

28.04.2001 - version 1.2 [Armel]

- Remise en forme du code html (DIR et PRE en blockquote et pre),
remise en forme des balises de numérotation des pages #TIJ_PAGE01#.
- Suppression des balises TIJ.

02.02.2001 - version 1.1 [Raczy]

- Tags + html nettoye

27.06.2000 - version 1.0 [Raczy]

- Dernière modification : 27 juin 2000

2 : Tout est Objet

Bien qu'il soit basé sur C++, Java est un langage orienté objet plus « pur ».

C++ et Java sont tous les deux des langages hybrides, mais dans Java, les concepteurs ont pensé que l'hybridation est moins importante qu'elle ne l'est en C++. Un langage hybride autorise plusieurs styles de programmation : C++ est hybride pour assurer la compatibilité avec le langage C. Comme C++ est une extension du langage C, il contient un grand nombre des particularités indésirables de ce langage, ce qui peut rendre certains aspects du C++ particulièrement embrouillés.

Le langage Java suppose qu'on ne veut faire que de la programmation orientée objet (POO). Ceci signifie qu'avant de pouvoir commencer il faut tourner sa vision des choses vers un monde orienté objets (à moins qu'elle ne le soit déjà). L'avantage de cet effort préliminaire est la capacité à programmer dans un langage qui est plus simple à apprendre et à utiliser que beaucoup d'autres langages de POO. Dans ce chapitre nous verrons les composantes de base d'un programme Java et nous apprendrons que tout dans Java est objet, même un programme Java.

Les objets sont manipulés avec des références

Chaque langage de programmation a ses propres façons de manipuler les données. Parfois le programmeur doit être constamment conscient du type des manipulations en cours. Manipulez-vous l'objet directement, ou avez-vous affaire à une sorte de représentation indirecte (un pointeur en C ou C++) qui doit être traité avec une syntaxe particulière ?

Tout ceci est simplifié en Java. On considère tout comme des objets, ainsi il n'y a qu'une seule syntaxe cohérente qui est utilisée partout. Bien qu'on *traite* tout comme des objets, les identificateurs qui sont manipulés sont en réalité des « références » vers des objets [\[21\]](#). On pourrait imaginer cette situation comme une télévision (l'objet) avec une télécommande (la référence). Tant qu'on conserve cette référence, on a une liaison vers la télévision, mais quand quelqu'un dit « change de chaîne » ou « baisse le volume », ce qu'on manipule est la référence, qui en retour modifie l'objet. Si on veut se déplacer dans la pièce tout en contrôlant la télévision, on emporte la télécommande/référence, pas la télévision.

De plus, la télécommande peut exister par elle même sans télévision. C'est à dire que le fait d'avoir une référence ne signifie pas nécessairement qu'un objet y soit associé. Ainsi, si on veut avoir un mot ou une phrase, on crée une référence sur une **String** :

```
String s;
```

Mais on a *seulement* créé la référence, pas un objet. À ce point, si on décidait d'envoyer un message à `s`, on aurait une erreur (lors de l'exécution) parce que `s` n'est pas rattachée à quoi que ce soit (il n'y a pas de télévision). Une pratique plus sûre est donc de toujours initialiser une référence quand on la crée :

```
String s = "asdf";
```

Toutefois, ceci utilise une caractéristique spéciale de Java : les chaînes de caractères peuvent être initialisées avec du texte entre guillemets. Normalement, on doit utiliser un type d'initialisation plus général pour les objets.

Vous devez créer tous les objets

Quand on crée une référence, on veut la connecter à un nouvel objet. Ceci se fait, en général, avec le mot-clef **new**. **new** veut dire « fabrique moi un de ces objets ». Ainsi, dans l'exemple précédent, on peut dire :

```
String s = new String("asdf");
```

Ceci ne veut pas seulement dire « fabrique moi un nouvel objet **String** », mais cela donne aussi une information sur *comment* fabriquer l'objet **String** en fournissant une chaîne de caractères initiale.

Bien sûr, **String** n'est pas le seul type qui existe. Java propose une pléthore de types tout prêts. Le plus important est qu'on puisse créer ses propres types. En fait, c'est l'activité fondamentale en programmation Java et c'est ce qu'on apprendra à faire dans la suite de ce livre.

Où réside la mémoire ?

Il est utile de visualiser certains aspects de comment les choses sont arrangées lorsque le programme tourne, en particulier comment la mémoire est organisée. Il y a six endroits différents pour stocker les données :

1. **Les registres.** C'est le stockage le plus rapide car il se trouve à un endroit différent des autres zones de stockage : dans le processeur. Toutefois, le nombre de registres est sévèrement limité, donc les registres sont alloués par le compilateur en fonction de ses besoins. On n'a aucun contrôle direct et il n'y a même aucune trace de l'existence des registres dans les programmes.
2. **La pile.** Elle se trouve dans la RAM (random access memory) mais elle est prise en compte directement par le processeur via son *pointeur de pile*. Le pointeur de pile est déplacé vers le bas pour créer plus d'espace mémoire et déplacé vers le haut pour libérer cet espace. C'est un moyen extrêmement efficace et rapide d'allouer de la mémoire, supplanté seulement par les registres. Le compilateur Java doit connaître, lorsqu'il crée le programme, la taille et la durée de vie exacte de toutes les données qui sont rangées sur la pile, parce qu'il doit générer le code pour déplacer le pointeur de pile vers le haut et vers le bas. Cette contrainte met des limites à la flexibilité des programmes, donc, bien qu'il y ait du stockage Java sur la pile -- en particulier les références aux objets -- les objets Java eux même ne sont pas placés sur la pile.
3. **Le segment.** C'est une réserve de mémoire d'usage général (aussi en RAM) où résident tous les objets java. La bonne chose à propos du segment est que, contrairement à la pile, le compilateur n'a pas besoin de savoir de combien de place il a besoin d'allouer sur le segment ni combien de temps cette place doit rester sur le segment. Ainsi, il y a une grande flexibilité à utiliser la mémoire sur le segment. Lorsqu'on a besoin de créer un objet, il suffit d'écrire le code pour le créer en utilisant **new** et la mémoire est allouée sur le segment lorsque le programme s'exécute. Bien entendu il y a un prix à payer pour cette flexibilité :

il faut plus de temps pour allouer de la mémoire sur le segment qu'il n'en faut pour allouer de la mémoire sur la pile (c'est à dire si on *avait* la possibilité de créer des objets sur la pile en Java, comme on peut le faire en C++).

