nouveau*, *Autre…*, puis *Java, Exécution/Débogage Java, Page de testeur de code* et entre le mot `Test` comme nom de ta page.

Entre maintenant dans la page ces cinq lignes de code utilisant `monScore`, sélectionne-les et clique sur la petite flèche verte accompagnée d'un J dans la barre d'outils.



Pour voir le résultat des calculs, clique simplement sur l'onglet *Console* en bas de l'écran :

```
Mon score est 6
```

Dans cet exemple, l'argument de la méthode `println()` a été fabriqué en accolant deux morceaux – le texte "Mon score est " et la valeur de la variable `monScore`, c'est-à-dire 6. On appelle *concaténation* la création d'une chaîne de caractères (`String`) à partir de morceaux. Bien que `monScore` soit un nombre, Java est suffisamment malin pour convertir cette variable en `String` puis l'accoler au texte "Mon score est ".

Voyons d'autres façons de modifier les valeurs des variables :

`monScore = monScore * 2;` est équivalent à `monScore *= 2;`

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

`monScore = monScore + 2;` est équivalent à `monScore += 2;`
`monScore = monScore - 2;` est équivalent à `monScore -= 2;`
`monScore = monScore / 2;` est équivalent à `monScore /= 2;`

Il y a huit types de données simples, ou *primaires* en Java, et tu dois décider lequel utiliser en fonction du type et de la taille des données que tu as l'intention de stocker dans chaque variable :

- ✓ Quatre types de données pour les valeurs entières – `byte`, `short`, `int` et `long`.
- ✓ Deux types de données pour les valeurs décimales – `float` et `double`.
- ✓ Un type de données qui permet de stocker un caractère isolé – `char`.
- ✓ Un type de données *logique* nommé `boolean` qui autorise seulement deux valeurs : `true` (vrai) ou `false` (faux).

On peut affecter une valeur initiale à une variable lors de sa déclaration. On parle alors d'*initialisation de la variable* :

```
char niveau = 'E';
int chaises = 12;
boolean sonActif = false;
double revenuNational = 23863494965745.78;
float prixJeu = 12.50f;
long totalVoitures = 4637283648392l;
```

Dans les deux dernières lignes, `f` signifie `float` et `l` signifie `long`.

Si tu n'initialises pas les variables, Java le fera pour toi en donnant la valeur 0 à chaque variable numérique, la valeur `false` aux variables de type `boolean` et le code spécial '\u0000' aux variables de type `char`.

Il y a aussi le mot-clé spécial `final` ; s'il est utilisé dans la déclaration d'une variable, on ne peut affecter de valeur à cette variable qu'une fois, cette valeur ne pouvant plus être modifiée par la suite. Dans certains langages, les variables invariantes sont appelées *constantes*. En Java les noms des variables invariantes sont généralement écrits en lettres majuscules :

```
final String CAPITALE_ETAT = "Washington";
```

Outre les types de données primaires, on peut aussi utiliser les classes Java pour déclarer des variables. A chaque type de données primaire correspond une classe *enveloppe*, par exemple `Integer`, `Double`,

`Boolean`, etc. Ces classes possèdent des méthodes utiles pour convertir les données d'un type à un autre.

Alors que le type de données `char` est utilisé pour stocker un caractère isolé, Java a aussi une classe `String` permettant de manipuler un texte plus long, par exemple :

```
String nom = "Dupont";
```

En Java, les noms de variables ne peuvent pas commencer par un chiffre ni contenir d'espaces.

Un *bit* est le plus petit bout de donnée que l'on puisse stocker en mémoire. Il contient soit 1, soit 0.

Un octet (`byte`) est composé de huit bits.

En Java, un `char` occupe deux octets en mémoire.

En Java, un `int` ou un `float` occupe quatre octets.

Les variables de type `long` ou `double` utilisent huit octets.

Les types de données numériques qui utilisent plus d'octets peuvent stocker de plus grands nombres.

1 kilooctet (KO) correspond à 1024 octets.

1 mégaoctet (MO) correspond à 1024 kilooctets.

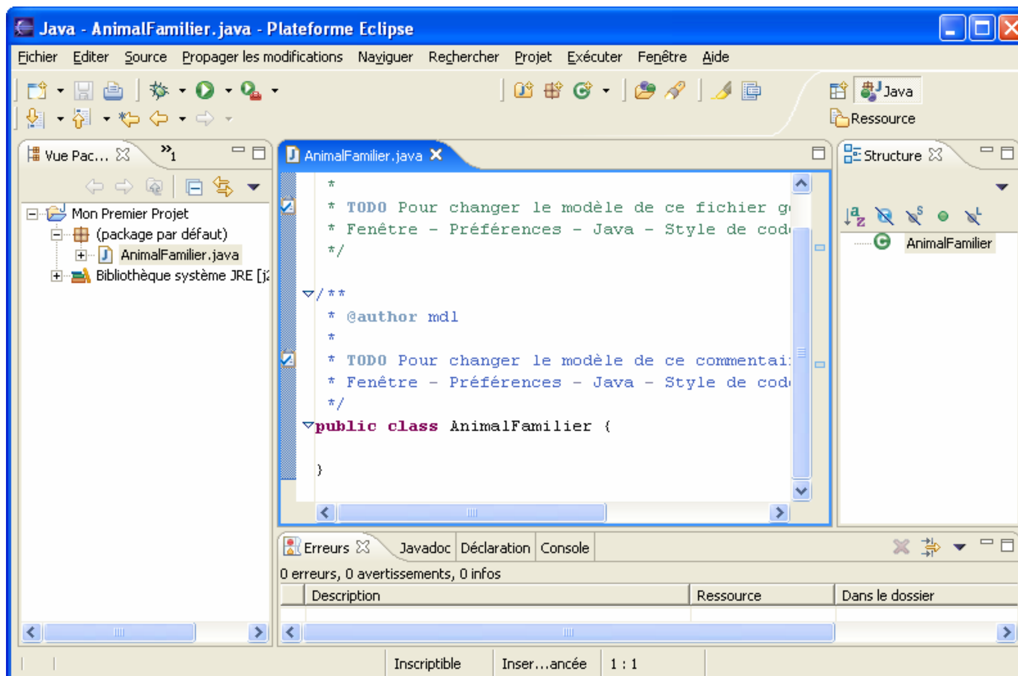
1 gigaoctet (GO) correspond à 1024 mégaoctets.

Création d'un animal familier

Nous allons concevoir et créer une classe `AnimalFamilier`. Tout d'abord, nous devons décider quelles actions notre animal familier sera capable d'effectuer. Que dirais-tu de manger, dormir et dire ? Nous allons programmer ces actions dans les méthodes de la classe `AnimalFamilier`. Nous allons aussi donner à notre animal les attributs suivants : `âge`, `taille`, `poids` et `couleur`.

Commençons par créer une nouvelle classe Java nommée `AnimalFamilier` dans `Mon Premier Projet` comme décrit dans le chapitre 2, mais sans cocher la case de création de la méthode `main()`.

Ton écran devrait ressembler à ceci :



Nous sommes maintenant prêts à déclarer les attributs et les méthodes de la classe `AnimalFamilier`. Le corps des méthodes et des classes Java est délimité par des accolades. A chaque accolade ouvrante doit correspondre une accolade fermante :

```
class AnimalFamilier {
}
```

Pour déclarer les variables constituant les attributs de la classe, nous devons choisir leur type. Prenons par exemple le type `int` pour l'âge, `float` pour la taille et le poids et `String` pour la couleur.

```
class AnimalFamilier {
  int âge;
  float poids;
  float taille;
  String couleur;
}
```

L'étape suivante consiste à ajouter des méthodes à notre classe. Avant de déclarer une méthode, il faut décider si elle prend des arguments et si elle retourne une valeur :

- ✓ La méthode `dormir()` ne fera qu'afficher le message "Bonne nuit, à demain" – elle n'a pas besoin d'arguments et ne retourne aucune valeur.
- ✓ Il en est de même pour la méthode `manger()`. Elle ne fera qu'afficher le message "J'ai si faim… Donne-moi un biscuit !".
- ✓ La méthode `dire()` affichera aussi un message, mais l'animal pourra "dire" (afficher) le mot ou la phrase que nous lui fournirons. Nous passerons ce mot à la méthode `dire()` comme un *argument de méthode*. La méthode construira une phrase en utilisant cet argument et la retournera au programme appelant.

La nouvelle version de la classe `AnimalFamilier` ressemble à ceci :

```
public class AnimalFamilier {
  int âge;
  float poids;
  float taille;
  String couleur;

  public void dormir() {
    System.out.println("Bonne nuit, à demain");
  }

  public void manger() {
    System.out.println(
            "J'ai si faim… Donne-moi un biscuit !");
  }

  public String dire(String unMot) {
    String réponseAnimal = "OK !! OK !! " + unMot;
    return réponseAnimal;
  }
}
```

Cette classe représente une sympathique créature du monde réel :

Voyons maintenant la signature de la méthode `dormir()`:

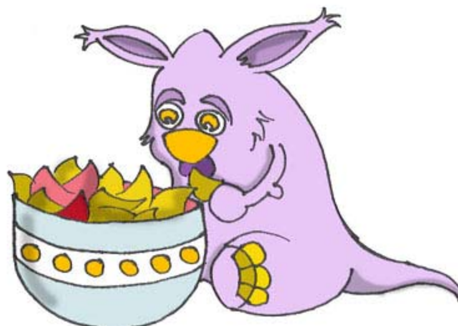
```
public void dormir()
```

Elle nous indique que cette méthode peut être appelée depuis n'importe quelle autre classe Java (`public`) et qu'elle ne retourne aucune donnée (`void`). Les parenthèses vides signifient que cette méthode ne prend pas d'argument, parce qu'elle n'a besoin d'aucune donnée du monde extérieur – elle affiche toujours le même texte.

La signature de la méthode `dire()` est celle-ci :

```
public String dire(String unMot)
```

Cette méthode peut aussi être appelée depuis n'importe quelle autre classe Java, mais elle doit retourner un texte, ce qu'indique le mot-clé `String` devant le nom de la méthode. Par ailleurs, elle attend une donnée textuelle de l'extérieur, d'où l'argument `String unMot`.



Comment décider si une méthode doit retourner une valeur ou pas ? Si une méthode manipule des données et doit fournir le résultat de ces manipulations à une classe appelante, elle doit retourner une valeur. Tu vas me dire que notre classe `AnimalFamilier` n'a aucune classe appelante ! C'est exact, alors créons-en une, que nous appellerons `MaîtreAnimal`. Cette classe aura une méthode `main()` contenant le code nécessaire pour communiquer avec la classe `AnimalFamilier`. Crée simplement une autre classe nommée `MaîtreAnimal`, mais cette fois sélectionne dans Eclipse l'option qui crée la méthode `main()`. Rappelle-toi, sans cette méthode il est impossible d'exécuter cette classe en tant que programme. Modifie le code généré par Eclipse pour obtenir ceci :

```
public class MaîtreAnimal {

  public static void main(String[] args) {

    String réactionAnimal;

    AnimalFamilier monAnimal = new AnimalFamilier();

    monAnimal.manger();
    réactionAnimal = monAnimal.dire("Cui !! Cui !!");
    System.out.println(réactionAnimal);

    monAnimal.dormir();

  }
}
```

N'oublie pas d'appuyer sur *Ctrl-S* pour enregistrer et compiler cette classe !

Pour exécuter la classe `MaîtreAnimal`, sélectionne le menu Eclipse *Exécuter*, *Exécuter…*, *Créer* et tape le nom de la classe principale : `MaîtreAnimal`. Appuie sur le bouton *Exécuter* et le programme affichera le texte suivant :

```
J'ai si faim… Donne-moi un biscuit !
OK !! OK !! Cui !! Cui !!
Bonne nuit, à demain
```

La classe `MaîtreAnimal` est la classe *appelante* ; elle commence par créer une *instance* de l'objet `AnimalFamilier`. Elle déclare une variable `monAnimal` et utilise l'opérateur Java `new` :

```
AnimalFamilier monAnimal = new AnimalFamilier();
```

Cette ligne déclare une variable du type `AnimalFamilier` (c'est exact, tu peux considérer chaque classe que tu crées comme un nouveau type Java). Maintenant, la variable `monAnimal` sait où a été créée l'instance d'`AnimalFamilier` dans la mémoire de l'ordinateur,

et tu peux utiliser cette variable pour appeler n'importe laquelle des méthodes de la classe `AnimalFamilier`, par exemple :

```
monAnimal.dormir();
```

Si une méthode retourne une valeur, tu dois l'appeler d'une autre façon. Déclare une variable du même type que la valeur retournée par la méthode. Tu peux maintenant appeler cette méthode :

```
String réactionAnimal;

réactionAnimal = monAnimal.dire("Cui !! Cui !!");
```

A ce stade, la valeur retournée est stockée dans la variable `réactionAnimal` et si tu veux voir ce qu'elle contient, ne te gêne pas :

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

Héritage – un Poisson est aussi un AnimalFamilier

Notre classe `AnimalFamilier` va nous aider à découvrir un autre concept important de Java, appelé *héritage*. Dans la vie réelle, chaque personne hérite des caractéristiques de l'un ou l'autre de ses parents. De la même façon, dans le monde Java, tu peux, à partir d'une classe, en créer une nouvelle.

La classe `AnimalFamilier` possède un comportement et des attributs partagés par de nombreux animaux familiers – ils mangent, dorment, certains d'entre eux émettent des bruits, leurs peaux peuvent être de différentes couleurs, etc. D'un autre côté, les animaux familiers sont différents les uns des autres – les chiens aboient, les poissons nagent et sont silencieux, les perroquets parlent mieux que les chiens. Mais tous mangent, dorment, ont un poids et une taille. C'est pourquoi il est plus facile de créer une classe `Poisson` qui *héritera* certains comportements et attributs communs de la classe `AnimalFamilier`, que de créer `Chien`, `Perroquet` ou `Poisson` à partir de rien à chaque fois.

Le mot-clé spécial `extends` est là pour ça :

```
class Poisson extends AnimalFamilier {

}
```

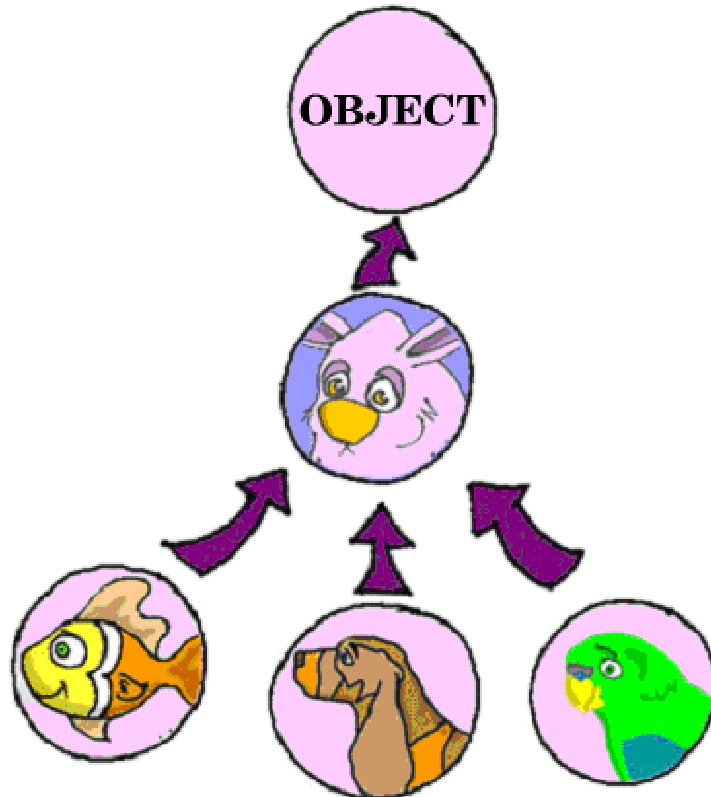
On peut dire que notre `Poisson` est une *sous-classe (subclass)* de la classe `AnimalFamilier` et que la classe `AnimalFamilier` est une *superclasse (superclass)* de la classe `Poisson`. Autrement dit, on utilise la classe `AnimalFamilier` comme un modèle pour créer la classe `Poisson`.

Même si tu te contentes de laisser la classe `Poisson` telle qu'elle est maintenant, tu peux toujours utiliser chacun des attributs et méthodes hérités de la classe `AnimalFamilier`. Regarde :

```
Poisson monPetitPoisson = new Poisson();
monPetitPoisson.dormir();
```

Même si nous n'avons pas encore déclaré de méthode dans la classe `Poisson`, nous avons le droit d'appeler la méthode `dormir()` de sa superclasse !

Dans Eclipse, la création de sous-classes est une partie de plaisir ! Sélectionne le menu *Fichier, Nouveau, Classe* et tape `Poisson` comme nom de la classe. Remplace la valeur `java.lang.Object` par le mot `AnimalFamilier` dans le champ *Superclasse*.



N'oublions pas cependant que nous sommes en train de créer une sous-classe d'`AnimalFamilier` pour ajouter certaines propriétés que seuls les poissons possèdent et réutiliser une partie du code que nous avons écrit pour un animal familier en général.

Il est temps de te révéler un secret – toutes les classes Java héritent de la super superclasse `Object`, que tu utilises le mot `extends` ou pas.

Mais une classe Java ne peut pas avoir deux parents distincts. Si c'était la même chose avec les gens, les enfants ne seraient pas des sous-classes de leurs parents, mais tous les garçons seraient des descendants d'Adam et toutes les filles des descendantes d'Eve ☺.

Tous les animaux ne sont pas capables de plonger, mais il est certain que les poissons le peuvent. Ajoutons maintenant une nouvelle méthode `plonger()` à la classe `Poisson`.

```
public class Poisson extends AnimalFamilier {

  int profondeurCourante = 0;

  public int plonger(int combienDePlus){
    profondeurCourante = profondeurCourante +
                                    combienDePlus;
    System.out.println("Plongée de " +
            combienDePlus + " mètres");
    System.out.println("Je suis à " +
            profondeurCourante +
            " mètres sous le niveau de la mer");
    return profondeurCourante;
  }
}
```

La méthode `plonger()` a un *argument* `combienDePlus` qui indique au poisson de combien il doit plonger. Nous avons aussi déclaré la variable de classe `profondeurCourante` qui enregistrera la nouvelle profondeur courante à chaque fois que tu appelleras la méthode `plonger()`. Cette méthode renvoie la valeur courante de la variable `profondeurCourante` à la classe appelante.

Crée maintenant une autre classe nommée `MaîtrePoisson` qui ressemble à ceci :

```
public class MaîtrePoisson {

  public static void main(String[] args) {

    Poisson monPoisson = new Poisson();

    monPoisson.plonger(2);
    monPoisson.plonger(3);

    monPoisson.dormir();
  }
}
```

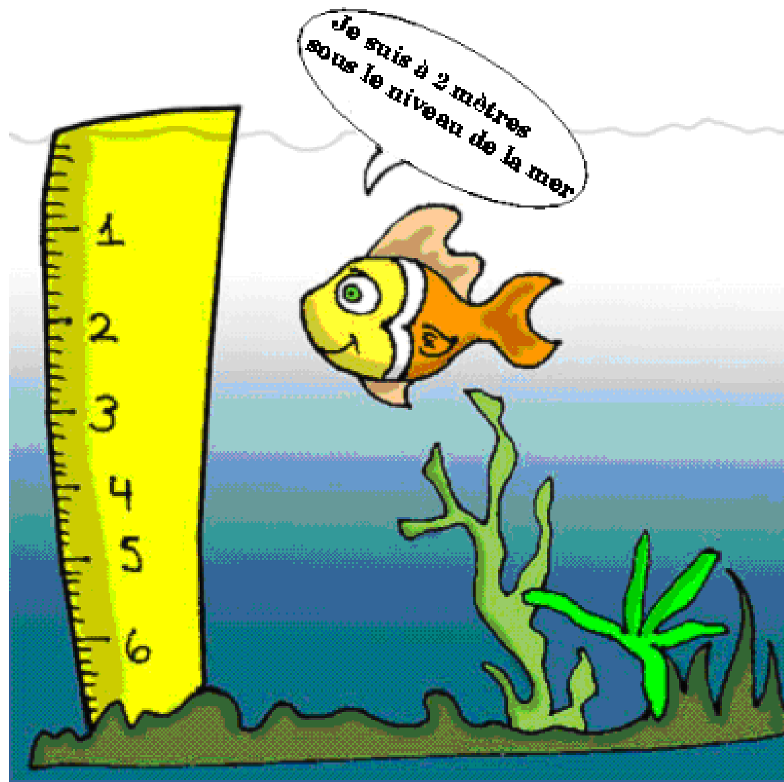
La méthode `main()` instancie l'objet `Poisson` et appelle sa méthode `plonger()` deux fois, avec des arguments différents. Ensuite, elle appelle la méthode `dormir()`. Quand tu exécutes le programme `MaîtrePoisson`, il affiche les messages suivants :

```
Plongée de 2 mètres
Je suis à 2 mètres sous le niveau de la mer
Plongée de 3 mètres
Je suis à 5 mètres sous le niveau de la mer
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com


```
Bonne nuit, à demain
```

As-tu noté que, outre la méthode définie dans la classe `Poisson`, le `MaîtrePoisson` appelle aussi des méthodes de sa superclasse `AnimalFamilier` ? C'est tout l'intérêt de l'héritage – tu n'as pas besoin de copier et coller le code de la classe `AnimalFamilier` – il suffit d'utiliser le mot `extends` et la classe `Poisson` peut utiliser les méthodes d'`AnimalFamilier` !



Encore une chose : bien que la méthode `plonger()` renvoie la valeur de `profondeurCourante`, notre `MaîtrePoisson` ne l'utilise pas. Très bien, notre `MaîtrePoisson` n'a pas besoin de cette valeur ; mais peut-être y a-t-il d'autres classes qui utilisent `Poisson`, qui pourraient trouver cette valeur utile. Par exemple, imagine une classe `ContrôleurTraficPoissons` qui doit connaître les positions des autres poissons dans la mer avant d'autoriser un poisson à plonger, pour éviter les accidents ☺.

Surcharge d'une méthode

Comme tu le sais, les poissons ne parlent pas (du moins ne parlent-ils pas à voix haute). Mais notre classe `Poisson` hérite de la classe `AnimalFamilier` qui possède la méthode `dire()`. Ceci signifie que rien ne t'empêche d'écrire quelque chose comme ça :

```
monPoisson.dire("Un poisson qui parle !");
```

Eh bien, notre poisson a commencé à parler… Pour éviter que cela se produise, il faut que la classe `Poisson` *surcharge (override)* la méthode `dire()` de la classe `AnimalFamilier`. Voilà comment ça marche : si tu déclares une méthode avec exactement la même signature dans la sous-classe que dans la superclasse, la méthode de la sous-classe sera utilisée à la place de celle de la superclasse. Ajoutons la méthode `dire()` à la classe `Poisson`.

```
public String dire(String unMot) {
  return "Ne sais-tu pas que les poissons ne parlent pas ?";
}
```

Ajoute maintenant l'appel suivant dans la méthode `main()` de la classe `MaîtrePoisson` :

```
String réactionPoisson;
réactionPoisson = monPoisson.dire("Salut");
System.out.println(réactionPoisson);
```

Exécute le programme et il affichera :

```
Ne sais-tu pas que les poissons ne parlent pas ?
```

Cela prouve que la méthode `dire()` de la classe `AnimalFamilier` a été surchargée, ou, autrement dit, supprimée.

> Si la signature d'une méthode inclut le mot-clé `final`, elle ne peut pas être surchargée. Par exemple:
>
> `final public void dormir(){…}`

Waouh ! Nous en avons appris, des choses, dans ce chapitre. Que dirais-tu d'une petite pause ?

Autres lectures

1. Types de données Java :
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

2. Héritage :
http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html

Exercices

1. Crée une nouvelle classe `Voiture` possédant les méthodes suivantes :

```
public void démarrer()
public void arrêter()
public int rouler(int durée)
```

La méthode `rouler()` doit renvoyer la distance totale parcourue par la voiture pendant un temps donné. Utilise la formule suivante pour calculer la distance :

```
distance = durée * 60;
```

2. Ecris une autre classe, `PropriétaireVoiture`, qui crée une instance de l'objet `Voiture` et appelle ses méthodes. Le résultat de chaque appel de méthode doit être affiché à l'aide de `System.out.println()`.

Exercices pour les petits malins

Crée une sous-classe de `Voiture` nommée `VoitureJamesBond` et surcharge la méthode `rouler()`. Utilise la formule suivante pour calculer la distance :

```
distance = durée*180;
```

Sois créatif, affiche des messages amusants !

Chapitre 4. Briques de base Java

Tu peux ajouter n'importe quels commentaires textuels à ton programme pour expliquer à quoi servent une ligne, une méthode ou une classe particulière. Tout d'abord, il arrive qu'on oublie pourquoi on a écrit un programme de cette façon. Ensuite, les commentaires aident les autres programmeurs à comprendre ton code.

Commentaires de programme

Il y a trois types de commentaires différents :

- Si tes commentaires tiennent sur une ligne, commence-les par deux barres obliques (*slashs*) :

```
// Cette méthode calcule la distance
```

- Les commentaires s'étendant sur plusieurs lignes doivent être entourés par les symboles `/*` et `*/`, par exemple :

```
/* les 3 lignes suivantes enregistrent
   la position courante du Poisson.
*/
```

- Java est fourni avec un programme spécial, `javadoc`, qui peut extraire certains des commentaires de tes programmes dans un fichier d'aide séparé. Ce fichier peut être utilisé comme *documentation technique* de tes programmes. Ces commentaires sont entourés par les symboles `/**` et `*/`. Seuls les commentaires les plus importants, comme la description d'une classe ou d'une méthode, devraient être repérés ainsi.

```
/** Cette méthode calcule la remise en fonction du prix.
   Si le prix est supérieur à 100 €, la remise est de
   20%, sinon, elle est seulement de 10%.
*/
```

A partir de maintenant, j'ajouterai des commentaires aux exemples de code pour que tu comprennes mieux comment et où les utiliser.

Prises de décisions à l'aide de la clause `if`

Nous prenons sans arrêt des décisions dans la vie : *Si elle me dit ceci, je lui répondrai cela ; sinon, je ferai autre chose.* Java possède une clause qui évalue une *expression* donnée pour savoir si sa valeur est `true` ou `false`.

Selon la valeur de cette expression, l'exécution de ton programme prend une direction ou une autre ; seule la portion de code correspondante est exécutée.

Par exemple, si l'expression *Ai-je envie d'aller voir grand-mère ?* retourne `true`, tourne à gauche, sinon, tourne à droite.

Si l'expression *retourne* `true`, Java exécute le code compris entre les deux premières accolades, sinon, il va directement au code suivant la clause `else`. Par exemple, si un prix est supérieur à cent euros, donner une remise de 20%, sinon, ne donner que 10%.

```
// Les biens les plus chers donnent une remise de 20%
if (prix > 100) {
  prix = prix * 0.8;
  System.out.println("Tu as 20% de remise");
}
else {
  prix = prix * 0.9;
  System.out.println("Tu as 10% de remise");
}
```

Modifions la méthode `plonger()` de la classe `Poisson` pour que notre poisson ne puisse pas plonger plus profond que 100 mètres :

```
public class Poisson extends AnimalFamilier {
  int profondeurCourante = 0;

  public int plonger(int combienDePlus) {
    profondeurCourante = profondeurCourante +
                                combienDePlus;
    if (profondeurCourante > 100) {
      System.out.println("Je suis un petit "
        + " poisson et je ne peux pas plonger"
        + " plus profond que 100 mètres");
      profondeurCourante = profondeurCourante
                                   - combienDePlus;
    }
    else {
      System.out.println("Plongée de " + combienDePlus
                                   + " mètres");
      System.out.println("Je suis à " + profondeurCourante
                    + " mètres sous le niveau de la mer");
    }
    return profondeurCourante;
  }

  public String dire(String unMot) {
    return "Ne sais-tu pas que les poissons ne"
                          + " parlent pas ?";
  }
}
```

Effectuons ensuite une petite modification de `MaîtrePoisson` – laissons-le essayer de faire plonger notre poisson très profond :

```
public class MaîtrePoisson {

  public static void main(String[] args) {

    Poisson monPoisson = new Poisson();

    // Essayons de plonger plus profond que 100 mètres
    monPoisson.plonger(2);
    monPoisson.plonger(97);
    monPoisson.plonger(3);

    monPoisson.dormir();
  }
}
```

Exécute ce programme et il affichera ceci :

```
Plongée de 2 mètres
Je suis à 2 mètres sous le niveau de la mer
Plongée de 97 mètres
Je suis à 99 mètres sous le niveau de la mer
Je suis un petit  poisson et je ne peux pas plonger plus profond
que 100 mètres
Bonne nuit, à demain
```

Opérateurs logiques

Parfois, pour prendre une décision, il peut être nécessaire de vérifier plus d'une expression conditionnelle. Par exemple, si le nom d'un état est "Texas" ou "Californie", ajouter la taxe d'état au prix de chaque article du magasin. C'est un cas d'emploi du *ou logique* (*logical or*) – soit "Texas" soit "Californie". En Java, un *ou* logique est représenté par une ou deux barres verticales. Ca fonctionne comme ceci : si l'une quelconque des conditions est vraie, le résultat de l'expression dans son ensemble vaut `true`. Dans les exemples suivants, j'utilise une variable de type `String`. Cette classe Java possède une méthode `equals()` que j'utilise pour comparer la valeur de la variable état avec "Texas" ou "Californie" :

```
if (état.equals("Texas") | état.equals("Californie"))
```

Tu peux aussi écrire ceci en utilisant deux barres verticales :

```
if (état.equals("Texas") || état.equals("Californie"))
```

La différence entre les deux est que, si tu utilises deux barres et que la première expression est vraie, la seconde expression ne sera même pas évaluée. Si tu ne mets qu'une seule barre, Java évalue les deux expressions.

Le *et logique* (*logical and*) est représenté par un ou deux "et commercial" (`&&`) et l'expression dans son ensemble vaut `true` si chacune de ses parties vaut `true`. Par exemple, imputer les taxes commerciales uniquement dans le cas où l'état est "New York" et où le prix est supérieur à 110. Les deux conditions doivent être vraies *en même temps* :

```
if (état.equals("New York") && prix > 110)
```

ou

```
if (état.equals("New York") & prix > 110)
```

Si tu utilises un double "et commercial" et que la première expression vaut `false`, la seconde ne sera même pas évaluée, car l'expression entière vaudra `false` de toutes façons. Avec un simple "et commercial", les deux expressions seront évaluées.

Le *non logique* (*logical not*) est représenté par le point d'exclamation et donne à l'expression le sens opposé. Par exemple, si tu veux effectuer certaines actions uniquement si l'état n'est pas "New York", utilise cette syntaxe :

```
if (!état.equals("New York"))
```

Voici un autre exemple – les deux expressions suivantes produiront le même résultat :

```
if (prix < 50)

if (!(prix >= 50))
```

Le *non logique* est ici appliqué à l'expression entre parenthèses.

Opérateur conditionnel

Il existe une autre forme de la clause `if` : *l'opérateur conditionnel*. On l'utilise pour affecter une valeur à une variable en fonction d'une expression terminée par un point d'interrogation. Si l'expression est vraie, la valeur suivant le point d'interrogation est utilisée ; sinon, la valeur suivant les deux points est affectée à la variable située à gauche du signe égal :

```
remise = prix > 50 ? 10 : 5;
```

Si le prix est supérieur à `50`, la variable `remise` prend la valeur `10` ; sinon, elle vaut `5`. C'est juste un raccourci pour exprimer une clause `if` normale :

```
if (prix > 50) {
  remise = 10;
}
else {
  remise = 5;
}
```

Utilisation de `else if`

Tu peux construire des clauses `if` plus complexes avec plusieurs blocs `else if`. Nous allons maintenant créer la classe `BulletinAppréciation`. Cette classe doit avoir la méthode `main()` ainsi qu'une méthode acceptant un argument – la note du devoir. En fonction du nombre, cette méthode affichera ton niveau sous la forme I (Insuffisant), P (Passable), A (Assez bien), B (Bien), T (Très bien) ou E (Excellent). Nous la nommerons `convertirNiveaux()`.

```
public class BulletinAppréciation {

/**
Cette méthode attend un argument entier - la note du devoir
- et retourne une mention, I, P, A, B, T ou E, en fonction
de sa valeur.
*/
  public char convertirNiveaux(int noteDevoir) {
    char niveau;

    if (noteDevoir >= 18) {
      niveau = 'E';
    }
    else if (noteDevoir >= 16 && noteDevoir < 18) {
      niveau = 'T';
    }
    else if (noteDevoir >= 14 && noteDevoir < 16) {
      niveau = 'B';
    }
    else if (noteDevoir >= 12 && noteDevoir < 14) {
      niveau = 'A';
    }
    else if (noteDevoir >= 10 && noteDevoir < 12) {
      niveau = 'P';
    }
    else {
      niveau = 'I';
    }
    return niveau;
  }

  public static void main(String[] args) {

    BulletinAppréciation convertisseur =
                                  new BulletinAppréciation();

    char tonNiveau = convertisseur.convertirNiveaux(17);
    System.out.println("Ton premier niveau est " +
                                          tonNiveau);

    tonNiveau = convertisseur.convertirNiveaux(15);
    System.out.println("Ton second niveau est " +
                                          tonNiveau);
  }
}
```

Outre l'utilisation de la condition `else if`, cet exemple te montre aussi comment utiliser des variables de type `char`. Tu peux aussi voir que tu peux utiliser l'opérateur `&&` pour vérifier si un nombre appartient à une plage de valeurs donnée. Tu ne peux pas écrire simplement `if résultatTest entre 16 et 18`, mais en Java nous écrivons que `résultatTest` doit être à la fois supérieur ou égal à `16` *et* inférieur à `18` :

```
résultatTest >= 16 && résultatTest < 18
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Réfléchis à la raison pour laquelle nous ne pouvons pas utiliser l'opérateur | | ici.

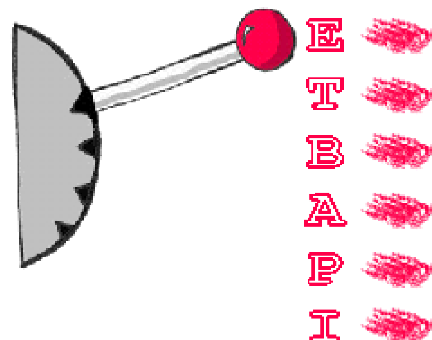
Prises de décisions à l'aide de la clause `switch`

La clause `switch` peut parfois être utilisée comme alternative à `if`. L'expression après le mot-clé `switch` est évaluée et le programme va directement au `case` (cas) correspondant :

```
public static void main(String[] args) {

  BulletinAppréciation convertisseur =
                              new BulletinAppréciation();
  char tonNiveau = convertisseur.convertirNiveaux(15);

  switch (tonNiveau) {

    case 'E':
       System.out.println("Excellent travail !");
       break;
    case 'T':
       System.out.println("Très bon travail !");
       break;
    case 'B':
       System.out.println("Bon travail !");
       break;
    case 'A':
       System.out.println("Tu peux mieux faire !");
       break;
    case 'P':
       System.out.println("Tu dois travailler plus !");
       break;
    case 'I':
       System.out.println("Change d'attitude !");
       break;
  }
}
```

N'oublie pas le mot-clé `break` à la fin de chaque cas – il faut que le code saute hors de la clause `switch`. Sans les instructions `break`, ce code imprimerait, par exemple, les quatre dernières lignes si la variable `tonNiveau` avait la valeur `'B'`.

La clause `switch` Java a une limitation – l'expression à évaluer doit être de l'un de ces types :




```
char
int
byte
short
```

Quelle est la durée de vie des variables ?

La classe `BulletinAppréciation` déclare la variable `niveau` dans la méthode `convertirNiveaux()`. Si tu déclares une variable à l'intérieur d'une méthode, cette variable est dite *locale*. Cela signifie que cette variable n'est accessible que pour le code *à l'intérieur de cette méthode*. Quand la méthode se termine, la variable est automatiquement effacée de la mémoire.

Les programmeurs utilisent aussi le mot *portée* (*scope*) pour préciser combien de temps une variable vivra. Par exemple, tu peux dire que les variables déclarées à l'intérieur d'une méthode ont une portée locale.

Si une variable doit être réutilisée par plusieurs appels de méthode, ou si elle doit être visible par plus d'une méthode d'une classe, il est préférable de la déclarer en dehors de toute méthode. Dans la classe `Poisson`, `profondeurCourante` est une *variable membre* (*member*). De telles variables sont "vivantes" tant que l'instance de l'objet `Poisson` existe en mémoire ; c'est pourquoi on les appelle des *variables d'instance*. Elles peuvent être partagées et réutilisées par toutes les méthodes de la classe et, dans certains cas, elles peuvent même être visibles depuis des classes extérieures. Par exemple, dans nos classes, l'instruction `System.out.println()` utilise la variable de classe `out` déclarée dans la classe `System`.

Attends une minute ! Pouvons-nous utiliser une variable membre de la classe `System` sans même avoir créé une instance de cette classe ? Oui, nous le pouvons, si cette variable a été déclarée avec le mot-clé `static`. Si la déclaration d'une variable membre ou d'une méthode commence par `static`, il n'est pas nécessaire de créer une instance de la classe pour l'utiliser. Les membres statiques d'une classe sont utilisés pour stocker les valeurs qui sont identiques pour toutes les instances d'une classe.

Par exemple, la méthode `convertirNiveaux()` peut être déclarée comme `static` dans la classe `BulletinAppréciation`, car son code n'utilise pas de variables membres pour lire ou stocker des valeurs spécifiques à une instance particulière de la classe. Voici comment on appelle une méthode statique :

```
char tonNiveau = BulletinAppréciation.convertirNiveaux(15);
```

Voici un autre exemple : il y a en Java une classe `Math` qui contient plusieurs douzaines de méthodes mathématiques telles que `sqrt()` (racine carrée), `sin()`, `abs()` et autres. Toutes ces méthodes sont statiques et tu n'as pas besoin de créer une instance de la classe `Math` pour les appeler. Par exemple :

```
double racineCarrée = Math.sqrt(4.0);
```

Méthodes spéciales : constructeurs

Java utilise l'opérateur `new` (nouveau) pour créer des instances d'objets en mémoire. Par exemple :

```
Poisson monPoisson = new Poisson();
```

Les parenthèses après le mot `Poisson` signifient que cette classe a une méthode nommée `Poisson()`. Oui, il y a des méthodes spéciales appelées *constructeurs* (*constructors*), qui ont les caractéristiques suivantes :

- Les constructeurs ne sont appelés qu'une fois au cours de la construction d'un objet en mémoire.
- Ils doivent avoir le même nom que la classe elle-même.
- Ils ne retournent pas de valeur ; il n'est même pas nécessaire d'utiliser le mot-clé `void` dans la signature d'un constructeur.

Toute classe peut avoir plusieurs constructeurs. Si tu ne crées pas de constructeur pour une classe, Java crée automatiquement, lors de la compilation, un *constructeur sans argument par défaut*. C'est pour cela que le compilateur Java n'a jamais réclamé une déclaration permettant d'écrire `new Poisson()`, alors que la classe `Poisson` ne définit aucun constructeur.