4. **La mémoire statique.** « Statique » est utilisé ici dans le sens « à un endroit fixe » (bien que ce soit aussi dans la RAM). La mémoire statique contient les données qui sont disponibles pendant tout le temps d'exécution du programme. On peut utiliser le mot-clef **static** pour spécifier qu'un élément particulier d'un objet est statique, mais les objets Java par eux-mêmes ne sont jamais placés dans la mémoire statique.
5. **Les constantes.** Les valeurs des constantes sont souvent placées directement dans le code du programme, ce qui est sûr puisqu'elles ne peuvent jamais changer. Parfois les constantes sont isolées de façon à pouvoir être optionnellement placées dans une mémoire accessible en lecture seulement (ROM).
6. **Stockage hors RAM.** Si les données résident entièrement hors du programme, elles peuvent exister même quand le programme ne tourne pas, en dehors du contrôle du programme. Les deux exemples de base sont les *flots de données*, pour lesquels les données sont transformées en flots d'octets, généralement pour être transmises vers une autre machine, et les *objets persistants*, pour lesquels les objets sont placés sur disque de façon à ce qu'ils conservent leur état même après que le programme soit terminé. L'astuce avec ces types de stockage est de transformer les objets en quelque chose qui peut exister sur l'autre support, tout en pouvant être ressuscité en un objet normal en mémoire, lorsque c'est nécessaire. Java fournit des outils pour la *persistance légère*, et les versions futures pourraient fournir des solutions plus complètes pour la persistance.

Cas particulier : les types primitifs

Il y a un ensemble de types qui sont soumis à un traitement particulier ; ils peuvent être considérés comme les types « primitifs » fréquemment utilisés en programmation. La raison de ce traitement particulier est que la création d'un objet avec **new** -- en particulier une simple variable -- n'est pas très efficace parce que **new** place les objets sur le segment. Pour ces types, Java a recours à l'approche retenue en C et en C++. Au lieu de créer la variable en utilisant **new**, une variable « automatique », qui n'est pas une référence, est créée. La variable contient la valeur et elle est placée sur la pile, ce qui est beaucoup plus efficace.

Java fixe la taille de chacun des types primitifs. Ces tailles ne changent pas d'une architecture de machine à une autre, comme c'est le cas dans la plupart des langages. L'invariance de la taille de ces types est l'une des raisons pour lesquelles Java est si portable.

Type primitif	Taille	Minimum	Maximum	Type wrapper
boolean	-	-	-	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	Short
int	32-bit	-2^{31}	$+2^{31}-1$	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	-	-	-	Void

Tous les types numériques sont signés, il est donc inutile d'aller chercher après des types non signés.

Les types de données primitifs sont aussi associés à des classes « wrapper ». Ceci signifie que pour faire un objet non primitif sur le segment pour représenter ce type primitif il faut utiliser le wrapper associé. Par exemple :

```
char c = 'x'; Character C =  
    new Character(c);
```

On peut aussi utiliser :

```
Character C = new  
    Character('x');
```

Les raisons pour lesquelles on fait ceci seront indiquées dans un prochain chapitre.

Nombres de grande précision

Java contient deux classes pour effectuer des opérations arithmétiques de grande précision : **BigInteger** et **BigDecimal**. Bien que ceux-ci soient dans la même catégorie que les classes « wrapper », aucun d'eux n'a d'analogue primitif.

Chacune de ces classes a des méthodes qui fournissent des opérations analogues à celles qu'on peut faire sur les types primitifs. C'est à dire qu'avec un **BigInteger** ou un **BigDecimal** on peut faire tout ce qu'on peut faire avec un **int** ou un **float**, seulement il faut utiliser des appels de méthodes au lieu des opérateurs. Par ailleurs, comme elles en font plus, les opérations sont plus lentes. On échange la vitesse contre la précision.

BigInteger sert aux entiers de précision arbitraire. C'est à dire qu'ils permettent de représenter des valeurs entières de n'importe quelle taille sans perdre aucune information au cours des opérations.

BigDecimal sert aux nombres à virgule fixe de précision arbitraire ; par exemple, on peut les utiliser pour des calculs monétaires précis.

Il faut se reporter à la documentation en ligne pour obtenir des détails sur les constructeurs et méthodes utilisables avec ces deux classes.

Tableaux en Java

Pratiquement tous les langages de programmation gèrent les tableaux. Utiliser des tableaux en C ou C++ est dangereux car ces tableaux ne sont que des blocs de mémoire. Si un programme accède à un tableau en dehors de son bloc mémoire, ou s'il utilise la mémoire avant initialisation (erreurs de programmation fréquentes) les résultats seront imprévisibles.

Un des principaux objectifs de Java est la sécurité, aussi, un grand nombre des problèmes dont souffrent C et C++ ne sont pas répétés en Java. On est assuré qu'un tableau Java est initialisé et qu'il ne peut pas être accédé en dehors de ses bornes. La vérification des bornes se fait au prix d'un petit excédent de mémoire pour chaque tableau ainsi que de la vérification de l'index lors de l'exécution, mais on suppose que le gain en sécurité et en productivité vaut la dépense.

Quand on crée un tableau d'objets, on crée en réalité un tableau de références, et chacune de ces références est automatiquement initialisée à une valeur particulière avec son propre mot clé : **null**. Quand Java voit **null**, il reconnaît que la référence en question ne pointe pas vers un objet. Il faut affecter un objet à chaque référence avant de l'utiliser et si on essaye d'utiliser une référence encore à **null**, le problème sera signalé lors de l'exécution. Ainsi, les erreurs typiques sur les tableaux sont évitées en Java.

On peut aussi créer des tableaux de variables de type primitif. À nouveau, le compilateur garantit l'initialisation car il met à zéro la mémoire utilisée par ces tableaux.

Les tableaux seront traités plus en détails dans d'autres chapitres.

Vous n'avez jamais besoin de détruire un objet

Dans la plupart des langages de programmation, le concept de durée de vie d'une variable monopolise une part significative des efforts de programmation. Combien de temps une variable existe-t-elle ? S'il faut la détruire, quand faut-il le faire ? Des erreurs sur la durée de vie des variables peuvent être la source de nombreux bugs et cette partie montre comment Java simplifie énormément ce problème en faisant le ménage tout seul.

Notion de portée

La plupart des langages procéduraux ont le concept de *portée*. Il fixe simultanément la visibilité et la durée de vie des noms définis dans cette portée. En C, C++ et Java, la portée est fixée par l'emplacement des accolades {}. Ainsi, par exemple :

```
{  
    int x = 12;  
  
    /* seul x est accessible */  
  
    {  
        int q = 96;  
  
        /* x & q sont tous les deux accessibles */  
    }  
  
    /* seul x est accessible */  
  
    /* q est « hors de portée » */  
}
```

Une variable définie dans une portée n'est accessible que jusqu'à la fin de cette portée.

L'indentation rend le code Java plus facile à lire. Étant donné que Java est un langage indépendant de la mise en page, les espaces, tabulations et retours chariots supplémentaires ne changent pas le programme.

Il faut remarquer qu'on ne peut pas faire la chose suivante, bien que cela soit autorisé en C et C++ :

```
{  
    int x = 12;
```

```
{  
    int x = 96; /* illegal */  
}  
}
```

Le compilateur annoncera que la variable **x** a déjà été définie. Ainsi, la faculté du C et du C++ à « cacher » une variable d'une portée plus étendue n'est pas autorisée parce que les concepteurs de Java ont pensé que ceci mène à des programmes confus.

Portée des objets

Les objets Java n'ont pas la même durée de vie que les variables primitives. Quand on crée un objet Java avec **new**, il existe toujours après la fin de la portée. Ainsi, si on fait :

```
{  
    String s = new String("a string");  
} /* fin de portée */
```

la référence **s** disparaît à la fin de la portée. Par contre l'objet **String** sur lequel **s** pointait occupe toujours la mémoire. Dans ce bout de code, il n'y a aucun moyen d'accéder à l'objet parce que son unique référence est hors de portée. Dans d'autres chapitres on verra comment la référence à un objet peut être transmise et dupliquée dans un programme.