En général, les constructeurs sont utilisés pour affecter des valeurs initiales aux variables membres d'une classe. Par exemple, la version suivante de la classe `Poisson` a un constructeur mono-argument qui se contente d'affecter la valeur de l'argument à la variable d'instance `profondeurCourante` pour une utilisation ultérieure.

```
public class Poisson extends AnimalFamilier {
  int profondeurCourante;

  Poisson(int positionDépart) {
    profondeurCourante = positionDépart;
  }
}
```

Maintenant, la classe `MaîtrePoisson` peut créer une instance de `Poisson` et affecter la position initiale du poisson. L'exemple suivant crée une instance de `Poisson` "submergée" sous 20 mètres d'eau :

```
Poisson monPoisson = new Poisson(20);
```

Si un constructeur avec arguments est défini dans une classe, il n'est plus possible d'utiliser le constructeur sans argument par défaut. Si tu souhaites avoir un constructeur sans argument, écris-en un.

Le mot-clé this

Le mot-clé `this` est utile lorsque tu as besoin de te référer à l'instance de l'objet dans laquelle tu te trouves. Regarde cet exemple :

```
class Poisson {
  int profondeurCourante ;

  Poisson(int profondeurCourante) {
    this.profondeurCourante = profondeurCourante;
  }
}
```

Le mot-clé `this` permet d'éviter les conflits de nom. Par exemple, `this.profondeurCourante` fait référence à la variable membre `profondeurCourante`, alors que `profondeurCourante` fait référence à la valeur de l'argument.

En d'autres termes, l'instance de l'objet `Poisson` pointe sur elle-même.

Tu verras un autre exemple important de l'utilisation du mot-clé `this` au Chapitre 6, dans la section *Comment passer des données entre classes*.

Tableaux

Disons que ton programme doit stocker les noms de quatre joueurs. Au lieu de déclarer quatre variables de type `String`, tu peux déclarer un *tableau* (*array*) qui a quatre *éléments* de type `String`.
Les tableaux sont repérés par des crochets, placés soit après le nom de la variable, soit après le type de données :

```
String [] joueurs;
```

ou

```
String joueurs[];
```

Ces lignes indiquent juste au compilateur Java que tu as l'intention de stocker plusieurs chaînes de caractères dans le tableau `joueurs`. Chaque élément possède son propre indice partant de zéro. L'exemple suivant crée en fait une instance de tableau qui peut stocker quatre éléments de type `String` puis leur affecte des valeurs :

```
joueurs = new String [4];

joueurs[0] = "David";
joueurs[1] = "Daniel";
joueurs[2] = "Anna";
joueurs[3] = "Gregory";
```

Tu dois connaître la taille du tableau avant d'affecter des valeurs à ses éléments. Si tu ne sais pas à l'avance combien d'éléments tu vas avoir, tu ne peux pas utiliser de tableaux ; tu devrais examiner d'autres classes Java, par exemple `Vector` (vecteur). Mais concentrons-nous pour l'instant sur nos tableaux.

Tous les tableaux ont un attribut nommé `length` (longueur) qui se "rappelle" le nombre d'éléments de ce tableau. Tu peux donc toujours savoir combien d'éléments il y a :

```
int nombreJoueurs = joueurs.length;
```

Si tu connais toutes les valeurs qui seront stockées dans le tableau au moment où tu le déclares, Java te permet de déclarer et d'initialiser le tableau d'un seul coup :

```
String [] joueurs = {"David", "Daniel", "Anna", "Gregory"};
```

Imagine que le second joueur soit un gagnant et que tu souhaites lui faire part de tes félicitations. Si le nom des joueurs est stocké dans un tableau, nous avons besoin de son deuxième élément :

```
String leGagnant = joueurs[1];
System.out.println("Félicitations, " + leGagnant + " !");
```

Voilà ce qu'affiche ce code :

```
Félicitations, Daniel !
```

Sais-tu pourquoi le deuxième élément a l'indice `[1]` ? Bien sûr que tu le sais : l'indice du premier élément est toujours `[0]`.

Le tableau de joueurs de notre exemple est *unidimensionnel*, car nous les stockons en quelque sorte en ligne. Si nous voulons stocker les valeurs sous forme de matrice, nous pouvons créer un tableau bidimensionnel. Java permet la création de tableaux *multidimensionnels*. Tu peux stocker n'importe quel objet dans un tableau ; je te montrerai comment au Chapitre 10.

Répétition d'actions à l'aide de boucles

Les boucles (*loops*) sont utilisées pour répéter la même action plusieurs fois, par exemple si nous avons besoin de féliciter plusieurs joueurs.
Si tu sais à l'avance combien de fois l'action doit être répétée, utilise une boucle avec le mot-clé `for` (pour) :

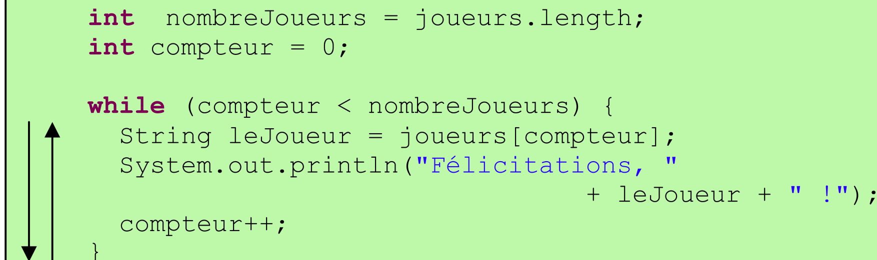
```
int nombreJoueurs = joueurs.length;
int compteur;

for (compteur = 0; compteur < nombreJoueurs; compteur++) {
  String leJoueur = joueurs[compteur];
  System.out.println("Félicitations, " +
                                   leJoueur + " !");
}
```

Java exécute chaque ligne entre les accolades puis retourne à la première ligne de la boucle, pour incrémenter le compteur et évaluer l'expression conditionnelle. Ce code signifie ceci :

Affiche la valeur de l'élément du tableau dont l'indice est identique à la valeur courante du compteur. Commence de l'élément 0 (`compteur = 0`) et incrémente de 1 la valeur du compteur (`compteur++`). Continue à faire ainsi tant que le compteur est inférieur à `nombreJoueurs` (`compteur < nombreJoueurs`).

Il y a un autre mot-clé permettant d'écrire des boucles - `while` (pendant). Avec ces boucles, il n'est pas nécessaire de savoir exactement combien de fois l'action doit être répétée, mais il faut tout de même savoir à quel moment terminer la boucle. Voyons comment

nous pouvons féliciter les joueurs en utilisant une boucle `while` qui se terminera lorsque la valeur de la variable `compteur` sera égale à la valeur de `nombreJoueurs` :



```
int  nombreJoueurs = joueurs.length;
int compteur = 0;

while (compteur < nombreJoueurs) {
  String leJoueur = joueurs[compteur];
  System.out.println("Félicitations, "
                               + leJoueur + " !");
  compteur++;
}
```

Au Chapitre 9, tu apprendras comment enregistrer les données sur les disques et comment les recharger dans la mémoire de l'ordinateur. Si tu charges les scores du jeu à partir d'un fichier sur le disque, tu ne sais pas à l'avance combien de scores y ont été enregistrés. Il est fort probable que tu chargeras les scores à l'aide d'une boucle `while`.

Tu peux aussi utiliser deux mots-clés importants dans les boucles : `break` et `continue`.

Le mot-clé `break` est utilisé pour sortir de la boucle quand une condition particulière est vraie. Disons que nous ne voulons pas afficher plus de 3 félicitations, indépendamment du nombre de joueurs que nous avons. Dans l'exemple suivant, après avoir affiché les éléments 0, 1 et 2 du tableau, le `break` fera sortir le code de la boucle et le programme continuera à partir de la ligne suivant l'accolade fermante.

Dans cet exemple de code, il y a un double signe égale dans la clause `if`. Cela signifie que tu compares la valeur de la variable `compteur` avec le nombre 3. Un seul signe égale à cet endroit signifierait qu'on affecte la valeur 3 à la variable `compteur`. Remplacer == par = dans une instruction `if` est un piège très subtil, qui peut mener à des erreurs du programme imprévisibles et difficiles à trouver.

```
int compteur = 0;
while (compteur < nombreJoueurs) {

  if (compteur == 3) {
    break; // Sortie de la boucle
  }
  String leJoueur = joueurs[compteur];
  System.out.println("Félicitations, " + leJoueur + " !");
  compteur++;
}
```

Le mot-clé `continue` permet au code de sauter des lignes et de revenir au début de la boucle. Imagine que tu veuilles féliciter tout le monde sauf David – le mot-clé `continue` fait revenir le programme au début de la boucle :

```
while (compteur < nombreJoueurs) {
  compteur++;

  String leJoueur = joueurs[compteur];

  if (leJoueur.equals("David")) {
    continue;
  }
  System.out.println("Félicitations, " + leJoueur + " !");
}
```

Il y a encore une autre forme de la boucle `while`, qui commence par le mot-clé `do` :

```
do {
  // Place ton code à cet endroit
} while (compteur < nombreJoueurs);
```

Une telle boucle évalue l'expression *après* avoir exécuté le code entre les accolades, ce qui signifie que le code à l'intérieur de la boucle est exécuté *au moins une fois*. Les boucles qui commencent par le mot-clé `while` peuvent ne pas être exécutées du tout si l'expression conditionnelle est fausse dès le début.

Autres lectures

1. jGuru: Language Essentials. Short Course:[5]
http://java.sun.com/developer/onlineTraining/JavaIntro/contents.html

2. Portée des variables:
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/scope.html

Exercices

1. Crée une nouvelle classe nommée `ConvertisseurTempératures` dont une méthode a la signature suivante :

```
public String convertirTempérature
      (int température, char convertirEn)
```

Si la valeur de l'argument `convertirEn` est `F`, la température doit être convertie en degrés Fahrenheit ; si c'est `C`, en degrés Celsius. Quand tu appelles cette méthode, mets la valeur de l'argument de type `char` entre apostrophes.

2. Essaie de déclarer la méthode `convertirNiveaux()` de la classe `BulletinAppréciation` comme `static` et supprime de la méthode `main()` la ligne qui instancie cette classe.

[5] NDT : Ce document propose, en anglais, une présentation des bases du langage et un cours d'introduction.

Exercices pour les petits malins

As-tu remarqué que dans l'exemple avec le mot-clé `continue` nous avons remonté la ligne `compteur++;` ?
Que ce serait-il passé si nous avions laissé cette ligne à la fin de la boucle comme dans l'exemple avec `break` ?

Chapitre 5. Une calculatrice graphique

Java est fourni avec un tas de classes dont tu vas te servir pour créer des applications graphiques. Il y a deux groupes principaux de classes (*librairies*) utilisées pour créer des fenêtres en Java : AWT et Swing.

AWT et Swing

Lorsque Java fut créé, seule la librairie AWT était disponible pour travailler avec du graphique. Cette librairie est un simple ensemble de classes telles que `Button` (bouton), `TextField` (champ textuel), `Label` (libellé) et autres. Peu après, une autre librairie, plus évoluée, apparut : Swing. Elle inclut aussi les boutons, les champs textuels et d'autres contrôles. Le nom des composants Swing commence par la lettre `J`, par exemple `JButton`, `JTextField`, `JLabel`, etc.

Tout est un peu mieux, plus rapide et plus simple d'utilisation avec Swing, mais dans certains cas, nos programmes sont exécutés sur des ordinateurs avec d'anciennes versions de Java qui peuvent ne pas supporter les classes Swing. Tu verras des exemples d'utilisation de AWT plus tard, au Chapitre 7, mais dans le présent chapitre nous allons créer une calculatrice utilisant Swing.

Il existe un autre ensemble de classes graphiques, une partie de la plate-forme Eclipse appelée Standard Widget Toolkit (SWT), mais nous ne l'utiliserons pas dans ce livre.

Paquetages et déclarations d'importation

Java est fourni avec de nombreuses classes utiles organisées en *paquetages* (*packages*). Certains paquetages contiennent des classes

responsables du dessin, d'autres des classes permettant de travailler avec Internet et ainsi de suite. Par exemple, la classe `String` fait partie du paquetage `java.lang` et le nom complet de la classe `String` est `java.lang.String`.

Le compilateur Java sait où trouver les classes de `java.lang`, mais il y a beaucoup d'autres paquetages contenant des classes utiles et c'est à toi de dire au compilateur où se trouvent les classes de ton programme. Par exemple, la plupart des classes Swing se trouvent dans l'un de ces deux paquetages :

```
javax.swing
javax.swing.event
```

Ce serait ennuyeux de devoir écrire le nom complet d'une classe à chaque fois qu'on l'utilise. Pour éviter cela, on peut effectuer une bonne fois des déclarations `import` (importer) avant la ligne de déclaration de la classe. Par exemple :

```
import javax.swing.JFrame;
import javax.swing.JButton;

class Calculatrice {
  JButton monBouton = new JButton();
  JFrame monCadre = new JFrame();
}
```

Ces déclarations `import` permettent d'utiliser ensuite les noms courts des classes, comme `JFrame` ou `JButton`. Le compilateur Java saura où chercher ces classes.

Si tu as besoin de plusieurs classes d'un même paquetage, tu n'as pas besoin d'en écrire la liste dans la déclaration `import`. Utilise simplement le *caractère joker* (*wildcard*). Dans l'exemple suivant, l'étoile (astérisque) rend toutes les classes du paquetage `javax.swing` *visibles* de ton programme :

```
import javax.swing.*;
```

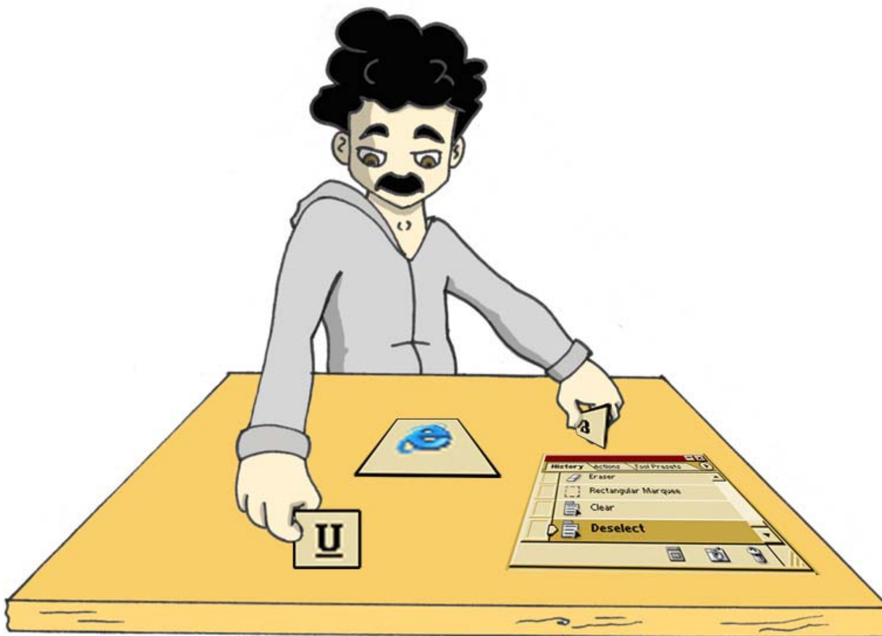
Cependant, il est préférable d'utiliser des déclarations d'importation distinctes, afin de voir quelle classe exacte est importée de chaque paquetage. Nous en reparlerons plus avant au Chapitre 10.

Principaux éléments de Swing

Voici les principaux objets qui constituent les applications Swing :

- Une fenêtre ou un *cadre* (*frame*) qui peut être créé en utilisant la classe `JFrame`.
- Un *panneau* (*panel*) ou un *carreau* (*pane*) contenant tous les boutons, champs textuels, libellés et autres composants. Les panneaux sont créés à l'aide de la classe `JPanel`.
- Les contrôles graphiques tels que les boutons (`JButton`), les champs textuels (`JTextField`), les listes (`JList`)...
- Les gestionnaires de disposition (*layout managers*) qui aident à organiser tous ces boutons et champs dans un panneau.

Habituellement, un programme crée une instance de `JPanel` et lui affecte un gestionnaire de disposition. Ensuite, il peut créer des contrôles graphiques et les ajouter au panneau. Enfin, il ajoute le panneau au cadre, fixe les dimensions du cadre et le rend visible.



Mais l'affichage du cadre n'est qu'une partie du travail, parce que les contrôles graphiques doivent savoir comment répondre à divers *événements* (*events*), tels qu'un clic sur un bouton.

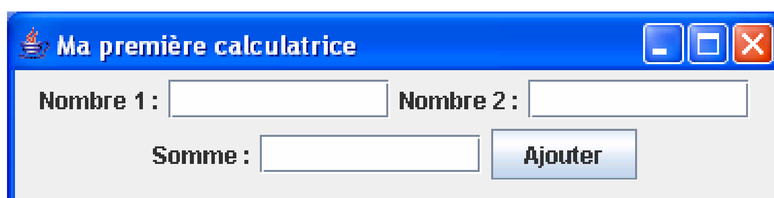
Dans ce chapitre, nous allons apprendre à afficher des fenêtres agréables à voir. Dans le chapitre suivant, nous verrons comment écrire le code qui traite les événements qui peuvent concerner les éléments de cette fenêtre.

Notre prochain objectif est la création d'une calculatrice simple capable d'ajouter deux nombres et d'afficher le résultat. Crée dans Eclipse le

projet `Ma Calculatrice` et la classe `CalculatriceSimple` avec le code suivant :

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

public class CalculatriceSimple {
  public static void main(String[] args) {
    // Crée un panneau
    JPanel contenuFenêtre = new JPanel();

    // Affecte un gestionnaire de disposition à ce panneau
    FlowLayout disposition = new FlowLayout();
    contenuFenêtre.setLayout(disposition);
    // Crée les contrôles en mémoire
    JLabel label1 = new JLabel("Nombre 1 :");
    JTextField entrée1 = new JTextField(10);
    JLabel label2 = new JLabel("Nombre 2 :");
    JTextField entrée2 = new JTextField(10);
    JLabel label3 = new JLabel("Somme :");
    JTextField résultat = new JTextField(10);
    JButton lancer = new JButton("Ajouter");

    // Ajoute les contrôles au panneau
    contenuFenêtre.add(label1);
    contenuFenêtre.add(entrée1);
    contenuFenêtre.add(label2);
    contenuFenêtre.add(entrée2);
    contenuFenêtre.add(label3);
    contenuFenêtre.add(résultat);
    contenuFenêtre.add(lancer);

    // Crée le cadre et y ajoute le panneau
    JFrame cadre = new JFrame("Ma première calculatrice");

    cadre.setContentPane(contenuFenêtre);

    // Positionne les dimensions et rend la fenêtre visible
    cadre.setSize(400,100);
    cadre.setVisible(true);
  }
}
```

Compile et exécute ce programme ; il affiche une fenêtre telle que celle-ci :



Ce n'est peut-être pas la calculatrice la plus jolie, mais elle nous permettra d'apprendre comment ajouter des composants et afficher une fenêtre. Dans la section suivante, nous tâcherons d'améliorer son apparence à l'aide des *gestionnaires de disposition*.

Gestionnaires de disposition

Certains langages de programmation vieillots t'obligent à régler les coordonnées et tailles exactes de chaque composant de la fenêtre. Ca marche très bien si tu connais les réglages de l'écran (sa *résolution*) de chacune des personnes qui utiliseront ton programme. A propos, on appelle les personnes qui utilisent nos programmes des *utilisateurs*. Il y a dans Java des gestionnaires de disposition qui t'aident à arranger les composants sur l'écran sans avoir à affecter des positions précises aux contrôles graphiques. Les gestionnaires de disposition assurent que l'écran aura une bonne tête quelles que soient les dimensions de la fenêtre.

Swing propose les gestionnaires de disposition suivants :

- `FlowLayout` (présentation en file)
- `GridLayout` (présentation en grille)
- `BoxLayout` (présentation en lignes ou colonnes)
- `BorderLayout` (présentation avec bordures)
- `CardLayout` (présentation en pile)
- `GridBagLayout` (présentation en grille composite)
- `SpringLayout` (présentation avec ressorts)

Pour utiliser un gestionnaire de disposition, un programme doit l'instancier puis affecter cet objet à un *conteneur* (*container*), par exemple à un panneau, comme dans la classe `CalculatriceSimple`.

FlowLayout

Dans cette présentation, les composants sont disposés dans la fenêtre ligne par ligne. Par exemple, les libellés, les champs textuels et les boutons sont ajoutés à la première ligne imaginaire tant qu'il y reste de la place. Quand la ligne courante est pleine, on passe à la ligne suivante. Si un utilisateur retaille la fenêtre, il risque de désorganiser son apparence. Essaie simplement de retailler la fenêtre de notre calculatrice en tirant sur l'un de ses coins. Regarde comment le gestionnaire `java.awt.FlowLayout` réorganise les contrôles lorsque les dimensions de la fenêtre varient.

Dans l'exemple de code suivant, le mot-clé `this` représente une instance de l'objet `CalculatriceSimple`.

```
FlowLayout disposition = new FlowLayout();
this.setLayoutManager(disposition);
```

Eh bien, `FlowLayout` n'est pas le meilleur choix pour notre calculatrice. Essayons quelque chose d'autre.

GridLayout

La classe `java.awt.GridLayout` te permet d'organiser les composants en *lignes* et en *colonnes* dans une grille. Tu vas ajouter les composants à des cellules imaginaires de cette grille. Si les dimensions de l'écran sont modifiées, les cellules de la grille peuvent changer de taille, mais les positions relatives des composants de la fenêtre ne changent pas. Notre calculatrice a sept composants – trois libellés, trois champs textuels et un bouton. Nous pouvons les organiser selon une grille de quatre lignes sur deux colonnes (il reste une cellule vide) :

```
GridLayout disposition = new GridLayout(4,2);
```

Tu peux aussi fixer un espace, horizontal ou vertical, entre les cellules, par exemple cinq pixels (les images sur l'écran de l'ordinateur sont constituées de minuscules points appelés pixels).

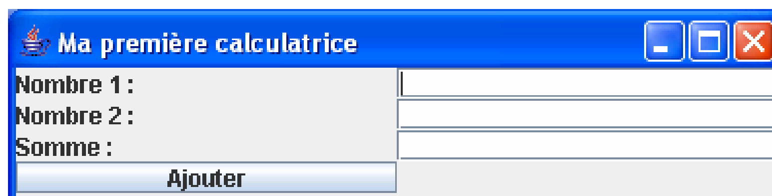
```
GridLayout disposition = new GridLayout(4,2,5,5);
```

Après quelques modifications mineures (en surbrillance ci-dessous), notre calculatrice a une bien meilleure allure.

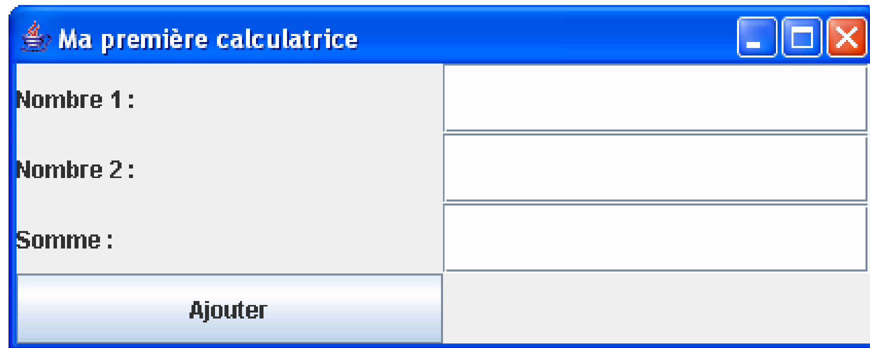
Crée et compile la classe `GrilleCalculatriceSimple` dans le projet `Ma Calculatrice`.

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

public class GrilleCalculatriceSimple {
  public static void main(String[] args) {
    // Crée un panneau
    JPanel contenuFenêtre = new JPanel();

    // Affecte un gestionnaire de présentation à ce panneau
    GridLayout disposition = new GridLayout(4,2);
    contenuFenêtre.setLayout(disposition);

    // Crée les contrôles en mémoire
    JLabel label1 = new JLabel("Nombre 1 :");
    JTextField entrée1 = new JTextField(10);
    JLabel label2 = new JLabel("Nombre 2 :");
    JTextField entrée2 = new JTextField(10);
    JLabel label3 = new JLabel("Somme :");
    JTextField résultat = new JTextField(10);
    JButton lancer = new JButton("Ajouter");

    // Ajoute les contrôles au panneau
    contenuFenêtre.add(label1);
    contenuFenêtre.add(entrée1);
    contenuFenêtre.add(label2);
    contenuFenêtre.add(entrée2);
    contenuFenêtre.add(label3);
    contenuFenêtre.add(résultat);
    contenuFenêtre.add(lancer);

    // Crée le cadre et y ajoute le panneau
    JFrame cadre = new JFrame("Ma première calculatrice");

    cadre.setContentPane(contenuFenêtre);

    // Affecte ses dimensions à la fenêtre et la rend visible
    cadre.setSize(400,100);
    cadre.setVisible(true);
  }
}
```

En exécutant le programme `GrilleCalculatriceSimple`, tu obtiens ceci :



Essaie de retailler cette fenêtre – les contrôles grandissent avec la fenêtre, mais leurs positions relatives ne changent pas :



Il y a encore une chose à retenir à propos de la disposition en grille – toutes les cellules de la grille ont la même largeur et la même hauteur.

BorderLayout

La classe `java.awt.BorderLayout` partage la fenêtre en cinq zones : `South` (sud), `West` (ouest), `North` (nord), `East` (est) et `Center` (centre). La région `North` est toujours en haut de la fenêtre, la région `South` en bas ; la région `West` est à gauche et la région `East` à droite.
Par exemple, dans la calculatrice de la page suivante, le champ textuel qui affiche les nombres est situé dans la zone `North`.

Voici comment créer une disposition `BorderLayout` et y placer un champ textuel :

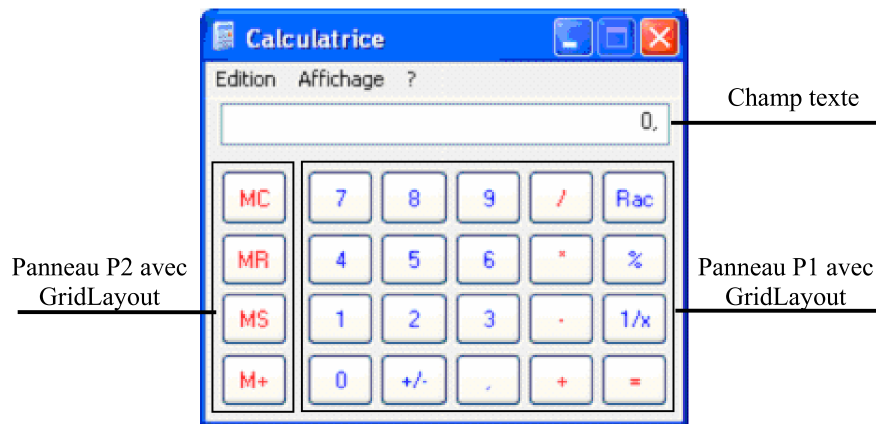
```
BorderLayout disposition = new BorderLayout();
this.setLayoutManager(disposition);

JTextField  affichageTexte = new JTextField(20);
this.add("North", affichageTexte);
```

Tu n'es pas obligé de mettre des contrôles graphiques dans chacune des cinq zones. Si tu n'as besoin que des régions `North`, `Center`, et `South`, la région `Center` sera plus large, car tu ne vas pas utiliser les régions `East` et `West`.

J'utiliserai un gestionnaire `BorderLayout` un peu plus tard dans la version suivante de notre calculatrice, `Calculatrice.java`.

Combiner les gestionnaires de disposition

Penses-tu que le gestionnaire `GridLayout` te permette de créer une fenêtre de calculatrice qui ressemble à celle de Microsoft Windows ?

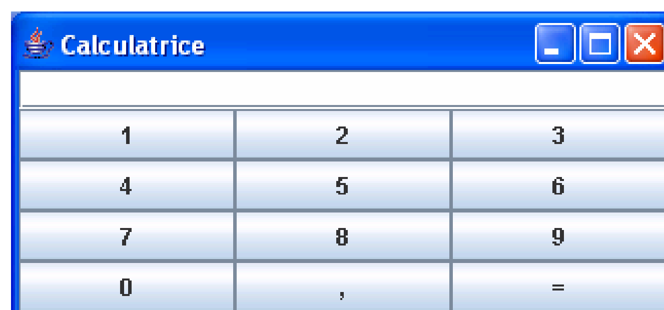


Malheureusement, non, parce que les cellules de cette calculatrice sont de tailles différentes – le champ textuel est plus large que les boutons, par exemple. Mais tu peux combiner les gestionnaires de disposition en utilisant des panneaux qui ont chacun leur propre gestionnaire de disposition.

Pour combiner les gestionnaires de disposition dans notre nouvelle calculatrice, procédons comme suit :

- ✓ Affecte une disposition `BorderLayout` au panneau de contenu.
- ✓ Ajoute un `JTextField` à la région `North` de l'écran pour afficher les nombres.
- ✓ Crée un panneau `p1` avec la présentation `GridLayout`, ajoute-lui 20 boutons et ajoute `p1` à la région `Center` du panneau de contenu.
- ✓ Crée un panneau `p2` avec la présentation `GridLayout`, ajoute-lui 4 boutons et ajoute `p2` à la région `West` du panneau de contenu.

Commençons avec une version un peu plus simple de la calculatrice qui ressemblera à ceci :



Crée la classe `Calculatrice` et exécute le programme. Lis les commentaires dans l'exemple de code suivant pour comprendre comment ça marche.

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.BorderLayout;
public class Calculatrice {
  // Déclaration de tous les composants de la calculatrice.
  JPanel contenuFenêtre;
  JTextField champAffichage;
  JButton bouton0;
  JButton bouton1;
  JButton bouton2;
  JButton bouton3;
  JButton bouton4;
  JButton bouton5;
  JButton bouton6;
  JButton bouton7;
  JButton bouton8;
  JButton bouton9;
  JButton boutonVirgule;
  JButton boutonEgale;
  JPanel panneauChiffres;

  // Le constructeur crée les composants en mémoire
  // et les ajoute au cadre en utilisant une combinaison
  // de Borderlayout et Gridlayout
  Calculatrice() {
    contenuFenêtre = new JPanel();

    // Affecte un gestionnaire de présentation à ce panneau
    BorderLayout dispositionl = new BorderLayout();
    contenuFenêtre.setLayout(dispositionl);

    // Crée le champ d'affichage et le positionne dans
    // la zone nord de la fenêtre
    champAffichage = new JTextField(30);
    contenuFenêtre.add("North", champAffichage);

    // Crée les boutons en utilisant le constructeur de
    // la classe JButton qui prend en paramètre le libellé
    // du bouton
    bouton0 = new JButton("0");
    bouton1 = new JButton("1");
    bouton2 = new JButton("2");
    bouton3 = new JButton("3");
    bouton4 = new JButton("4");
    bouton5 = new JButton("5");
    bouton6 = new JButton("6");
    bouton7 = new JButton("7");
    bouton8 = new JButton("8");
    bouton9 = new JButton("9");
    boutonVirgule = new JButton(",");
    boutonEgale = new JButton("=");
```

Classe `Calculatrice` (partie 1 de 2)

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com


```
    // Crée le panneau avec le quadrillage qui contient
    // les 12 boutons - les 10 boutons numériques et ceux
    // représentant la virgule et le signe égale
    panneauChiffres = new JPanel();
    GridLayout disposition2 = new GridLayout(4, 3);
    panneauChiffres.setLayout(disposition2);

    // Ajoute les contrôles au panneau panneauChiffres
    panneauChiffres.add(bouton1);
    panneauChiffres.add(bouton2);
    panneauChiffres.add(bouton3);
    panneauChiffres.add(bouton4);
    panneauChiffres.add(bouton5);
    panneauChiffres.add(bouton6);
    panneauChiffres.add(bouton7);
    panneauChiffres.add(bouton8);
    panneauChiffres.add(bouton9);
    panneauChiffres.add(bouton0);
    panneauChiffres.add(boutonVirgule);
    panneauChiffres.add(boutonEgale);

    // Ajoute panneauChiffres à la zone centrale de la
    // fenêtre
    contenuFenêtre.add("Center", panneauChiffres);

    // Crée le cadre et lui affecte son contenu
    JFrame frame = new JFrame("Calculatrice");
    frame.setContentPane(contenuFenêtre);

    // Affecte à la fenêtre des dimensions suffisantes pour
    // prendre en compte tous les contrôles
    frame.pack();

    // Enfin, affiche la fenêtre
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    Calculatrice calc = new Calculatrice();
  }
}
```

Classe `Calculatrice` (partie 2 de 2)

BoxLayout

La classe `javax.swing.BoxLayout` permet de disposer de multiples composants soit horizontalement (selon l'axe des X) ou verticalement (selon l'axe des Y). Contrairement au gestionnaire `FlowLayout`, les contrôles ne changent pas de ligne quand la fenêtre est retaillée. Avec `BoxLayout`, les contrôles peuvent avoir des tailles différentes (ce qui n'est pas possible avec `GridLayout`).

Les deux lignes de code suivantes mettent en place une présentation de type `BoxLayout` avec un alignement vertical dans un `JPanel`.

```
JPanel panneauChiffres = new JPanel();
setLayout(new BoxLayout(panneauChiffres, BoxLayout.Y_AXIS));
```

Pour rendre ce code plus compact, je ne déclare pas de variable pour stocker une référence à l'objet BoxLayout ; je crée plutôt une instance de cet objet et la passe immédiatement en argument à la méthode setLayout().

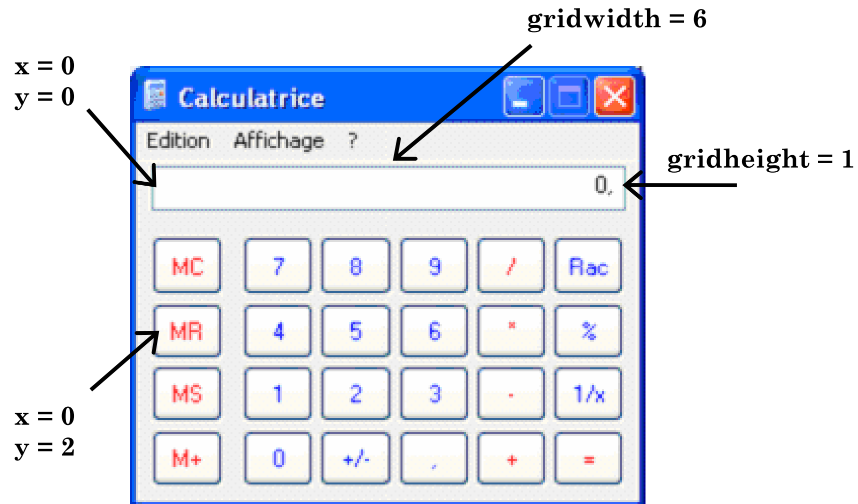
GridBagLayout

Dans cette section, je vais encore te montrer un autre moyen de créer la fenêtre de la calculatrice en utilisant le gestionnaire java.awt.GridBagLayout au lieu de combiner des dispositions et des panneaux.

Notre calculatrice a des lignes et des colonnes, mais dans une présentation en grille tous les composants doivent avoir les mêmes dimensions. Ca ne marche pas pour notre calculatrice à cause du champ textuel, en haut, qui est aussi large que trois boutons numériques.

La disposition GridBagLayout est une grille évoluée, qui permet d'avoir des cellules de tailles différentes. La classe GridBagLayout fonctionne en collaboration avec la classe GridBagConstraints (contraintes de la grille). Les contraintes ne sont rien d'autre que des attributs de la cellule, que tu dois positionner pour chaque cellule séparément. Toutes les contraintes d'une cellule doivent être positionnées *avant* de placer un composant dans la cellule. Par exemple, l'un des attributs de contrainte est appelé gridwidth. Il permet de rendre une cellule aussi large que plusieurs autres.

Quand tu travailles avec une présentation en grille composite, tu dois d'abord créer une instance de l'objet contrainte puis donner une valeur à ses propriétés. Une fois ceci fait, tu peux ajouter le composant à la cellule dans ton conteneur.

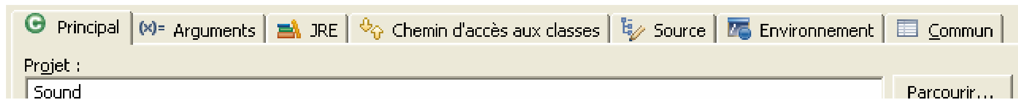


L'exemple de code suivant est largement commenté pour t'aider à comprendre comment utiliser `GridBagLayout`.