Il s'avère que du simple fait qu'un objet créé avec **new** reste disponible tant qu'on le veut, tout un tas de problèmes de programmation du C++ disparaissent tout simplement en Java. Il semble que les problèmes les plus durs surviennent en C++ parce que le langage ne fournit aucune aide pour s'assurer que les objets sont disponibles quand on en a besoin. Et, encore plus important, en C++ on doit s'assurer qu'on détruit bien les objets quand on en a terminé avec eux.

Ceci amène une question intéressante. Si Java laisse les objets traîner, qu'est-ce qui les empêche de complètement remplir la mémoire et d'arrêter le programme ? C'est exactement le problème qui surviendrait dans un programme C++. C'est là qu'un peu de magie apparaît. Java a un ramasse-miettes qui surveille tous les objets qui ont été créés avec **new** et qui arrive à deviner lesquels ne sont plus référencés. Ensuite il libère la mémoire de ces objets de façon à ce que cette mémoire puisse être utilisée pour de nouveaux objets. Ceci signifie qu'il ne faut jamais s'embêter à récupérer la mémoire soi-même. On crée simplement les objets, et quand on n'en a plus besoin, ils disparaissent d'eux même. Ceci élimine toute une classe de problèmes de programmation : les soi-disant « fuites de mémoire » qui arrivent quand un programmeur oublie de libérer la mémoire.

Créer de nouveaux types de données : class

Si tout est objet, qu'est-ce qui définit à quoi ressemble une classe particulière d'objets et comment elle se comporte ? Autrement dit, qu'est-ce qui constitue le *type* d'un objet ? On pourrait s'attendre à avoir un mot-clef appelé « type », et cela serait parfaitement sensé. Historiquement, toutefois, la plupart des langages orientés objet ont utilisé le mot-clef **class** qui signifie « je vais décrire à quoi ressemble un nouveau type d'objet ». Le mot-clef **class** (qui est si commun qu'il ne sera plus mis en gras dans la suite de ce livre) est suivi par le nom du

nouveau type. Par exemple :

```
class ATypeName { /* le corps de la classe vient ici */ }
```

Ceci introduit un nouveau type, on peut alors créer un objet de ce type en utilisant **new** :

```
ATypeName a = new ATypeName();
```

Dans **ATypeName**, le corps de la classe ne consiste qu'en un commentaire (les étoiles et barres obliques et ce qu'il y a à l'intérieur, ce qui sera décrit ultérieurement dans ce chapitre), donc il n'y a grand chose à faire avec. En fait, on ne peut pas lui dire de faire quoi que ce soit (c'est à dire qu'on ne peut pas lui transmettre de message intéressant) tant qu'on n'y définit pas de méthodes.

#/TIJ_PAGE01# #/TIJ_PAGE02#

Champs et méthodes

Lorsqu'on définit une classe (et tout ce que l'on fait en Java consiste à définir des classes, fabriquer des objets à partir de ces classes et envoyer des messages à ces objets) on peut mettre deux types d'éléments dans ces classes : des données membres de la classe (aussi appelées champs) et des fonctions membres de la classe (habituellement appelées méthodes). Une donnée membre est un objet de n'importe quel type avec lequel on peut communiquer via sa référence. Il peut aussi s'agir d'un des types primitifs (dans ce cas, ce n'est pas une référence). S'il s'agit d'une référence à un objet, il faut initialiser cette référence pour la connecter à un objet réel (en utilisant **new** comme indiqué précédemment) grâce à une fonction particulière appelée un *constructeur* (entièrement décrit dans le chapitre 4). S'il s'agit d'un type primitif il est possible de l'initialiser directement lors de sa définition dans la classe (comme on le verra plus tard, les références peuvent aussi être initialisées lors de la définition).

Chaque objet met ses données membres dans sa zone de mémoire propre, les données membres ne sont pas partagées entre les objets. Voici un exemple de classe avec des données membres :

```
class DataOnly {  
    int i;  
    float f;  
  
    boolean b;  
}
```

Cette classe ne fait rien mais on peut créer un objet :

```
DataOnly d = new DataOnly();
```

On peut affecter des valeurs aux données membres mais il faut d'abord savoir comment faire référence à un membre d'un objet. Ceci s'effectue en indiquant le nom de la référence à l'objet, suivi par un point, suivi par le nom du membre dans l'objet :

```
objectReference.member
```

Par exemple :

```
d.i = 47;  
d.f = 1.1f;  
d.b = false;
```

Il est aussi possible que l'objet puisse contenir d'autres objets qui contiennent des données que l'on souhaite modifier. Pour cela il suffit de continuer à « associer les points ». Par exemple :

```
myPlane.leftTank.capacity = 100;
```

La classe **DataOnly** ne peut pas faire grand chose à part contenir des données car elle n'a pas de fonctions membres (méthodes). Pour comprendre comment celles-ci fonctionnent il faut d'abord comprendre les notions de *paramètres* et de *valeurs de retour*, qui seront brièvement décrites.

Valeurs par défaut des membres primitifs

Quand une donnée d'un type primitif est membre d'une classe on est assuré qu'elle a une valeur par défaut si on ne l'initialise pas :

Type primitif	Valeur par défaut
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Par prudence, il faut remarquer que les valeurs par défaut sont celles que Java garantit quand la variable est utilisée *comme un membre d'une classe*. Ceci assure que les variables membres de type primitif sont toujours initialisées (parfois C++ ne le fait pas), ce qui supprime une source de bugs. Toutefois, cette valeur initiale peut ne pas être correcte ou même légale pour le programme qui est écrit. Il est préférable de toujours initialiser explicitement les variables.

Cette garantie ne s'applique pas aux variables « locales » -- celles qui ne sont pas des champs d'une classe. Ainsi, si dans la définition d'une fonction, on a :

```
int x;
```


Alors **x** aura une valeur arbitraire (comme en C et C++), il ne sera pas initialisé automatiquement à zéro. On a la responsabilité d'affecter une valeur appropriée avant d'utiliser **x**. Si on oublie de le faire, Java est sans aucun doute mieux conçu que C++ sur ce point : on obtient une erreur de compilation qui dit que la variable pourrait ne pas être initialisée (avec beaucoup de compilateurs C++ on a des avertissements concernant les variables non initialisées, mais avec Java ce sont des erreurs).

Méthodes, paramètres et valeurs de retour

Jusqu'à présent le terme *fonction* a été employé pour désigner une sous-routine nommée. Le terme qui est plus généralement employé en Java est *méthode*, en tant que « moyen de faire quelque chose ». Il est possible, si on le souhaite, de continuer à raisonner en terme de fonctions. Il s'agit simplement d'une différence de syntaxe, mais à partir de maintenant on utilisera « méthode » plutôt que fonction, dans ce livre.

Les méthodes en Java définissent les messages qu'un objet peut recevoir. Dans cette partie on verra à quel point il est simple de définir une méthode.

Les éléments fondamentaux d'une méthode sont le nom, les paramètres, le type de retour et le corps. Voici la forme de base :

```
returnType methodName( /* liste de paramètres */ ) {  
    /* corps de la méthode */  
}
```

Le type de retour est le type de la valeur qui est retournée par la méthode après son appel. La liste de paramètres donne le type et le nom des informations qu'on souhaite passer à la méthode. L'association du nom de la méthode et de la liste de paramètres identifie de façon unique la méthode.

En Java, les méthodes ne peuvent être créées que comme une composante d'une classe. Une méthode ne peut être appelée que pour un objet [\[22\]](#) et cet objet doit être capable de réaliser cet appel de méthode. Si on essaye d'appeler une mauvaise méthode pour un objet, on obtient un message d'erreur lors de la compilation. On appelle une méthode pour un objet en nommant l'objet suivi d'un point, suivi du nom de la méthode et de sa liste d'arguments, comme ça : **objectName.methodName(arg1, arg2, arg3)**. Par exemple, si on suppose qu'on a une méthode **f()** qui ne prend aucun paramètre et qui retourne une valeur de type **int**. Alors, si on a un objet appelé **a** pour lequel **f()** peut être appelé, on peut écrire :

```
int x = a.f();
```

Le type de la valeur de retour doit être compatible avec le type de **x**.

On appelle généralement *envoyer un message à un objet* cet acte d'appeler une méthode. Dans l'exemple précédent le message est **f()** et l'objet est **a**. La programmation orientée objet est souvent simplement ramenée à « envoyer des messages à des objets ».

La liste de paramètres

La liste de paramètres de la méthode spécifie quelles informations on passe à la méthode. Comme on peut le supposer, ces informations -- comme tout le reste en Java -- sont sous la forme d'objets. On doit donc indiquer dans la liste de paramètres les types des objets à transmettre et les noms à employer pour chacun. Comme dans

toutes les situation où on a l'impression de manipuler des objets, en Java on passe effectivement des références [23]. Toutefois, le type de la référence doit être correct. Si le paramètre est censé être un objet de type **String**, ce qu'on transmet doit être de ce type.

Considérons une méthode qui prend un objet de classe **String** en paramètre. Voici la définition qui doit être mise à l'intérieur de la définition d'une classe pour qu'elle soit compilée :

```
int storage(String s) {  
    return s.length() * 2;  
}
```

Cette méthode indique combien d'octets sont nécessaires pour contenir une **String** donnée (chaque **char** dans une **String** fait 16 bits, ou deux octets, pour permettre les caractères Unicode). Le paramètre est de type **String** et il est appelé **s**. Une fois que **s** est passé à une méthode, il peut être traité comme n'importe quel autre objet (on peut lui envoyer des messages). Ici, on appelle la méthode **length()** qui est une des méthodes de la classe **String** ; elle retourne le nombre de caractères que contient la chaîne.

On peut aussi voir l'utilisation du mot-clef **return** qui fait deux choses. D'abord, il signifie « quitte la méthode, j'ai terminé ». Ensuite, si la méthode retourne une valeur, cette valeur est placée juste après la déclaration du **return**. Dans le cas présent, la valeur de retour est produite en évaluant l'expression **s.length() * 2**.

On peut retourner des valeurs de tous les types qu'on souhaite, mais si on souhaite ne rien retourner du tout on peut le faire en indiquant que la méthode retourne **void**. Voici quelques exemples :

```
boolean flag() { return true; }  
  
float naturalLogBase() { return 2.718f; }  
  
void nothing() { return; }  
  
void nothing2() {}
```

Quand le type de retour est **void**, alors le mot-clef **return** n'est utilisé que pour sortir de la méthode, il n'est donc pas nécessaire quand on atteint la fin de la méthode. On peut retourner d'une méthode à n'importe quel endroit mais si on a indiqué un type de retour qui n'est pas **void** alors le compilateur imposera (avec des messages d'erreur) un retour avec une valeur d'un type approprié sans tenir compte de l'endroit auquel le retour se produit.

À ce point, on peut penser qu'un programme n'est qu'un paquet d'objets avec des méthodes qui prennent d'autres objets en paramètres pour transmettre des messages à ces autres objets. C'est effectivement l'essentiel de ce qui se passe mais dans les chapitres suivants on verra comment faire le travail de bas niveau en prenant des décisions au sein d'une méthode. Pour ce chapitre, envoyer des messages est suffisant.

Construction d'un programme Java

Il y a plusieurs autres éléments à comprendre avant de voir le premier programme Java.

Visibilité des noms

Un problème commun à tous les langages de programmation est le contrôle des noms. Si on utilise un nom dans

un module du programme et si un autre programmeur utilise le même nom dans un autre module, comment distingue-t-on un nom d'un autre et comment empêche-t-on les « collisions » de noms ? En C c'est un problème particulier car un programme est souvent un océan de noms incontrôlable. Les classes C++ (sur lesquelles les classes Java sont basées) imbriquent les fonctions dans les classes de telle sorte qu'elles ne peuvent pas entrer en collision avec les noms de fonctions imbriqués dans d'autres classes. Toutefois, C++ autorise toujours les données et les fonctions globales, donc les collisions sont toujours possibles. Pour résoudre ce problème, C++ a introduit les *domaines de noms* (namespace) en utilisant des mots-clefs supplémentaires.

Java a pu éviter tout cela en employant une approche originale. Pour générer sans ambiguïté un nom pour une bibliothèque, le spécificateur utilisé n'est pas très différent d'un nom de domaine Internet. En fait, les créateurs de Java veulent qu'on utilise son propre nom de domaine Internet inversé, puisqu'on est assuré que ceux-ci sont uniques. Puisque mon nom de domaine est **BruceEckel.com**, ma bibliothèque d'utilitaires (utility) pour mes marottes (foibles) devrait être appelée **com.bruceeckel.utility.foibles**. Après avoir inversé le nom de domaine, les points sont destinés à représenter des sous-répertoires.

Dans Java 1.0 et Java 1.1 les extensions de domaines com, edu, org, net, etc. étaient mises en lettres capitales par convention, ainsi la bibliothèque serait : **COM.bruceeckel.utility.foibles**. Toutefois, au cours du développement de Java 2, on s'est rendu compte que cela causait des problèmes et par conséquent les noms de packages sont entièrement en lettres minuscules.

Ce mécanisme signifie que tous les fichiers existent automatiquement dans leur propre domaine de nom et toutes les classe contenues dans un fichier donné doivent avoir un identificateur unique. Ainsi, on n'a pas besoin d'apprendre de particularités spécifiques au langage pour résoudre ce problème -- le langage s'en occupe à votre place.

Utilisation d'autres composantes

Lorsqu'on souhaite utiliser une classe prédéfinie dans un programme, le compilateur doit savoir comment la localiser. Bien entendu, la classe pourrait déjà exister dans le même fichier source que celui d'où elle est appelée. Dans ce cas on utilise simplement la classe -- même si la classe n'est définie que plus tard dans le fichier. Java élimine le problème des « référence anticipées », il n'y a donc pas à s'en préoccuper.