```
// Positionne le GridBagLayout pour le contenu de la fenêtre
GridBagLayout disposition = new GridBagLayout();
contenuFenêtre.setLayout(disposition);

// Tu dois répéter ces lignes pour chaque composant
// que tu souhaites ajouter au quadrillage
// Crée une instance de GridBagConstraints
GridBagConstraints contr = new GridBagConstraints();

// Affecte les contraintes du champ Affichage

// coordonnée x dans le quadrillage
contr.gridx = 0;
// coordonnée y dans le quadrillage
contr.gridy = 0;

// cette cellule a la même hauteur que les autres
contr.gridheight = 1;

// cette cellule est 6 fois plus large que les autres
contr.gridwidth = 6;

// remplit tout l'espace dans la cellule
contr.fill = GridBagConstraints.BOTH;

// proportion d'espace horizontal occupée par ce composant
contr.weightx = 1.0;

// proportion d'espace vertical occupée par ce composant
contr.weighty = 1.0;

// position du composant dans la cellule
contr.anchor = GridBagConstraints.CENTER;

// Crée le champ textuel
champAffichage = new JTextField();

// Affecte les contraintes à ce champ
disposition.setConstraints(champAffichage, contr);

// Ajoute le champ à la fenêtre
contenuFenêtre.add(champAffichage);
```

CardLayout

Imagine une pile de cartes posées les unes sur les autres, de telle sorte qu'on ne puisse voir complètement que la carte du dessus. Le gestionnaire `java.awt.CardLayout` permet de créer un composant qui ressemble à un classeur à onglets.



Quand tu cliques sur un onglet, le contenu de l'écran change. En fait, tous les panneaux nécessaires à cet écran dont déjà préchargés et se trouvent les uns au-dessus des autres. Quand l'utilisateur clique sur un onglet, le programme se contente de "mettre cette carte" au-dessus et rend les autres cartes invisibles.

Il y a peu de chances que tu utilises cette disposition car la librairie Swing comporte un meilleur composant pour les fenêtres à onglet. Il s'agit du composant `JTabbedPane`.

SpringLayout

Le gestionnaire de disposition à ressorts est inspiré de véritables ressorts et tringles. Imagine que tu aies un paquet de crochets, ressorts et tringles et que tu doives fixer nos boutons et champs textuels aux bordures de la fenêtre. Les composants attachés par des tringles ne peuvent pas bouger et changeront de taille si tu agrandis la fenêtre, mais ceux attachés par des ressorts pourront s'écarter du bord.

Les écarts entre composants sont définis par les propriétés des ressorts (minimum, maximum, taille préférée et taille réelle). Ces paramètres sont spécifiés dans l'instance de `SpringLayout.Constraints` associée à chaque composant. C'est un peu comme pour le `GridBagLayout`.

Au contraire du `GridLayout`, le `SpringLayout` te permet de gérer la disposition de composants de tailles différentes, tout en maintenant leurs positions relatives.

Tu peux en lire davantage sur ce gestionnaire de disposition dans un article en ligne de Joe Winchester : "SpringLayout: A Powerful and Extensible Layout Manager" (http://jdj.sys-con.com/read/37300.htm).

Puis-je créer des fenêtres sans utiliser de gestionnaire de disposition ?

Bien sûr que tu peux ! Tu peux fixer les coordonnées dans l'écran de chaque composant lorsque tu l'ajoutes à la fenêtre. Dans ce cas, ta classe doit annoncer explicitement qu'elle n'utilisera pas de gestionnaire de disposition. En Java, il y a le mot-clé spécial `null`, qui signifie précisément "n'a pas de valeur". Nous utiliserons ce mot-clé assez souvent par la suite. Dans l'exemple suivant, il signifie qu'il n'y a pas de gestionnaire de disposition :

```
contenuFenêtre.setLayout(null);
```

Mais si tu fais ceci, ton code doit affecter les coordonnées du coin supérieur gauche, la largeur et la hauteur de chaque composant de la fenêtre. Cet exemple montre comment tu peux donner à un bouton une largeur de 40 pixels, une hauteur de 20 pixels et le placer à 100 pixels à droite et 200 pixels au-dessous du coin supérieur gauche de la fenêtre :

```
JButton monBouton = new JButton("Nouvelle partie");
monBouton.setBounds(100, 200, 40, 20);
```

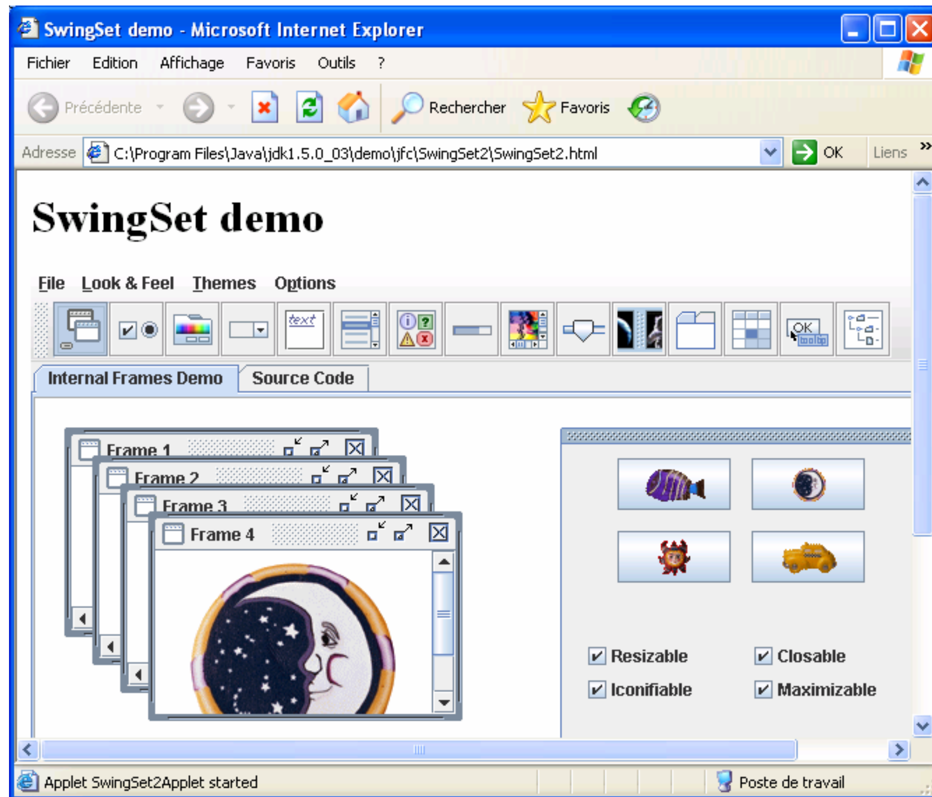
Composants des fenêtres

Je ne vais pas décrire dans ce livre tous les composants de Swing, mais tu trouveras dans la section *Autres lectures* les références du didacticiel Swing en ligne. Ce didacticiel fournit des explications détaillées de tous les composants Swing. Nos calculatrices n'utilisent que JButton, JLabel et JTextField, mais voici la liste de tout ce qui est disponible :

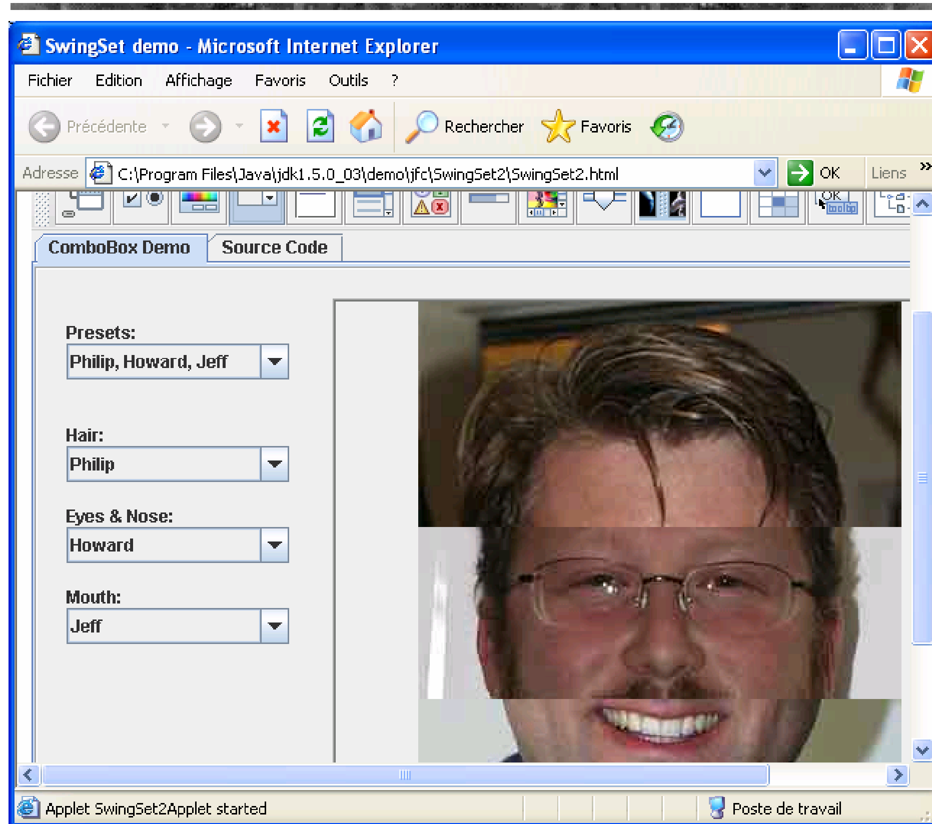
- ✓ JButton
- ✓ JLabel
- ✓ JCheckBox
- ✓ JRadioButton
- ✓ JToggleButton
- ✓ JScrollPane
- ✓ JSpinner
- ✓ JTextField
- ✓ JTextArea
- ✓ JPasswordField
- ✓ JFormattedTextField
- ✓ JEditorPane
- ✓ JScrollBar
- ✓ JSlider
- ✓ JProgressBar
- ✓ JComboBox
- ✓ JList
- ✓ JTabbedPane
- ✓ JTable
- ✓ JToolTip
- ✓ JTree
- ✓ JViewPort
- ✓ ImageIcon

Tu peux aussi créer des menus (JMenu et JPopupMenu), des fenêtres à la demande (*popup*), des cadres imbriqués dans d'autres cadres (JInternalFrame) et utiliser les fenêtres standard de manipulation d'informations : JFileChooser (choix de fichier), JColorChooser (choix de couleur) et JOptionPane (choix d'option).

Java est accompagné d'une excellente application de démonstration qui présente tous les composants Swing disponibles en action. Elle se trouve dans le répertoire `demo\jfc\SwingSet2` du répertoire d'installation de J2SDK. Ouvre simplement le fichier `SwingSet2.html`, et tu verras un écran similaire à celui-ci :



Clique sur une image de la barre d'outils pour voir comment le composant Swing correspondant fonctionne. Tu peux aussi consulter le code utilisé pour créer chaque fenêtre en sélectionnant l'onglet *Source Code*. Par exemple, si tu cliques sur la quatrième icône à partir de la gauche (nommée *combobox*), tu obtiendras une fenêtre comme celle-ci :



Il y a tant de différents composants Swing permettant de rendre tes fenêtres agréables à voir !

Dans ce chapitre, nous avons créé des composants Swing simplement en tapant le code, sans utiliser d'outils particuliers. Mais il existe des outils permettant de sélectionner un composant depuis une barre d'outils et de le déposer sur la fenêtre. Ces outils génèrent automatiquement le bon code Java pour les composants Swing. L'un des ces outils gratuits de conception d'interface utilisateur graphique (Graphic User Interface ou GUI), permettant de créer facilement des composants Swing et SWT, est *jigloo*, édité par CloudGarden. Tu trouveras la référence de la page web de ce produit dans la section *Autres lectures*.

Dans le chapitre suivant, tu apprendras comment une fenêtre peut réagir aux actions de l'utilisateur.

Autres lectures

1.Didacticiel Swing :
http://java.sun.com/docs/books/tutorial/uiswing/

2. Classe `JFormattedTextField` :
http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/JFormattedTextField.html

3. Didacticiel et articles SWT :
http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-swt-home/SWT_Resources.html

4. Constructeur de GUI Jigloo :
http://www.cloudgarden.com/jigloo/index.html

Exercices

1. Modifie la classe `Calculatrice.java` en lui ajoutant les boutons `+`, `-`, `/` et `*`. Ajoute ces boutons au panneau `panneauOpérations` et place celui-ci dans la région est du panneau contenu de la fenêtre.

2. Va lire sur le web les informations concernant `JFormattedTextField` et modifie le code de la calculatrice en utilisant cette classe à la place de `JTextField`. L'objectif est de créer un champ aligné à droite comme dans les vraies calculatrices.

Exercices pour les petits malins

Modifie la classe `Calculatrice.java` pour stocker tous les boutons numériques dans un tableau de 10 éléments déclaré ainsi :

```
JButton[] boutonsChiffres =
                          new JButton[10];
```

Remplace les 10 lignes à partir de :

```
bouton0 = new JButton("0");
```

par une boucle qui crée les boutons et les stocke dans le tableau.

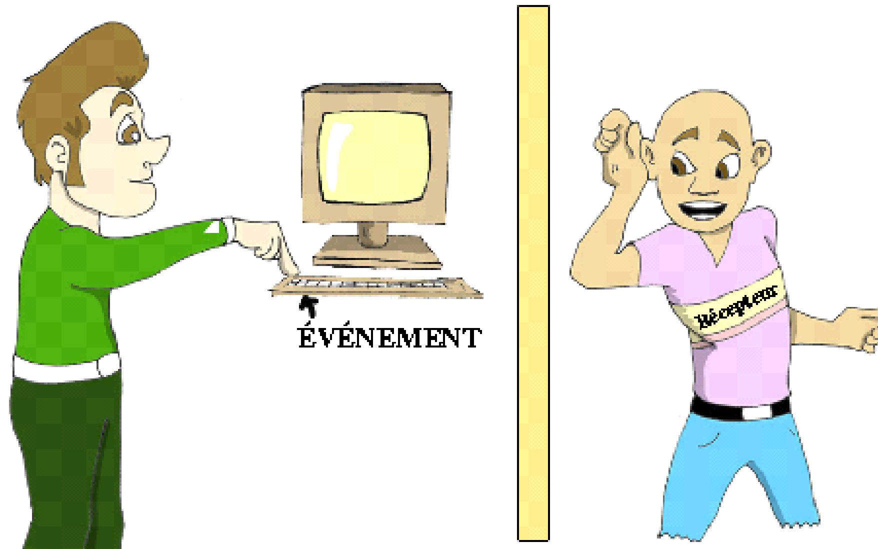
Suggestion : jette un œil au code du jeu de morpion au Chapitre 7.

Chapitre 6. Evénements de la fenêtre

Au cours de son exécution, un programme peut être confronté à divers événements : un utilisateur clique sur un bouton dans une fenêtre, le navigateur web décide de repeindre la fenêtre, etc. Je suis sûr que tu as essayé de cliquer sur les boutons de notre calculatrice du Chapitre 5, mais ces boutons n'étaient pas encore prêts à réagir à tes actions.

Chaque composant d'une fenêtre peut traiter un certain nombre d'événements ou, comme on dit, *être à l'écoute* (*listen*) de ces événements, et, à son tour, *émettre des événements*, possiblement différents de ceux qu'il a reçu. Pour réagir aux actions de l'utilisateur, ton programme doit s'enregistrer auprès des composants de la fenêtre au moyen de classes Java appelées *récepteurs* (*listeners*). Il vaut mieux que ton programme ne reçoive que les événements qui l'intéressent. Par exemple, quand une personne déplace le pointeur de la souris au-dessus d'un bouton de la calculatrice : peu importe l'endroit exact où se trouve la souris quand la personne appuie sur le bouton ; ce qui compte, c'est que c'est à l'intérieur de la surface du bouton, et que ton programme sache que le bouton a été cliqué. C'est pourquoi tu n'as pas besoin d'enregistrer ton programme auprès du bouton en tant que `MouseMotionListener`, qui est le récepteur traitant les déplacements de la souris. Par contre, ce récepteur est bien utile pour toutes sortes de programmes de dessin.

Ton programme devrait s'enregistrer auprès des boutons de la calculatrice en tant que récepteur `ActionListener`, qui peut traiter les clics sur un bouton. Tous les récepteurs sont des classes Java spéciales appelées *interfaces*.



Interfaces

La plupart des classes définissent des méthodes qui effectuent diverses actions, par exemple *réagir aux clics sur les boutons*, *réagir aux mouvements de la souris*, etc. La combinaison de telles actions est appelée le *comportement de la classe* (*class behavior*).

Les interfaces sont des classes spéciales qui donnent juste des noms à un ensemble d'actions particulières sans écrire le code réel qui implanterait ces actions. Par exemple :

```
interface MouseMotionListener {
  void mouseDragged(MouseEvent événement);
  void mouseMoved(MouseEvent événement);
}
```

Comme tu vois, les méthodes `mouseDragged()` et `mouseMoved()` ne contiennent pas de code – elles sont juste déclarées dans l'interface `MouseMotionListener`. Mais si ta classe a besoin de réagir quand la souris est déplacée ou qu'elle fait glisser un objet, elle doit *implanter*[6] (*to implement*) cette interface. Le terme *implanter* signifie que cette classe va inclure pour de bon les méthodes déclarées dans l'interface, par exemple :

[6] NDT on rencontre fréquemment aussi le terme *implémenter*, mais implanter est la recommandation officielle.

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;


class monBlocADessin implements MouseMotionListener {

  // Insère ici ton code capable de dessiner

  public void mouseDragged(MouseEvent événement) {
    // Insère ici le code à exécuter quand un "glisser"
    // est effectué avec la souris
  }

  public void mouseMoved(MouseEvent événement) {
    // Insère ici le code à exécuter quand la souris est
    // déplacée
  }
}
```

Tu te demandes peut-être à quoi bon prendre la peine de créer des interfaces sans même y écrire le code ? C'est pour la bonne raison que, une fois que cette interface est créée, elle peut être réutilisée par de nombreuses classes. Par exemple, quand d'autres classes (ou Java lui-même) voient que la classe `monBlocADessin` implante l'interface `MouseMotionListener`, elles sont certaines que cette classe définit les méthodes `mouseDragged()` et `mouseMoved()`. Chaque fois qu'un utilisateur déplace la souris, Java appelle la méthode `mouseMoved()` et exécute le code que tu y a écrit. Imagine ce qui se passerait si un programmeur John décidait de nommer une telle méthode `mouseMoved()`, mais que Marie la nommait `mouvementSouris()` et Pierre préférait `sourisDéplacée()` ? Java serait embrouillé et ne saurait pas quelle méthode de ta classe appeler pour lui signaler le déplacement de la souris.

Une classe Java peut implanter plusieurs interfaces. Par exemple, elle peut avoir besoin de traiter les déplacements de la souris et le clic sur un bouton :

```
class MonProgrammeDeDessin implements
                     MouseMotionListener, ActionListener {

  // Tu dois écrire ici le code de toutes les méthodes
  // définies par les deux interfaces

}
```

Une fois à l'aise avec les interfaces fournies avec Java, tu seras à même de créer tes propres interfaces, mais il s'agit d'un sujet avancé. N'en parlons même pas pour l'instant.

Récepteur d'événements

Revenons à notre calculatrice. Si tu as fait tes devoirs du précédent chapitre, la partie visuelle est terminée. Nous allons maintenant créer une autre classe, un récepteur qui effectuera certaines actions quand l'utilisateur cliquera sur l'un des boutons. En réalité, nous aurions pu ajouter le code traitant les événements clic à la classe `Calculatrice.java`, mais les bons programmeurs mettent toujours les parties affichage et traitement dans des classes distinctes.

Nous allons appeler la deuxième classe `MoteurCalcul`. Elle doit implanter l'interface `java.awt.ActionListener` qui ne déclare qu'une méthode - `actionPerformed(ActionEvent)`. Java appelle cette méthode de la classe qui implémente l'interface à chaque fois qu'une personne clique sur un bouton.

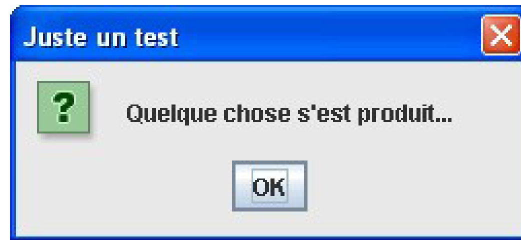
Crée s'il te plaît cette classe simple :

```
import java.awt.event.ActionListener;
public class MoteurCalcul implements ActionListener {

}
```

Si tu essaies de compiler cette classe (ou simplement de l'enregistrer dans Eclipse), tu obtiens un message d'erreur disant que cette classe doit implanter la méthode `actionPerformed(ActionEvent événement)`. Corrigeons cette erreur :

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class MoteurCalcul implements ActionListener {

  public void actionPerformed(ActionEvent événement) {
    // On peut laisser ici une méthode vide, même s'il
    // ne se produira rien quand Java l'appellera
  }
}
```

La version suivante de cette classe affiche une *boîte de message (message box)* depuis la méthode `actionPerformed()`. Tu peux afficher n'importe quel message en utilisant la classe `JOptionPane` et sa méthode `showConfirmDialog()`. Par exemple, la classe `MoteurCalcul` affiche cette boîte de message :



Il y a différentes version de la méthode `showConfirmDialog()`. Nous allons utiliser celle qui prend quatre arguments. Dans le code ci-dessous, le mot-clé `null` signifie que cette boîte de message n'a pas de fenêtre mère, le deuxième argument contient le message, le troisième le titre de la boîte de message et le quatrième te permet de choisir le(s) bouton(s) à inclure dans la boîte (`PLAIN_MESSAGE` signifie que seul un bouton OK est affiché).

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;
public class MoteurCalcul implements ActionListener {

  public void actionPerformed(ActionEvent événement) {
    JOptionPane.showConfirmDialog(null,
                  "Quelque chose s'est produit...",
                  "Juste un test",
                  JOptionPane.PLAIN_MESSAGE);
  }
}
```

Dans la prochaine section, je t'expliquerai comment compiler et exécuter une nouvelle version de notre calculatrice, qui affichera la boîte de message "Quelque chose s'est produit".

Enregistrement d'un `ActionListener` auprès d'un composant

Par qui et quand sera appelé le code que nous avons écrit dans la méthode `actionPerformed()` ? C'est Java qui appellera cette méthode *si tu enregistres* (ou relies) la classe `MoteurCalcul` auprès des boutons de la calculatrice ! Ajoute simplement les deux lignes suivantes à la fin du constructeur de la classe `Calculatrice.java` pour enregistrer notre récepteur d'actions auprès du bouton zéro :

```
MoteurCalcul moteurCalcul = new MoteurCalcul();
bouton0.addActionListener(moteurCalcul);
```

Désormais, chaque fois qu'un utilisateur cliquera sur `bouton0`, Java appellera la méthode `actionPerformed()` de l'objet `MoteurCalcul`.

Maintenant, compile et exécute la classe `Calculatrice`. Clique sur le bouton zéro – ton programme affiche la boîte de message "Quelque chose s'est produit" ! Les autres boutons restent inactifs car ils ne connaissent pas encore notre récepteur d'actions. Continue à ajouter des lignes semblables pour donner vie à tous les boutons :

```
bouton1.addActionListener(moteurCalcul);
bouton2.addActionListener(moteurCalcul);
bouton3.addActionListener(moteurCalcul);
bouton4.addActionListener(moteurCalcul);
…
```

Quelle est la source d'un événement ?

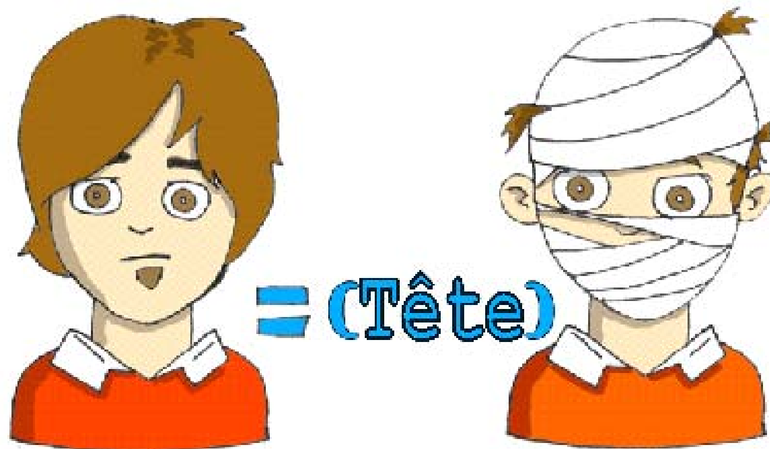
La prochaine étape consiste à rendre notre récepteur un peu plus subtil – il affichera différentes boîtes de message, en fonction du bouton sélectionné. Quand un *événement d'action* (*action event*) se produit, Java appelle la méthode `actionPerformed (ActionEvent événement)` de ta classe récepteur et lui fournit de précieuses informations sur l'événement dans l'argument `événement`. Tu peux obtenir cette information en appelant les méthodes appropriées de cet objet.

Conversion de type explicite

Dans l'exemple suivant, nous appelons la méthode `getSource()` de la classe `ActionEvent` pour savoir sur quel bouton a appuyé l'utilisateur – la variable `événement` est une référence à cet objet qui vit quelque part dans la mémoire de l'ordinateur. Mais d'après la documentation Java, cette méthode retourne la source de l'événement sous la forme d'une instance du type `Object`, qui est la superclasse de toutes les classes Java, y compris les composants de fenêtre. C'est fait ainsi de façon à avoir une méthode universelle qui fonctionne avec tous les composants. Mais nous savons bien que, dans notre fenêtre, seuls les boutons peuvent être à l'origine de l'événement d'action ! C'est pourquoi nous effectuons une *conversion de type explicite* (*casting*) de l'`Object` retourné en un `JButton`, en indiquant le type `(JButton)` entre parenthèses devant l'appel de la méthode :

```
JButton boutonCliqué =  (JButton) événement.getSource();
```

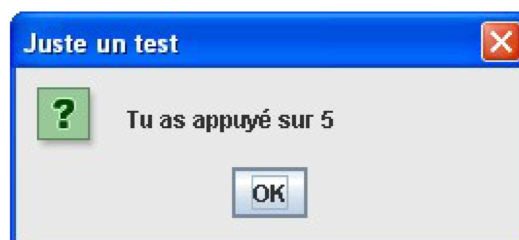
Nous déclarons une variable de type `JButton` à la gauche du signe égale et, bien que la méthode `getSource()` retourne des données du type `Object`, nous disons à Java : *Ne t'inquiète pas, je suis certain d'obtenir une instance de* `JButton`.



C'est seulement après avoir effectué la conversion d'`Object` en `JButton` que nous avons le droit d'appeler la méthode `getText()` définie par la classe `JButton`.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;
import javax.swing.JButton;
public class MoteurCalcul implements ActionListener {
  public void actionPerformed(ActionEvent événement) {
    // Retrouve la source de l'action
    JButton boutonCliqué = (JButton) événement.getSource();
    // Retrouve le libellé du bouton
    String libelléBoutonCliqué =
                             boutonCliqué.getText();

    // Concatène le libellé du bouton au texte
    // de la boîte de message
    JOptionPane.showConfirmDialog(null,
                "Tu as appuyé sur " + libelléBoutonCliqué,
                "Juste un test",
                JOptionPane.PLAIN_MESSAGE);
  }
}
```

Maintenant, si tu appuies par exemple sur le bouton cinq, tu obtiendras la boîte de message suivante :



Mais que faire si les événements d'une fenêtre proviennent non seulement des boutons, mais aussi d'autres composants ? Nous ne

voulons pas convertir n'importe quel objet en JButton ! Pour traiter ces cas, tu dois utiliser l'opérateur Java spécial instanceof pour effectuer la conversion de type appropriée. L'exemple suivant vérifie d'abord quel type d'objet est à l'origine de l'événement, puis effectue une conversion de type, soit en JButton soit en JTextField :

```
public void actionPerformed(ActionEvent événement) {

  JTextField monChampAffichage = null;
  JButton boutonCliqué = null;

  Object sourceEvénement = événement.getSource();

  if (sourceEvénement instanceof JButton) {
    boutonCliqué = (JButton) sourceEvénement;
  }
  else if (sourceEvénement instanceof JTextField) {
    monChampAffichage = (JTextField) sourceEvénement;
  }
}
```

Notre calculatrice doit exécuter différentes portions de code pour les différents boutons. Le fragment de code suivant montre comment faire.

```
public void actionPerformed(ActionEvent événement) {

  Object source = événement.getSource();

  if (source == boutonPlus) {
    // Insère ici le code qui ajoute deux nombres
   }
  else if (source == boutonMoins) {
    // Insère ici le code qui soustrait deux nombres
   }
  else if (source == boutonDiviser) {
    // Insère ici le code qui divise deux nombres
   }
  else if (source == boutonMultiplier) {
    // Insère ici le code qui multiplie deux nombres
   }
}
```

Comment passer des données entre classes

En fait, quand tu appuies sur une touche numérique d'une calculatrice réelle, elle n'affiche pas une boîte de message, mais plutôt le chiffre dans le champ textuel en haut. Voici un nouveau challenge – il nous faut parvenir à atteindre l'attribut displayField de la classe Calculatrice depuis la méthode actionPerformed() de la classe MoteurCalcul. C'est faisable si nous définissons, dans la classe MoteurCalcul, une variable qui stockera une *référence à l'instance* de l'objet Calculatrice.

Nous allons déclarer un constructeur dans la prochaine version de la classe `MoteurCalcul`. Ce constructeur a un argument de type `Calculatrice`. Ne sois pas étonné, les types des arguments de méthodes peuvent être des classes que tu as toi-même créées !

Java exécute le constructeur de l'instance de `MoteurCalcul` au cours de sa création en mémoire. La classe `Calculatrice` instancie le `MoteurCalcul` et passe au constructeur du moteur *une référence à elle-même* :

```
MoteurCalcul moteurCalcul = new MoteurCalcul(this);
```

Cette référence contient l'adresse de la calculatrice en mémoire. Le constructeur du moteur stocke cette valeur dans la variable membre `parent` et enfin l'utilise dans la méthode `actionPerformed()` pour accéder au champ textuel de la calculatrice.

```
parent.champAffichage.getText();
…
parent.champAffichage.setText(textChampAffichage +
                                    libelléBoutonCliqué);
```

Ces deux lignes sont extraites de l'exemple de code suivant.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JButton;

public class MoteurCalcul implements ActionListener {

  Calculatrice parent; // une référence à la Calculatrice

  // Le constructeur stocke la référence à la fenêtre
  // Calculatrice dans la variable membre parent
  MoteurCalcul(Calculatrice parent) {
    this.parent = parent;
  }

  public void actionPerformed(ActionEvent événement) {
    // Retrouve la source de cette action
    JButton boutonCliqué = (JButton) événement.getSource();

    // Retrouve le texte existant dans le champ Affichage
    // de la Calculatrice
    String texteChampAffichage =
                        parent.champAffichage.getText();

    // Retrouve le libellé du bouton
    String libelléBoutonCliqué = boutonCliqué.getText();

    // Affecte le nouveau texte au champ Affichage
    parent.champAffichage.setText(texteChampAffichage +
                                  libelléBoutonCliqué);
  }
}
```

Quand tu déclares une variable pour stocker une référence à l'instance d'une classe particulière, cette variable doit être du type de cette classe ou de l'une de ses superclasses.

En Java, chaque classe hérite de la classe `Object`. Si la classe `Poisson` est une sous-classe de `AnimalFamilier`, toutes ces lignes sont correctes :

```
Poisson monPoisson = new Poisson();
AnimalFamilier monPoisson = new Poisson();
Object monPoisson = new Poisson();
```

Fin de la calculatrice

Intéressons-nous à quelques règles (un *algorithme*) concernant la façon dont notre calculatrice doit fonctionner :

1. L'utilisateur entre tous les chiffres du premier nombre.

2. Si l'utilisateur tape l'un des boutons d'action `+`, `-`, `/` ou `*`, alors stocker le premier nombre et l'action sélectionnée dans des variables membres, puis effacer le nombre du champ textuel.

3. L'utilisateur entre un deuxième nombre et appuie sur le bouton *égale*.

4. Convertir la valeur de type `String` du champ textuel dans le type numérique `double` pour pouvoir stocker de grands nombres décimaux. Exécuter l'action sélectionnée en utilisant cette valeur et le nombre stocké dans la variable à l'étape 2.

5. Afficher le résultat de l'étape 4 dans le champ textuel et stocker cette valeur dans la variable utilisée à l'étape 2.

Nous allons programmer toutes ces actions dans la classe `MoteurCalcul`. En lisant le code ci-dessous, rappelle-toi que la méthode `actionPerformed()` est appelée après chaque clic sur un bouton et qu'entre ces appels de méthode les données sont stockées dans les variables `actionSélectionnée` et `résultatCourant`.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com


```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

import java.text.NumberFormat;
import java.text.ParsePosition;

import javax.swing.JButton;

public class MoteurCalcul implements ActionListener {

  Calculatrice parent; // une référence à la Calculatrice
  char actionSélectionnée = ' ';  // +, -, /, ou *

  double résultatCourant = 0;

  NumberFormat formatNombres = NumberFormat.getInstance();
      // un objet capable de lire et présenter les nombres

  // Le constructeur stocke la référence à la fenêtre
  // Calculatrice dans la variable membre parent
  MoteurCalcul(Calculatrice parent) {
    this.parent = parent;
  }

  public void actionPerformed(ActionEvent événement) {

    // Retrouve la source de l'action
    JButton boutonCliqué = (JButton) événement.getSource();
    String texteChampAffichage =
                           parent.champAffichage.getText();

    double valeurAffichée = 0;

    // Retrouve le nombre présenté dans le champ texte
    // s'il n'est pas vide
    if (!"".equals(texteChampAffichage)) {
      valeurAffichée =
        // analyse la chaîne de caractères
        formatNombres.parse(
          texteChampAffichage,
          new ParsePosition(0) /* ne sert pas */).
          // puis donne sa valeur en tant que double
          doubleValue();
    }
    Object sourceEvénement = événement.getSource();

    // Pour chaque bouton d'action, mémorise l'action
    // sélectionnée, +, -, /, ou *, stocke la valeur courante
    // dans la variable résultatCourant et vide le champ
    // Affichage avant l'entrée du nombre suivant
```

Classe `MoteurCalcul` (partie 1 de 2)

```
    if (sourceEvénement == parent.boutonPlus) {
      actionSélectionnée = '+';
      résultatCourant = valeurAffichée;
      parent.champAffichage.setText("");
    }
    else if (sourceEvénement == parent.boutonMoins) {
      actionSélectionnée = '-';
      résultatCourant = valeurAffichée;
      parent.champAffichage.setText("");
    }
    else if (sourceEvénement == parent.boutonDiviser) {
      actionSélectionnée = '/';
      résultatCourant = valeurAffichée;
      parent.champAffichage.setText("");
    }
    else if (sourceEvénement == parent.boutonMultiplier) {
      actionSélectionnée = '*';
      résultatCourant = valeurAffichée;
      parent.champAffichage.setText("");
    }
    else if (sourceEvénement == parent.boutonEgale) {
    // Effectue les calculs en fonction de actionSélectionnée
    // Modifie la valeur de la variable résultatCourant
    // et affiche le résultat
      if (actionSélectionnée == '+') {
        résultatCourant += valeurAffichée;
        // Convertit le résultat en le transformant en String
        // à l'aide de formatNombres
        parent.champAffichage.setText(
                      formatNombres.format(résultatCourant));
      }
      else if (actionSélectionnée == '-') {
        résultatCourant -= valeurAffichée;
        parent.champAffichage.setText(
                      formatNombres.format(résultatCourant));
      }
      else if (actionSélectionnée == '/') {
        résultatCourant /= valeurAffichée;
        parent.champAffichage.setText(
                      formatNombres.format(résultatCourant));
      }
      else if (actionSélectionnée == '*') {
        résultatCourant *= valeurAffichée;
        parent.champAffichage.setText(
                      formatNombres.format(résultatCourant));
      }
    }
    else {
      // Pour tous les boutons numériques, ajoute le libellé
      // du bouton au champ texte
      String libelléBoutonCliqué = boutonCliqué.getText();
      parent.champAffichage.setText(texteChampAffichage +
                                          libelléBoutonCliqué);
    }
  }
}
```

Classe `MoteurCalcul` (partie 2 de 2)

La version finale de la fenêtre de la calculatrice ressemblera à ceci :

La classe `Calculatrice` effectue les étapes suivantes :

1. Créer et afficher tous les composants de la fenêtre.
2. Créer une instance du récepteur d'événements `MoteurCalcul`.
3. Passer au moteur une référence à elle-même.
4. Enregistrer le moteur en tant que récepteur auprès de tous les composants qui peuvent générer des événements.

Voici la version finale de la classe `Calculatrice` :

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.BorderLayout;

public class Calculatrice {
  // Déclare et instancie les composants de la fenêtre
  JButton bouton0 = new JButton("0");
  JButton bouton1 = new JButton("1");
  JButton bouton2 = new JButton("2");
  JButton bouton3 = new JButton("3");
  JButton bouton4 = new JButton("4");
  JButton bouton5 = new JButton("5");
  JButton bouton6 = new JButton("6");
  JButton bouton7 = new JButton("7");
  JButton bouton8 = new JButton("8");
  JButton bouton9 = new JButton("9");
  JButton boutonVirgule = new JButton(",");
  JButton boutonEgale = new JButton("=");
  JButton boutonPlus = new JButton("+");
  JButton boutonMoins = new JButton("-");
```

Classe `Calculatrice` (partie 1 de 3)

```
  JButton boutonDiviser = new JButton("/");
  JButton boutonMultiplier = new JButton("*");
  JPanel contenuFenêtre = new JPanel();
  JTextField champAffichage = new JTextField(30);

  // Constructeur
  Calculatrice() {
    // Affecte le gestionnaire de disposition pour ce panneau
    BorderLayout disposition = new BorderLayout();
    contenuFenêtre.setLayout(disposition);

    // Ajoute le champ d'affichage en haut de la fenêtre
    contenuFenêtre.add("North", champAffichage);

    // Crée le panneau avec le quadrillage qui contient
    // 12 boutons - les 10 boutons numériques et ceux
    // représentant la virgule et le signe égale

    JPanel panneauChiffres = new JPanel();
    GridLayout dispositionChiffres = new GridLayout(4, 3);
    panneauChiffres.setLayout(dispositionChiffres);

    panneauChiffres.add(bouton1);
    panneauChiffres.add(bouton2);
    panneauChiffres.add(bouton3);
    panneauChiffres.add(bouton4);
    panneauChiffres.add(bouton5);
    panneauChiffres.add(bouton6);
    panneauChiffres.add(bouton7);
    panneauChiffres.add(bouton8);
    panneauChiffres.add(bouton9);
    panneauChiffres.add(bouton0);
    panneauChiffres.add(boutonVirgule);
    panneauChiffres.add(boutonEgale);

    // Ajoute le panneau des chiffres à la zone centrale
    // de la fenêtre
    contenuFenêtre.add("Center", panneauChiffres);

    // Crée le panneau avec le quadrillage qui contient 4
    // boutons d'opération - Plus, Moins, Diviser, Multiplier
    JPanel panneauOpérations = new JPanel();
    GridLayout dispositionOpérations = new GridLayout(4, 1);
    panneauOpérations.setLayout(dispositionOpérations);
    panneauOpérations.add(boutonPlus);
    panneauOpérations.add(boutonMoins);
    panneauOpérations.add(boutonMultiplier);
    panneauOpérations.add(boutonDiviser);
```

Classe `Calculatrice` (partie 2 de 3)

```
    // Ajoute le panneau des opérations à la zone est
    // de la fenêtre
    contenuFenêtre.add("East", panneauOpérations);

    // Crée le cadre et lui affecte son contenu
    JFrame frame = new JFrame("Calculatrice");
    frame.setContentPane(contenuFenêtre);

    // Affecte à la fenêtre des dimensions suffisantes pour
    // prendre en compte tous les contrôles
    frame.pack();

    // Affiche la fenêtre
    frame.setVisible(true);

    // Instancie le récepteur d'événements et l'enregistre
    // auprès de chaque bouton
    MoteurCalcul moteurCalcul = new MoteurCalcul(this);

    bouton0.addActionListener(moteurCalcul);
    bouton1.addActionListener(moteurCalcul);
    bouton2.addActionListener(moteurCalcul);
    bouton3.addActionListener(moteurCalcul);
    bouton4.addActionListener(moteurCalcul);
    bouton5.addActionListener(moteurCalcul);
    bouton6.addActionListener(moteurCalcul);
    bouton7.addActionListener(moteurCalcul);
    bouton8.addActionListener(moteurCalcul);
    bouton9.addActionListener(moteurCalcul);

    boutonVirgule.addActionListener(moteurCalcul);
    boutonPlus.addActionListener(moteurCalcul);
    boutonMoins.addActionListener(moteurCalcul);
    boutonDiviser.addActionListener(moteurCalcul);
    boutonMultiplier.addActionListener(moteurCalcul);
    boutonEgale.addActionListener(moteurCalcul);
  }

  public static void main(String[] args) {
   // Instancie la classe Calculatrice
   Calculatrice calc = new Calculatrice();
  }
}
```

Classe `Calculatrice` (partie 3 de 3)

Maintenant, compile le projet et exécute la classe `Calculatrice`. Elle marche presque comme les calculatrices du monde réel.

Félicitations ! C'est ton premier programme utilisable par plein d'autres personnes – fais-en cadeau à tes amis.

Pour mieux comprendre comment fonctionne ce programme, je te recommande de te familiariser avec le *déboguage* (*debugging*) de

programmes. Je te prie de lire l'Annexe B sur le débogueur et de revenir ici ensuite.

Autres récepteurs d'événements

Il y a dans le paquetage `java.awt` d'autres récepteurs Java qu'il est bon de connaître :

- Le récepteur d'activation (`FocusListener`) envoie un signal à ta classe quand un composant devient actif ou inactif. Par exemple, on dit qu'un champ textuel est actif si il a un curseur clignotant.
- Le récepteur de changement d'élément (`ItemListener`) réagit à la sélection d'éléments dans une liste ou une liste déroulante (*combobox*).
- Le récepteur de touche (`KeyListener`) répond à la frappe des touches du clavier.
- Le récepteur de souris (`MouseListener`) réagit si on clique sur la souris ou si le pointeur entre ou sort de la surface d'un composant de la fenêtre.
- Le récepteur de mouvement de souris (`MouseMotionListener`) indique les déplacements ou les glissements de la souris. *Glisser* (*to drag*) signifie déplacer la souris tout en maintenant son bouton enfoncé.
- Le récepteur de fenêtre (`WindowListener`) te prévient lorsque l'utilisateur ouvre, ferme, iconise ou active une fenêtre.

La table suivante indique pour chaque récepteur le nom de son interface et les méthodes qu'elle déclare.

Interface	Méthodes à implanter
`FocusListener`	`focusGained(FocusEvent)` `focusLost(FocusEvent)`
`ItemListener`	`itemStateChanged(ItemEvent)`

`KeyListener`	`keyPressed(KeyEvent)`
	`keyReleased(KeyEvent)`
	`keyTyped(KeyEvent)`

`MouseListener`	`mouseClicked(MouseEvent)`
	`mouseEntered(MouseEvent)`
	`mouseExited(MouseEvent)`
	`mousePressed(MouseEvent)`
	`mouseReleased(MouseEvent)`

`MouseMotionListener`	`mouseDragged(MouseEvent)` `mouseMoved(MouseEvent)`

`WindowListener`	`windowActivated(WindowEvent)`
	`windowClosed(WindowEvent)`
	`windowClosing(WindowEvent)`
	`windowDeactivated (WindowEvent)`

	windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)

Par exemple, l'interface `FocusListener` déclare deux méthodes : `focusGained()` et `focusLost()`. Cela implique que, même si ta classe n'est intéressée que par le traitement des événements correspondant à l'activation d'un champ particulier – et n'a donc besoin d'implanter que `focusGained()`, tu dois aussi inclure la méthode vide `focusLost()`. Cela peut être pénible, mais Java fournit des *classes adapteurs* (*adapter classes*) spéciales pour chaque récepteur afin de faciliter le traitement des événements.

Utilisation des adapteurs

Disons que tu as besoin d'enregistrer des données sur le disque quand l'utilisateur ferme la fenêtre. Selon la table ci-dessus, la classe qui implémente l'interface `WindowsListener` doit inclure sept méthodes. C'est-à-dire que tu dois écrire le code de la méthode `windowClosing()` et en outre inclure six méthodes vides.

Le paquetage `java.awt` fournit des adapteurs, qui sont des classes qui implantent déjà toutes les méthodes requises (les corps de ces méthodes sont vides). L'une de ces classes est nommée `WindowAdapter`. Tu peux créer la classe qui doit traiter les événements en héritant de `WindowAdapter` et en surchargeant juste les méthodes qui t'intéressent, comme la méthode `windowClosing()`.

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class MonProcesseurEvénements extends WindowAdapter {
  public void windowClosing(WindowEvent événement) {
    // Insère ici le code qui enregistre
    // les données sur disque
  }
}
```

Le reste est facile – enregistre simplement cette classe comme récepteur d'événements dans la classe de la fenêtre :

```
MonProcesseurEvénements monRécepteur =
                                   new MonProcesseurEvénements();
addWindowListener(monRécepteur);
```

On peut obtenir le même résultat en utilisant ce qu'on appelle des *classes internes anonymes* (*anonymous inner classes*), mais c'est un sujet un peu trop compliqué pour ce livre.