Qu'en est-il des classes qui existent dans un autre fichier ? On pourrait penser que le compilateur devrait être suffisamment intelligent pour aller simplement la chercher lui même, mais il y a un problème. Imaginons que l'on veuille utiliser une classe ayant un nom spécifique mais qu'il existe plus d'une classe ayant cette définition (il s'agit probablement de définitions différentes). Ou pire, imaginons que l'on écrive un programme et qu'en le créant on ajoute à sa bibliothèque une nouvelle classe qui entre en conflit avec le nom d'une classe déjà existante.

Pour résoudre ce problème il faut éliminer les ambiguïtés potentielles. Ceci est réalisé en disant exactement au compilateur Java quelles classes on souhaite, en utilisant le mot-clef **import**. **import** dit au compilateur d'introduire un *package* qui est une bibliothèque de classes (dans d'autres langages, une bibliothèque pourrait comporter des fonctions et des données au même titre que des classes mais il faut se rappeler que tout le code Java doit être écrit dans des classes).

La plupart du temps on utilise des composantes des bibliothèques Java standard qui sont fournies avec le compilateur. Avec celles-ci il n'y a pas à se tracasser à propos des longs noms de domaines inversés ; il suffit de dire, par exemple :

```
import java.util.ArrayList;
```

pour dire au compilateur que l'on veut utiliser la classe Java **ArrayList**. Toutefois, **util** contient de nombreuses classes et on pourrait vouloir utiliser plusieurs d'entre elles sans les déclarer explicitement. Ceci est facilement réalisé en utilisant '*' pour indiquer un joker :

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

Il est plus courant d'importer une collection de classes de cette manière que d'importer les classes individuellement.

Le mot-clef static

Normalement, quand on crée une classe, on décrit ce à quoi ressemblent les objets de cette classe et comment ils se comportent. Rien n'existe réellement avant de créer un objet de cette classe avec **new** ; à ce moment la zone de données est créée et les méthodes deviennent disponibles.

Mais il y a deux situation pour lesquelles cette approche n'est pas suffisante. L'une, si on veut avoir une zone de stockage pour des données spécifiques, sans tenir compte du nombre d'objets créés, ou même si aucun objet n'a été créé. L'autre, si on a besoin d'une méthode qui n'est associée à aucun objet particulier de la classe. C'est à dire si on a besoin d'une méthode qui puisse être appelée même si aucun objet n'a été créé. On peut obtenir ces deux effets avec le mot-clef **static**. Dire que quelque chose est **static** signifie que la donnée ou la méthode n'est pas spécifiquement rattachée à un objet instance de cette classe. Donc, même si aucun objet de cette classe n'a jamais été créé il est possible d'appeler une méthode **static** ou d'accéder à une donnée **static**. Avec des données et des méthodes non **static** ordinaires il faut connaître l'objet spécifique avec lequel elles fonctionnent. Bien entendu, étant donné que les méthodes **static** n'ont pas besoin qu'un objet soit créé avant d'être utilisées, elles ne peuvent pas accéder *directement* à des membres ou des méthodes non **static** en appelant ces autres membres sans faire référence à un objet nommé (puisque les membres et méthodes non **static** doivent être rattachés à un objet spécifique).

Certains langages orientés objet emploient les expressions *données de classe* et *méthodes de classe*, ce qui signifie que les données et les méthodes n'existent que pour la classe en tant que tout et pas pour des objets particuliers de la classe. Parfois la littérature Java utilise aussi ces expressions.

Pour rendre statique une méthode ou une donnée membre il suffit de mettre le mot-clef **static** avant la définition. Par exemple, le code suivant crée une donnée membre **static** et l'initialise :

```
class StaticTest {  
    static int i = 47;  
}
```

Maintenant, même en créant deux objet **StaticTest**, il n'y aura qu'une seule zone de stockage pour **StaticTest.i**. Tous les objets partageront le même **i**. Considérons :

```
StaticTest st1 = new StaticTest();  
StaticTest st2 = new StaticTest();
```

à ce point, **st1.i** et **st2.i** ont la même valeur 47 puisqu'elles font référence à la même zone mémoire.

Il y a deux façons de faire référence à une variable **static**. Comme indiqué ci-dessus, il est possible de la nommer via un objet, en disant par exemple `st2.i`. Il est aussi possible d'y faire référence directement par le nom de la classe, ce qui ne peut pas être fait avec un membre non **static** (c'est le moyen de prédilection pour faire référence à une variable **static** puisque cela met en évidence la nature **static** de la variable).

```
StaticTest.i++;
```

L'opérateur `++` incrémente la variable. À ce point, `st1.i` et `st2.i` auront tous deux la valeur 48.

Une logique similaire s'applique aux méthodes statiques. On peut faire référence à une méthode statique soit par l'intermédiaire d'un objet, comme on peut le faire avec n'importe quelle méthode, ou avec la syntaxe spécifique supplémentaire **ClassName.method()**. Une méthode statique est définie de façon similaire :

```
class StaticFun {  
    static void incr() { StaticTest.i++; }  
}
```

On peut voir que la méthode **incr()** de **StaticFun** incrémente la donnée **static i**. On peut appeler **incr()** de façon classique, par le biais d'un objet :

```
StaticFun sf = new StaticFun();  
sf.incr();
```

Ou, parce que **incr()** est une méthode statique, il est possible de l'appeler directement par sa classe :

```
StaticFun.incr();
```

Alors que **static**, lorsqu'il est appliqué à une donnée membre, change sans aucun doute la façon dont la donnée est créée (une pour chaque classe par opposition à une pour chaque objet dans le cas des données non statiques), lorsqu'il est appliqué à une méthode, le changement est moins significatif. Un cas important d'utilisation des méthodes **static** est de permettre d'appeler cette méthode sans créer d'objet. C'est essentiel, comme nous le verrons, dans la définition de la méthode **main()** qui est le point d'entrée pour exécuter une application.

Comme pour toute méthode, une méthode statique peut créer ou utiliser des objets nommés de son type, ainsi les méthodes statiques sont souvent utilisées comme « berger » pour un troupeau d'instances de son propre type.

Votre premier programme Java

Voici enfin notre premier programme . Il commence par écrire une chaîne de caractères, puis il écrit la date en utilisant la classe **Date** de la bibliothèque standard de Java. Il faut remarquer qu'un style de commentaire supplémentaire est introduit ici : le `/**` qui est un commentaire jusqu'à la fin de la ligne.

```
/** HelloDate.java  
import java.util.*;
```

```
public class HelloDate {  
    public static void main(String[] args) {  
        System.out.println("Hello, it's: ");  
        System.out.println(new Date());  
    }  
}
```

#/TIJ_PAGE02# #/TIJ_PAGE03#

Au début de chaque fichier de programme il faut mettre la déclaration **import** pour importer toutes les classes supplémentaires qui seront nécessaires pour le code dans ce fichier. Il faut noter que je dis « supplémentaires » ; c'est parce qu'il y a une bibliothèque de classes qui est automatiquement importée dans tous les fichiers Java : **java.lang**. En démarrant un browser Web et en chargeant la documentation de Sun (si elle n'a pas été téléchargée de *java.sun.com* ou si la documentation Java n'a pas encore été installée, il faut le faire maintenant), si on regarde à la liste des package, on voit toutes les bibliothèques de classes qui sont fournies avec Java. Sélectionnons **java.lang**. Ceci fait apparaître la liste de toutes les classes contenues dans cette bibliothèque. Étant donné que **java.lang** est implicitement inclus dans tous les fichiers source Java, ces classes sont automatiquement disponibles. Il n'y a pas de classe **Date** dans **java.lang**, ce qui veut dire qu'il faut importer une autre bibliothèque pour l'utiliser. Si on ne sait pas dans quelle bibliothèque se trouve une classe donnée, ou si on veut voir toutes les classes, on peut sélectionner « Tree » dans la documentation Java. Maintenant on peut trouver n'importe laquelle des classes fournies avec Java. On peut alors utiliser la fonction « chercher » du browser pour trouver **Date**. Une fois que c'est fait on voit qu'elle est connue en tant que **java.util.Date**, ce qui permet de savoir qu'elle se trouve dans la bibliothèque **util** et qu'il faut déclarer **import java.util.*** de façon à pouvoir utiliser **Date**.