Autres lectures

Ecriture de récepteurs d'événements :

http://java.sun.com/docs/books/tutorial/uiswing/events/

Exercices

Essaie de diviser un nombre par zéro à l'aide de notre calculatrice – le champ textuel affiche un signe ∞ qui veut dire "infini". Modifie la classe `MoteurCalcul` pour afficher le message "Impossible de diviser par zéro" si l'utilisateur tente une division par zéro.

Exercices pour les petits malins

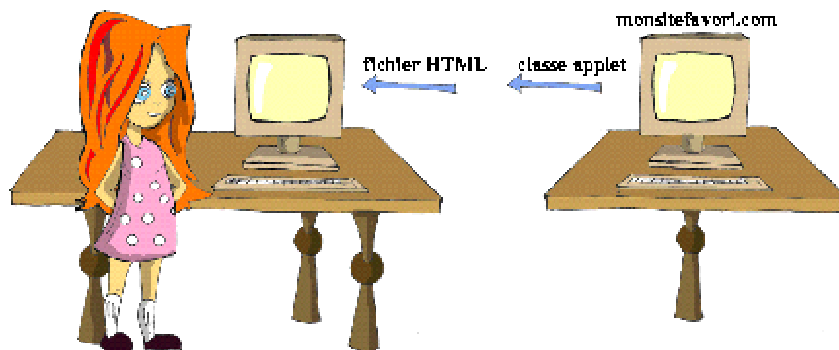
Modifie la classe `MoteurCalcul` pour qu'on ne puisse plus saisir plus d'une virgule dans le nombre.

Indice : jette un œil à la méthode `indexOf()` de la classe `String` ; elle te permettra de vérifier si le champ contient déjà une virgule.

Chapitre 7. L'applet Morpion

Quand tu es en ligne sur ton site web favori, il y a des chances pour que certains des jeux et autres programmes qui te sont proposés soient écrits en Java à l'aide de ce qu'on appelle des *applets* ou, parfois, des appliquettes. Ces applications spéciales existent et s'exécutent dans la fenêtre du navigateur web. Les navigateurs web comprennent un langage simple appelé HTML, qui te permet d'insérer des marques spéciales, ou balises (*tags*), dans les fichiers texte, afin de leur donner une apparence agréable lorsqu'ils sont affichés par les navigateurs. En plus du texte, tu peux inclure dans un fichier HTML la balise `<applet>`, qui indique au navigateur où trouver une applet Java et comment l'afficher.

Les applets Java sont téléchargées sur ton ordinateur depuis Internet en tant que parties d'une page web. Le navigateur est suffisamment malin pour lancer son propre Java afin de les exécuter.



Dans ce chapitre, tu vas apprendre comment créer des applets sur ton ordinateur. L'annexe C explique comment publier tes pages web sur Internet pour que d'autres personnes puissent les utiliser.

Les gens naviguent sur Internet sans savoir si les pages web contiennent des applets Java ou pas, mais veulent être certains que leurs ordinateurs sont à l'abri des dégâts que pourraient causer des personnes malintentionnées en incorporant à une page une applet

nuisible. C'est pourquoi les applets ont été conçues avec les limitations suivantes :

- Les applets ne peuvent pas accéder aux fichiers de ton disque dur, à moins que tu n'aies sur ton disque un fichier spécial, appelé *certificat*, qui leur en donne l'autorisation.
- Les applets ne peuvent se connecter qu'à l'ordinateur duquel elles ont été téléchargées.
- Les applets ne peuvent démarrer aucun des programmes situés sur ton disque dur.

Pour exécuter une applet, il te faut une classe Java écrite d'une certaine manière, un fichier texte HTML contenant la balise `<applet>` pointant sur cette classe et un navigateur web qui supporte Java. Tu peux aussi tester les applets dans Eclipse ou à l'aide d'un programme spécial appelé *visualisateur d'applet* (*appletviewer*). Mais avant d'apprendre à créer des applets, prenons 15 minutes pour nous familiariser avec quelques balises HTML.

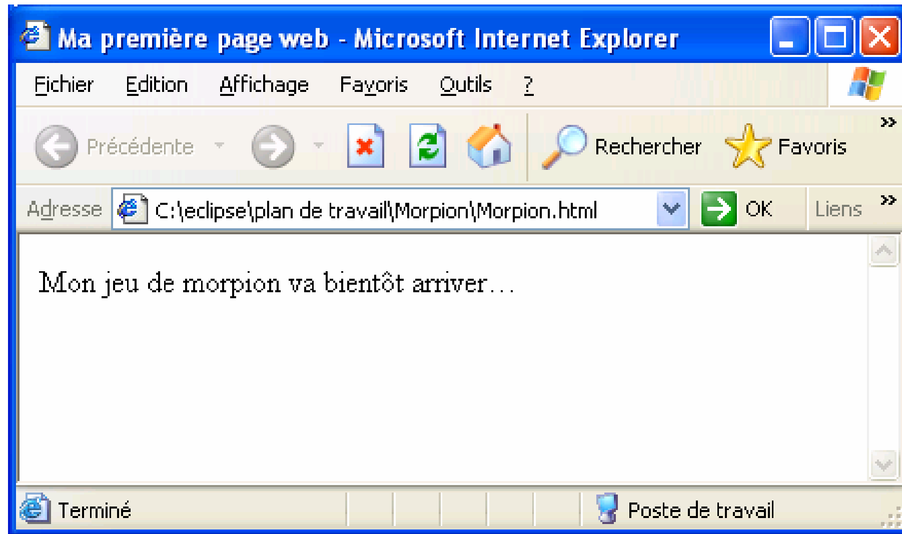
Apprendre HTML en 15 minutes

Imagine un instant que tu aies écrit et compilé une applet de jeu appelée `Morpion`. Tu dois maintenant créer le fichier HTML contenant les informations la concernant. Crée tout d'abord le fichier texte `Morpion.html` (à propos, Eclipse peut aussi créer des fichiers texte). Les fichiers HTML ont des noms terminés soit par *.html* soit par *.htm*. Ils contiennent en général les sections *header* (en-tête) et *body* (corps). La plupart des balises HTML ont une balise fermante correspondante, commençant par une barre oblique. Par exemple : `<Head>` et `</Head>`. Voici à quoi pourrait ressembler le fichier `Morpion.html` :

```
<HTML>
<Head>
<Title>Ma première page web</Title>
</Head>
<BODY>
  Mon jeu de morpion va bientôt arriver…
</BODY>
</HTML>
```

Les balises peuvent être placées soit sur la même ligne, comme nous l'avons fait avec les balises `<Title>` et `</Title>`, soit sur des lignes distinctes. Ouvre ce fichier avec ton navigateur web à l'aide des menus *Fichier* et *Ouvrir*. La barre de titre bleue de la fenêtre affiche "Ma

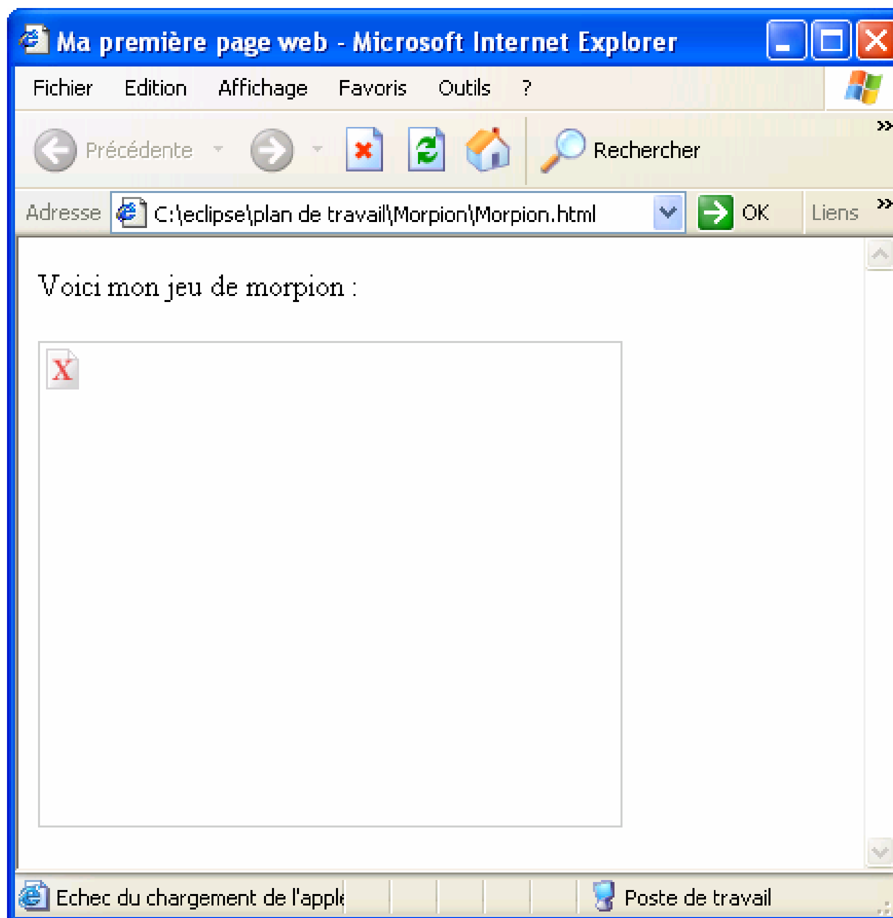
première page web" et à l'intérieur de la page sont affichés les mots "Mon jeu de morpion va bientôt arriver…" :



Modifie maintenant ce fichier en ajoutant la balise pour l'applet `Morpion` :

```
<HTML>
<Head>
<Title>Ma première page web</Title>
</Head>
<BODY>
  Voici mon jeu de morpion :
  <p><APPLET code = "Morpion.class" width=300
                                    height=250>
  </APPLET>
</BODY>
</HTML>
```

L'écran n'a plus la même apparence :



Evidemment, puisque le navigateur web ne peut pas trouver `Morpion.class`, il affiche simplement un rectangle vide. Nous créerons cette classe un peu plus loin dans ce chapitre.

Les balises HTML sont entourées par des signes inférieur (<) et supérieur (>). Certaines balises peuvent avoir des *attributs* supplémentaires. La balise `<APPLET>` de notre exemple utilise les attributs suivants :

- `code` : nom de la classe Java de l'applet.
- `width` : largeur en *pixels* de la surface rectangulaire de l'écran qui sera occupée par l'applet.
- `height` : hauteur de la surface occupée par l'applet.

Si une applet Java est constituée de plusieurs classes, rassemble-les dans un fichier archive à l'aide du programme *jar* fourni avec J2SDK. Dans ce cas, l'attribut `archive` doit préciser le nom de cette archive. L'annexe A propose des lectures relatives aux jars.

Choix de la librairie AWT pour écrire des applets

Pourquoi utiliser AWT pour écrire des applets si la librairie Swing est meilleure ? Est-il possible d'écrire des applets à l'aide de classes Swing ? Oui, c'est possible ; mais il faut savoir ceci.

Les navigateurs web sont fournis avec leur propre Java, qui supporte AWT, mais pas nécessairement les classes Swing incluses dans ton applet. Bien sûr, les utilisateurs peuvent télécharger et installer le dernier Java et il existe des convertisseurs HTML spéciaux permettant de modifier le fichier HTML pour faire pointer les navigateurs vers ce nouveau Java ; mais souhaites-tu vraiment demander aux utilisateurs de faire tout ça ? Une fois ta page web publiée sur Internet, tu ne sais pas qui peut l'utiliser. Imagine un vieux type, quelque part dans un désert avec un ordinateur vieux de 10 ans ; il laissera simplement tomber ta page, plutôt que de se lancer dans tous ces soucis d'installation. Imagine que notre applet aide à vendre des jeux en ligne et que nous ne voulions pas abandonner cette personne : elle pourrait être un client potentiel (les gens vivant dans le désert ont aussi des cartes de crédit). ☺

Utilise AWT si tu n'es pas sûr du type de navigateur web de tes utilisateurs.

En réalité, l'autre option consiste à demander aux utilisateurs de télécharger un *plugin* (module d'extension) Java spécial et de configurer leur navigateur pour utiliser ce plugin à la place du Java fourni avec leur navigateur. Tu trouveras plus d'explications à ce sujet sur ce site :
http://java.sun.com/j2se/1.5.0/docs/guide/plugin/.

Comment écrire des applets AWT

Les applets Java AWT doivent hériter de la classe `java.applet.Applet`. Par exemple :

```
class Morpion extends java.applet.Applet {

}
```

Contrairement aux applications Java, les applets n'ont pas besoin de méthode `main()` car le navigateur web les *télécharge* et les exécute dès qu'il rencontre la balise `<applet>` dans une page web. Le navigateur envoie aussi des signaux aux applets quand des événements importants se produisent, par exemple quand l'applet est lancée, est repeinte, etc. Pour être sûr que l'applet réagisse à ces événements, tu dois programmer des méthodes de *rappel* (*callback methods*) spéciales : `init()` (initialiser), `start()` (démarrer), `paint()` (peindre), `stop()` (arrêter) et `destroy()` (détruire). Le Java du navigateur appelle ces méthodes dans les cas suivants :

- `init()` est appelée quand l'applet est chargée par le navigateur. Elle n'est appelée qu'une fois : elle joue donc un rôle similaire à celui des constructeurs des classes Java normales.

- `start()` est appelée juste après `init()`. Elle est aussi appelée si l'utilisateur revient à une page web après avoir visité une autre page.

- `paint()` est appelée quand la fenêtre de l'applet a besoin d'être affichée ou rafraîchie après une activité quelconque sur l'écran. Par exemple, l'applet a été recouverte par une autre fenêtre et le navigateur a besoin de la repeindre.

- `stop()` est appelée quand l'utilisateur quitte la page web contenant l'applet.

- `destroy()` est appelée quand le navigateur détruit l'applet. Tu n'as besoin d'écrire du code pour cette méthode que si l'applet utilise d'autres ressources, par exemple quand elle maintient une connexion avec l'ordinateur duquel elle a été téléchargée.

Même s'il n'est pas nécessaire de programmer toutes ces méthodes, chaque applet doit définir au moins `init()` et `paint()`. Voici le code d'une applet qui affiche les mots "Bonjour Monde !". Cette applet ne contient que la méthode `paint()`, qui reçoit une instance de l'objet `Graphics` du Java du navigateur. Cet objet dispose d'un tas de

méthodes graphiques. L'exemple suivant utilise la méthode `drawString()` (dessiner une chaîne de caractères) pour dessiner le texte "Bonjour Monde !".

```
public class AppletBonjour extends java.applet.Applet {
  public void paint(java.awt.Graphics contexteGraphique) {
    contexteGraphique.drawString("Bonjour Monde !", 70, 40);
  }
}
```

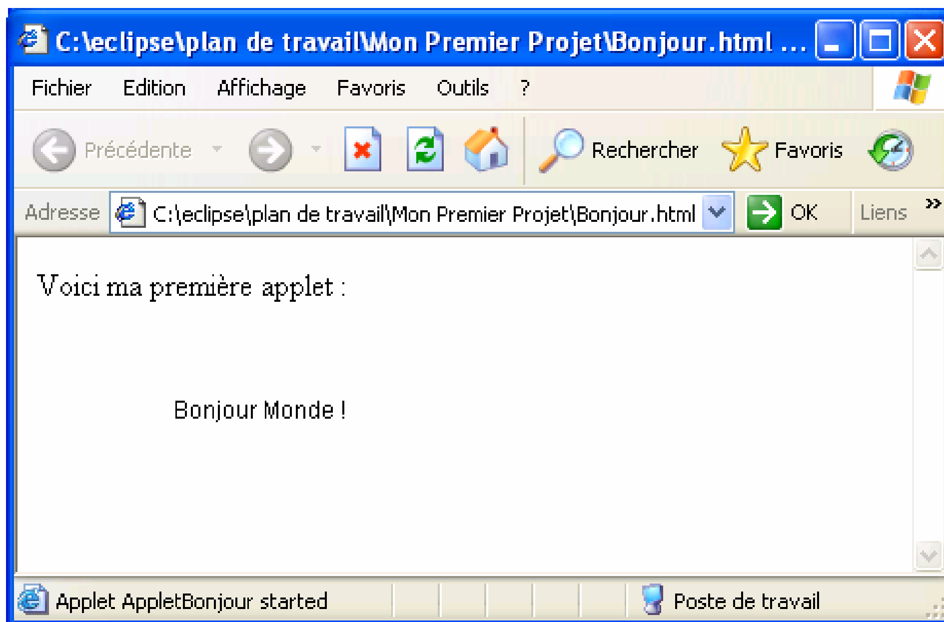
Crée cette classe dans Eclipse. Ensuite, dans la fenêtre *Exécuter*, sélectionne *Applet Java* dans le coin supérieur gauche, appuie sur le bouton *Créer* et entre `AppletBonjour` dans le champ *Applet*.

Pour tester cette applet dans le navigateur web, crée le fichier `Bonjour.html` dans le dossier où est enregistrée ton applet.

```
<HTML>
 <BODY>
  Voici ma première applet :<P>
 <APPLET code = AppletBonjour.class width = 200 height = 100>
 </APPLET>
 </BODY>
</HTML>
```

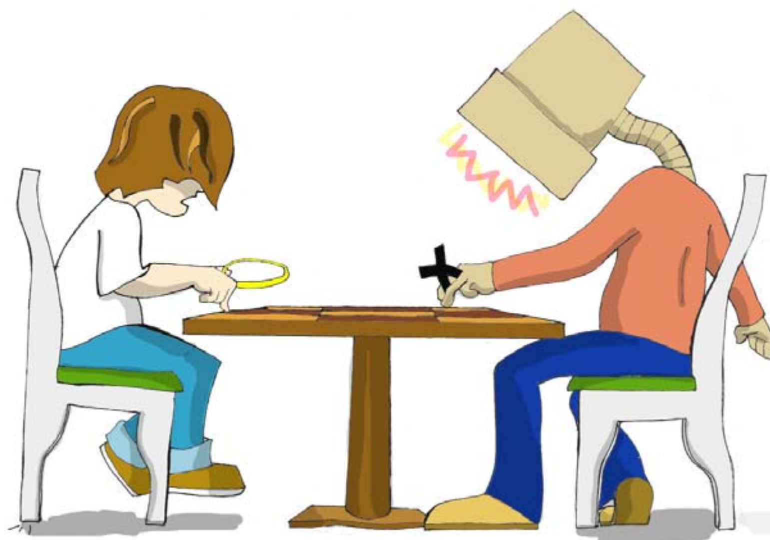
Ouvre maintenant le fichier `Bonjour.html` à l'aide des menus *Fichier* et *Ouvrir* de ton navigateur.

L'écran devrait ressembler à ceci :



Crois-tu qu'après ce simple exemple nous soyons prêts pour écrire un programme de jeu ? Tu m'étonnes ! Boucle juste ta ceinture…

Ecriture d'un jeu de morpion



Stratégie

Tous les jeux utilisent un algorithme – un ensemble de règles ou une stratégie à appliquer en fonction des actions du joueur. Pour un même jeu, il peut y avoir des algorithmes simples ou très compliqués. Quand on dit que le champion du monde d'échecs Gary Kasparov joue contre un ordinateur, il joue en fait contre un programme. Des équipes

d'experts essaient d'inventer des algorithmes sophistiqués pour le battre. Le jeu de morpion peut aussi être programmé selon différentes stratégies ; nous allons en utiliser la plus simple :

1. Nous utilisons un plateau 3 x 3.
2. L'utilisateur joue avec le symbole *X* et l'ordinateur avec le *O*.
3. Pour gagner, il faut avoir complété une ligne, une colonne ou une diagonale avec le même symbole.
4. Après chaque coup, le programme vérifie s'il y a un gagnant.
5. S'il y a un gagnant, la combinaison gagnante est mise en surbrillance et la partie se termine.
6. La partie se termine aussi s'il n'y a plus de cases libres.
7. Pour faire une nouvelle partie, le joueur appuie sur le bouton *Nouvelle partie*.
8. Quand l'ordinateur décide où placer le prochain *O*, il essaie de trouver une ligne, une colonne ou une diagonale où se trouvent déjà deux *O* pour la compléter.
9. S'il n'y pas deux *O* alignés, l'ordinateur essaie de trouver deux *X* alignés afin de parer un coup gagnant du joueur en plaçant un *O*.
10. Si aucun coup gagnant ou bloquant n'a été trouvé, l'ordinateur essaie d'occuper la case centrale, ou choisit la prochaine case vide *au hasard*.

Code

Je vais juste te donner ici une brève description du programme car le code de l'applet contient de nombreux commentaires qui t'aideront à comprendre comment il fonctionne.

L'applet utilise un gestionnaire de disposition `BorderLayout`.
La zone nord de la fenêtre contient le bouton *Nouvelle partie*.
La zone centrale affiche neuf boutons représentant les cases et la zone sud affiche les messages :

Tous les composants de la fenêtre sont créés par la méthode `init()` de l'applet. Tous les événements sont traités par le récepteur `ActionListener` dans la méthode `actionPerformed()`. La méthode `chercherUnGagnant()` est appelée après chaque coup pour vérifier si la partie est terminée.

Les règles 8, 9 et 10 de notre stratégie sont codées dans la méthode `coupOrdinateur()`, qui peut avoir besoin de générer un *nombre aléatoire* (*random number*). On utilise pour cela la classe Java `Math` et sa méthode `random()`.

Tu rencontreras aussi une syntaxe assez inhabituelle où plusieurs appels de méthodes sont effectués en une seule *expression*, comme dans cet exemple :

```
if(cases[0].getLabel().equals(cases[1].getLabel())){…}
```

Cette ligne raccourcit le code car elle effectue en fait les mêmes actions que l'ensemble des lignes suivantes :

```
String label0 = cases[0].getLabel();
String label1 = cases[1].getLabel();
if(label0.equals(label1)){…}
```

Dans les expressions complexes, Java évalue le code entre parenthèses avant d'effectuer d'autres calculs. La version courte de ce code cherche d'abord le résultat de l'expression entre parenthèses, puis l'utilise comme argument de la méthode `equals()`, qui est appliquée au résultat du premier appel à `getLabel()`.

Même si le code du jeu occupe plusieurs pages, il n'est pas très difficile à comprendre. Lis simplement tous les commentaires qui s'y trouvent.

```
/**
 * Un jeu de morpion sur un plateau 3x3
 */

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Morpion extends Applet implements
                                        ActionListener {

  Button cases[];
  Button boutonNouvellePartie;
  Label score;
  int casesLibresRestantes = 9;

  /**
   * La méthode init est comme un constructeur pour l'applet
   */
  public void init() {

    // Affecte le gestionnaire de disposition et la couleur
    // de l'applet
    this.setLayout(new BorderLayout());
    this.setBackground(Color.CYAN);

    // Passe la police de l'applet en style gras et taille 20
    Font policeApplet = new Font("Monospaced", Font.BOLD, 20);
    this.setFont(policeApplet);

    // Crée le bouton Nouvelle partie et enregistre
    // le récepteur d'actions auprès de lui
    boutonNouvellePartie = new Button("Nouvelle partie");
    boutonNouvellePartie.addActionListener(this);

    // Crée deux panneaux et un label et les agence en
    // utilisant le border layout
    Panel panneauSupérieur = new Panel();
    panneauSupérieur.add(boutonNouvellePartie);
```

Classe `Morpion` (partie 1 de 8)

```
    this.add(panneauSupérieur, "North");

    Panel panneauCentral = new Panel();
    panneauCentral.setLayout(new GridLayout(3, 3));
    this.add(panneauCentral, "Center");

    score = new Label("A vous de jouer !");
    this.add(score, "South");

    // Crée un tableau pour stocker les références des
    // 9 boutons
    cases = new Button[9];

    // Instancie les boutons, stocke leurs références dans le
    // tableau, enregistre le récepteur auprès d'eux, peint
    // les boutons en orange et les ajoute au panneau central
    for(int i = 0; i < 9; i++) {
      cases[i]= new Button();
      cases[i].addActionListener(this);
      cases[i].setBackground(Color.ORANGE);
      panneauCentral.add(cases[i]);
    }
  }
  /**
  * Cette méthode traite tous les événements d'action
  * @param événement l'événement à traiter
  */
  public void actionPerformed(ActionEvent événement) {
    Button leBouton = (Button) événement.getSource();
    // S'agit-il du bouton Nouvelle partie ?
    if (leBouton == boutonNouvellePartie) {
      for(int i = 0; i < 9; i++) {
        cases[i].setEnabled(true);
        cases[i].setLabel("");
        cases[i].setBackground(Color.ORANGE);
      }
      casesLibresRestantes = 9;
      score.setText("A vous de jouer !");
      boutonNouvellePartie.setEnabled(false);
      return;  // Sort de la méthode
    }

    String gagnant = "";
```

Classe `Morpion` (partie 2 de 8)

```
    // S'agit-il de l'une des cases ?
    for (int i = 0; i < 9; i++) {
      if (leBouton == cases[i]) {
        cases[i].setLabel("X");
        gagnant = chercherUnGagnant();

        if(!"".equals(gagnant)) {
          terminerLaPartie();
        } else {
          coupOrdinateur();
          gagnant = chercherUnGagnant();
          if (!"".equals(gagnant)) {
            terminerLaPartie();
          }
        }
        break;
      }
    } // Fin de la boucle for

    if (gagnant.equals("X")) {
      score.setText("Vous avez gagné !");
    } else if (gagnant.equals("O")) {
      score.setText("Vous avez perdu !");
    } else if (gagnant.equals("T")) {
      score.setText("Partie nulle !");
    }
  } // Fin de actionPerformed

 /**
  *  Cette méthode est appelée après chaque coup joué pour
  *  voir s'il y a un gagnant. Elle vérifie pour chaque ligne,
  *  colonne et diagonale, s'il y a trois symboles identiques
  *  @return "X", "O", "T" (terminé, partie nulle) ou "" (pas
  *  fini)
  */
  String chercherUnGagnant() {

    String leGagnant = "";
    casesLibresRestantes--;
```

Classe `Morpion` (partie 3 de 8)

```
    // Vérifie la ligne 1 - éléments 0, 1 et 2 du tableau
    if (!cases[0].getLabel().equals("") &&
        cases[0].getLabel().equals(cases[1].getLabel()) &&
        cases[0].getLabel().equals(cases[2].getLabel())) {
      leGagnant = cases[0].getLabel();
      montrerGagnant(0, 1, 2);

    // Vérifie la ligne 2 - éléments 3, 4 et 5 du tableau
    } else if (!cases[3].getLabel().equals("") &&
        cases[3].getLabel().equals(cases[4].getLabel()) &&
        cases[3].getLabel().equals(cases[5].getLabel())) {
      leGagnant = cases[3].getLabel();
      montrerGagnant(3, 4, 5);

    // Vérifie la ligne 3 - éléments 6, 7 et 8 du tableau
    } else if (!cases[6].getLabel().equals("") &&
        cases[6].getLabel().equals(cases[7].getLabel()) &&
        cases[6].getLabel().equals(cases[8].getLabel())) {
      leGagnant = cases[6].getLabel();
      montrerGagnant(6, 7, 8);

    // Vérifie la colonne 1 - éléments 0, 3 et 6 du tableau
    } else if (!cases[0].getLabel().equals("") &&
        cases[0].getLabel().equals(cases[3].getLabel()) &&
        cases[0].getLabel().equals(cases[6].getLabel())) {
      leGagnant = cases[0].getLabel();
      montrerGagnant(0, 3, 6);

    // Vérifie la colonne 2 - éléments 1, 4 et 7 du tableau
    } else if (!cases[1].getLabel().equals("") &&
        cases[1].getLabel().equals(cases[4].getLabel()) &&
        cases[1].getLabel().equals(cases[7].getLabel())) {
      leGagnant = cases[1].getLabel();
      montrerGagnant(1, 4, 7);

    // Vérifie la colonne 3 - éléments 2, 5 et 8 du tableau
    } else if (!cases[2].getLabel().equals("") &&
        cases[2].getLabel().equals(cases[5].getLabel()) &&
        cases[2].getLabel().equals(cases[8].getLabel())) {
      leGagnant = cases[2].getLabel();
      montrerGagnant(2, 5, 8);
```

Classe `Morpion` (partie 4 de 8)

```
    // Vérifie la première diagonale - éléments 0, 4 et 8
    } else if (!cases[0].getLabel().equals("") &&
        cases[0].getLabel().equals(cases[4].getLabel()) &&
        cases[0].getLabel().equals(cases[8].getLabel())) {
      leGagnant = cases[0].getLabel();
      montrerGagnant(0, 4, 8);

    // Vérifie la seconde diagonale - éléments 2, 4 et 6
    } else if (!cases[2].getLabel().equals("") &&
        cases[2].getLabel().equals(cases[4].getLabel()) &&
        cases[2].getLabel().equals(cases[6].getLabel())) {
      leGagnant = cases[2].getLabel();
      montrerGagnant(2, 4, 6);
    } else if (casesLibresRestantes == 0) {
      return "T";  // Partie nulle
    }

    return leGagnant;
  }
  /**
   * Cette méthode applique un ensemble de règles afin de
   * trouver le meilleur coup pour l'ordinateur. Si un bon
   * coup ne peut être trouvé, elle choisit une case au
   * hasard.
   */
  void coupOrdinateur() {
    int caseSélectionnée;
    // L'ordinateur essaie d'abord de trouver une case
    // vide près de deux case marquées "O" pour gagner
    caseSélectionnée = trouverCaseVide("O");
    // S'il n'y a pas deux "O" alignés, essaie au moins
    // d'empêcher l'adversaire d'aligner trois "X" en
    // plaçant un "O" près de deux "X"
    if (caseSélectionnée == -1) {
      caseSélectionnée = trouverCaseVide("X");
    }
    // Si caseSélectionnée vaut toujours -1, essaie d'occuper
    // la case centrale
    if ((caseSélectionnée == -1)
          && (cases[4].getLabel().equals(""))) {
      caseSélectionnée = 4;
    }
    // Pas de chance avec la case centrale non plus...
    // Choisit une case au hasard
    if (caseSélectionnée == -1) {
      caseSélectionnée = choisirCaseAuHasard();
    }
    cases[caseSélectionnée].setLabel("O");
  }
```

Classe `Morpion` (partie 5 de 8)

```
/**
 * Cette méthode examine chaque ligne, colonne et diagonale
 * pour voir si elle contient deux cases avec le même label
 * et une case vide.
 * @param joueur "X" pour l'utilisateur ou "O" pour
 *               l'ordinateur
 * @return numéro de la case vide à utiliser ou le nombre
 *         négatif -1 si la recherche est infructueuse
 */
int trouverCaseVide(String joueur) {

  int poids[] = new int[9];

  for (int i = 0; i < 9; i++ ) {
    if (cases[i].getLabel().equals("O"))
      poids[i] = -1;
    else if (cases[i].getLabel().equals("X"))
      poids[i] = 1;
    else
      poids[i] = 0;
  }

 int deuxPoids = joueur.equals("O") ? -2 : 2;

 // Regarde si la ligne 1 a 2 cases identiques et une vide
 if (poids[0] + poids[1] + poids[2] == deuxPoids) {
   if (poids[0] == 0)
     return 0;
   else if (poids[1] == 0)
     return 1;
   else
     return 2;
 }
 // Regarde si la ligne 2 a 2 cases identiques et une vide
 if (poids[3] + poids[4] + poids[5] == deuxPoids) {
   if (poids[3] == 0)
     return 3;
   else if (poids[4] == 0)
     return 4;
   else
     return 5;
 }
```

Classe `Morpion` (partie 6 de 8)

```
    // Regarde si la ligne 3 a 2 cases identiques et une vide
    if (poids[6] + poids[7] + poids[8] == deuxPoids) {
      if (poids[6] == 0)
        return 6;
      else if (poids[7] == 0)
        return 7;
      else
        return 8;
    }
    // Regarde si la colonne 1 a 2 cases identiques et une vide
    if (poids[0] + poids[3] + poids[6] == deuxPoids) {
      if (poids[0] == 0)
        return 0;
      else if (poids[3] == 0)
        return 3;
      else
        return 6;
    }
    // Regarde si la colonne 2 a 2 cases identiques et une vide
    if (poids[1] + poids[4] + poids[7] == deuxPoids) {
      if (poids[1] == 0)
        return 1;
      else if (poids[4] == 0)
        return 4;
      else
        return 7;
    }
    // Regarde si la colonne 3 a 2 cases identiques et une vide
    if (poids[2] + poids[5] + poids[8] == deuxPoids) {
      if (poids[2] == 0)
        return 2;
      else if (poids[5] == 0)
        return 5;
      else
        return 8;
    }

    // Regarde si la diagonale 1 a 2 cases identiques et une
    // vide
    if (poids[0] + poids[4] + poids[8] == deuxPoids) {
      if (poids[0] == 0)
        return 0;
      else if (poids[4] == 0)
        return 4;
      else
        return 8;
    }
```

Classe `Morpion` (partie 7 de 8)

```
    // Regarde si la diagonale 2 a 2 cases identiques et une
    // vide
    if (poids[2] + poids[4] + poids[6] == deuxPoids) {
      if (poids[2] == 0)
        return 2;
      else if (poids[4] == 0)
        return 4;
      else
        return 6;
    }
    // Il n'y a pas de cases alignées identiques
    return -1;
  } // Fin de trouverCaseVide()
  /**
   * Cette méthode sélectionne une case vide quelconque.
   * @return un numéro de case choisi au hasard
   */
  int choisirCaseAuHasard() {
    boolean caseVideTrouvée = false;
    int caseSélectionnée = -1;

    do {
      caseSélectionnée = (int) (Math.random() * 9);
      if (cases[caseSélectionnée].getLabel().equals("")) {
        caseVideTrouvée = true; // Pour terminer la boucle
      }
    } while (!caseVideTrouvée);

    return caseSélectionnée;
  } // Fin de choisirCaseAuHasard()
  /**
   * Cette méthode affiche la ligne gagnante en surbrillance.
   * @param gagnante1 première case à montrer.
   * @param gagnante2 deuxième case à montrer.
   * @param gagnante3 troisième case à montrer.
   */
  void montrerGagnant(int gagnante1, int gagnante2, int
                           gagnante3) {
    cases[gagnante1].setBackground(Color.CYAN);
    cases[gagnante2].setBackground(Color.CYAN);
    cases[gagnante3].setBackground(Color.CYAN);
  }
  // Désactive les cases et active le bouton Nouvelle partie
  void terminerLaPartie() {
    boutonNouvellePartie.setEnabled(true);
    for (int i = 0; i < 9; i++) {
      cases[i].setEnabled(false);
    }
  }
} // Fin de la classe
```

Classe Morpion (partie 8 de 8)

Félicitations ! Tu as terminé ton premier jeu en Java.

Tu peux exécuter cette applet directement depuis Eclipse ou en ouvrant le fichier `Morpion.html` que nous avons créé au début de ce chapitre (copie simplement les fichiers `Morpion.html` et `Morpion.class` dans le même répertoire). Notre classe `Morpion` a un petit bogue (*bug*) - tu pourrais même ne pas t'en apercevoir, mais je suis sûr qu'il sera corrigé lorsque tu auras terminé le deuxième exercice ci-dessous.

Notre classe `Morpion` utilise une stratégie simple parce que notre objectif est juste d'apprendre à programmer. Mais si tu souhaites améliorer ce jeu, applique l'algorithme appelé *minimax* qui te permet de choisir le meilleur coup pour l'ordinateur. La description de cette stratégie n'est pas du ressort de ce livre, mais tu peux facilement la trouver sur le web.

Autres lectures

Applets Java :
http://java.sun.com/docs/books/tutorial/applet/

Classe Java `Math`
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html

Exercices

1. Ajoute au panneau supérieur de la classe `Morpion` deux champs affichant les nombres de victoires et de défaites. Pour ce faire, déclare deux variables de classe que tu incrémenteras à chaque victoire ou défaite. Le score doit être rafraîchi juste après que le programme ait affiché le message "Vous avez gagné" ou "Vous avez perdu".

2. Notre programme permet de cliquer sur une case qui est déjà marquée d'un *X* ou d'un *O*. C'est un bogue ! Le programme continue comme si on avait joué un coup valide. Modifie le code pour ignorer les clics sur des cases déjà marquées.

3. Ajoute la méthode `main()` à la classe `Morpion` pour qu'on puisse lancer le jeu non seulement comme une applet, mais aussi comme une application Java.

Exercices pour les petits malins

1. Réécris le `Morpion` en remplaçant le tableau à une dimension qui stocke les neuf boutons :

```
JButton cases[]
```

par un tableau à deux dimensions 3 x 3 :

```
JButton cases[][]
```

Va sur le web pour t'informer à propos des tableaux multidimensionnels.

Chapitre 8. Erreurs et exceptions

Imaginons que tu aies oublié une accolade fermante dans ton code Java. Il se produirait une erreur de compilation, que tu pourrais corriger facilement. Mais il se produit aussi ce qu'on appelle des *erreurs d'exécution* (*run-time errors*), lorsque ton programme cesse tout à coup de fonctionner correctement. Par exemple, une classe Java lit un fichier contenant les scores du jeu. Que se passe-t-il si quelqu'un a supprimé ce fichier ? Le programme s'écroule-t-il en affichant un message d'erreur déroutant, ou reste-t-il en vie en affichant un message compréhensible comme : "Cher ami, pour une raison inconnue il m'est impossible de lire le fichier scores.txt. Merci de vérifier que ce fichier existe." ? Il est préférable de créer des programmes capables de gérer les situations inhabituelles. Dans beaucoup de langages de programmation, le traitement des erreurs dépend de la bonne volonté du programmeur. Mais Java rend obligatoire l'inclusion de code de gestion des erreurs. A défaut, les programmes ne peuvent même pas être compilés.

En Java, les erreurs d'exécution sont appelées des *exceptions* et le traitement des erreurs est appelé *gestion des exceptions* (*exception handling*). Tu dois placer le code susceptible de produire des erreurs dans des blocs `try/catch` (*essayer/capturer*). C'est comme si tu disais ceci à Java : *Essaie de lire le fichier contenant les scores. Si quelque chose d'anormal se produit, capture l'erreur et exécute le code qui doit la gérer.*

```
try {
  fichierScores.read();
}
catch (IOException exception) {
  System.out.println(
  "Cher ami, je ne peux pas lire le fichier scores.txt");
}
```

Nous apprendrons comment travailler avec des fichiers au Chapitre 9, mais pour l'instant familiarise-toi avec l'expression *I/O* ou *input/output* (*entrée/sortie*). Les opérations de lecture et d'écriture (sur disque ou sur d'autres périphériques) sont appelées I/O ; ainsi,

IOException est une classe qui contient des informations relatives aux erreurs d'entrée/sortie.
Une méthode *lève une exception* en cas d'erreur. Différentes exceptions sont levées pour différents types d'erreurs. Si le bloc catch existe dans le programme pour un type particulier d'erreur, l'erreur est capturée et le programme se débranche sur le bloc catch pour exécuter le code qui s'y trouve. Le programme reste vivant et cette exception est considérée comme prise en charge.

L'instruction qui affiche le message dans le code ci-dessus n'est exécutée que dans le cas d'une erreur de lecture de fichier.

Lecture de la trace de la pile

S'il se produit une exception inattendue, non gérée par le programme, un message d'erreur multilignes est affiché à l'écran. Un tel message est appelé *trace de la pile* (*stack trace*). Si ton programme a appelé plusieurs méthodes avant de rencontrer un problème, la trace de la pile peut t'aider à suivre la trace de ton programme et à trouver la ligne qui a causé l'erreur.

Ecrivons le programme TestTracePile qui effectue volontairement une division par 0 (les numéros de ligne ne font pas partie du code).

```
1  class TestTracePile {
2    TestTracePile()
3    {
4      diviserParZéro();
5    }
6
7    int diviserParZéro()
8    {
9      return 25 / 0;
10   }
11
12   public static void main(String[] args)
13   {
14     new TestTracePile();
15   }
16 }
```

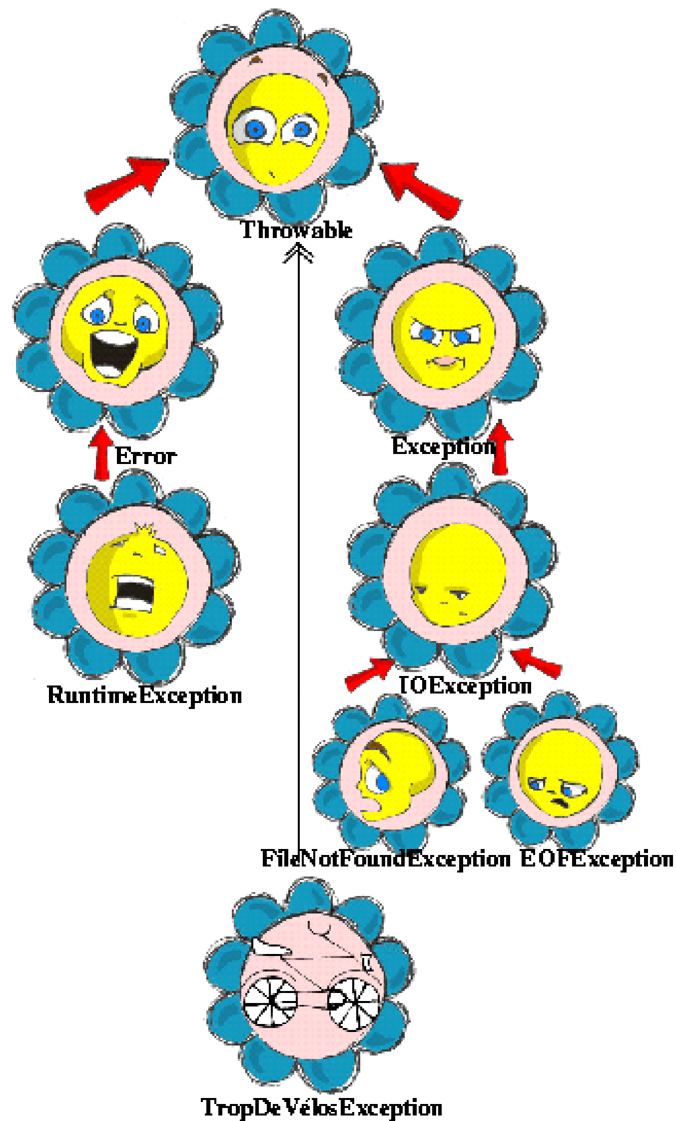
La sortie de ce programme montre la séquence des appels de méthodes effectués jusqu'au moment où l'erreur d'exécution s'est produite. Lis cette sortie en remontant à partir de la dernière ligne.

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
   at TestTracePile.diviserParZéro(TestTracePile.java:9)
   at TestTracePile.<init>(TestTracePile.java:4)
   at TestTracePile.main(TestTracePile.java:14)
```

Ceci signifie que le programme a commencé par la méthode `main()`, puis est entré dans `init()`, qui est un constructeur, et a ensuite appelé la méthode `diviserParZéro()`. Les nombres 14, 4 et 9 indiquent dans quelles lignes du programme ont eu lieu ces appels de méthodes. Enfin, une exception `ArithmeticException` a été levée : la ligne numéro 9 a essayé d'effectuer une division par 0.

Arbre généalogique des exceptions

En Java, les exceptions sont aussi des classes. Certaines d'entre elles apparaissent dans l'arbre d'héritage ci-dessous :



Les sous-classes de la classe `Exception` sont appelées *exceptions contrôlées* (*checked exceptions*) et tu dois les traiter dans ton code.

Les sous-classes de la classe `Error` sont des erreurs fatales et le programme en cours d'exécution ne peut en général pas les gérer.

`TropDeVélosException` est un exemple d'exception créée par un développeur.

Comment un développeur est-il sensé savoir à l'avance si une méthode Java peut lever une exception et qu'un bloc `try/catch` est nécessaire ? Pas de soucis ; si tu appelles une méthode qui peut lever une exception, le compilateur Java affiche un message d'erreur comme celui-ci :

```
"LecteurDeScore.java": unreported exception:
java.io.IOException; must be caught or declared to be thrown at
line 57
```

Bien sûr, tu es libre de lire la documentation Java qui décrit les exceptions qui peuvent être levées par une méthode donnée. La suite de ce chapitre explique comment gérer ces exceptions.

Bloc try/catch

Il y a cinq mot-clés Java qui peuvent être utilisés dans le traitement des erreurs : `try`, `catch`, `finally`, `throw` et `throws`.

Après un bloc `try`, tu peux mettre plusieurs blocs `catch`, si tu penses que différentes erreurs sont susceptibles de se produire. Par exemple, quand un programme essaie de lire un fichier, le fichier peut ne pas être là, ce qui génère l'exception `FileNotFoundException`, ou le fichier peut être là mais le code continuer à le lire après avoir atteint la fin du fichier, ce qui génère l'exception `EOFException`. L'extrait de code suivant affiche des messages en bon français si le programme ne trouve pas le fichier contenant les scores ou atteint prématurément la fin du fichier. Pour toute autre erreur, il affiche le message "Problème de lecture de fichier" et une description technique de l'erreur.

```
public void chargerScores() {
  try {
    fichierScores.read();
    System.out.println("Scores chargés avec succès");
  } catch(FileNotFoundException e) {
    System.out.println("Fichier Scores introuvable");
  } catch(EOFException e1) {
    System.out.println("Fin de fichier atteinte");
  } catch(IOException e2) {
    System.out.println("Problème de lecture de fichier" +
                                        e2.getMessage());
  }
}
```

Si la méthode read() (*lire*) échoue, le programme saute la ligne println() et essaie de trouver le bloc catch qui correspond à l'erreur. S'il le trouve, l'instruction println() appropriée est exécutée ; s'il n'y a pas de bloc catch correspondant à l'erreur, la méthode chargerScores() remonte l'exception à la méthode qui l'a appelée.

Si tu écris plusieurs blocs catch, tu dois faire attention à l'ordre dans lequel tu les écris si les exceptions que tu traites héritent les unes des autres. Par exemple, puisque EOFException est une sous-classe de IOException, tu dois placer le bloc catch de EOFException, la sous-classe, en premier. Si tu traitais IOException en premier, le programme n'atteindrait jamais les exceptions FileNotFound ou EOFException, car elles seraient interceptées par le premier catch.

Les fainéants pourraient programmer la méthode chargerScores() comme ceci :

```
public void chargerScores() {
  try {
    fichierScores.read();
  } catch(Exception e) {
    System.out.println("Problème de lecture du fichier "
                                         + e.getMessage());
  }
}
```

C'est un exemple de mauvais style de code Java. Quand tu écris un programme, n'oublie jamais que quelqu'un d'autre pourrait le lire ; et tu ne veux pas avoir honte de ton code.

Les blocs catch reçoivent une instance de l'objet Exception contenant une courte explication du problème. La méthode getMessage() retourne cette information. Parfois, lorsque la description d'une erreur n'est pas claire, tu peux essayer la méthode toString() à la place :

```
catch(Exception exception) {
  System.out.println("Problème de lecture du fichier " +
                                          exception.toString());
}
```

Si tu as besoin d'informations plus détaillées à propos de l'exception, utilise la méthode printStackTrace(). Elle affiche la séquence d'appels de méthodes qui a abouti à l'exception, de la même façon que dans l'exemple de la section *Lecture de la trace de la pile*.

Essayons de "tuer" le programme de calculatrice du Chapitre 6. Exécute la classe Calculatrice et entre au clavier les caractères abc. Appuie sur n'importe lequel des boutons d'opérations ; la console affiche quelque chose de ce genre :

```
Exception in thread "AWT-EventQueue-0"
java.lang.NullPointerException
      at MoteurCalcul.actionPerformed(MoteurCalcul.java:43)
      at javax.swing.AbstractButton.fireActionPerformed(AbstractB
utton.java:1849)
      at javax.swing.AbstractButton$Handler.actionPerformed(Abstr
actButton.java:2169)
      at javax.swing.DefaultButtonModel.fireActionPerformed(Defau
ltButtonModel.java:420)
```

C'est un exemple d'exception non gérée. Dans la méthode actionPerformed() de la classe MoteurCalcul, il y a ces lignes :

```
valeurAffichée =
  // analyse la chaîne de caractères
  formatNombres.parse(
    texteChampAffichage,
    new ParsePosition(0) /* ne sert pas */).
    // puis donne sa valeur en tant que double
   doubleValue();
```

Si la variable texteChampAffichage ne représente pas une valeur numérique, la méthode parse() est incapable de la convertir dans un nombre et rend null. Du coup l'appel de la méthode doubleValue() lève une exception NullPointerException.

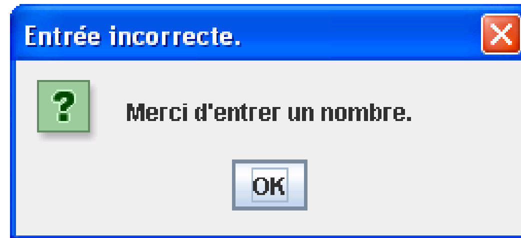
Gérons cette exception et affichons un message d'erreur qui explique le problème à l'utilisateur. Les lignes qui commencent par valeurAffichée doivent être placées dans un bloc try/catch ; Eclipse va t'y aider. Sélectionne tout le texte depuis valeurAffichée jusqu'au point-virgule après doubleValue()et clique sur le bouton droit de la souris. Dans le menu popup, sélectionne les sous-menus *Source* et *Entourer d'un bloc try/catch*. Réponds *Oui* à la question que te pose Eclipse et voilà ! Le code est modifié :

```
try {
  valeurAffichée =
    // analyse la chaîne de caractères
    formatNombres.parse(
      texteChampAffichage,
      new ParsePosition(0) /* ne sert pas */).
        // puis donne sa valeur en tant que double
        doubleValue();
} catch (RuntimeException e) {
  // TODO Bloc catch auto-généré
  e.printStackTrace();
}
```

Remplace la ligne `e.printStackTrace();` par ceci :

```
javax.swing.JOptionPane.showConfirmDialog(null,
   "Merci d'entrer un nombre.", "Entrée incorrecte.",
   javax.swing.JOptionPane.PLAIN_MESSAGE);
return;
```

Nous nous sommes débarrassés des messages d'erreur déroutants de la trace de la pile, pour afficher à la place le message facile à comprendre "Merci d'entrer un nombre" :



Maintenant, l'exception `NullPointerException` est gérée.

Le mot-clé throws

Dans certains cas, il est plus approprié de traiter l'exception non pas dans la méthode où elle se produit, mais dans la méthode appelante.

Dans de tels cas, la signature de la méthode doit déclarer (avertir) qu'elle peut lever une exception particulière. On utilise pour cela le mot-clé spécial `throws`. Reprenons notre exemple de lecture de fichier. Puisque la méthode `read()` peut déclencher une exception `IOException`, tu dois la gérer ou la déclarer. Dans l'exemple suivant, nous déclarons que la méthode `chargerTousLesScores()` peut émettre une `IOException` :

```
class MonSuperJeu {
  void chargerTousLesScores() throws IOException {
    // …
    // N'utilise pas try/catch si tu ne gères pas
    // d'exceptions dans cette méthode
    fichier.read();
  }

  public static void main(String[] args) {
    MonSuperJeu jeu = new MonSuperJeu();
    System.out.println("Liste des scores");

    try {
     // Puisque la méthode chargerTousLesScores() déclare
     // une exception, nous la gérons ici
       jeu.chargerTousLesScores();
    } catch(IOException e) {
       System.out.println(
       "Désolé, la liste des scores n'est pas disponible");
    }
  }
}
```

Comme nous n'essayons même pas de capturer d'exception dans la méthode `chargerTousLesScores()`, l'exception `IOException` est *propagée* à sa méthode appelante, `main()`. Celle-ci doit maintenant gérer cette exception.

Le mot-clé `finally`

Le code inclus dans un bloc `try/catch` peut se terminer de trois façons :

- Le code à l'intérieur du bloc `try` est exécuté avec succès et le programme se poursuit.

- Le code à l'intérieur du bloc `try` rencontre une instruction `return` et le programme sort de la méthode.

- Le code à l'intérieur du bloc `try` lève une exception et le contrôle passe au bloc `catch` correspondant. Soit celui-ci gère l'erreur et l'exécution de la méthode se poursuit ; soit il réémet l'exception à destination de la méthode appelante.

Si un morceau de code doit être exécuté quoi qu'il arrive, il faut le placer dans un bloc `finally` :

```
try {
  fichier.read();
} catch(Exception e) {
  e.printStackTrace();
} finally {
  // Le code qui doit toujours être exécuté,
  // par exemple file.close(), est placé ici.
}
```

Le code ci-dessus doit fermer le fichier indépendamment du succès ou de l'échec de l'opération de lecture. En général, on trouve dans le bloc `finally` le code qui libère des ressources de l'ordinateur, par exemple la déconnexion d'un réseau ou la fermeture d'un fichier.

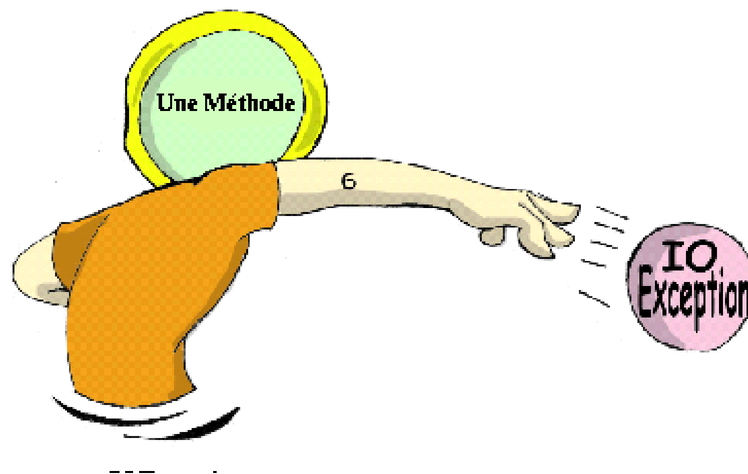
Si tu n'as pas l'intention de traiter les exceptions dans la méthode courante, elles sont propagées à l'appelant. Dans ce cas, tu peux utiliser `finally` même si tu n'as pas de bloc `catch` :

```
void maMéthode() throws IOException {
  try {
    // Place ici le code qui lit le fichier.
  }
  finally {
    // Place ici le code qui ferme le fichier.
  }
}
```

Le mot-clé `throw`

Si une exception se produit dans une méthode, mais que tu penses que c'est à l'appelant de la traiter, il suffit de la réémettre vers celui-ci. Parfois, tu peux vouloir capturer une exception mais en lever une autre avec une description différente de l'erreur, comme dans le bout de code ci-dessous.

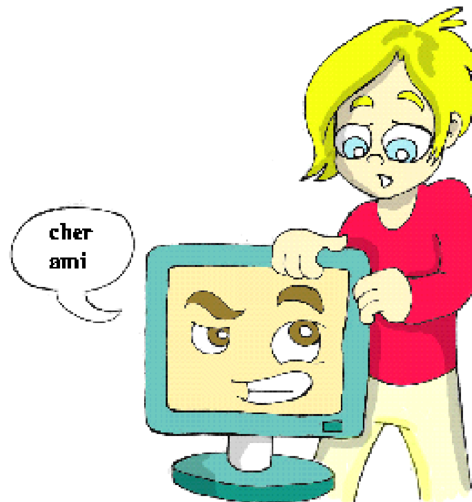
L'instruction `throw` est utilisée pour lever des exceptions en lançant des objets Java. L'objet lancé par un programme doit être *émissible* (*throwable*). C'est-à-dire que tu ne peux lancer que les objets qui sont des sous-classes directes ou indirectes de la classe `Throwable` - ce que sont toutes les exceptions Java.



Le fragment de code suivant montre comment la méthode `chargerTousLesScores()` capture une `IOException`, crée un nouvel objet `Exception` avec une description plus sympathique de l'erreur, et le lance vers la méthode `main()`. Du coup, pour que la méthode `main()` puisse être compilée, il faut ajouter une ligne qui appelle `chargerTousLesScores()` dans le bloc `try/catch`, car cette méthode peut émettre un objet `Exception` qui doit être traité ou réémis. La méthode `main()` ne devant lever aucune exception, elle doit traiter celle-ci.

```
class ListeDesScores {
  // Ce code doit être complété pour pouvoir le compiler

  static void chargerTousLesScores() throws Exception {
    try {
      fichier.read(); // Cette ligne peut lever
                      // une exception
    } catch (IOException e) {
      throw new Exception(
        "Cher ami, le fichier des scores a un problème.");
    }
  }

  public static void main(String[] args) {
    System.out.println("Scores");

    try {
      chargerTousLesScores();
    }
    catch(Exception e1) {
      System.out.println(e1.getMessage());
    }
  }
}
```

Si une erreur fichier se produit, la méthode `main()` la prend en charge et l'appel à `e1.getMessage()` renvoie le message "Cher ami ..."



Création de nouvelles exceptions

Les programmeurs peuvent aussi créer des classes d'exception Java entièrement nouvelles. Une telle classe doit être dérivée de l'une des classes d'exception Java. Disons que tu travailles dans une affaire de vente de bicyclettes et que tu aies besoin de *valider* les commandes des clients. Le nombre de vélos que tu peux mettre dans ta camionnette dépend de leur modèle. Par exemple, tu ne peux pas y mettre plus de trois vélos du modèle "Eclair". Crée une nouvelle sous-classe de `Exception` et si quelqu'un essaie de commander plus de trois de ces bicyclettes, lève cette exception :

```
class TropDeVélosException extends Exception {

  // Constructeur
  TropDeVélosException() {
  // Appelle simplement le constructeur de la superclasse
  // et lui passe le message d'erreur à afficher
    super(
        "Impossible de livrer autant de vélos en une fois.");
  }
}
```

Cette classe a juste un constructeur qui prend le message décrivant l'erreur et le passe à sa superclasse pour qu'elle le stocke. Quand un bloc `catch` reçoit cette exception, il peut savoir ce qu'il se passe exactement en appelant la méthode `getMessage()`.

Imagine qu'un utilisateur sélectionne plusieurs bicyclettes d'un certain modèle dans la fenêtre `EcranCommande` et appuie sur le bouton *Commander*. Comme tu le sais depuis le Chapitre 6, cette action aboutit à l'appel de la méthode `actionPerformed()`, laquelle vérifie

si la commande peut être livrée. L'exemple de code suivant montre comment la méthode `vérifierCommande()` déclare qu'elle est susceptible de lever l'exception `TropDeVélosException`. Si la commande ne rentre pas dans la camionnette, la méthode lève l'exception, le bloc `catch` l'intercepte et affiche un message d'erreur dans le champ textuel de la fenêtre.

```
class EcranCommande implements ActionListener {
// Place ici le code pour créer les composants de la fenêtre.

  public void actionPerformed(ActionEvent evt) {
    // L'utilisateur a cliqué sur le bouton Commander
    String modèleChoisi = champTexteModèle.getText();
    String quantitéChoisie = champTexteQuantité.getText();
    int quantité = Integer.parseInt(quantitéChoisie);

    try {
      commandeVélos.vérifierCommande(modèleChoisi,
                                     quantitéChoisie);
      // Cette ligne sera sautée en cas d'exception
      champTexteConfirmationCommande.setText(
                         "Votre commande est prête.");
    } catch(TropDeVélosException e) {
      champTexteConfirmationCommande.setText(e.getMessage());
    }
  }

  void vérifierCommande(String modèleVélo, int quantité)
                        throws TropDeVélosException {

    // Ecris le code qui vérifie que la quantité demandée
    // du modèle sélectionné entre bien dans la camionnette.
    // Si ça n'entre pas, faire ce qui suit :

    throw new TropDeVélosException ("Impossible de livrer " +
      quantité + " vélos du modèle " + modèleVélo +
                            " en une fois.");
  }
}
```

Dans un monde parfait, tous les programmes fonctionneraient parfaitement, mais il faut être réaliste et être prêt à réagir aux situations inattendues. Java nous aide vraiment en nous forçant à écrire un code adapté à ces situations.

Autres lectures

Gestion des erreurs à l'aide des exceptions :
http://java.sun.com/docs/books/tutorial/essential/exceptions/

Exercices

Crée une application Swing pour gérer des commandes de vélos. Elle doit contenir les deux champs textuels *Modèle de vélo* et *Quantité*, le bouton *Commander* et le libellé pour la confirmation des commandes.

Utilise le code des exemples `EcranCommande` et `TropDeVélosException`. Prévois plusieurs combinaisons de modèles et de quantités qui déclenchent une exception.

Exercices pour les petits malins

Modifie l'application de l'exercice précédent en remplaçant le champ textuel *Modèle de vélo* par une boîte de liste déroulante contenant plusieurs modèles, afin que l'utilisateur puisse choisir dans la liste au lieu d'entrer son choix au clavier.

Vas lire en ligne ce qui est dit du composant Swing `JComboBox` et de `ItemListener` pour traiter les événements correspondant au choix d'un modèle par l'utilisateur.

Chapitre 9. Enregistrement du score

Après qu'il se soit terminé, un programme est effacé de la mémoire. Cela signifie que toutes les classes, méthodes et variables disparaissent jusqu'à ce que tu exécutes à nouveau ce programme. Si tu souhaites sauvegarder certains résultats de l'exécution du programme, il faut les enregistrer dans des fichiers sur un disque, une cassette, une carte mémoire ou un autre périphérique capable de stocker des données pendant une longue période. Dans ce chapitre, tu vas apprendre comment enregistrer des données sur disque à l'aide des *flux* (*streams*) Java. Fondamentalement, tu ouvres un flux entre ton programme et un fichier sur disque. Si tu veux lire des données sur le disque, il te faut *un flux d'entrée* (*input stream)* ; si tu écris des données sur le disque, ouvre un *flux de sortie* (*output stream).* Par exemple, si un joueur gagne une partie et que tu veux sauvegarder son score, tu peux l'enregistrer dans le fichier `scores.txt` en utilisant un flux de sortie.

Un programme lit ou écrit les données dans un flux *en série* – octet après octet, caractère après caractère, etc. Comme ton programme utilise différents types de données tels que `String`, `int`, `double` et autres, tu dois utiliser un flux Java approprié, par exemple un flux d'octets (*byte stream*), un flux de caractères (*character stream*) ou un flux de données (*data stream*).

Les classes qui fonctionnent avec des flux de fichiers sont situées dans les paquetages `java.io`. et `java.nio`.

Quel que soit le type de flux que tu vas utiliser, tu dois respecter les trois étapes suivantes dans ton programme :

- Ouvrir un flux qui pointe sur un fichier.
- Lire ou écrire des données dans ce flux.
- Fermer le flux.

Flux d'octets

Si tu crées un programme qui lit un fichier puis affiche son contenu sur l'écran, tu dois savoir quel type de données contient le fichier. Par contre, un programme qui se contente de copier des fichiers d'un endroit à un autre n'a même pas besoin de savoir s'il s'agit d'images, de texte ou de musique. De tels programmes chargent le fichier original en mémoire sous la forme d'un ensemble d'octets, puis les écrivent dans le fichier de destination, octet après octet, à l'aide des classes Java `FileInputStream` et `FileOutputStream`.

L'exemple suivant montre comment utiliser la classe `FileInputStream` pour lire un fichier graphique nommé `abc.gif`, situé dans le répertoire `c:\exercices`. Si tu utilises un ordinateur sous Microsoft Windows, pour éviter la confusion avec les caractères spéciaux Java qui commencent par une barre oblique inversée, utilise des barres doubles dans ton code pour séparer les noms de répertoires et de fichier : `"c:\\exercices"`. Ce petit programme n'affiche pas l'image, mais des nombres qui correspondent à la façon dont l'image est stockée sur un disque. Chaque octet a une valeur entière positive comprise entre 0 et 255, que la classe `LecteurOctets` affiche en la délimitant par des espaces.

Je te prie de noter que la classe `LecteurOctets` ferme le flux dans le bloc `finally`. N'appelle jamais la méthode `close()` à l'intérieur du bloc `try/catch` juste après avoir fini de lire le fichier, mais fais-le dans le bloc `finally`. Sinon, en cas d'exception, le programme sauterait par-dessus l'instruction `close()` barrée et le flux ne serait pas fermé ! La lecture se termine quand la méthode `FileInputStream.read()` retourne la valeur négative -1.

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

public class LecteurOctets {

  public static void main(String[] args) {

    FileInputStream monFichier = null;

    try {
      // Ouvre un flux pointant sur le fichier
      monFichier = new
              FileInputStream("c:\\exercices\\abc.gif");

      while (true) {
        int valeurEntièreOctet = monFichier.read();
        System.out.print(" " + valeurEntièreOctet);

        if (valeurEntièreOctet  == -1) {
          // Nous avons atteint la fin du fichier
          // Sortons de la boucle
          break;
        }
      } // Fin de la boucle while
      // monFichier.close(); pas à cet endroit
    } catch (IOException exception) {
      System.out.println("Impossible de lire le fichier : "
                                  + exception.toString());
    } finally {
      try {
        monFichier.close();
      } catch (Exception exception1){
        exception1.printStackTrace() ;
      }
      System.out.println("Lecture du fichier terminée.");
    }
  }
}
```

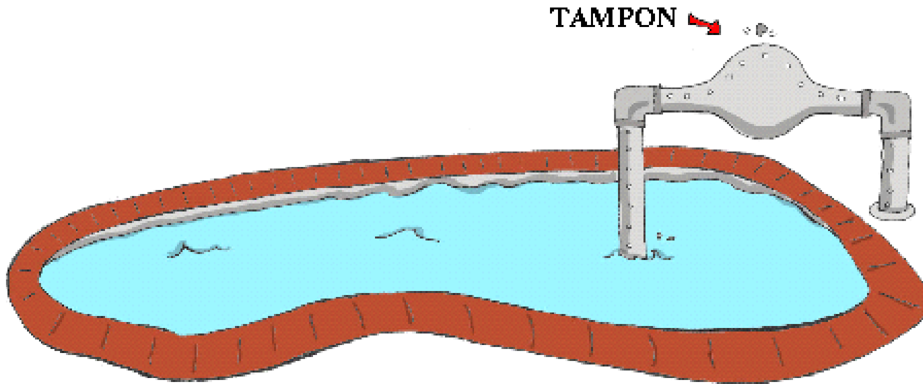
L'extrait de code suivant écrit plusieurs octets, représentés par des nombres entiers, dans le fichier `xyz.dat`, à l'aide de la classe `FileOutputStream` :

```
int données[] = {56, 230, 123, 43, 11, 37};

FileOutputStream monFichier = null;

try {
  // Ouvre le fichier xyz.dat et y enregistre
  // les données du tableau
  monFichier = new FileOutputStream("xyz.dat");
  for (int i = 0; i < données.length; i++) {
    monFichier.write(données[i]);
  }
} catch (IOException exception) {
  System.out.println("Impossible d'écrire dans le fichier :"
                                   + exception.toString());
} finally{
  try{
    monFichier.close();
  } catch (Exception exception1) {
    exception1.printStackTrace();
  }
}
```

Flux à tampon

Jusqu'ici nous avons lu et écrit les données un octet à la fois, ce qui implique que le programme `LecteurOctets` devra accéder 1000 fois au disque pour lire un fichier de 1000 octets. Mais l'accès aux données sur le disque est bien plus lent que la manipulation de données en mémoire. Pour minimiser le nombre de tentatives d'accès au disque, Java fournit ce qu'on appelle des tampons (*buffers*), qui sont des sortes de "réservoirs de données".

La classe `BufferedInputStream` permet de remplir rapidement la mémoire tampon avec des données de `FileInputStream`. Un flux à tampon charge d'un seul coup dans un tampon en mémoire un gros paquet d'octets depuis un fichier. Ensuite, la méthode `read()` lit chaque octet dans le tampon beaucoup plus rapidement qu'elle ne le ferait sur le disque.



Ton programme peut connecter des flux comme un plombier connecte deux tuyaux. Modifions l'exemple qui lit un fichier. Les données sont d'abord déversées du `FileInputStream` dans le `BufferedInputStream`, puis passées à la méthode `read()` :

```
FileInputStream monFichier = null;
BufferedInputStream tampon = null;

try {
  monFichier =
              new FileInputStream("c:\\exercices\\abc.gif");
  // Connecte les flux
  tampon = new BufferedInputStream(monFichier);
  while (true) {
    int valeurOctet = tampon.read();
    System.out.print(valeurOctet + " ");
    if (valeurOctet == -1)
      break;
  }
} catch (IOException exception) {
  exception.printStackTrace();
} finally {
  try {
    tampon.close();
    monFichier.close();
  } catch(IOException exception1) {
    exception1.printStackTrace();
  }
}
```

Quelle est la taille de ce tampon ? Cela dépend du Java, mais tu peux régler sa taille et voir si cela rend la lecture de fichier un peu plus rapide. Par exemple, pour affecter au tampon une taille de 5000 octets, utilise le constructeur à deux arguments :

```
BufferedInputStream tampon =
                     new BufferedInputStream(monFichier, 5000);
```

Les flux à tampon ne modifient pas le type de lecture ; ils la rendent seulement plus rapide.

BufferedOutputStream fonctionne de la même façon, mais avec la classe FileOutputStream.

```
int données[] = {56, 230, 123, 43, 11, 37};
FileOutputStream monFichier = null;
BufferedOutputStream tampon = null;

try {
  monFichier = new FileOutputStream("xyz.dat");
  // Connecte les flux
  tampon = new BufferedOutputStream(monFichier);
  for (int i = 0; i < données.length; i++) {
    tampon.write(données[i]);
  }
} catch (IOException exception) {
  exception.printStackTrace();
} finally {
  try {
    tampon.flush();
    tampon.close();
    monFichier.close();
  } catch (IOException exception1) {
    exception1.printStackTrace();
  }
}
```

Pour t'assurer que tous les octets du tampon sont envoyés au fichier, appelle la méthode flush() (*vider*) lorsque l'écriture dans le BufferedOutputStream est terminée.

Arguments de la ligne de commande

Notre programme LecteurOctets stocke le nom du fichier abc.gif directement dans son code, ou, comme disent les développeurs, le nom de fichier est *écrit en dur* dans le programme. Cela signifie que, pour créer un programme similaire qui lit le fichier xyz.gif, tu dois modifier le code et recompiler le programme, ce qui n'est pas agréable. Il vaudrait mieux passer le nom du fichier depuis la ligne de commande, lors du lancement du programme.

Tu peux exécuter tous les programmes Java avec des *arguments de ligne de commande*, par exemple :

```
java LecteurOctets xyz.gif
```

Dans cet exemple, nous passons à la méthode main() de LecteurOctets un seul argument - xyz.gif. Si tu t'en souviens, la méthode main() a un argument :

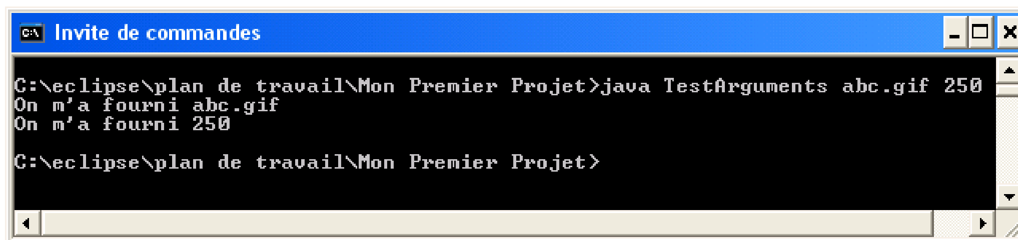
```
public static void main(String[] arguments)
```

Effectivement, c'est un tableau de `String` que Java passe à la méthode `main()`. Si tu lances un programme sans aucun argument sur la ligne de commande, ce tableau est vide. Dans le cas contraire, le nombre d'éléments de ce tableau est exactement le même que celui des arguments passés au programme sur la ligne de commande.

Voyons comment utiliser ces arguments de ligne de commande dans une classe très simple qui ne fait que les afficher :

```
public class TestArguments {

  public static void main(String[] arguments) {

    // Combien d'arguments m'a-t-on fourni ?
    int nombreArguments = arguments.length;

    for (int i = 0; i < nombreArguments; i++) {
      System.out.println("On m'a fourni " + arguments[i]);
    }
  }
}
```

La capture d'écran suivante montre ce qu'il se passe si on exécute ce programme avec deux arguments – `xyz.gif` et `250`. La valeur `xyz.gif` est placée par Java dans l'élément `arguments[0]` et la seconde dans `arguments[1]`.



Les arguments de la ligne de commande sont toujours passés à un programme comme des `Strings`. Il est de la responsabilité du programme de convertir les données dans le type de données approprié. Par exemple :

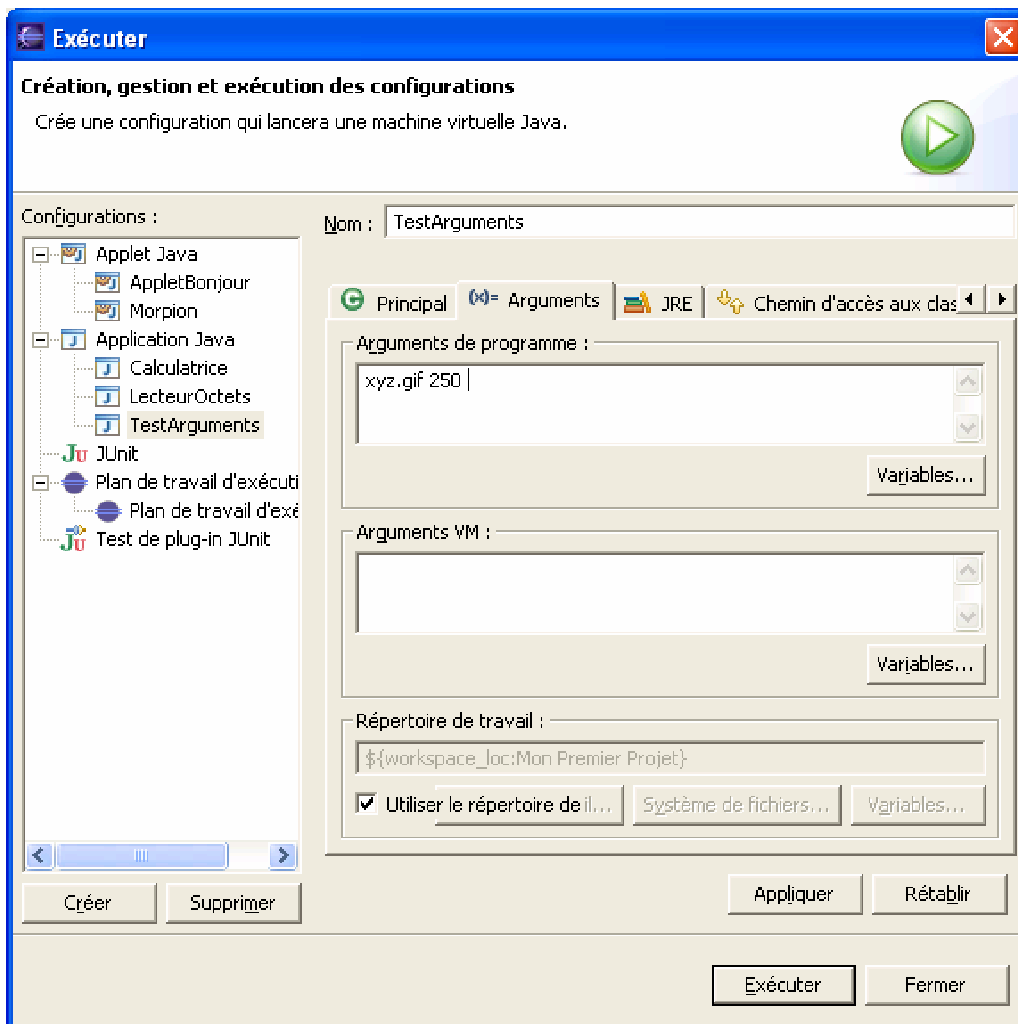
```
int monScore = Integer.parseInt(arguments[1]);
```

C'est toujours une bonne chose de vérifier si la ligne de commande contient le bon nombre d'arguments. Fais-le au tout début de la méthode `main()`. Si le programme ne reçoit pas les arguments attendus, il doit le signaler en affichant un message bref et s'arrêter immédiatement en utilisant la méthode spéciale `System.exit()`:

```
public static void main(String[] arguments) {
  if (arguments.length != 2) {
    System.out.println(
          "Merci de fournir deux arguments, par exemple : ");
    System.out.println("java TestArguments xyz.gif 250");

    // Sortie du programme
    System.exit(0);
  }
}
```

A la fin de ce chapitre, tu devras écrire un programme qui copie des fichiers. Pour que ce programme fonctionne avec n'importe quels fichiers, les noms des fichiers source et destination doivent être passés au programme en tant qu'arguments de la ligne de commande.