Si on retourne au début pour sélectionner **java.lang** puis **System**, on voit que la classe **System** a plusieurs champs et si on sélectionne **out** on découvre que c'est un objet **static PrintStream**. Puisqu'il est statique, on n'a pas besoin de créer quoique ce soit. L'objet **out** est toujours là et il suffit de l'utiliser. Ce qu'on peut faire avec **out** est défini par son type : **PrintStream**. D'une façon très pratique, **PrintStream** est affiché dans la description comme un hyper lien, ainsi, en cliquant dessus on voit la liste de toutes les méthodes qu'on peut appeler pour **PrintStream**. Il y en a un certain nombre et elles seront décrites ultérieurement dans ce livre. Pour l'instant nous ne nous intéressons qu'à **println()**, qui veut dire « écrit ce que je te donne sur la console et passe à la ligne ». Ainsi, dans tout programme Java on peut dire **System.out.println("quelque chose")** chaque fois qu'on souhaite écrire quelque chose sur la console.

Le nom de la classe est le même que celui du fichier. Quand on crée un programme autonome comme celui-là une des classes du fichier doit avoir le même nom que le fichier (le compilateur se plaint si on ne le fait pas). Cette classe doit contenir une méthode appelée **main()** avec la signature suivante :

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

Le mot-clef **public** signifie que la méthode est accessible au monde extérieur (décrit en détail dans le chapitre 5). Le paramètre de **main()** est un tableau d'objets de type **String**. Le paramètre **args** n'est pas utilisé dans ce programme mais le compilateur Java insiste pour qu'il soit là car il contient les paramètres invoqués sur la ligne de commande.

La ligne qui écrit la date est assez intéressante :

```
System.out.println(new Date());
```

Considérons le paramètre : un objet **Date** est créé juste pour transmettre sa valeur à **println()**. Dès que cette instruction est terminée, cette date est inutile et le ramasse-miettes peut venir le récupérer n'importe quand. On n'a pas à s'occuper de s'en débarrasser.

Compilation et exécution

Pour compiler et exécuter ce programme, et tous les autres programmes de ce livre, il faut d'abord avoir un environnement de programmation Java. Il y a un grand nombre d'environnements de développement mais dans ce livre nous supposons que vous utilisez le JDK de Sun qui est gratuit. Si vous utilisez un autre système de développement, vous devrez vous reporter à la documentation de ce système pour savoir comment compiler et exécuter les programmes.

Connectez vous à Internet et allez sur <http://java.sun.com>. Là, vous trouverez des informations et des liens qui vous guideront pour télécharger et installer le JDK pour votre plate-forme.

Une fois que le JDK est installé et que vous avez configuré les informations relatives au chemin sur votre ordinateur afin qu'il puisse trouver **javac** et **java**, téléchargez et décompressez le code source de ce livre (on peut le trouver sur le CD-ROM qui est livré avec le livre ou sur www.BruceEckel.com). Ceci créera un sous répertoire pour chacun des chapitres de ce livre. Allez dans le sous-répertoire **c02** et tapez :

```
javac HelloDate.java
```

Cette commande ne devrait produire aucune réponse. Si vous obtenez un message d'erreur de quelque sorte que ce soit cela signifie que vous n'avez pas installé le JDK correctement et que vous devez corriger le problème.

Par contre, si vous obtenez simplement votre invite de commande vous pouvez taper :

```
java HelloDate
```

et vous obtiendrez en sortie le message ainsi que la date.

C'est le procédé que vous pouvez employer pour compiler et exécuter chacun des programmes de ce livre. Toutefois, vous verrez que le code source de ce livre a aussi un fichier appelé **makefile** dans chaque chapitre, et celui-ci contient les commandes « make » pour construire automatiquement les fichiers de ce chapitre. Reportez-vous à la page Web de ce livre sur www.BruceEckel.com pour plus de détails sur la manière d'utiliser ces *makefiles*.

Commentaires et documentation intégrée

Il y a deux types de commentaires en Java. Le premier est le commentaire traditionnel, style C, dont a hérité C++. Ces commentaires commencent par **/*** et continuent, éventuellement sur plusieurs lignes, jusqu'à un ***/**. Il faut remarquer que de nombreux programmeurs commencent chaque ligne de continuation de commentaire avec *****, on voit donc souvent :


```
/* Ceci est un commentaire  
 * qui continue  
 * sur plusieurs lignes  
 */
```

Il faut toutefois se rappeler que tout ce qui se trouve entre `/*` et `*/` est ignoré, il n'y a donc aucune différence avec :

```
/* Ceci est un commentaire qui  
   continue sur plusieurs lignes */
```

La seconde forme de commentaires vient du C++. C'est le commentaire sur une seule ligne qui commence avec `//` et continue jusqu'à la fin de la ligne. Ce type de commentaire est pratique et souvent rencontré car il est simple. Il n'y a pas à se démener sur le clavier pour trouver `/` puis `*` (à la place il suffit d'appuyer deux fois sur la même touche) et il n'est pas nécessaire de fermer le commentaire. On trouve donc fréquemment :

```
// Ceci est un commentaire sur une seule ligne
```

Commentaires de documentation

Un des plus solides éléments de Java est que les concepteurs n'ont pas considéré que l'écriture du code est la seule activité importante -- ils ont aussi pensé à sa documentation. Le plus gros problème avec la documentation de code est probablement la maintenance de cette documentation. Si la documentation et le code sont séparés, ça devient embêtant de changer la documentation chaque fois que le code change. La solution a l'air simple : relier le code et la documentation. Le moyen le plus simple de le faire est de tout mettre dans le même fichier. Toutefois, pour compléter le tableau il faut une syntaxe de commentaire particulière pour marquer la documentation particulière et un outil pour extraire ces commentaires et les mettre sous une forme exploitable. C'est ce que Java a fait.

L'outil pour extraire les commentaires est appelé *javadoc*. Il utilise certaines technologies du compilateur Java pour rechercher les marqueurs spécifiques des commentaires qui ont été mis dans les programmes. Il ne se contente pas d'extraire les informations repérées par ces marqueurs, mais il extrait aussi le nom de classe ou le nom de méthode adjoint au commentaire. Ainsi on parvient avec un travail minimal à générer une documentation de programme convenable.