Tu peux tester tes programmes dans Eclipse, qui permet aussi de fournir des arguments de ligne de commande à tes programmes. Dans la fenêtre *Exécuter*, sélectionne l'onglet *(x)=Arguments* et entre les valeurs requises dans la boîte *Arguments de programme*.



La boîte *Arguments VM* est utilisée pour passer des paramètres à Java. Ces paramètres permettent de demander plus de mémoire pour ton programme, régler finement la performance de Java… Tu trouveras dans la section *Autres lectures* la référence d'un site web qui décrit ces paramètres en détail.

Lecture de fichiers texte

Java utilise des caractères de deux octets pour stocker les lettres. Les classes `FileReader` et `FileWriter` sont très pratiques pour travailler avec des fichiers texte. Ces classes peuvent lire un fichier texte, soit caractère par caractère à l'aide de la méthode `read()`, soit ligne par ligne à l'aide de la méthode `readLine()`. Les classes `FileReader` et `FileWriter` ont aussi leurs contreparties `BufferedReader` et `BufferedWriter` pour accélérer le travail avec des fichiers.

La classe `LecteurScores` lit le fichier `scores.txt` ligne à ligne et le programme se termine quand la méthode `readLine()` renvoie `null`, ce qui signifie fin de fichier.

A l'aide d'un éditeur de texte quelconque, crée le fichier `c:\scores.txt` avec le contenu suivant :

```
David 235
Daniel 190
Anna  225
Gregory 160
```

Exécute le programme `LecteurScores` ci-dessous et il affichera le contenu de ce fichier. Ajoute d'autres lignes au fichier de scores et exécute à nouveau le programme pour vérifier que les nouvelles lignes sont aussi affichées.

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

public class LecteurScores {

  public static void main(String[] arguments) {
    FileReader monFichier = null;
    BufferedReader tampon = null;

    try {
      monFichier = new FileReader("c:\\scores.txt");
      tampon = new BufferedReader(monFichier);

      while (true) {
        // Lit une ligne de scores.txt
        String ligne = tampon.readLine();
        // Vérifie la fin de fichier
        if (ligne == null)
          break;
        System.out.println(ligne);
      } // Fin du while
    } catch (IOException exception) {
      exception.printStackTrace();
    } finally {
      try {
        tampon.close();
        monFichier.close();
      } catch(IOException exception1) {
          exception1.printStackTrace();
      }
    }
  } // Fin de main
}
```

Si ton programme doit écrire un fichier texte sur un disque, utilise l'une des méthodes `write()` surchargées de la classe `FileWriter`. Ces méthodes permettent d'écrire un caractère, un `String` ou un tableau entier de caractères.

`FileWriter` possède plusieurs constructeurs surchargés. Si tu ouvres un fichier en écriture en ne fournissant que son nom, ce fichier sera remplacé par un nouveau à chaque fois que tu exécuteras le programme :

```
FileWriter fichierSortie = new FileWriter("c:\\scores.txt");
```

Si tu souhaites ajouter des données à la fin d'un fichier existant, utilise le constructeur à deux arguments (`true` signifie ici mode ajout) :

```
FileWriter fichierSortie = new FileWriter("c:\\scores.txt",
true);
```

La classe `EnregistreurScores` écrit trois lignes dans le fichier `c:\scores.txt` à partir du tableau `scores`.

```
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class EnregistreurScores {

  public static void main(String[] arguments) {

    FileWriter monFichier = null;
    BufferedWriter tampon = null;
    String[] scores = new String[3];

    // Entre des scores dans le tableau
    scores[0] = "M. Dupont 240";
    scores[1] = "M. Durand 300";
    scores[2] = "M. Pemieufer 190";

    try {
      monFichier = new FileWriter("c:\\scores.txt");
      tampon = new BufferedWriter(monFichier);

      for (int i = 0; i < scores.length; i++) {
        // Ecrit le tableau de chaînes dans scores.txt
        tampon.write(scores[i]);

        System.out.println("Ecriture de : " + scores[i]);
      }
      System.out.println("Ecriture du fichier terminée.");

    } catch (IOException exception) {
      exception.printStackTrace();
    } finally {
      try {
        tampon.flush();
        tampon.close();
        monFichier.close();
      } catch (IOException e1) {
        e1.printStackTrace();
      }
    }
  } // Fin de main
}
```

La sortie de ce programme ressemble à ceci :

```
Ecriture de : M. Dupont 240
Ecriture de : M. Durand 300
Ecriture de : M. Pemieufer 190
Ecriture du fichier terminée.
```

Classe `File` (*fichier*)

La classe `java.io.File` fournit de nombreuses méthodes utiles, qui permettent de renommer un fichier, de supprimer un fichier, de

vérifier si le fichier existe, etc. Mettons que ton programme enregistre des données dans un fichier et qu'il ait besoin d'afficher un message pour avertir l'utilisateur si ce fichier existe déjà. Pour ce faire, tu dois créer une instance de l'objet `File` en lui donnant le nom du fichier, puis appeler la méthode `exists()`. Si cette méthode retourne `true`, le fichier a été trouvé et tu dois afficher un message d'avertissement. Sinon, c'est que ce fichier n'existe pas.

```
File unFichier = new File("abc.txt");

if (unFichier.exists()) {
  // Affiche un message ou utilise un JOptionPane
  // pour afficher un avertissement.
}
```

Le constructeur de la classe `File` *ne crée pas réellement un fichier* – il crée juste en mémoire une instance de cet objet qui pointe sur le fichier réel. Si tu dois vraiment créer un fichier sur le disque, utilise la méthode `createNewFile()`.

Voici quelques unes des méthodes utiles de la classe `File`.

Nom de la méthode	Fonctionnalité
`createNewFile()`	Crée un nouveau fichier, vide, du nom utilisé pour l'instanciation de la classe *File*. Ne crée un nouveau fichier que s'il n'existe pas déjà un fichier du même nom.
`delete()`	Supprime un fichier ou un répertoire.
`renameTo()`	Renomme un fichier.
`length()`	Retourne la longueur d'un fichier en octets.
`exists()`	Retourne `true` si le fichier existe.
`list()`	Retourne un tableau de chaînes contenant les noms des fichiers/répertoires contenus dans un répertoire donné.
`lastModified()`	Retourne l'heure et la date de dernière modification du fichier.
`mkDir()`	Crée un répertoire.

L'extrait de code ci-dessous renomme le fichier `clients.txt` en `clients.txt.bak`. Si le fichier `.bak` existe déjà, il est remplacé.

```
File fichier = new File("clients.txt");
File sauvegarde = new File("clients.txt.bak");

if (sauvegarde.exists()) {
  sauvegarde.delete();
}
fichier.renameTo(sauvegarde);
```

Même si nous n'avons travaillé dans ce chapitre que sur des fichiers situés sur le disque de ton ordinateur, Java te permet de créer des flux pointant vers des machines distantes sur un réseau d'ordinateurs. Ces ordinateurs peuvent être situés assez loin les uns des autres. Par exemple, la NASA utilise Java pour contrôler les robots de la mission Mars Rovers et je suis sûr qu'ils se sont contentés de pointer leurs flux sur Mars. ☺

Autres lectures

1. Options de ligne de commande Java
http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html

2. Utilisation des flux fichier
http://java.sun.com/docs/books/tutorial/essential/io/filestreams.html

Exercices

Ecris le programme de copie de fichier `FileCopy` en combinant les fragments de code de la section sur les flux d'octets.

Ouvre deux flux (entrée et sortie) et appelle les méthodes `read()` et `write()` *dans la même boucle*. Utilise les arguments de la ligne de commande pour passer au programme les noms des fichiers source et cible, par exemple :

```
java CopieFichier
            c:\temp\scores.txt
            c:\sauvegardes\scores2.txt
```

Exercices pour les petits malins

Crée un programme Swing qui permette à l'utilisateur de sélectionner les noms de fichiers à copier en utilisant la classe `JFileChooser`, qui crée une fenêtre standard de sélection de fichier. Cette fenêtre doit s'ouvrir quand l'utilisateur clique sur l'un des boutons *Parcourir*. Tu as quelques lignes de code à écrire pour afficher le nom de fichier sélectionné dans le champ textuel correspondant.

Quand l'utilisateur clique sur le bouton *Copier*, le code de la méthode `actionPerformed()` doit copier le fichier sélectionné. Essaie de réutiliser le code de l'exercice précédent sans effectuer de copier/coller.

Chapitre 10. Autres briques de base Java

Nous avons eu l'occasion d'utiliser pas mal d'éléments Java différents dans les chapitres précédents. Nous avons même créé un jeu de morpion. Mais j'ai laissé de côté d'importants éléments et techniques Java et il est temps de corriger ces omissions.

Utilisation des valeurs de date et d'heure

Tous les ordinateurs ont une horloge interne. N'importe quel programme Java peut déterminer la date et l'heure courantes et les afficher dans différents formats, par exemple "15/06/2004" ou "15 juin 2004". De nombreuses classes Java permettent de gérer les dates et les heures, mais deux d'entre elles, `java.util.Date` et `java.text.SimpleDateFormat`, suffiront à couvrir la plupart de tes besoins.

Il est facile de créer un objet qui stocke les date et heure système courantes précises à la milliseconde près :

```
Date maintenant = new Date();
System.out.println("Nous sommes le " +
         SimpleDateFormat.getDateInstance().format(maintenant));
```

La sortie de ces lignes peut ressembler à ceci[7] :

```
Nous sommes le 23 juin 2005
```

La classe `SimpleDateFormat` permet d'afficher les dates et les heures selon différents formats. D'abord, tu dois créer une instance de cette classe avec le format souhaité. Ensuite, appelle sa méthode `format()` en lui passant un objet `Date` en argument. Le programme suivant formate et affiche la date courante sous différents formats.

---

[7] NDT le code minimal est encore plus simple pour nos amis anglophones, qui n'ont pas besoin de faire appel à `SimpleDateFormat` ; la ligne `System.out.println("Date anglaise " + maintenant);` imprimerait par exemple `Date anglaise Thu Jun 23 11:48:55 CEST 2005`.

```
import java.util.Date;
import java.text.SimpleDateFormat;

public class MonFormatDate {

  public static void main( String [] args ) {
    // Crée un objet Date
    // et l'affiche dans le format par défaut
    Date maintenant = new Date();
    System.out.println("Nous sommes le : " +
       SimpleDateFormat.getDateInstance().format(maintenant));

    // Affiche la date comme ceci : 23-06-05
    SimpleDateFormat formatDate =
                           new SimpleDateFormat("dd-MM-yy");
    String dateFormatée = formatDate.format(maintenant);
    System.out.println("Nous sommes le (jj-MM-aa) : "
                                         + dateFormatée);

    // Affiche la date comme ceci : 23-06-2005
    formatDate = new SimpleDateFormat("dd/MM/yyyy");
    dateFormatée = formatDate.format(maintenant);
    System.out.println("Nous sommes le (jj/MM/aaaa) : "
                                         + dateFormatée);

    // Affiche la date comme ceci : jeudi 23 juin 2005
    formatDate = new SimpleDateFormat("EEEE d MMM yyyy");
    dateFormatée = formatDate.format(maintenant);
    System.out.println("Nous sommes le (JJJJ jj MMMM aaaa) : "
                                         + dateFormatée);
    // Affiche l'heure comme ceci : 20h 30m 17s
    formatDate = new SimpleDateFormat("kk'h' mm'm' ss's'");
    dateFormatée = formatDate.format(maintenant);
    System.out.println("Il est (HHh MMm SSs) : "
                                         + dateFormatée);
  }
}
```

Compile et exécute la classe `MonFormatDate`. Elle affiche ceci :

```
Nous sommes le : 23 juin 2005
Nous sommes le (jj-MM-aa) : 23-06-05
Nous sommes le (jj/MM/aaaa) : 23/06/2005
Nous sommes le (JJJJ jj MMMM aaaa) : jeudi 23 juin 2005
Il est (HHh MMm SSs) : 20h 30m 17s
```

La documentation Java de la classe `SimpleDateFormat` décrit d'autres formats. Tu peux aussi trouver d'autres méthodes de gestion des dates dans la classe `java.util.Calendar`.

Surcharge de méthode

Il peut y avoir, dans une même classe, plusieurs méthodes de même nom mais ayant des listes d'arguments différentes. C'est ce qu'on appelle la *surcharge de méthode* (*method overloading*). Par exemple, la méthode `println()` de la classe `System` peut être appelée avec différents types d'arguments : `String`, `int`, `char`, etc.

```
System.out.println("Bonjour");

System.out.println(250);

System.out.println('A');
```

Même s'il semble que l'on appelle trois fois la même méthode `println()`, il s'agit en fait de trois méthodes différentes. Tu te demandes peut-être pourquoi on ne crée pas des méthodes avec des noms différents, par exemple `printString()`, `printInt()` et `printChar()` ? Une raison en est qu'il est plus facile de mémoriser un nom de méthode d'affichage plutôt que plusieurs. Il y a d'autres raisons justifiant l'utilisation de la surcharge de méthode, mais elles sont un peu compliquées à expliquer et sont abordées dans des livres d'un niveau plus avancé.

Tu te rappelles notre classe `Poisson` du Chapitre 4 et sa méthode `plonger()`, qui attend un argument :

```
public int plonger(int combienDePlus)
```

Créons une autre version de cette méthode qui n'aura pas besoin d'argument. Cette méthode va forcer le poisson à plonger de cinq mètres, sauf si la profondeur courante vient à dépasser 100 mètres. La nouvelle version de la classe `Poisson` a une nouvelle variable invariante, `PROFONDEUR_PLONGEE`, dont la valeur est cinq.

La classe `Poisson` a maintenant deux méthodes `plonger()` surchargées.

```
public class Poisson extends AnimalFamilier {
  int profondeurCourante = 0;
  final int PROFONDEUR_PLONGEE = 5;

  public int plonger() {
    profondeurCourante = profondeurCourante +
                                      PROFONDEUR_PLONGEE;
    if (profondeurCourante > 100) {
      System.out.println("Je suis un petit"
                       + " poisson et je ne peux pas plonger"
                       + " plus profond que 100 mètres");
      profondeurCourante = profondeurCourante
                                    - PROFONDEUR_PLONGEE;
    } else {
      System.out.println("Plongée de " + PROFONDEUR_PLONGEE
                                            + " mètres");
      System.out.println("Je suis à " + profondeurCourante
                       + " mètres sous le niveau de la mer");
    }
    return profondeurCourante;
  }
  public int plonger(int combienDePlus) {
    profondeurCourante = profondeurCourante + combienDePlus;
    if (profondeurCourante > 100) {
      System.out.println("Je suis un petit"
                       + " poisson et je ne peux pas plonger"
                       + " plus profond que 100 mètres");
      profondeurCourante = profondeurCourante
                                          - combienDePlus;
    } else {
      System.out.println("Plongée de " + combienDePlus
                                            + " mètres");
      System.out.println("Je suis à " + profondeurCourante
                       + " mètres sous le niveau de la mer");
    }
    return profondeurCourante;
  }

  public String dire(String unMot){
    return "Ne sais-tu pas que les poissons ne"
                                      + " parlent pas ?";
  }
  // Constructeur
  Poisson(int positionDépart) {
    profondeurCourante = positionDépart;
  }
}
```

`MaîtrePoisson` peut désormais appeler l'une ou l'autre des méthodes `plonger()` surchargées :

```
public class MaîtrePoisson {

  public static void main(String[] args) {

    Poisson monPoisson = new Poisson(20);

    monPoisson.plonger(2);

    monPoisson.plonger();  // nouvelle méthode surchargée

    monPoisson.plonger(97);
    monPoisson.plonger(3);

    monPoisson.dormir();
  }
}
```

Les constructeurs peuvent aussi être surchargés, mais l'un d'entre eux seulement est utilisé lorsqu'un objet est créé. Java appelle le constructeur qui a la bonne liste d'arguments. Par exemple, si tu ajoutes un constructeur sans argument à la classe `Poisson`, `MaîtrePoisson` peut en créer une instance par l'un des moyens suivants :

```
Poisson monPoisson = new Poisson(20);
```

ou

```
Poisson monPoisson = new Poisson();
```

Lecture des entrées clavier

Dans cette section, tu vas apprendre comment un programme peut afficher des questions dans la fenêtre de commande et comprendre les réponses que l'utilisateur entre au clavier. Cette fois, nous allons retirer de la classe `MaîtrePoisson` toutes les valeurs codées en dur qu'il passait à la classe `Poisson`. Le programme va maintenant demander "De combien ?" et le poisson va plonger en fonction des réponses de l'utilisateur.

A ce stade, tu dois être plutôt à l'aise dans l'utilisation de la *sortie standard* (*standard output*) `System.out`. A propos, la variable `out` est du type `java.io.OutputStream`. Je vais maintenant t'expliquer comment utiliser *l'entrée standard* (*standard input*) `System.in`. Au fait, comme tu peux t'en douter, le type de données de la variable `in` est `java.io.InputStream`.

La version suivante de la classe `MaîtrePoisson` affiche une *invite de commande* (*prompt*) sur la console système et attend la réponse de

l'utilisateur. Une fois que l'utilisateur a tapé un ou plusieurs caractères et appuyé sur la touche *Entrée*, Java place ces caractères dans l'objet InputStream pour les passer au programme.

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

public class MaîtrePoisson {

  public static void main(String[] args) {

    Poisson monPoisson = new Poisson(20);
    String chaîneNombreDeMètres = "";
    int entierNombreDeMètres;
    // Crée un lecteur de flux d'entrée connecté à
    // System.in et le passe au lecteur à tampon
    BufferedReader entréeStandard = new BufferedReader
                           (new InputStreamReader(System.in));

    // Continue à plonger tant que l'utilisateur
    // ne tape pas "Q"
    while (true) {
      System.out.println("Prêt à plonger. De combien ?");
      try {
        chaîneNombreDeMètres = entréeStandard.readLine();
        if (chaîneNombreDeMètres.equals("Q")) {
          // Sort du programme
          System.out.println("Au revoir !");
          System.exit(0);
        } else {
          // Convertit chaîneNombreDeMètres en entier
          // et plonge de la valeur de entierNombreDeMètres
          entierNombreDeMètres =
                      Integer.parseInt(chaîneNombreDeMètres);
          monPoisson.plonger(entierNombreDeMètres);
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    } // Fin du while
  } // Fin de main
}
```

Un dialogue entre l'utilisateur et le programme MaîtrePoisson ressemble à ceci :

```
Prêt à plonger. De combien ?
14
Plongée de 14 mètres
Je suis à 34 mètres sous le niveau de la mer
Prêt à plonger. De combien ?
30
Plongée de 30 mètres
Je suis à 64 mètres sous le niveau de la mer
Prêt à plonger. De combien ?
Q
Au revoir !
```

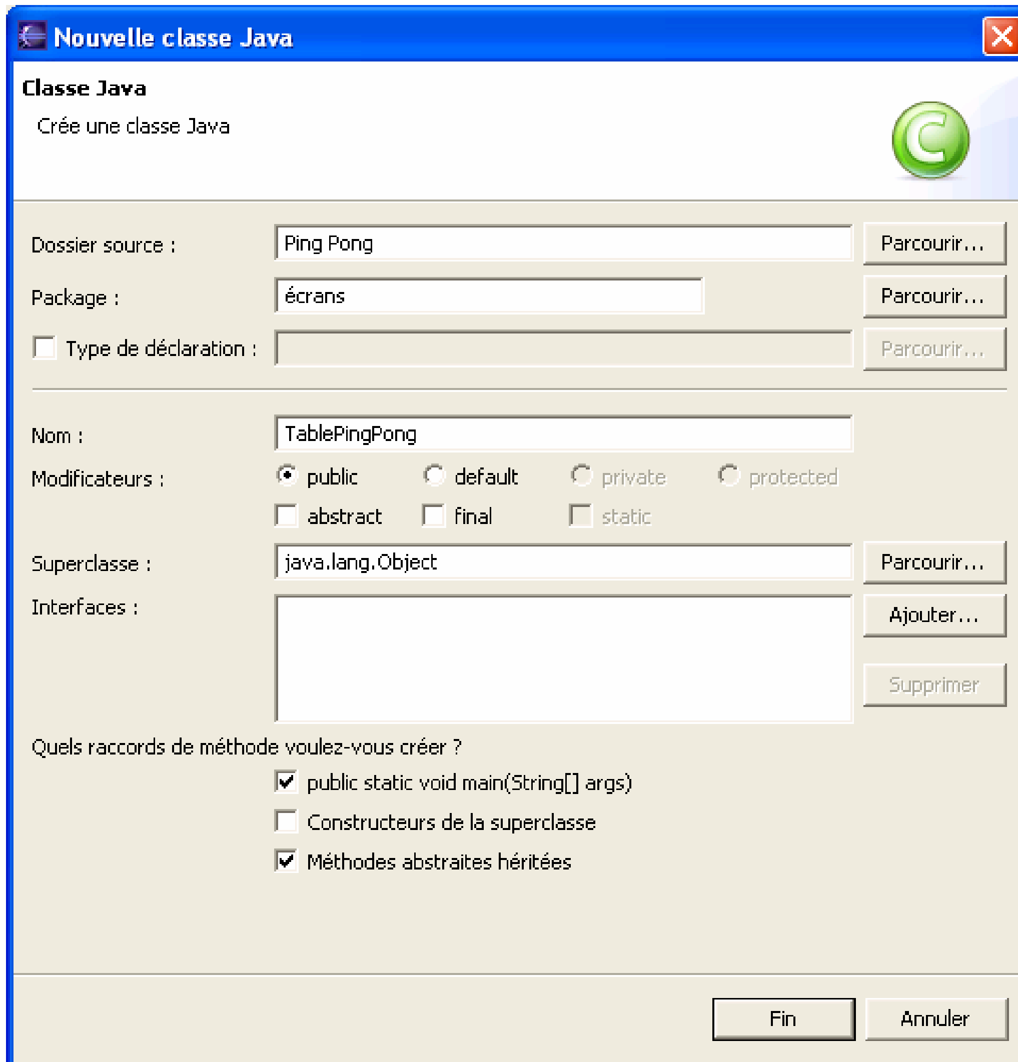
En premier lieu, `MaîtrePoisson` crée un flux `BufferedReader` connecté à l'entrée standard `System.in`. Il affiche ensuite le message "Prêt à plonger. De combien ?" et la méthode `readLine()` met le programme en attente jusqu'à ce que l'utilisateur appuie sur *Entrée*. La valeur saisie est passée au programme sous forme de `String`, que `MaîtrePoisson` convertit en entier avant d'appeler la méthode `plonger()` de la classe `Poisson`.
Ces actions sont répétées en boucle jusqu'à ce que l'utilisateur tape la lettre *Q* pour quitter le programme. La ligne `chaîneNombreDeMètres.equals("Q")` compare la valeur de la variable de type `String chaîneNombreDeMètres` avec la lettre *Q*.

Nous avons utilisé la méthode `readLine()` pour lire en une seule fois toute la ligne entrée par l'utilisateur, mais il existe une autre méthode, `System.in.read()`, qui permet de traiter les entrées de l'utilisateur caractère par caractère.

Compléments sur les paquetages Java

Quand les programmeurs travaillent sur des projets de grande taille, contenant de nombreuses classes, ils les organisent généralement en différents *paquetages* (*packages*). Par exemple, un paquetage pourrait contenir toutes les classes qui affichent les fenêtres (écrans), alors qu'un autre contiendrait tous les récepteurs d'événements. Java aussi range ses classes dans des paquetages, tels que `java.io` pour les classes responsables des opérations d'entrée/sortie ou `javax.swing` pour les classes Swing.

Créons un nouveau projet Eclipse appelé *Ping Pong*. Ce projet organisera ses classes en deux paquetages : `écrans` et `moteur`. Crée la classe `TablePingPong` et entre le mot `écrans` dans le champ *Package* :



Appuie sur le bouton *Fin* et Eclipse génère un code qui inclut une ligne contenant le nom du paquetage.

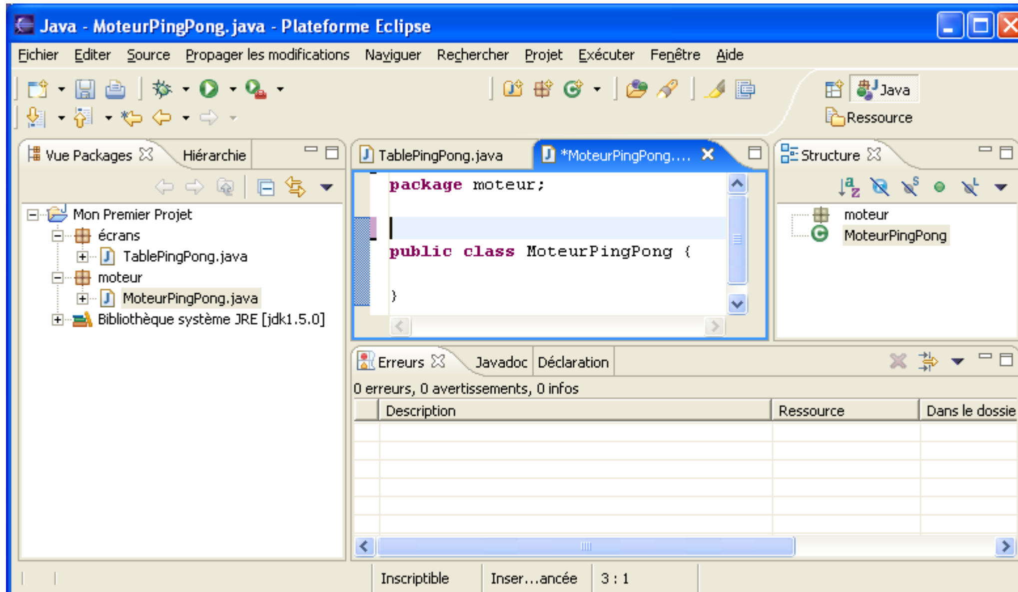
```
package écrans;

public class TablePingPong {

  public static void main(String[] args) {
  }

}
```

A propos, si ta classe inclut une ligne avec le mot-clé `package`, tu n'as le droit d'écrire que des commentaires au-dessus de cette ligne.

Comme chaque paquetage est stocké dans un répertoire différent du disque, Eclipse crée le répertoire `écrans` et y place le fichier `TablePingPong.java`. Vérifie – il doit y avoir sur ton disque un répertoire `c:\eclipse\plan de travail\Ping Pong\écrans`,

contenant les fichiers `TablePingPong.java` et `TablePingPong.class`.

Crée maintenant une autre classe, nommée `MoteurPingPong`, et indique `moteur` comme nom de paquetage. Le projet `Ping Pong` contient maintenant deux paquetages :



Puisque nos deux classes sont situées dans deux paquetages (et répertoires) différents, la classe `TablePingPong` ne peut voir la classe `MoteurPingPong` que si tu ajoutes une instruction `import`.

```
package écrans;

import moteur.MoteurPingPong;

public class TablePingPong {

  public static void main(String[] args) {
    MoteurPingPong moteurJeu = new MoteurPingPong();
  }
}
```

Non seulement les paquetages Java t'aident-ils à mieux organiser tes classes, mais ils peuvent aussi être utilisés pour limiter l'accès à leurs classes par des "étrangers" installés dans d'autres paquetages.

Niveaux d'accès

Les classes, méthodes et variables membres Java peuvent avoir les niveaux d'accès `public`, `private`, `protected` et `package`. Notre classe `MoteurPingPong` a le niveau d'accès `public`, ce qui signifie que n'importe quelle classe peut y accéder. Faisons une petite expérience : ôtons le mot-clé `public` de la déclaration de la classe `MoteurPïngPong`. La classe `TablePingPong` ne peut même plus compiler ; le message d'erreur *MoteurPingPong ne peut pas être résolu ou ne correspond pas à un type* signifie que la classe `TablePingPong` *ne voit plus* la classe `MoteurPingPong`.

Si aucun niveau d'accès n'est spécifié, la classe a un niveau d'accès `package`, c'est-à-dire qu'elle n'est accessible qu'aux classes du même paquetage.

De la même façon, si tu oublies de donner un accès public aux méthodes de la classe `MoteurPingPong`, `TablePingPong` se plaindra en disant que ces méthodes ne sont pas visibles. Tu verras comment les niveaux d'accès sont utilisés dans le prochain chapitre, en créant le jeu de ping pong.

Le niveau d'accès `private` est utilisé pour cacher les variables ou méthodes de la classe aux yeux du monde extérieur. Pense à une voiture : la plupart des gens n'ont aucune idée de tout ce qui se cache sous le capot ou de ce qui se passe quand le conducteur appuie sur la pédale de frein.

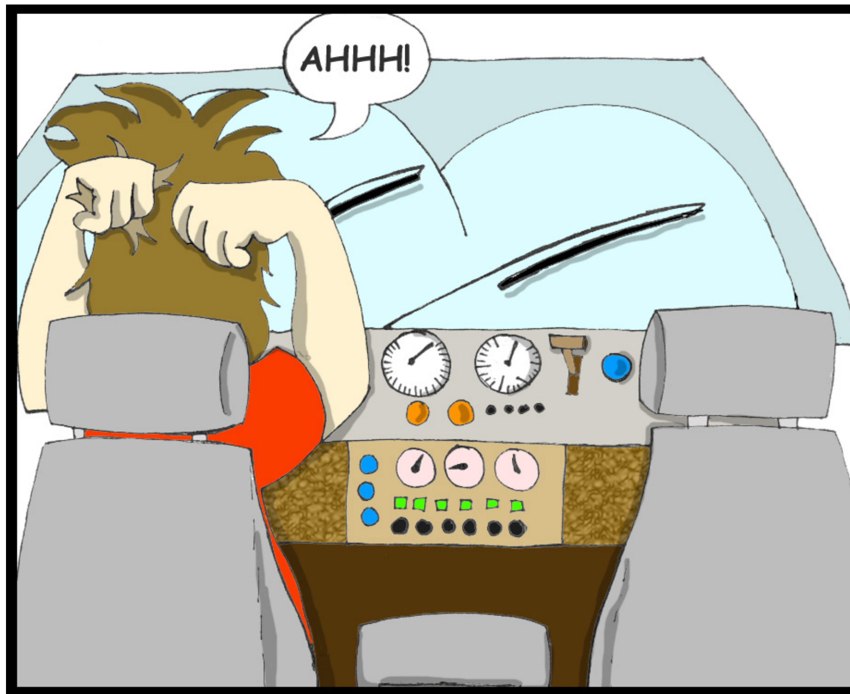
Examine l'exemple de code suivant - en Java, on peut dire que l'objet `Voiture` *expose* une seule méthode publique – `freiner()`, qui en interne peut appeler plusieurs autres méthodes qu'un conducteur n'a pas besoin de connaître. Par exemple, si le conducteur appuie trop fort sur la pédale, l'ordinateur de bord peut mettre en action des freins spéciaux anti-blocage. J'ai déjà dit auparavant que des programmes Java contrôlaient des robots aussi compliqués que ceux de la mission Mars Rovers, sans même parler des simples voitures.

```
public class Voiture {

  // Cette variable privée ne peut être utilisée
  // qu'à l'intérieur de cette classe
  private String conditionFreins;

  // La méthode publique freiner() appelle des méthodes
  // privées pour décider quels freins utiliser
  public void freiner(int pressionPédale) {
    boolean utiliserFreinsNormaux;
    utiliserFreinsNormaux =
              vérifierBesoinABS(pressionPédale);

    if (utiliserFreinsNormaux == true) {
      utiliserFreinsNormaux();
    } else {
      utiliserFreinsAntiBlocage();
    }
  }

  // Cette méthode privée ne peut être appelée
  // qu'à l'intérieur de cette classe
  private boolean vérifierBesoinABS(int pression) {
    if (pression > 100) {
      return true;
    } else {
      return false;
    }
  }

  // Cette méthode privée ne peut être appelée
  // qu'à l'intérieur de cette classe
  private void utiliserFreinsNormaux() {
    // Code qui envoie un signal aux freins normaux
  }

  // Cette méthode privée ne peut être appelée
  // qu'à l'intérieur de cette classe
  private void utiliserFreinsAntiBlocage() {
    // Code qui envoie un signal aux freins anti-blocage
  }
}
```

Le dernier mot-clé Java pour contrôler le niveau d'accès est `protected`. Si tu utilises ce mot-clé dans la signature d'une méthode, celle-ci est visible à l'intérieur de la classe, de ses sous-classes et des autres classes du même paquetage. Mais elle n'est pas utilisable par les classes indépendantes situées dans d'autres paquetages.

> L'une des caractéristiques principales des langages orientés objet est appelée *encapsulation*, c'est-à-dire la capacité à cacher et protéger les éléments d'une classe.

Quand tu conçois une classe, cache les méthodes et les variables membres qui ne doivent pas être visibles de l'extérieur. Si les concepteurs de voitures ne masquaient pas une partie des contrôles de la mécanique, le conducteur devrait gérer des centaines de boutons, d'interrupteurs et de jauges.



Dans la section suivante, la classe `Score` masque ses attributs en les déclarant `private`.

Retour sur les tableaux

Le programme `EnregistreurScores` du Chapitre 9 crée un tableau d'objets `String` pour stocker les noms et les scores des joueurs dans un fichier. Il est temps d'apprendre à utiliser des tableaux pour stocker tous types d'objets.

Cette fois, nous allons créer un objet représentant un score et lui donner des attributs tels que les nom et prénom du joueur, son score et la date de sa dernière partie.

La classe `Score` ci-dessous définit des *méthodes d'accès en lecture et en écriture* (*getters* et *setters*) pour chacun des ses attributs, qui sont déclarés privés. Bon, il n'est peut-être pas évident de comprendre pourquoi la classe appelante ne peut pas simplement affecter la valeur de l'attribut `score` comme ceci :

```
Score.score = 250;
```

au lieu de :

```
Score.affecterScore(250);
```

Essaie d'élargir ton horizon. Imagine que, plus tard, nous décidions que notre programme doive jouer un morceau de musique quand un joueur atteint le score de 500 ? Si la classe `Score` a une méthode `affecterScore()`, il suffit de modifier cette méthode en lui ajoutant le code qui vérifie le score et joue de la musique si nécessaire. La classe appelante appelle toujours de la même façon la version musicale de la méthode `affecterScore()`. Si la classe appelante affectait directement la valeur, elle devrait elle-même implanter les changements musicaux. Et si tu voulais réutiliser la classe `Score` dans deux programmes de jeu différents ? En utilisant la modification directe d'attributs, tu devrais implanter les changements dans deux classes appelantes, alors qu'une méthode d'affectation *encapsule* les changements qui sont immédiatement fonctionnels pour chaque classe appelante.

```
import java.util.Date;
import java.text.SimpleDateFormat;

public class Score {
  private String prénom;
  private String nom;
  private int score;
  private Date dateDernièrePartie;

  public String lirePrénom() {
    return prénom;
  }
  public void affecterPrénom(String prénom) {
    this.prénom = prénom;
  }
  public String lireNom() {
    return nom;
  }
  public void affecterNom(String nom) {
    this.nom = nom;
  }
  public int lireScore() {
    return score;
  }
  public void affecterScore(int score) {
    this.score = score;
  }
  public Date lireDateDernièrePartie() {
    return dateDernièrePartie;
  }
  public void affecterDateDernièrePartie(Date
                                  dateDernièrePartie) {
    this.dateDernièrePartie = dateDernièrePartie;
  }
  // Concatène tous les attributs en une chaîne
  // et y ajoute un caractère fin de ligne.
  // Cette méthode est pratique, par exemple pour
  // afficher toutes les valeurs d'un coup, comme ceci :
  // System.out.println(myScore.toString());
  // NDT : comme cette méthode surcharge Object.toString(),
  // tu es certain que la bonne représentation de Score
  // est utilisée partout où Java en a besoin (c'est pour
  // cela que nous ne l'avons pas appelée convertirEnString).
  public String toString() {
    String chaîneScore = prénom + " " +
        nom + " " +  score + " " + SimpleDateFormat.
        getDateInstance().format(dateDernièrePartie) +
        System.getProperty("line.separator");
    return chaîneScore;
  }
}
```

Le programme `EnregistreurScores2` crée des instances de l'objet `Score` et affecte des valeurs à leurs attributs.

```java
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;
import java.util.Date;

public class EnregistreurScores2 {

  /**
   La méthode main exécute les actions suivantes :
   1. Crée une instance de tableau
   2. Crée les objets Score et les stocke dans le tableau
   3. Ecrit les données de scores dans un fichier
  */
  public static void main(String[] args) {

    FileWriter monFichier = null;
    BufferedWriter tampon = null;

    Date ceJour = new Date();
    Score scores[] = new Score[3];

    // Joueur n°1
    scores[0]= new Score();
    scores[0].affecterPrénom("Jean");
    scores[0].affecterNom("Dupont");
    scores[0].affecterScore(250);
    scores[0].affecterDateDernièrePartie(ceJour);

    // Joueur n°2
    scores[1]= new Score();
    scores[1].affecterPrénom("Anne");
    scores[1].affecterNom("Durand");
    scores[1].affecterScore(300);
    scores[1].affecterDateDernièrePartie(ceJour);

    // Joueur n°3
    scores[2]= new Score();
    scores[2].affecterPrénom("Eskil");
    scores[2].affecterNom("Pemieufer");
    scores[2].affecterScore(190);
    scores[2].affecterDateDernièrePartie(ceJour);
```

Classe `EnregistreurScores2` (partie 1 de 2)

```
    try {
      monFichier = new FileWriter("c:\\scores2.txt");
      tampon = new BufferedWriter(monFichier);

      for (int i = 0; i < scores.length; i++) {
        // Convertit chaque Score en String
        // et l'écrit dans scores2.txt
        tampon.write(scores[i].toString());
        System.out.println("Ecriture des données de " +
                                        scores[i].lireNom());
      }
      System.out.println("Ecriture du fichier terminée");
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        tampon.flush();
        tampon.close();
        monFichier.close();
      } catch (IOException e1) {
        e1.printStackTrace();
      }
    }
  }
}
```

Classe `EnregistreurScores2` (partie 2 de 2)

Si un programme essaie d'accéder à un élément du tableau dont l'indice est trop grand, par exemple ici `scores[5].lireNom()`, Java lève l'exception `ArrayIndexOutOfBoundsException`.

Classe `ArrayList`

Le paquetage `java.util` contient des classes plutôt pratiques pour stocker plusieurs instances (*des collections*) d'objets en mémoire. Certaines des collections populaires de ce paquetage sont les classes `ArrayList` (liste stockée dans un tableau), `Vector` (vecteur), `Hashtable` et `HashMap` (deux sortes de tables de hachage) et `List` (liste). Je vais te montrer comment utiliser la classe `java.util.ArrayList`.

L'inconvénient des tableaux normaux est que tu dois connaître à l'avance le nombre d'éléments du tableau. Rappelle-toi, pour créer une instance de tableau, tu dois mettre un nombre entre les crochets :

```
String[] mesAmis = new String[5];
```

La classe `ArrayList` n'a pas cette contrainte. Tu peux créer une instance de cette collection sans savoir combien d'objets il y aura : ajoute simplement de nouveaux éléments quand tu en as besoin.

Pourquoi utiliser des tableaux, alors, et ne pas utiliser systématiquement des `ArrayList` ? Malheureusement, rien n'est gratuit et il faut payer le prix du confort : `ArrayList` est un peu plus lent qu'un tableau normal et on ne peut y stocker que des objets, pas des nombres de type `int`.

Pour créer et remplir un objet `ArrayList`, il faut l'instancier, créer des instances des objets que tu souhaites y stocker, puis les ajouter à l'`ArrayList` en appelant sa méthode `add()`. Le petit programme ci-dessous remplit un `ArrayList` avec des objets de type `String` puis affiche le contenu de cette collection.

```
import java.util.ArrayList;

public class DémoArrayList {
  public static void main(String[] args) {
    // Crée et remplit un ArrayList
    ArrayList amis = new ArrayList();
    amis.add("Marie");
    amis.add("Anne");
    amis.add("David");
    amis.add("Rémi");

    // Combien d'amis y a-t-il ?
    int nombreAmis = amis.size();

    // Affiche le contenu de l'ArrayList
    for (int i = 0; i < nombreAmis; i++) {
      System.out.println("L'ami(e) n°" + i + " est "
                                        + amis.get(i));
    }
  }
}
```

Ce programme affiche les lignes suivantes :

```
L'ami(e) n°0 est Marie
L'ami(e) n°1 est Anne
L'ami(e) n°2 est David
L'ami(e) n°3 est Rémi
```

La méthode `get()` *extrait* d'un `ArrayList` l'élément situé à une position donnée. Puisqu'on peut stocker n'importe quel objet dans une collection, la méthode `get()` renvoie chaque élément comme un `Object` Java ; le programme a la responsabilité de convertir explicitement cet objet en un type de données approprié. Nous n'avons pas besoin de le faire dans l'exemple précédent uniquement parce que

nous stockons des objets de type `String` dans la collection `amis` et que Java convertit automatiquement `Object` en `String`. Mais si tu stockes d'autres objets dans un `ArrayList`, par exemple des instances de la classe `Poisson`, le code correct pour ajouter et extraire un `Poisson` ressemble plutôt à celui du programme `BocalAPoissons` ci-dessous. Le programme crée une paire d'instances de la classe `Poisson`, affecte des valeurs à leurs attributs `couleur`, `poids` et `profondeurCourante`, puis les stocke dans l'`ArrayList` `bocalAPoissons`. Le programme extrait ensuite les objets de cette collection, convertit chacun d'entre eux en `Poisson` et affiche les valeurs de ses attributs.

```
import java.util.ArrayList;

public class BocalAPoissons {
  public static void main(String[] args) {
    ArrayList bocalAPoissons = new ArrayList();
    Poisson lePoisson;

    Poisson unPoisson = new Poisson(20);

    unPoisson.couleur = "Rouge";
    unPoisson.poids = 1;
    bocalAPoissons.add(unPoisson);

    unPoisson = new Poisson(10);
    unPoisson.couleur = "Vert";
    unPoisson.poids = 2;
    bocalAPoissons.add(unPoisson);

    int nombrePoissons = bocalAPoissons.size();

    for (int i = 0; i < nombrePoissons; i++) {
      lePoisson = (Poisson) bocalAPoissons.get(i);
      System.out.println("Attrapé le poisson " +
        lePoisson.couleur + " qui pèse " +
        lePoisson.poids + " livres. Profondeur : " +
        lePoisson.profondeurCourante);
    }
  }
}
```

Voici la sortie du programme `BocalAPoissons` :

```
Attrapé le poisson Rouge qui pèse 1.0 livres. Profondeur : 20
Attrapé le poisson Vert qui pèse 2.0 livres. Profondeur : 10
```

Maintenant que tu connais les niveaux d'accès Java, on peut modifier un peu les classes `AnimalFamilier` et `Poisson`. Des variables telles que `age`, `couleur`, `poids` et `taille` devraient être déclarées comme `protected` et la variable `profondeurCourante` devrait être `private`. Tu dois ajouter de nouvelles méthodes publiques, telles que

`lireAge()` pour renvoyer la valeur de la variable `age` et `affecterAge()` pour affecter la valeur de cette variable.

Les programmeurs élégants ne permettent pas à une classe de modifier directement les propriétés d'une autre ; la classe doit fournir les méthodes qui modifient ses internes. C'est pourquoi la classe `Score` de la section précédente est conçue avec des variables `private`, qui peuvent être modifiées et lues avec les méthodes d'accès appropriées.

Dans ce chapitre, je t'ai présenté divers éléments et techniques Java qui semblent sans relation les uns avec les autres. Mais tous ces éléments sont fréquemment utilisés par les programmeurs Java professionnels. Après avoir effectué les exercices pratiques de ce chapitre, tu devrais mieux comprendre comment tous ces éléments fonctionnent ensemble.

Autres lectures

1. Collections Java :
http://java.sun.com/docs/books/tutorial/collections/intro/

2. Classe `ArrayList`:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/ArrayList.html

3. Classe `Vector`:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/Vector.html

4. Classe `Calendar`:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html

Exercices

1. Ajoute un constructeur surchargé sans argument à la classe `Poisson`. Ce constructeur doit initialiser à 10 mètres la profondeur de départ. La classe `MaîtrePoisson` va créer une instance de l'objet `Poisson` comme ceci :

```
Poisson monPoisson = new Poisson();
```

2. Ajoute un constructeur à quatre arguments à la classe `Score`. Crée un programme `EnregistreurScores3` qui génère les instances des objets `Score` non pas en utilisant des méthodes d'affectation, mais plutôt au moment de la création des scores, comme ceci :

```
Score unScore = new Score("Jean",
                  "Dupont", 250, ceJour);
```

Exercices pour les petits malins

Regarde en ligne comment s'utilise la classe `Vector` et essaie de créer un programme `DémoVector` similaire au programme `DémoArrayList`.

Chapitre 11. Retour sur le graphique – le jeu de ping-pong

Dans les chapitres 5, 6 et 7, nous avons utilisé certains composants AWT et Swing. Je vais maintenant te montrer comment dessiner et déplacer dans une fenêtre des objets tels que des ovales, des rectangles et des lignes. Nous verrons aussi comment traiter les événements de la souris et du clavier. Pour rendre plus amusant ces sujets ennuyeux, nous allons explorer toutes ces choses en créant un jeu de ping-pong. Il y aura deux joueurs, que j'appelle *l'enfant* et *l'ordinateur*.

Stratégie

Commençons par les règles du jeu :

1. La partie se poursuit jusqu'à ce que l'un des joueurs (l'enfant ou l'ordinateur) atteigne le score de 21.
2. Les mouvements de la raquette de l'enfant sont contrôlés par la souris.
3. Le score est affiché en bas de la fenêtre.
4. Une nouvelle partie commence quand un joueur appuie sur la touche *N* du clavier. *Q* met fin à la partie. *S* effectue le service.
5. Seul l'enfant peut servir.
6. Pour gagner un point, la balle doit dépasser la ligne à la verticale de la raquette (sans avoir été bloquée par la raquette).
7. Quand l'ordinateur renvoie la balle, elle ne peut se déplacer qu'horizontalement vers la droite.
8. Si la balle rencontre la raquette de l'enfant dans la moitié supérieure de la table, elle doit se déplacer dans la direction haut-gauche. Si la balle se trouve dans la partie inférieure de la table, elle doit se déplacer dans la direction bas-gauche.

Tu dois penser que ça va être difficile à programmer. L'astuce consiste à découper une tâche complexe en un ensemble de tâches plus petites et plus simples, puis d'essayer de les résoudre une par une.

Cette méthode de travail est appelée *réflexion analytique* (*analytical thinking*). Elle est utile non seulement en programmation, mais aussi dans la vie en général : ne sois pas frustré si tu ne peux pas atteindre un objectif important, mais découpe-le en un ensemble d'objectifs plus petits que tu peux atteindre un par un !

C'est pour cette raison que la première version du jeu n'implante que quelques-unes de ces règles : elle dessine la table, déplace la raquette et affiche les coordonnées du pointeur de la souris quand on clique.

Au lieu de te contenter de dire "mon ordinateur ne fonctionne pas" (vaste problème), essaye de voir ce qui ne va pas exactement (trouve un problème plus petit).

1. L'ordinateur est-il branché au secteur (oui/non) ? *Oui.*

2. Quand je mets en route l'ordinateur, l'écran avec toutes ses icônes est-il affiché (oui/non) ? *Oui.*

3. Puis-je déplacer le pointeur de la souris sur l'écran (oui/non) ? *Non.*

4. Le câble de la souris est-il branché correctement (oui/non) ? *Non.*

Il suffit de brancher la souris et l'ordinateur fonctionnera à nouveau ! Un gros problème a été réduit à la simple correction du branchement du câble de la souris.

Code

Ce jeu est constitué des trois classes suivantes :

- La classe `TableVertePingPong` se charge de la partie visuelle. Pendant la partie, elle affiche la table, les raquettes et la balle.

- La classe `MoteurJeuPingPong` est responsable du calcul des coordonnées de la balle et des raquettes, du démarrage et de l'arrêt de la partie, et du service. Cette classe passe les coordonnées des composants à la classe `TableVertePingPong`, qui rafraîchit son affichage en fonction de celles-ci.

- L'interface `ConstantesDuJeu` contient les déclarations de toutes les constantes nécessaires au jeu, telles que la largeur et la hauteur de la table, les positions de départ des raquettes, etc..

La table de ping-pong ressemble à ceci :

La première version de ce jeu ne fait que trois choses :

- Afficher une table de ping pong verte.
- Afficher les coordonnées du pointeur de la souris quand on clique.
- Déplacer la raquette de l'enfant vers le haut et vers le bas.

Le code de notre classe `TableVertePingPong`, qui est une sous-classe de la classe Swing `JPanel`, se trouve deux pages plus bas. Regarde-le pendant que tu lis le texte ci-dessous.

Puisque notre jeu a besoin de connaître les coordonnées exactes du pointeur de la souris, le constructeur de la classe `TableVertePingPong` crée une instance du récepteur d'événements `MoteurJeuPingPong`. Cette classe effectue certaines actions quand l'enfant clique ou simplement déplace la souris.

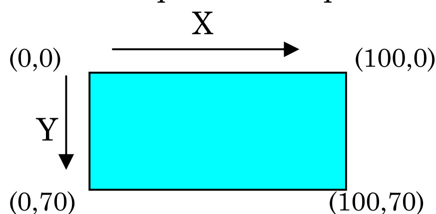
La méthode `ajouteAuCadre()` crée un libellé qui affichera les coordonnées de la souris.

Cette classe n'est pas une applet, raison pour laquelle elle utilise la méthode `paintComponent()` au lieu de la méthode `paint()`. Cette méthode est appelée soit par Java quand il est nécessaire de rafraîchir la fenêtre, soit quand notre programme appelle la méthode `repaint()`. Tu as bien lu, la méthode `repaint()` appelle en interne la méthode `paintComponent()` et fournit à ta classe un objet `Graphics` pour que tu puisses dessiner dans la fenêtre. Nous appellerons cette méthode à chaque fois que nous aurons recalculé les

coordonnées des raquettes ou de la balle pour les afficher au bon endroit.

Pour dessiner une raquette, il faut lui affecter une couleur, puis remplir un rectangle avec cette peinture à l'aide de la méthode `fillRect()` (*remplir rectangle*). Cette méthode doit connaître les coordonnées `X` et `Y` du coin haut gauche du rectangle, ainsi que sa largeur et sa hauteur en pixels. La balle est dessinée à l'aide de la méthode `fillOval()` (*remplir ovale*), qui a besoin de connaître les coordonnées du centre de l'ovale, sa hauteur et sa largeur. Si la hauteur et la largeur de l'ovale sont identiques, c'est un cercle.

Dans une fenêtre, la coordonnée `X` augmente de la gauche vers la droite et la coordonnée `Y` augmente du haut vers le bas. Par exemple, les coordonnées des coins de ce rectangle de largeur 100 pixels et de hauteur 70 sont indiquées entre parenthèses :



Une autre méthode intéressante est la méthode `getPreferredSize()`. Nous créons une instance de la classe Swing `Dimension` pour fixer la taille de la table. Java a besoin de connaître les dimensions de la fenêtre et appelle pour cela la méthode `getPreferredSize()` de l'objet `TableVertePingPong`. Cette méthode retourne à Java un objet `Dimension` que nous avons créé dans le code en fonction de la taille de notre table.

Les deux classes table et moteur utilisent des valeurs constantes, qui ne changent jamais. Par exemple, la classe `TableVertePingPong` utilise la largeur et la hauteur de la table et `MoteurJeuPingPong` doit connaître les incréments de déplacement de la balle (plus l'incrément est petit, plus le mouvement est fluide). Il est commode de réunir toutes les constantes (variables `final`) au sein d'une interface. Dans notre jeu, l'interface se nomme `ConstantesDuJeu`. Si une classe a besoin de ces valeurs, il suffit d'ajouter `implements ConstantesDuJeu` à la déclaration de la classe pour utiliser n'importe laquelle des variables `final` de cette interface comme si elle était déclarée dans la classe elle-même. C'est pourquoi nos deux classes table et moteur implantent l'interface `ConstantesDuJeu`.

Si tu décides de modifier la taille de la table, de la balle ou des raquettes, tu n'as besoin de le faire qu'à un endroit : dans l'interface `ConstantesDuJeu`. Examinons le code de la classe `TableVertePingPong` et de l'interface `ConstantesDuJeu`.

```
package écrans;

import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.BoxLayout;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import java.awt.Point;
import java.awt.Dimension;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.Color;
import moteur.MoteurJeuPingPong;


/**
* Cette classe dessine une table de ping-pong verte
* et affiche les coordonnées du point où l'utilisateur
* a cliqué.
*/
public class TableVertePingPong
    extends JPanel
    implements ConstantesDuJeu {

  JLabel label;
  public Point point = new Point(0,0);

  public int raquetteOrdinateur_X = 15;
  private int raquetteEnfant_Y = RAQUETTE_ENFANT_Y_DEPART;

  Dimension taillePréférée = new
                   Dimension(LARGEUR_TABLE, HAUTEUR_TABLE);

  // Cette méthode affecte sa taille au cadre.
  // Elle est appelée par Java.
  public Dimension getPreferredSize() {
    return taillePréférée;
  }
```

Classe `TableVertePingPong` (partie 1 de 3)

```
// Constructeur.
TableVertePingPong() {

  MoteurJeuPingPong moteurJeu =
                              new MoteurJeuPingPong(this);
  // Reçoit les clics pour l'affichage de leurs coordonnées
  addMouseListener(moteurJeu);
  // Reçoit les mouvements de la souris pour
  // le déplacement des raquettes
  addMouseMotionListener(moteurJeu);
}

// Ajoute à une fenêtre un panneau contenant cette table et
// un JLabel
void ajouteAuCadre(Container conteneur) {
  conteneur.setLayout(new BoxLayout(conteneur,
                                  BoxLayout.Y_AXIS));
  conteneur.add(this);
  label = new JLabel("Coordonnées...");
  conteneur.add(label);
}

// Repeint la fenêtre. Cette méthode est appelée par Java
// quand il est nécessaire de rafraîchir l'écran ou quand
// la méthode repaint() est appelée par le
// MoteurJeuPingPong.
public void paintComponent(Graphics contexteGraphique) {
  super.paintComponent(contexteGraphique);

  // Dessine la table verte
  contexteGraphique.setColor(Color.GREEN);
  contexteGraphique.fillRect(
                  0, 0, LARGEUR_TABLE, HAUTEUR_TABLE);

  // Dessine la raquette droite
  contexteGraphique.setColor(Color.yellow);
  contexteGraphique.fillRect(RAQUETTE_ENFANT_X_DEPART,
                           raquetteEnfant_Y, 5, 30);

  // Dessine la raquette gauche
  contexteGraphique.setColor(Color.blue);
  contexteGraphique.fillRect(
                  raquetteOrdinateur_X, 100, 5, 30);

  // Dessine la balle
  contexteGraphique.setColor(Color.red);
  contexteGraphique.fillOval(25, 110, 10, 10);

  // Dessine les lignes
  contexteGraphique.setColor(Color.white);
  contexteGraphique.drawRect(10, 10, 300, 200);
  contexteGraphique.drawLine(160, 10, 160, 210);
```

Classe `TableVertePingPong` (partie 2 de 3)

```
    // Affiche un point sous forme de rectangle de 2x2 pixels
    if (point != null) {
      label.setText("Coordonnées (x, y) : " +
                      point.x + ", " + point.y);
      contexteGraphique.fillRect(point.x, point.y, 2, 2);
    }
  }

  // Affecte sa position courante à la raquette de l'enfant
  public void positionnerRaquetteEnfant_Y(int y) {
    this.raquetteEnfant_Y = y;
  }

  // Retourne la position courante de la raquette de l'enfant
  public int coordonnéeRaquetteEnfant_Y() {
    return raquetteEnfant_Y;
  }

  public static void main(String[] args) {
    // Crée une instance du cadre
    JFrame monCadre = new JFrame("Table verte de ping-pong");
    // Permet la fermeture de la fenêtre par clic sur la
    // petite croix dans le coin.
    monCadre.setDefaultCloseOperation(
                 WindowConstants.EXIT_ON_CLOSE);

    TableVertePingPong table = new TableVertePingPong();
    table.ajouteAuCadre(monCadre.getContentPane());
    // Affecte sa taille au cadre et le rend visible.
    monCadre.pack();
    monCadre.setVisible(true);
  }
}
```

Classe `TableVertePingPong` (partie 3 de 3)

Voici l'interface `ConstantesDuJeu`. Toutes les valeurs des variables sont en pixels. Ecris les noms des variables `final` en lettres majuscules.

```
package écrans;

public interface ConstantesDuJeu {
  public final int LARGEUR_TABLE = 320;
  public final int HAUTEUR_TABLE = 220;
  public final int RAQUETTE_ENFANT_Y_DEPART = 100;
  public final int RAQUETTE_ENFANT_X_DEPART = 300;
  public final int HAUT_TABLE = 12;
  public final int BAS_TABLE = 180;

  public final int INCREMENT_RAQUETTE = 4;
}
```

Un programme en cours d'exécution ne peut pas modifier les valeurs de ces variables, puisqu'elles sont déclarées comme `final`. Mais si, par exemple, tu décides d'augmenter la taille de la table, tu dois modifier

les valeurs de LARGEUR_TABLE et de HAUTEUR_TABLE, puis recompiler l'interface ConstantesDuJeu.

Dans ce jeu, les décisions sont prises par la classe MoteurJeuPingPong, qui implante deux interfaces concernant la souris. MouseListener n'a de code que dans la méthode mousePressed(). A chaque clic, cette méthode dessine un point blanc sur la table et affiche ses coordonnées. En réalité, ce code n'est d'aucune utilité pour notre jeu, mais il te montre d'une façon simple comment obtenir les coordonnées de la souris de l'objet MouseEvent passé au programme par Java.

La méthode mousePressed() affecte les coordonnées de la variable point en fonction de la position du pointeur de la souris au moment où l'utilisateur a cliqué. Une fois les coordonnées affectées, elle demande à Java de repeindre la table.

MouseMotionListener répond aux mouvements de la souris au-dessus de la table. Nous utilisons sa méthode mouseMoved() pour déplacer la raquette de l'enfant vers le haut et vers le bas.

La méthode mouseMoved() calcule la position suivante de la raquette de l'enfant. Quand le pointeur de la souris se trouve au-dessus de la raquette (la coordonnée Y de la souris est inférieure à la coordonnée Y de la raquette), cette méthode assure que la raquette n'ira pas plus loin que le haut de la table.

Quand le constructeur de la table crée l'objet moteur, il lui passe une référence à l'instance de table (le mot-clé this représente une référence à l'emplacement en mémoire de l'objet TableVertePingPong). Le moteur peut maintenant "parler" à la table, par exemple pour affecter de nouvelles coordonnées à la balle ou pour repeindre la table si nécessaire. Si cette partie n'est pas claire, tu peux relire la section concernant le passage de données entre classes, au Chapitre 6.

Dans notre jeu, les raquettes se déplacent verticalement d'une position à une autre en utilisant un incrément de quatre pixels, comme le définit l'interface ConstantesDuJeu (la classe moteur implante cette interface). Par exemple, la ligne suivante soustrait quatre à la valeur de la variable raquetteEnfant_Y :

```
raquetteEnfant_Y -= INCREMENT_RAQUETTE;
```

Si la coordonnée Y de la raquette valait 100, elle devient 96 après cette ligne de code, ce qui signifie que la raquette doit être déplacée vers le haut. Tu obtiendrais le même résultat en utilisant la syntaxe suivante :

```
raquetteEnfant_Y = raquetteEnfant_Y - INCREMENT_RAQUETTE;
```

Souviens-toi, nous avons abordé les différents moyens de modifier la valeur d'une variable au Chapitre 3.

Voici la classe `MoteurJeuPingPong`.

```
package moteur;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import écrans.*;
public class MoteurJeuPingPong  implements
       MouseListener, MouseMotionListener, ConstantesDuJeu {

  TableVertePingPong table;
  public int raquetteEnfant_Y = RAQUETTE_ENFANT_Y_DEPART;
  // Constructeur. Stocke une référence à la table.
  public MoteurJeuPingPong(TableVertePingPong tableVerte) {
    table = tableVerte;
  }
  // Méthodes requises par l'interface MouseListener.
  public void mousePressed(MouseEvent événement) {
    // Récupère les coordonnées X et Y du pointeur de la
    // souris et les affecte au "point blanc" sur la table.
    table.point.x = événement.getX();
    table.point.y = événement.getY();
    // La méthode repaint appelle en interne la méthode
    // paintComponent() de la table qui rafraîchit la
    // fenêtre.
    table.repaint();
  }
  public void mouseReleased(MouseEvent événement) {}
  public void mouseEntered(MouseEvent événement) {}
  public void mouseExited(MouseEvent événement) {}
  public void mouseClicked(MouseEvent événement) {}

  // Méthodes requises par l'interface MouseMotionListener.
  public void mouseDragged(MouseEvent événement) {}

  public void mouseMoved(MouseEvent événement) {
    int souris_Y = événement.getY();
    // Si la souris est au-dessus de la raquette de l'enfant
    // et que la raquette n'a pas dépassé la limite
    // supérieure de la table, la déplace vers le haut ;
    // sinon, la déplace vers le bas.
    if (souris_Y < raquetteEnfant_Y
                   && raquetteEnfant_Y > HAUT_TABLE) {
      raquetteEnfant_Y -= INCREMENT_RAQUETTE;
    } else if (raquetteEnfant_Y < BAS_TABLE) {
      raquetteEnfant_Y += INCREMENT_RAQUETTE;
    }
    // Affecte la nouvelle position de la raquette dans la
    // classe table
    table.positionnerRaquetteEnfant_Y(raquetteEnfant_Y);
    table.repaint();
  }
}
```

Bases des fils d'exécution Java

Jusqu'ici, tous nos programmes s'exécutent en séquence – une action après l'autre. Si un programme appelle deux méthodes, la seconde méthode attend que la première se soit terminée. Autrement dit, chacun de nos programmes n'a qu'un *fil d'exécution* (*thread of execution*).

Cependant, dans la vraie vie, nous pouvons faire plusieurs choses en même temps, comme manger, parler au téléphone, regarder la télévision et faire nos devoirs. Pour mener à bien toutes ces actions *en parallèle*, nous utilisons plusieurs *processeurs* : les mains, les yeux et la bouche.

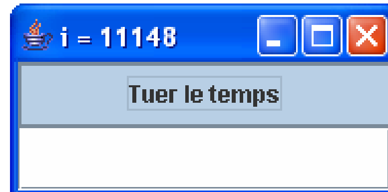
Certains des ordinateurs les plus chers ont aussi deux processeurs ou plus. Mais sans doute ton ordinateur n'a-t-il qu'un processeur qui effectue les calculs, envoie les commandes à l'écran, au disque, aux ordinateurs distants, etc.

Même un unique processeur peut exécuter plusieurs actions à la fois si le programme utilise des *fils d'exécution multiples* (*multiple threads*). Une classe Java peut lancer plusieurs fils d'exécution qui obtiennent chacun à leur tour des tranches du temps de l'ordinateur.

Un bon exemple de programme capable de créer de multiples fils d'exécution est un navigateur web. Tu peux naviguer sur Internet tout en téléchargeant des fichiers : un seul programme exécute deux fils d'exécution.

La version suivante de notre jeu de ping-pong a un fil d'exécution dédié à l'affichage de la table. Le second fil d'exécution calcule les coordonnées de la balle et des raquettes et envoie les commandes au premier fil d'exécution pour repeindre la fenêtre. Mais tout d'abord, je vais te montrer deux programmes très simples pour mieux te faire comprendre pourquoi les fils d'exécution sont nécessaires.

Chacun de ces programmes d'exemple affiche un bouton et un champ textuel.



Quand on appuie sur le bouton *Tuer le temps*, le programme entre dans une boucle qui incrémente une variable trente mille fois. La valeur courante de la variable compteur est affichée dans la barre de titre de la fenêtre.

La classe `ExempleSansFils` n'a qu'un fil d'exécution et *il est impossible de saisir quelque chose dans le champ textuel tant que la boucle n'est pas terminée*. Cette boucle accapare tout le temps de calcul du processeur, c'est pourquoi la fenêtre est verrouillée.

Compile et exécute cette classe et constate par toi-même que la fenêtre est verrouillée pendant un moment. Note que cette classe crée une instance de `JTextField` et la passe au contenu de la fenêtre sans déclarer de variable représentant cette instance. Si tu ne comptes pas lire ou modifier d'attributs de cet objet dans ton programme, tu n'as pas besoin de mémoriser une telle référence.

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ExempleSansFils extends JFrame
                            implements ActionListener {
  // Constructeur
  ExempleSansFils() {
    // Crée un cadre contenant un bouton et un champ textuel
    GridLayout disposition = new GridLayout(2,1);
    this.getContentPane().setLayout(disposition);
    JButton monBouton = new JButton("Tuer le temps");
    monBouton.addActionListener(this);
    this.getContentPane().add(monBouton);
    this.getContentPane().add(new JTextField());
  }
  // Traite les clics sur le bouton
  public void actionPerformed(ActionEvent événement) {
    // Tue juste un peu le temps pour montrer que
    // les contrôles de la fenêtre sont verrouillés.
    for (int i = 0; i < 30000; i++) {
      this.setTitle("i = " + i);
    }
  }

  public static void main(String[] args) {
    // Crée une instance du cadre
    ExempleSansFils maFenêtre = new ExempleSansFils();
    // Permet la fermeture de la fenêtre par clic sur la
    // petite croix dans le coin.
    maFenêtre.setDefaultCloseOperation(
                  WindowConstants.EXIT_ON_CLOSE);

    // Affecte sa taille au cadre - coordonnées du coin haut
    // gauche, largeur et hauteur.
    maFenêtre.setBounds(0, 0, 200, 100);
    // Rend la fenêtre visible.
    maFenêtre.setVisible(true);
  }
}
```

La version suivante de cette petite fenêtre crée et lance un fil d'exécution séparé pour la boucle ; le fil d'exécution principal de la fenêtre te permet de taper dans le champ textuel pendant que la boucle s'exécute.

En Java, on peut créer un fil d'exécution par l'un des moyens suivants :

1. Créer une instance de la classe Java `Thread` et lui passer un objet qui implante l'interface `Runnable`. Si cette classe implante l'interface `Runnable`, le code ressemble à ceci :

```
Thread travailleur = new Thread(this);
```

Cette interface t'impose d'écrire dans la méthode `run()` le code qui doit être exécuté comme un fil d'exécution séparé. Mais pour lancer le fil d'exécution, tu dois appeler la méthode `start()`, qui va en fait appeler ta méthode `run()`. D'accord, c'est un peu troublant, mais c'est comme ça que tu démarres le fil d'exécution :

```
travailleur.start();
```

2. Créer une sous-classe de la classe `Thread` et y implanter la méthode `run()`. Pour démarrer le fil d'exécution, appeler la méthode `start()`.

```
public class MonFil extends Thread {

  public static void main(String[] args) {
      MonFil travailleur = new MonFil();
      travailleur.start();
  }
  public void run() {
    // Place ton code ici.
  }
}
```

J'utilise la première méthode dans la classe `ExempleAvecFils` parce que cette classe hérite déjà de `JFrame` et qu'on ne peut pas hériter de plus d'une classe en Java.

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ExempleAvecFils extends JFrame
                       implements ActionListener, Runnable {

  // Constructeur
  ExempleAvecFils() {
    // Crée un cadre contenant un bouton et un champ textuel.
    GridLayout disposition = new GridLayout(2,1);
    this.getContentPane().setLayout(disposition);
    JButton monBouton = new JButton("Tuer le temps");
    monBouton.addActionListener(this);
    this.getContentPane().add(monBouton);
    this.getContentPane().add(new JTextField());
  }

  public void actionPerformed(ActionEvent événement) {
    // Crée un fil et exécute le code "tuer le temps"
    // sans bloquer la fenêtre.
    Thread travailleur = new Thread(this);
    travailleur.start();  // Ceci appelle la méthode run()
  }

  public void run() {
    // Tue juste un peu le temps pour montrer que
    // les contrôles de la fenêtre NE sont PAS verrouillés.
    for (int i = 0; i < 30000; i++) {
      this.setTitle("i = " + i);
    }
  }

  public static void main(String[] args) {
    ExempleAvecFils maFenêtre = new ExempleAvecFils();
    // Permet la fermeture de la fenêtre par clic sur la
    // petite croix dans le coin.
    maFenêtre.setDefaultCloseOperation(
                         WindowConstants.EXIT_ON_CLOSE);

    // Affecte sa taille au cadre et le rend visible.
    maFenêtre.setBounds(0, 0, 200, 100);
    maFenêtre.setVisible(true);
  }
}
```

La classe `ExempleAvecFils` démarre un nouveau fil d'exécution lorsque tu cliques sur le bouton *Tuer le temps*. Après quoi, le fil d'exécution contenant la boucle et le fil d'exécution principal utilisent chacun leur tour des tranches du temps du processeur. Tu peux maintenant saisir du texte dans le champ textuel (le fil d'exécution principal) alors que l'autre fil d'exécution exécute la boucle !

Les fils d'exécution méritent une étude bien plus approfondie que ces quelques pages et je t'encourage à en améliorer ta compréhension par d'autres lectures.

Fin du jeu de ping-pong

Après cette rapide introduction des fils d'exécution, nous sommes en mesure de modifier le code des classes de notre jeu de ping-pong.

Commençons par la classe `TableVertePingPong`. Nous n'avons pas besoin d'afficher un point blanc quand l'utilisateur clique – il s'agissait juste d'un exercice pour apprendre comment afficher les coordonnées du pointeur de la souris. Nous allons donc retirer la déclaration de la variable `point` et les lignes qui dessinent le point blanc de la méthode `paintComponent()`. Le constructeur n'a plus besoin non plus d'ajouter `MouseListener`, qui ne sert qu'à afficher les coordonnées du point.

Par contre, cette classe devrait traiter certaines touches du clavier (*N* pour nouvelle partie, *S* pour servir et *Q* pour quitter le jeu). La méthode `addKeyListener()` s'en charge.

Pour que notre code soit un peu plus encapsulé, j'ai aussi déplacé les appels à la méthode `repaint()` de la classe moteur à `TableVertePingPong`. Celle-ci a maintenant la responsabilité de se repeindre lorsque c'est nécessaire.

J'ai aussi ajouté deux méthodes, pour modifier les positions de la balle et de la raquette de l'ordinateur et pour afficher des messages.

```
package écrans;

import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.BoxLayout;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import java.awt.Dimension;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.Color;
import moteur.MoteurJeuPingPong;
/**
*  Cette classe dessine la table de ping-pong, la balle et
*  les raquettes et affiche le score
*/
public class TableVertePingPong extends JPanel
                           implements ConstantesDuJeu {
  private JLabel label;

  private int raquetteOrdinateur_Y =
                             RAQUETTE_ORDINATEUR_Y_DEPART;
  private int raquetteEnfant_Y = RAQUETTE_ENFANT_Y_DEPART;
  private int balle_X = BALLE_X_DEPART;
  private int balle_Y = BALLE_Y_DEPART;

  Dimension taillePréférée = new
                     Dimension(LARGEUR_TABLE, HAUTEUR_TABLE);

  // Cette méthode affecte sa taille au cadre.
  // Elle est appelée par Java.
  public Dimension getPreferredSize() {
    return taillePréférée;
  }

  // Constructeur. Crée un récepteur d'événements souris.
  TableVertePingPong() {

    MoteurJeuPingPong moteurJeu =
                             new MoteurJeuPingPong(this);
    // Reçoit les mouvements de la souris pour déplacer la
    // raquette.
    addMouseMotionListener(moteurJeu);
    // Reçoit les événements clavier.
    addKeyListener(moteurJeu);
  }
```

Classe `TableVertePingPong` (partie 1 de 3)

```
  // Ajoute à un cadre la table et un JLabel
  void ajouteAuCadre(Container conteneur) {
    conteneur.setLayout(new BoxLayout(conteneur,
                                      BoxLayout.Y_AXIS));
    conteneur.add(this);
    label = new JLabel(
      "Taper N pour une nouvelle partie, S pour servir" +
      " ou Q pour quitter");
    conteneur.add(label);
  }

  // Repeint la fenêtre. Cette méthode est appelée par Java
  // quand il est nécessaire de rafraîchir l'écran ou quand
  // la méthode repaint() est appelée.
  public void paintComponent(Graphics contexteGraphique) {

    super.paintComponent(contexteGraphique);
    // Dessine la table verte
    contexteGraphique.setColor(Color.GREEN);
    contexteGraphique.fillRect(
                  0, 0, LARGEUR_TABLE, HAUTEUR_TABLE);

    // Dessine la raquette droite
    contexteGraphique.setColor(Color.yellow);
    contexteGraphique.fillRect(
                  RAQUETTE_ENFANT_X,
                  raquetteEnfant_Y,
                  LARGEUR_RAQUETTE, LONGUEUR_RAQUETTE);

    // Dessine la raquette gauche
    contexteGraphique.setColor(Color.blue);
    contexteGraphique.fillRect(RAQUETTE_ORDINATEUR_X,
                       raquetteOrdinateur_Y,
                       LARGEUR_RAQUETTE, LONGUEUR_RAQUETTE);

    // Dessine la balle
    contexteGraphique.setColor(Color.red);
    contexteGraphique.fillOval(balle_X, balle_Y, 10, 10);

    // Dessine les lignes blanches
    contexteGraphique.setColor(Color.white);
    contexteGraphique.drawRect(10, 10, 300, 200);
    contexteGraphique.drawLine(160, 10, 160, 210);

    // Donne le focus à la table, afin que le récepteur de
    // touches envoie les commandes à la table
    requestFocus();
 }

 // Affecte sa position courante à la raquette de l'enfant
  public void positionnerRaquetteEnfant_Y(int y) {
    this.raquetteEnfant_Y = y;
    repaint();
  }
```

Classe `TableVertePingPong` (partie 2 de 3)

```
  // Retourne la position courante de la raquette de l'enfant
  public int coordonnéeRaquetteEnfant_Y() {
     return raquetteEnfant_Y;
  }

  // Affecte sa position courante à la raquette de
  // l'ordinateur
  public void positionnerRaquetteOrdinateur_Y(int y) {
    this.raquetteOrdinateur_Y = y;
    repaint();
  }

  // Affecte le texte du message du jeu
  public void affecterTexteMessage(String texte) {
    label.setText(texte);
    repaint();
  }

  // Positionne la balle
  public void positionnerBalle(int x, int y) {
    balle_X = x;
    balle_Y = y;
    repaint();
  }

  public static void main(String[] args) {

    // Crée une instance du cadre
    JFrame monCadre = new JFrame("Table verte de ping-pong");

    // Permet la fermeture de la fenêtre par clic sur la
    // petite croix dans le coin.
    monCadre.setDefaultCloseOperation(
                                WindowConstants.EXIT_ON_CLOSE);
    TableVertePingPong table = new TableVertePingPong();
    table.ajouteAuCadre(monCadre.getContentPane());

    // Affecte sa taille au cadre et le rend visible.
    monCadre.setBounds(0, 0, LARGEUR_TABLE + 5,
                                         HAUTEUR_TABLE + 40);
    monCadre.setVisible(true);
  }
}
```

Classe `TableVertePingPong` (partie 3 de 3)

J'ai ajouté quelques variables déclarées `final` à l'interface `ConstantesDuJeu`. Tu devrais pouvoir en deviner le rôle d'après leurs noms.

```
package écrans;
/**
 * Cette interface contient toutes les définitions des
 * variables invariantes utilisées dans le jeu.
 */
public interface ConstantesDuJeu {
  // Taille de la table de ping-pong
  public final int LARGEUR_TABLE = 320;
  public final int HAUTEUR_TABLE = 220;
  public final int HAUT_TABLE = 12;
  public final int BAS_TABLE = 180;

  // Incrément du mouvement de la balle en pixels
  public final int INCREMENT_BALLE = 4;

  // Coordonnées maximum et minimum permises pour la balle
  public final int BALLE_X_MIN = 1 + INCREMENT_BALLE;
  public final int BALLE_Y_MIN = 1 + INCREMENT_BALLE;
  public final int BALLE_X_MAX =
                        LARGEUR_TABLE - INCREMENT_BALLE;
  public final int BALLE_Y_MAX =
                        HAUTEUR_TABLE - INCREMENT_BALLE;

  // Position de départ de la balle
  public final int BALLE_X_DEPART = LARGEUR_TABLE / 2;
  public final int BALLE_Y_DEPART = HAUTEUR_TABLE / 2;

  // Taille, positions et incrément des raquettes
  public final int RAQUETTE_ENFANT_X = 300;
  public final int RAQUETTE_ENFANT_Y_DEPART = 100;
  public final int RAQUETTE_ORDINATEUR_X = 15;
  public final int RAQUETTE_ORDINATEUR_Y_DEPART = 100;
  public final int INCREMENT_RAQUETTE = 2;
  public final int LONGUEUR_RAQUETTE = 30;
  public final int LARGEUR_RAQUETTE = 5;

  public final int SCORE_GAGNANT = 21;

  // Ralentit mes ordinateurs rapides ;
  // modifier la valeur si nécessaire.
  public final int DUREE_SOMMEIL = 10; // En millisecondes.

}
```

Voici les modifications les plus marquantes que j'ai effectuées dans la classe `MoteurJeuPingPong` :

- ✓ J'ai supprimé l'interface `MouseListener` et toutes ses méthodes, puisque nous ne nous intéressons plus aux clics. `MouseMotionListener` se charge de tous les déplacements de la souris.
- ✓ Cette classe implante maintenant l'interface `Runnable` ; les prises de décisions sont codées dans la méthode `run()`. Regarde le constructeur : j'y crée et lance un nouveau fil d'exécution. La

méthode `run()` applique les règles de la stratégie du jeu en plusieurs étapes, programmées à l'intérieur de la clause `if (balleServie)`. C'est une version courte de `if (balleServie == true)`.