La sortie de *javadoc* est un fichier HTML qui peut être visualisé avec un browser Web. Cet outil permet de créer et maintenir un unique fichier source et à générer automatiquement une documentation utile. Grâce à *javadoc* on a un standard pour créer la documentation et il est suffisamment simple pour qu'on puisse espérer ou même exiger une documentation avec toute bibliothèque Java.

Syntaxe

Toutes les commandes *javadoc* n'apparaissent que dans les commentaires `/**`. Les commentaires finissent avec `*/` comme d'habitude. Il y a deux principales façons d'utiliser *javadoc* : du HTML intégré ou des « onglets doc ». Les onglets doc sont des commandes qui commencent avec un `'@'` et qui sont placées au début d'une ligne de commentaire (toutefois, un `'*'` en tête est ignoré).

Il y a trois types de commentaires de documentation qui correspondent aux éléments suivant le commentaire : classe, variable ou méthode. C'est à dire qu'un commentaire de classe apparaît juste avant la définition de la classe, un commentaire de variable apparaît juste avant la définition de la variable et un commentaire de méthode apparaît juste avant la définition de la méthode. Voici un exemple simple :

```
/** Un commentaire de classe */

public class docTest {

    /** Un commentaire de variable */

    public int i;

    /** Un commentaire de méthode */

    public void f() {}

}
```

Il faut noter que javadoc ne traite les commentaires de documentation que pour les membres **public** et **protected**. Les commentaires pour les membres **private** et « amis » (voir Chapitre 5) sont ignorés et on ne verra aucune sortie (toutefois on peut utiliser le flag **private** pour inclure aussi les membres **private**). Ceci est sensé puisque seuls les membres **public** et **protected** sont accessibles en dehors du fichier, ce qui est la perspective du client du programmeur. Toutefois, tous les commentaires de classe sont inclus dans le fichier de sortie.

La sortie du code précédent est un fichier HTML qui a le même format standard que tout le reste de la documentation Java, ainsi les utilisateurs seront à l'aise avec le format et pourront facilement naviguer dans les classes. Ça vaut la peine de taper le code précédent, de le passer dans javadoc et d'étudier le fichier HTML résultant pour voir le résultat.

HTML intégré

Javadoc passe les commandes HTML dans les documents HTML générés. Ceci permet une utilisation complète de HTML, la motivation principale étant de permettre le formatage du code comme suit :

```
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
```

On peut aussi utiliser HTML comme on pourrait le faire dans n'importe quel autre document Web pour formater du texte courant dans les descriptions :

```
/**
 * On peut <em>même</em> insérer une liste :
 *
 * <ol>
 *
 * <li> élément un
```

```
* <li> élément deux
* <li> élément trois
* </ol>
* /
```

Il faut noter que dans les commentaires de documentation, les astérisques en début de ligne sont éliminés par javadoc, ainsi que les espaces en tête de ligne. Javadoc reformate tout pour assurer la conformité avec l'apparence des documentations standard. Il ne faut pas utiliser des titres tels que **<h1>** ou **<hr>** dans le HTML intégré car javadoc insère ses propres titres et il y aurait des interférences.

Tous les types de commentaires de documentation -- classes, variables et méthodes -- acceptent l'intégration de HTML.

@see : faire référence aux autres classes

Les trois types de commentaires de documentation (classes, variables et méthodes) peuvent contenir des onglets **@see**, qui permettent de faire référence à de la documentation dans d'autres classes. Javadoc générera du HTML où les onglets **@see** seront des hyper liens vers d'autres documentations. Les différentes formes sont :

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Chacune d'entre elles ajoute un hyper lien de type « Voir aussi » à la documentation générée. Javadoc ne vérifie pas si l'hyper lien indiqué est valide.

Class documentation tags

En plus du HTML intégré et des références **@see**, les documentations de classe peuvent inclure des onglets pour les informations de version et pour le nom de l'auteur. Les documentations de classe peuvent aussi être utilisées pour les *interfaces* (voir Chapitre 8).

@version

Voici le format :

```
@version version-information
```

dans lequel **version-information** est n'importe quelle information significative que l'on souhaite inclure. Quand le flag **-version** est mis sur la ligne de commande de javadoc, les informations de version seront exploitées, particulièrement dans la documentation HTML.

@author

Voici le format :

```
@author author-information
```

dans lequel **author-information** est, a priori, votre nom mais il peut aussi contenir une adresse email ou toute autre information appropriée. Quand le flag **-author** est mis sur la ligne de commande javadoc, les informations sur l'auteur seront exploitées, particulièrement dans la documentation HTML.

On peut avoir plusieurs onglets d'auteur pour une liste d'auteurs mais ils doivent être placés consécutivement. Toutes les informations d'auteurs seront regroupées dans un unique paragraphe dans le code HTML généré.

@since

Cet onglet permet d'indiquer la version du code qui a commencé à utiliser une caractéristique particulière. On la verra apparaître dans la documentation HTML de Java pour indiquer quelle version de JDK est utilisée.

Les onglets de documentation de variables

Les documentations de variables ne peuvent contenir que du HTML intégré et des références @see.

Les onglets de documentation de méthodes

En plus du HTML intégré et des références @see, les méthodes acceptent des onglets de documentation pour les paramètres, les valeurs de retour et les exceptions.

@param

Voici le format :

```
@param parameter-name description
```

dans lequel **parameter-name** est l'identificateur dans la liste de paramètres et **description** est du texte qui peut se prolonger sur les lignes suivantes. La description est considérée comme terminée quand un nouvel onglet de documentation est trouvé. On peut en avoir autant qu'on veut, a priori un pour chaque paramètre.

@return

Voici le format :

```
@return description
```

dans lequel **description** indique la signification de la valeur de retour. Le texte peut se prolonger sur les lignes suivantes.

@throws

Les exceptions seront introduites dans le Chapitre 10 mais, brièvement, ce sont des objets qui peuvent être émis (thrown) par une méthode si cette méthode échoue. Bien qu'une seule exception puisse surgir lors de l'appel d'une méthode, une méthode donnée est susceptible de produire n'importe quel nombre d'exceptions de types

différents, chacune d'entre elles nécessitant une description. Ainsi, le format de l'onglet d'exception est :

```
@throws fully-qualified-class-name description
```

dans lequel **fully-qualified-class-name** indique sans ambiguïté un nom de classe d'exception qui est définie quelque part, et **description** (qui peut se prolonger sur les lignes suivantes) précise pourquoi ce type particulier d'exception peut survenir lors de l'appel de la méthode.

@deprecated

Ceci est utilisé pour marquer des fonctionnalités qui ont été remplacées par d'autres qui sont meilleures. L'onglet **deprecated** suggère de ne plus utiliser cette fonctionnalité particulière étant donné qu'elle sera probablement supprimée ultérieurement. Une méthode marquée **@deprecated** fait produire un warning par le compilateur si elle est utilisée.