- ✓ Je te prie de noter l'utilisation de la clause conditionnelle `if` pour affecter une valeur à la variable `rebondPossible` à l'étape 1. En fonction de l'expression en surbrillance, cette variable prendra la valeur `true` ou la valeur `false`.
- ✓ La classe implante l'interface `KeyListener` ; la méthode `keyPressed()` examine la lettre entrée au clavier pour démarrer la partie, quitter le jeu ou servir la balle. Le code de cette méthode permet à l'utilisateur de taper aussi bien des majuscules que des minuscules, par exemple `N` et `n`.
- ✓ J'ai ajouté plusieurs méthodes `private` telles que `afficherScore()`, `serviceEnfant()` et `balleSurLaTable()`. Ces méthodes sont déclarées privées parce qu'elles ne sont utilisées que dans cette classe et que les autres classes n'ont même pas à connaître leur existence. C'est un exemple d'*encapsulation*.
- ✓ Certains ordinateurs sont trop rapides, ce qui rend les déplacements de la balle difficiles à contrôler. C'est pourquoi j'ai ralenti le jeu en appelant la méthode `Thread.sleep()`. La méthode statique `sleep()` met ce fil d'exécution en pause pour le nombre de millisecondes qui lui est passé en argument.
- ✓ Pour ajouter un peu de piment au jeu, la balle se déplace en diagonale quand la raquette de l'enfant la frappe. C'est pourquoi le code modifie non seulement la coordonnée `X`, mais aussi la coordonnée `Y` de la balle.

```
package moteur;

import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import écrans.*;
/**
 * Cette classe est un récepteur de souris et de clavier.
 * Elle calcule les déplacements de la balle et des raquettes
 * et change leurs coordonnées.
 */
public class MoteurJeuPingPong implements Runnable,
       MouseMotionListener, KeyListener, ConstantesDuJeu {

  private TableVertePingPong table; // Référence à la table.
  private int raquetteEnfant_Y = RAQUETTE_ENFANT_Y_DEPART;
  private int raquetteOrdinateur_Y =
                             RAQUETTE_ORDINATEUR_Y_DEPART;
  private int scoreEnfant;
  private int scoreOrdinateur;
  private int balle_X; // position X de la balle
  private int balle_Y; // position Y de la balle
  private boolean déplacementGauche = true;
  private boolean balleServie = false;

  //Valeur en pixels du déplacement vertical de la balle.
  private int déplacementVertical;

  // Constructeur. Stocke une référence à la table.
  public MoteurJeuPingPong(TableVertePingPong tableVerte) {
    table = tableVerte;
    Thread travailleur = new Thread(this);
    travailleur.start();
  }
  // Méthodes requises par l'interface MouseMotionListener
  // (certaines sont vides, mais doivent être incluses dans
  // la classe de toute façon).

   public void mouseDragged(MouseEvent événement) {
   }
```

Classe `MoteurJeuPingPong` (partie 1 de 5)

```
public void mouseMoved(MouseEvent événement) {

  int souris_Y = événement.getY();

  // Si la souris est au-dessus de la raquette de l'enfant
  // et que la raquette n'a pas dépassé le haut de la
  // table, la déplace vers le haut ;
  // sinon, la déplace vers le bas.
  if (souris_Y < raquetteEnfant_Y &&
          raquetteEnfant_Y > HAUT_TABLE) {
    raquetteEnfant_Y -= INCREMENT_RAQUETTE;
  } else if (raquetteEnfant_Y < BAS_TABLE) {
    raquetteEnfant_Y += INCREMENT_RAQUETTE;
  }

  // Affecte la nouvelle position de la raquette
  table.positionnerRaquetteEnfant_Y(raquetteEnfant_Y);
}

// Méthodes requises par l'interface KeyListener.
public void keyPressed(KeyEvent événement) {
  char touche = événement.getKeyChar();
  if ('n' == touche || 'N' == touche) {
    démarrerNouvellePartie();
  } else if ('q' == touche || 'Q' == touche) {
    terminerJeu();
  } else if ('s' == touche || 'S' == touche) {
    serviceEnfant();
  }
}

public void keyReleased(KeyEvent événement) {}

public void keyTyped(KeyEvent événement) {}

// Démarre une nouvelle partie.
public void démarrerNouvellePartie() {
  scoreOrdinateur = 0;
  scoreEnfant = 0;
  table.affecterTexteMessage("Scores - Ordinateur : 0"
                         + "Enfant : 0");
  serviceEnfant();
}

// Termine le jeu.
public void terminerJeu(){
  System.exit(0);
}
```

Classe `MoteurJeuPingPong` (partie 2 de 5)

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com


```
  // La méthode run() est requise par l'interface Runnable.
 public void run() {

    boolean rebondPossible = false;
    while (true) {

      if (balleServie) { // Si la balle est en mouvement
        // Etape 1. La balle se déplace-t-elle vers la
        // gauche ?
        if (déplacementGauche && balle_X > BALLE_X_MIN) {
          rebondPossible = (balle_Y >= raquetteOrdinateur_Y
            && balle_Y < (raquetteOrdinateur_Y +
            LONGUEUR_RAQUETTE) ? true : false);
          balle_X -= INCREMENT_BALLE;

          // Ajoute un déplacement vertical à chaque
          // mouvement horizontal de la balle.
          balle_Y -= déplacementVertical;

          table.positionnerBalle(balle_X, balle_Y);
          // La balle peut-elle rebondir ?
          if (balle_X <= RAQUETTE_ORDINATEUR_X
              && rebondPossible) {
            déplacementGauche = false;
          }
        }

        // Etape 2. La balle se déplace-t-elle vers la
        // droite ?
        if (!déplacementGauche && balle_X <= BALLE_X_MAX) {
          rebondPossible = (balle_Y >= raquetteEnfant_Y &&
            balle_Y < (raquetteEnfant_Y +
            LONGUEUR_RAQUETTE) ? true : false);

          balle_X += INCREMENT_BALLE;
          table.positionnerBalle(balle_X, balle_Y);
          // La balle peut-elle rebondir ?
          if (balle_X >= RAQUETTE_ENFANT_X &&
                                                rebondPossible) {
            déplacementGauche = true;
        }
      }

      // Etape 3. Déplace la raquette de l'ordinateur vers le
      //          haut ou vers le bas pour bloquer la balle.
```

Classe `MoteurJeuPingPong` (partie 3 de 5)

```
        if (raquetteOrdinateur_Y < balle_Y
                  && raquetteOrdinateur_Y < BAS_TABLE) {
          raquetteOrdinateur_Y += INCREMENT_RAQUETTE;
        } else if (raquetteOrdinateur_Y > HAUT_TABLE) {
          raquetteOrdinateur_Y -= INCREMENT_RAQUETTE;
        }
        table.positionnerRaquetteOrdinateur_Y(
                                    raquetteOrdinateur_Y);

        // Etape 4. Sommeiller un peu
        try {
          Thread.sleep(DUREE_SOMMEIL);
        } catch (InterruptedException exception) {
          exception.printStackTrace();
        }

        // Etape 5. Mettre le score à jour si la balle est
        // dans la surface verte mais ne bouge plus.
        if (balleSurLaTable()) {
          if (balle_X > BALLE_X_MAX ) {
            scoreOrdinateur++;
            afficherScore();
          } else if (balle_X < BALLE_X_MIN) {
            scoreEnfant++;
            afficherScore();
          }
        }
      } // Fin du if balleServie
    } // Fin du while
  }// Fin de run()

  // Sert depuis la position courante de la raquette
  // de l'enfant.
  private void serviceEnfant() {

    balleServie = true;
    balle_X = RAQUETTE_ENFANT_X - 1;
    balle_Y = raquetteEnfant_Y;

    if (balle_Y > HAUTEUR_TABLE / 2) {
      déplacementVertical = -1;
    } else {
      déplacementVertical = 1;
    }

    table.positionnerBalle(balle_X, balle_Y);
    table.positionnerRaquetteEnfant_Y(raquetteEnfant_Y);
  }
```

Classe `MoteurJeuPingPong` (partie 4 de 5)

```
  private void afficherScore() {
    balleServie = false;

    if (scoreOrdinateur == SCORE_GAGNANT) {
      table.affecterTexteMessage("L'ordinateur a gagné ! " +
                                 scoreOrdinateur +
                                 " : " +  scoreEnfant);
    } else if (scoreEnfant == SCORE_GAGNANT) {
      table.affecterTexteMessage ("Tu as gagné ! "+
                                   scoreEnfant +
                                   " : " + scoreOrdinateur);
    } else {
      table.affecterTexteMessage ("Ordinateur : "+
                                  scoreOrdinateur +
                                  " Enfant: " + scoreEnfant);
    }
  }

  // Vérifie que la balle n'a pas dépassé la limite
  // inférieure ou supérieure de la table.
  private boolean balleSurLaTable() {
    if (balle_Y >= BALLE_Y_MIN && balle_Y <= BALLE_Y_MAX) {
      return true;
    } else {
      return false;
    }
  }
}
```

Classe `MoteurJeuPingPong` (partie 5 de 5)

Félicitations ! Tu as terminé ton second jeu. Compile les classes et fais une partie. Lorsque tu te sentiras à l'aise avec le code, essaie de le modifier ; je suis sûr que tu as des idées d'amélioration.

Que lire d'autre sur la programmation de jeux ?

1. CodeRally est un jeu de programmation temps réel Java parrainé par IBM, basé sur la plate-forme Eclipse. Il permet aux utilisateurs peu familiers de Java de rentrer facilement dans la compétition tout en apprenant le langage Java. Les joueurs développent une voiture de course et prennent des décisions sur le bon moment pour accélérer, tourner ou ralentir en fonction de la position des autres joueurs ou des points de contrôle, de leur niveau de carburant et d'autres facteurs.

http://www.alphaworks.ibm.com/tech/codeRally

2. Robocode est un jeu de programmation très amusant qui t'apprend le Java en te faisant créer des robots.

http://www.alphaworks.ibm.com/tech/robocode

Autres lectures

Didacticiel sur les fils d'exécution Java :
http://java.sun.com/docs/books/tutorial/essential/threads/

Introduction aux fils d'exécution Java :
http://www-106.ibm.com/developerworks/edu/j-dw-javathread-i.html

Classe java.awt.Graphics :
http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Graphics.html

Exercices

1. La classe `MoteurJeuPingPong` affecte ses coordonnées au point blanc à l'aide de ce code :
`table.point.x = événement.getX();`.

Dans la classe `TableVertePingPong`, rend privée la variable `point` et ajoute la méthode publique :
`affecteCoordonnéesPoint(int x, int y)`.

Modifie le code de la classe moteur en utilisant cette méthode.

2. Notre jeu de ping-pong a un bogue : après qu'un gagnant ait été annoncé, on peut toujours appuyer sur la touche *S* du clavier et le jeu continue. Corrige ce bogue.

Exercices pour les petits malins

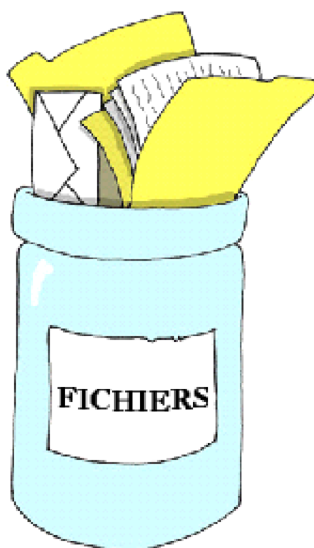
1. Essaie de modifier les valeurs de `INCREMENT_RAQUETTE` et de `INCREMENT_BALLE`. Des valeurs plus élevées augmentent les vitesses de déplacement de la raquette et de la balle. Modifie le code pour permettre à l'utilisateur de sélectionner un niveau de 1 à 10. Utilise la valeur sélectionnée comme incrément pour la balle et la raquette.

2. Quand la raquette de l'enfant frappe la balle dans la partie supérieure de la table, la balle se déplace en diagonale vers le haut et sort rapidement de la table. Modifie le programme pour que la balle se déplace en diagonale vers le bas quand elle est dans la partie supérieure de la table et vers le haut lorsqu'elle est dans la partie inférieure.

Annexe A. Archives Java - JARs

Les utilisateurs d'ordinateurs ont assez souvent besoin d'échanger des fichiers. Ils peuvent les copier sur des disquettes ou sur des CD, utiliser le courrier électronique ou juste envoyer les données à travers le réseau. Il existe des programmes spéciaux capables de *compresser* de multiples fichiers en un unique fichier d'*archive*.

La taille d'une telle archive est en générale inférieure à la somme des tailles des fichiers inclus, ce qui rend la copie plus rapide et économise de la place sur tes disques.



Java est fourni avec un programme appelé `jar`, utilisé pour archiver de multiples classes Java et d'autres fichiers en un fichier d'extension `.jar`.

Le format interne des fichiers jar est le même que celui du populaire WinZip (nous l'avons utilisé au Chapitre 2).

Les trois commandes suivantes illustrent l'utilisation de l'outil `jar`.

Pour créer un jar contenant tous les fichiers dont l'extension est `.class`, ouvre une fenêtre de commande, place-toi dans le répertoire où se trouvent tes classes et tape la commande suivante :

```
jar cvf mesClasses.jar *.class
```

Après le mot `jar`, tu dois préciser les options de cette commande. Dans l'exemple précédent, `c` signifie création d'une nouvelle archive, `v` demande l'affichage de ce que fait la commande et `f` indique que le nom du nouveau fichier d'archive est fourni.

Tu peux maintenant copier ce fichier sur un autre disque ou l'envoyer par courrier électronique à un ami. Pour extraire (*unjar*) les fichiers de l'archive `mesClasses.jar`, tape la commande suivante :

```
jar xvf mesClasses.jar
```

Tous les fichiers seront extraits dans le répertoire courant. Dans cet exemple, l'option `x` signifie extraction des fichiers de l'archive.

Si tu veux juste voir le contenu d'un fichier jar sans extraire les fichiers, utilise cette commande, où `t` signifie sommaire (*table of contents)* :

```
jar tvf mesClasses.jar
```

En réalité, je préfère utiliser le programme WinZip pour voir ce qu'il y a dans un fichier jar.

Très souvent, les applications Java du monde réel sont constituées de multiples clases stockées dans des fichiers jar. Même s'il y a beaucoup d'autres options utilisables avec le programme `jar`, les trois exemples de ce chapitre te suffiront pour la plupart de tes projets.

Autres lectures

Outil d'archivage Java :
http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/jar.html

Annexe B. Astuces Eclipse

Il y a dans Eclipse de nombreuses petites commandes utiles qui rendent la programmation Java un peu plus rapide. Je te propose une liste de commandes Eclipse utiles, mais je suis sûr que tu en trouveras d'autres quand tu commenceras à utiliser cet outil.

- ✓ Si tu vois un petit astérisque dans l'onglet classe, cela signifie que des modifications du code de la classe n'ont pas été enregistrées.
- ✓ Met en surbrillance le nom d'une classe ou d'une méthode utilisée dans ton code et appuie sur la touche *F3* de ton clavier pour aller à la ligne où a été déclarée cette classe ou cette méthode.
- ✓ Si une ligne est marquée avec un rond rouge, symbole d'erreur, déplace la souris au-dessus du rond pour voir le texte de l'erreur.
- ✓ Appuie sur *Ctrl-F11* pour exécuter à nouveau le dernier programme exécuté.
- ✓ Place le curseur après une accolade pour marquer l'accolade correspondante.
- ✓ Pour modifier la superclasse lorsque tu crées une classe, clique sur le bouton *Parcourir*, supprime `java.lang.Object` et entre la première lettre de la classe que tu souhaites utiliser. Tu peux alors choisir ta classe dans une liste.
- ✓ Pour copier une classe d'un paquetage à un autre, sélectionne-la et appuie sur *Ctrl-C*. Sélectionne ensuite sa destination et appuie sur *Ctrl-V*.
- ✓ Pour renommer une classe, une variable ou une méthode, clique dessus avec le bouton droit de la souris et sélectionne *Propager*

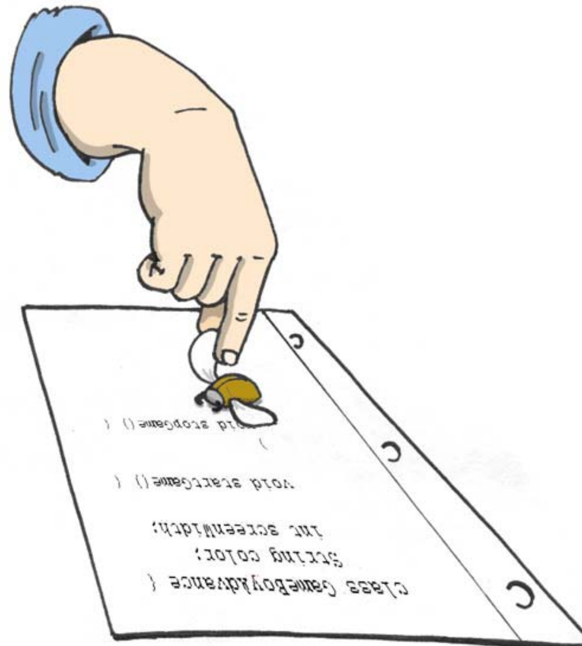
les modifications puis *Renommer* dans le menu contextuel. Toutes les occurrences de ce nom seront renommées.

- ✓ Si ton projet nécessite des fichiers jar externes, clique avec le bouton droit sur le nom du projet, sélectionne *Propriétés*, *Chemin de compilation Java*, choisis l'onglet *Bibliothèques* et clique sur le bouton *Ajouter des fichiers JAR externes*.

Débogueur Eclipse

Selon la rumeur, il y a 40 ans, quand les ordinateurs étaient tellement gros qu'ils n'auraient même pas pu entrer dans ta chambre, l'un des programmes se mit soudain à donner des résultats erronés. Tous ces problèmes étaient le fait d'un petit insecte installé quelque part dans les câbles de l'ordinateur[8]. Quand on retira l'insecte, le programme se remit à fonctionner correctement. Depuis lors, *déboguer un programme* signifie trouver ce qui ne donne pas les résultats attendus.

Ne confonds pas les bogues et les erreurs de compilation. Disons par exemple que, au lieu de multiplier une variable par 2, tu la multiplies par 22. Cette erreur de frappe ne génère aucune erreur de compilation, mais le résultat sera faux. Les débogueurs te permettent de dérouler l'exécution d'un programme ligne à ligne et de voir ou modifier la valeur des variables à chaque étape de l'exécution du programme.



Je vais t'expliquer comment utiliser le débogueur d'Eclipse en prenant l'exemple du programme `MaîtrePoisson` du Chapitre 4.

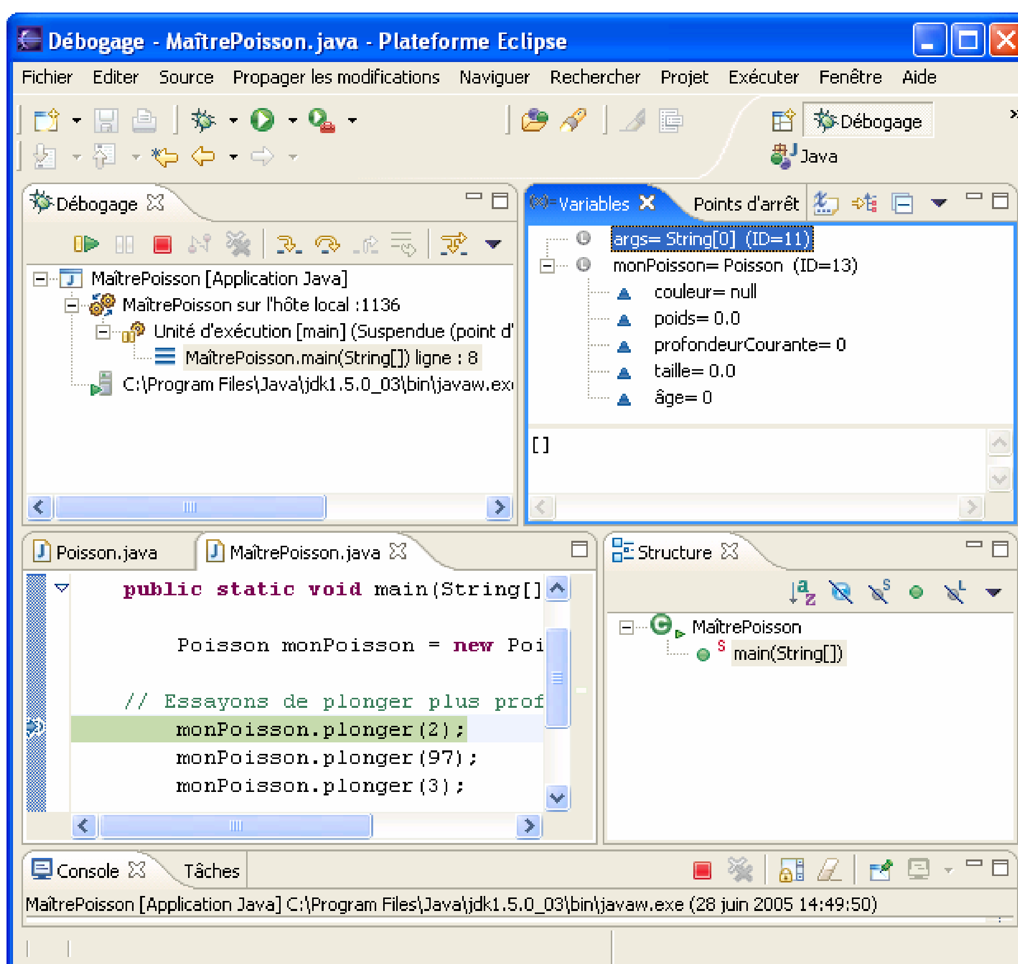
---

[8] NDT : l'un des mots anglais pour insecte est *bug*.

Un *point d'arrêt* (*breakpoint*) est une ligne du code où tu voudrais que ton programme s'arrête afin de voir ou changer la valeur courante des variables et d'autres informations d'exécution. Pour définir un point d'arrêt, double-clique simplement sur la gauche de la ligne où tu veux que le programme s'arrête. Essayons sur la ligne `monPoisson.plonger(2)`. Tu peux voir sur cette ligne une puce ronde qui indique un point d'arrêt. Maintenant, sélectionne les menus *Exécuter*, *Déboguer…*. Sélectionne l'application `MaîtrePoisson` et appuie sur le bouton *Déboguer*.

`MaîtrePoisson` démarre *en mode débogage* (*in the debug mode*) et s'arrête en attendant de nouvelles instructions de ta part dès qu'il atteint la ligne `monPoisson.plonger(2)`.

Une fenêtre comme celle-ci est affichée :

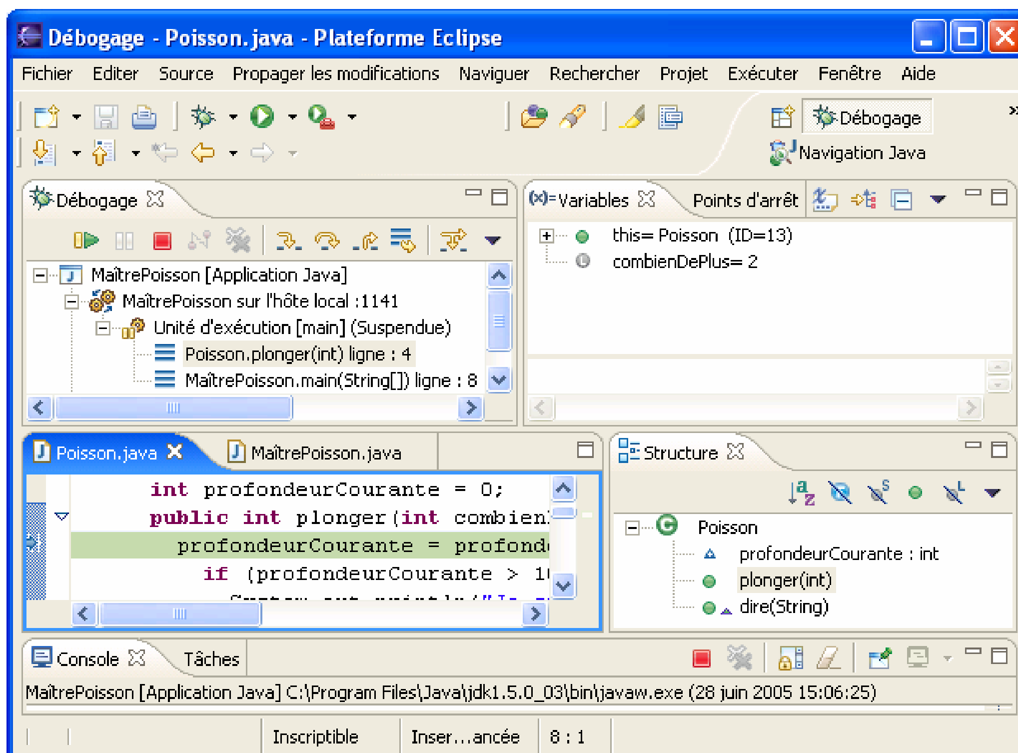


En bas à gauche de la perspective de débogage, la ligne correspondant au point d'arrêt est en surbrillance. La flèche bleue pointe sur la ligne qui est sur le point d'être exécutée. Sur le côté droit (dans la vue *Variables*), clique sur le petit signe + près de la variable `monPoisson`. Comme cette variable pointe sur l'objet `Poisson`, on peut examiner

toutes les variables membres de cette classe et leurs valeurs courantes ; par exemple, `profondeurCourante = 0`.

Les flèches de la partie haut gauche de la fenêtre permettent de poursuivre l'exécution du programme dans différents modes. La première flèche jaune signifie *Avance d'un pas avec entrée*, ou *entre dans* (*step into*) la méthode. Si tu appuies sur cette flèche (ou sur *F5*), tu te retrouveras à l'intérieur de la méthode `plonger()`. La fenêtre change et te présente la valeur de l'argument `combienDePlus = 2` comme dans la capture d'écran ci-dessous. Clique sur le petit signe + à côté du mot `this` pour voir quelles sont les valeurs courantes des variables membres de cet objet.

Pour modifier la valeur de la variable, clique dessus avec le bouton droit et entre la nouvelle valeur. Cela peut t'aider quand tu n'es pas sûr de comprendre pourquoi le programme ne fonctionne pas correctement et que tu veux jouer au jeu du *et si*.



Pour poursuivre l'exécution ligne à ligne, clique sur la flèche suivante, *Avance d'un pas sans entrée* (ou appuie sur le bouton *F6*).

Si tu veux reprendre l'exécution normale du programme, clique sur le bouton *Reprise* (petit triangle vert à gauche) ou appuie sur le bouton *F8*.

Pour supprimer le point d'arrêt, double-clique sur la petite puce ronde, qui disparaît. J'aime utiliser le débogueur même si mon programme ne

contient pas de bogue : ça m'aide à mieux comprendre ce qu'il se passe exactement dans le programme qui s'exécute.

Où placer un point d'arrêt ? Si tu as une idée de la méthode qui te pose problème, mets-le juste avant la ligne suspecte. Si tu n'en es pas sûr, place-le simplement sur la première ligne de la méthode `main()` et avance petit à petit dans le programme.

Annexe C. Comment publier une page web

Les pages Internet sont constituées de fichiers HTML, d'images, de fichiers son, etc. Nous avons brièvement mentionné HTML au Chapitre 7, mais si tu envisages de devenir un designer web, tu dois passer plus de temps à apprendre l'HTML ; l'un des bons endroits par où commencer est la page web www.w3schools.com. En réalité, il y a de nombreux sites web et programmes permettant de créer une page web en quelques minutes sans même savoir comment c'est fait. Ces programmes génèrent de toutes façons de l'HTML, mais ils te le cachent. Cela dit, si tu as maîtrisé ce livre, je te déclare **Programmeur Java Junior** (je ne plaisante pas !) et l'apprentissage d'HTML sera du gâteau pour toi.

Pour développer une page web, tu crées en général un ou plusieurs fichiers HTML sur le disque de ton ordinateur ; mais le problème, c'est que ton ordinateur n'est *pas visible* par les autres utilisateurs d'Internet. C'est pourquoi, une fois la page terminée, tu dois remonter (*upload*) ces fichiers en un lieu visible de tous. C'est le cas d'un disque situé dans l'ordinateur de la compagnie qui est ton *Fournisseur d'accès à Internet* (*FAI*).

Premièrement, tu dois avoir ton propre répertoire sur l'ordinateur de ton FAI. Contacte ton FAI par téléphone ou par courrier électronique en lui expliquant que tu as créé une page HTML et que tu veux la publier. Sa réponse contient en général les informations suivantes :

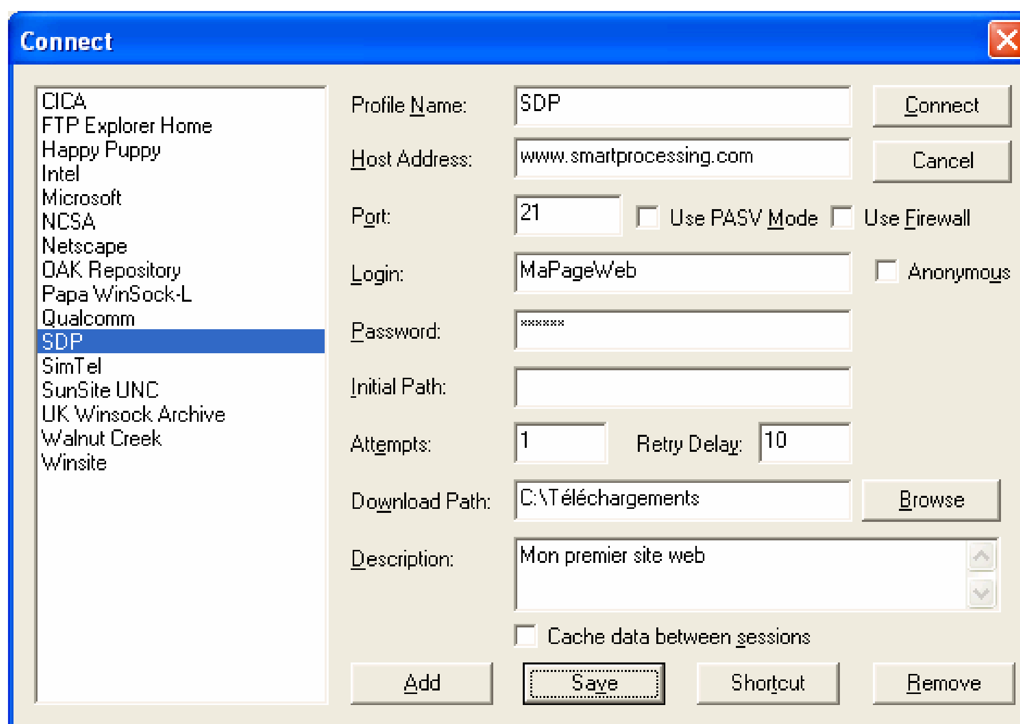
- Le nom de réseau de son ordinateur *hôte* (*host machine*).
- Le nom du répertoire de son ordinateur où il t'est permis de stocker tes fichiers.
- L'adresse web (*URL*) de ta nouvelle page – que tu indiqueras aux personnes qui sont intéressées par ta page.

- L'identifiant utilisateur (*user id*) et le mot de passe (*password*) dont tu auras besoin pour télécharger de nouveaux fichiers ou pour en modifier d'anciens.

Ces temps-ci, la plupart des FAI mettent gratuitement à disposition au moins 10 Mo d'espace sur leur disque, ce qui est plus qu'assez pour la plupart des gens.

Il te faut maintenant un programme qui te permette de copier des fichiers de ton ordinateur sur celui de ton FAI. La copie de fichiers de ton ordinateur vers un ordinateur sur Internet est appelée *téléchargement ascendant* (*uploading*) et la copie de fichiers d'Internet vers ta machine est appelée *téléchargement descendant* (*downloading*). Tu peux télécharger des fichiers dans les deux sens à l'aide de ce qu'on appelle un *programme FTP client*.

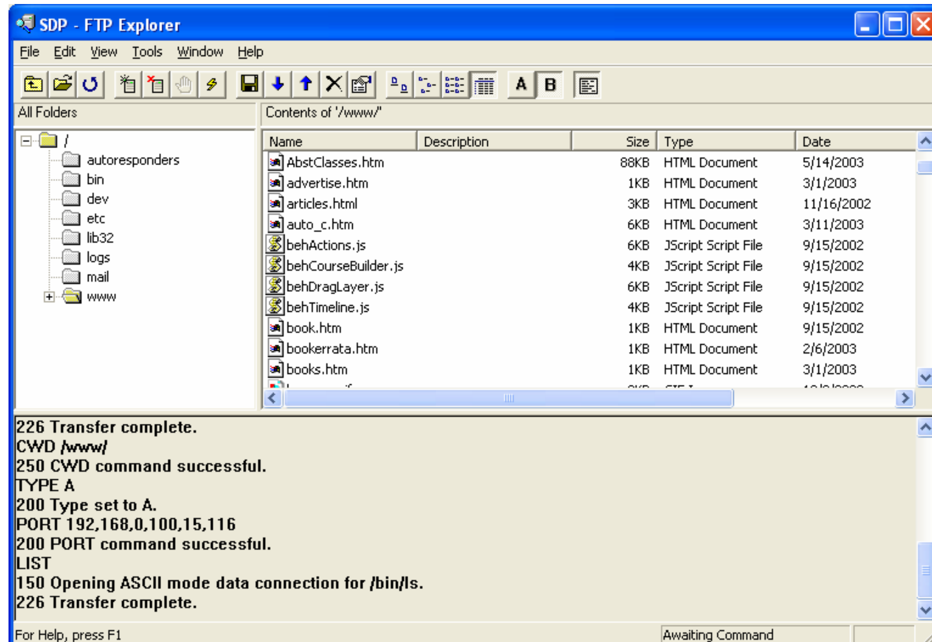
L'un des clients FTP simples et faciles d'utilisation est *FTP Explorer* ; tu peux le télécharger depuis www.ftpx.com. Installe ce programme et ajoute la machine de ton FAI à la liste de connexions de ton client FTP : démarre *FTP Explorer* ; la première fenêtre qui s'ouvre est un écran de connexion. Tu peux aussi cliquer sur l'item *Connection* du menu *Tools*.



Clique sur le bouton *Add* et entre l'hôte, l'identifiant de connexion et le mot de passe que t'a fournis ton FAI. Entre le nom de ton FAI dans le champ *Profile Name*. Si tu as fais les choses correctement, tu dois voir ton nouveau profil de connexion dans la liste des serveurs FTP disponibles. Clique sur le bouton *Connect* et tu verras les répertoires de

la machine de ton FAI. Trouve ton répertoire et suis la procédure de téléchargement décrite ci-dessous.

La barre d'outils contient deux flèches bleues. Celle qui pointe vers le haut sert aux téléchargements ascendants. Clique sur cette flèche pour ouvrir une fenêtre standard qui te permet d'aller dans le répertoire qui contient tes fichiers HTML. Sélectionne les fichiers que tu souhaites télécharger et appuie sur le bouton *Open*. Au bout de quelques secondes, ces fichiers sont visibles sur la machine de ton FAI.



Surveille le bas de la fenêtre pour t'assurer qu'il n'y a pas eu de problèmes pendant le téléchargement.

Nomme le fichier principal de ta page `index.html`. De cette façon, ton URL sera plus courte et les gens n'auront pas besoin d'y ajouter celui de ton fichier. En effet, si le nom du répertoire sur le disque de ton FAI est www.xyz.com/~David et que le fichier principal de ta page web est `maPagePrincipale.html`, l'adresse de ta page web est www.xyz.com/~David/maPagePrincipale.html. Mais si le nom de ton fichier principal est `index.html`, l'URL de ta page est plus court : www.xyz.com/~David. Désormais, tous ceux qui connaissent cette URL pourront voir ta page en ligne. Si, plus tard, tu décides de modifier cette page web, répète la même procédure : effectue les corrections sur ton disque, puis télécharge-les pour remplacer les anciens fichiers par les nouveaux.

Si tu décides de devenir un concepteur web, le prochain langage à apprendre est JavaScript. Ce langage est beaucoup plus simple que Java et te permettra de fabriquer des pages web plus attractives.

Autres lectures

1. Webmonkey pour les enfants :
http://hotwired.lycos.com/webmonkey/kids/

2. Le World Wide Web
http://www.w3schools.com/html/html_www.asp

Exercices

Crée une page web et publie le jeu de morpion du Chapitre 7. Pour commencer, télécharge simplement les fichiers `Morpion.html` et `Morpion.class` sur ta page web.

Fin

Index

! 58
&& 57
== 66
argument 33, 49
arguments de ligne de commande 147
ArrayList 173, 174, 175
AWT 70
boîte de message 94, 95
BorderLayout 77
boucles 65
break 66
BufferedInputStream 145
BufferedOutputStream 147
cadre 72
CardLayout 83
catch 133, 134
CLASSPATH 16
concaténation 39
constantes 40
constructeur 62
continue 67
conversion de type explicite 95
Eclipse 22
else if 58
equals() 57
événements 72, 95
Exception 131
extends 50
FAI 212
File 153
FileInputStream 143
FileOutputStream 144
FileReader 150
FileWriter 150
final 40
finally 136, 137
flux à tampon 145
flux de sortie 142
flux d'entrée 142
for 65
gestionnaire de disposition 72
Graphics 180
GridBagLayout 81
GridLayout 75
GUI 87

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

HTML 109, 111
IDE 22
implanter 91
import 71
instance 97
instanceof 97
interfaces 90, 184
jar 205, 206
Javadoc 54
JRE 20
Math 61
MouseListener 185
MouseMotionListener 185
new 62
niveaux d'accès 166
Object 99
package 166
panneau 72
paquetages 70, 163
Path 16
portée 61
primaire 40
private 166, 167
protected 166
public 32, 166
récepteurs 90
signature de méthode 31
sous-classe 47
static 32, 61
String 41
style orienté objet 35
superclasse 47, 50
surcharge de méthode 50, 51, 159
Swing 71, 178
switch 60
SWT 70
tableau 63
this 63
throw 137
throws 135
try 132, 134
types de données 38
variable d'instance 61
variable membre 61
void 32
while 67
WinZip 23, 206