Exemple de documentation

Voici à nouveau le premier programme Java, cette fois après avoir ajouté les commentaires de documentation :

```
///  
import java.util.*;  
  
/** Le premier exemple de programme de Thinking in Java.  
 * Affiche une chaîne de caractères et la date du jour.  
 * @author Bruce Eckel  
 * @author http://www.BruceEckel.com  
 * @version 2.0  
 */  
  
public class HelloDate {  
    /** Unique point d'entrée de la classe et de l'application  
     * @param args tableau de paramètres sous forme de chaînes de caractères  
     * @return Pas de valeur de retour  
     * @exception exceptions Pas d'exceptions émises  
     */  
  
    public static void main(String[] args) {  
        System.out.println("Hello, it's: ");  
        System.out.println(new Date());  
    }  
} ///  
///  
}
```

La première ligne du fichier utilise une technique personnelle qui consiste à mettre un ':' comme marqueur

spécifique pour la ligne de commentaire contenant le nom du fichier source. Cette ligne contient le chemin du fichier (dans ce cas, c02 indique le Chapitre 2) suivi du nom de fichier [25]. La dernière ligne finit aussi avec un commentaire et celui-ci indique la fin du listing du code source, ce qui permet de l'extraire automatiquement du texte de ce livre et de le contrôler avec un compilateur.

Style de programmation

Le standard non officiel de Java consiste à mettre en majuscule la première lettre des noms de classes. Si le nom de classe est composé de plusieurs mots, ils sont accolés (c'est à dire qu'on ne sépare pas les noms avec un trait bas) et la première lettre de chaque mot est mise en majuscule ainsi :

```
class AllTheColorsOfTheRainbow { // ...
```

Pour pratiquement tout le reste : méthodes, champs (variables membres) et les noms des références d'objets, le style retenu est comme pour les classes *sauf* que la première lettre de l'identificateur est une minuscule. Par exemple :

```
class AllTheColorsOfTheRainbow {  
    int anIntegerRepresentingColors;  
    void changeTheHueOfTheColor(int newHue) {  
        // ...  
    }  
    // ...  
}
```

Bien entendu il faut se rappeler que l'utilisateur doit aussi taper tous ces longs noms, donc soyez clément.

Le code Java qu'on voit dans les bibliothèques de Sun respecte aussi le placement des accolades ouvrantes et fermantes qui est utilisé dans ce livre.

#/TIJ_PAGE03# #TIJ_PAGE04#

Sommaire

Dans ce chapitre on en a vu suffisamment sur la programmation Java pour comprendre comment écrire un programme simple et on a eu une vue d'ensemble du langage et de certaines des idées de base. Toutefois, les exemples vus jusqu'à présent ont tous été du type « faire ceci, puis faire cela, puis faire autre chose ». Qu'en advient-il si on veut faire un programme pour réaliser des choix, comme dans « si le résultat de ceci est rouge, faire cela, sinon faire autre chose » ? Les outils disponibles en Java pour cette activité de programmation fondamentale seront vus dans le prochain chapitre.

Exercices

1. En suivant l'exemple **HelloDate.java** de ce chapitre, créez un programme « hello, world » qui affiche simplement cette phrase. Vous n'avez besoin que d'une seule méthode dans la classe (la méthode « main »

qui est exécutée quand le programme démarre). Pensez à la rendre **static** et à indiquer la liste de paramètres, même si la liste de paramètres n'est pas utilisée. Compilez ce programme avec **javac**. Si vous utilisez un environnement de développement autre que JDK, apprenez à compiler et exécuter les programmes dans cet environnement.

2. Trouver le morceau de code concernant **ATypeName** et faites-en un programme qui compile et s'exécute.
3. Transformez le morceau de code **DataOnly** en un programme qui compile et qui s'exécute.
4. Modifiez l'exercice 3 de telle sorte que les valeurs des données dans **DataOnly** soient affectées et affichées dans **main()**.
5. Écrivez un programme qui inclus et appelle la méthode **storage()** définie comme morceau de code dans ce chapitre.
6. Transformez le morceau de code **StaticFun** en un programme qui fonctionne.
7. Écrivez un programme qui imprime trois paramètres saisis sur la ligne de commande. Pour faire ceci il faut indexer le tableau de **String** représentant la ligne de commande.
8. Transformez l'exemple **AllTheColorsOfTheRainbow** en un programme qui compile et s'exécute.
9. Trouvez le code de la seconde version de **HelloDate.java** qui est le petit exemple de commentaire de documentation. Exécutez javadoc sur le fichier et visualisez le résultat avec votre browser Web.
10. Transformez docTest en un fichier qui compile et exécutez javadoc dessus. Contrôlez la documentation qui en résulte avec votre browser Web.
11. Ajoutez une liste d'éléments en HTML, à la documentation de l'exercice 10.
12. Prenez le programme de l'exercice 1 et ajoutez lui une documentation. Sortez cette documentation dans un fichier HTML à l'aide de javadoc et visualisez la avec votre browser Web.

[21] Ceci peut être une poudrière. Il y a ceux qui disent « c'est clairement un pointeur », mais ceci suppose une implémentation sous-jacente. De plus, les références Java sont plutôt apparentées aux références C++ qu'aux pointeurs dans leur syntaxe. Dans la première édition de ce livre, j'ai choisi d'inventer un nouveau terme, « manipulateur » (*handle*), car les références C++ et les références Java ont des différences importantes. Je venais du C++ et je ne voulais pas embrouiller les programmeurs C++ que je supposais être le principal public de Java. Dans la seconde édition, j'ai décidé que « référence » était le terme le plus généralement utilisé et que tous ceux qui venaient du C++ auraient à s'occuper de bien d'autres choses que de la terminologie de référence, donc qu'ils pourraient aussi bien y plonger les deux pieds en avant. Toutefois, il y a des gens qui ne sont pas d'accord, même avec le terme « référence ». J'ai lu dans un livre qu'il était « complètement faux de dire que Java supportait les passages par références », parce que les identificateurs des objets Java (selon cet auteur) sont *en fait* des « références à des objets ». Et (continue-t-il) tout est *en fait* passé par valeur. Donc on ne passe pas une référence, on « passe une référence à un objet, par valeur ». On pourrait discuter de la précision d'une explication si alambiquée mais je pense que mon approche simplifie la compréhension du concept sans blesser qui que ce soit (d'accord, les avocats du langage pourraient prétendre que je mens, mais je répondrai que je fournis une abstraction appropriée).

[22] Les méthodes **static**, que nous découvrirons ultérieurement, peuvent être appelées *pour la classe*, sans passer par un objet.

[23] Avec les exceptions habituelles pour les types de données précités **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**, et **double**. En général, toutefois, on passe des objets, ce qui veut dire qu'en réalité on passe des références à des objets.

[24] Certains environnements de programmations feront apparaître brièvement le programme à l'écran et le fermeront avant d'avoir eu une chance de voir le résultat. On peut mettre le morceau de code suivant à la fin du **main()** pour obtenir une pause sur la sortie :

```
try {
    System.in.read();
} catch (Exception e) {}
```

Cette pause durera jusqu'à ce qu'on appuye sur « Entrée » (ou toute autre touche). Ce code met en jeu des concepts qui ne seront introduits que bien plus tard dans ce livre, donc vous ne le comprendrez pas d'ici là, mais il fera l'affaire.

[25] Un outil que j'ai créé avec Python (voir www.Python.org) utilise cette information pour extraire les fichiers sources, les mettre dans les sous-répertoires appropriés et créer les makefiles.

#/TIJ_PAGE04#