Tiger : la révolution de Java 5

(Les principales nouveautés de J2SE 5.0)

-

Lionel Roux

-

version 1.0

Introduction.

Avec sa suite de développement Visual Studio .Net, ses composants totalement intégrés et des langages simples, la plateforme .Net de Microsoft semble séduire de plus en plus de développeurs et de décideurs.

Face à cette montée en puissance, Sun se devait de réagir en proposant une nouvelle version de sa technologie Java, plus simple et offrant une meilleure productivité.

Avec son J2SE[a] 5.0, Sun a décidé de frapper fort en révolutionnant littéralement sa plateforme Java et en tentant de la rendre à nouveau incontournable.

Le J2SE 5.0 (nom de code ***Tiger***) est la prochaine révision majeure de la plateforme Java. Elle est prévue pour être dévoilé en version finale pour l'automne 2004.

Elle doit intégrer pas moins de 15 composants issus des JSR[b] formulées, et c'est une première, via le JCP[c], c'est-à-dire que sous la supervision de Sun MicroSystem, un certain nombre d'industriels et d'institutions (BEA, Borland, IBM, Doug Lea, Oracle, Apache Software Fundation, …) du secteur sont impliqués dans ces spécifications.

Cette version doit apporter des améliorations majeures dans la qualité, la supervision et la gestion, la performance et la faculté de montée en charge, ainsi que la facilité de développement.

Dans cet article, nous nous focaliserons sur les fonctionnalités les plus significatives et les plus excitantes de Tiger, notamment celles qui sont attendues depuis nombres d'années par les développeurs Java.

Nous ferons tout d'abord une revue des quelques fonctionnalités ou améliorations du langage censées améliorer la lisibilité, la facilité et la rapidité de développement.
Puis nous verrons rapidement quelques unes des nouvelles API majeures ajoutées dans cette nouvelle version.
Enfin, nous nous arrêterons plus longuement sur le principe de généricité via les *Generics* et sur la programmation par annotation via les *MetaData* introduits dans Tiger.

Attention : Les exemples donnés dans cet article ont été validés avec la révision J2SE 5.0 Beta2. Il se peut que lors de la sortie de la version finale du J2SE 5.0, certains aspects aient changés.

Table des matières.

Les améliorations et nouveautés du langage.

Un des leitmotivs de Sun et des utilisateurs de Java depuis l'avènement de cette technologie, est de simplifier un langage qui, à force d'enrichissement depuis le JDK 1.1, est devenu très complexe. Ceci est d'autant plus vrai que les programmes deviennent de plus en plus imposants par leurs designs et leurs fonctionnalités.

Avec Tiger, Sun a souhaité apporter une réelle réponse à ce problème.
La plus grande partie des nouveautés apportées au langage va d'ailleurs dans ce sens, en améliorant la lisibilité, la simplicité et la rapidité des développements.

Désormais, nul besoin de transtyper les types primitifs. L'*autoboxing* s'en charge pour vous.
Une nouvelle boucle `for` simplifiée fait aussi son apparition et l'ellipse est introduite pour représenter un nombre variable d'arguments, comme le fait la sortie formatée `printf` chère au cœur des développeurs C. Et ce n'est pas tout, beaucoup de ces petites choses qui changent la vie d'un développeur sont au rendez vous.

Nous allons dresser dans ce chapitre une liste non exhaustive des améliorations les plus intéressantes.

L'autoboxing des types primitifs.

Le langage Java s'appuie sur des types primitifs pour décrire les types de base. Il s'agit essentiellement des types représentant des nombres (`byte`, `short`, `int`, `long`, `double` et `float`), des booléens (`boolean`) et des caractères (`char`).
Cependant, en Java, tout se doit d'être « objet ».

Ainsi, un code comme celui qui suit lève une erreur de compilation avec J2SE 1.4:

```
List list = new ArrayList();
int myInt = 100;

list.add(myInt); //erreur de compilation ici (java.lang.ClassCastException)
```

Pour résoudre cette erreur; il faut encapsuler l'entier dans une classe référence (*wrapper*) comme la classe *Integer* :

```
list.add(new Integer(myInt)) ;
```

Ce type de transformation (aussi appelé *boxing*) peut rapidement s'avérer pénible. D'autant que le processus inverse (appelé *unboxing*) est nécessaire pour retrouver son type initial.

Avec Tiger, cette conversion explicite devient obsolète. En effet, Tiger introduit l'*autoboxing* qui convertit de manière transparente les types de bases en références.
La table de conversion qui suit donne les correspondances entre type de base et *wrapper* (enrobeur) :

Type primitif	Wrapper
boolean	Boolean
byte	Byte
double	Double
short	Short
int	Integer
long	Long
float	Float
char	Character

On peut ainsi écrire directement :

```
List list = new ArrayList();
int myInt = 100;

list.add(myInt);
```

Le compilateur se chargera de transtyper automatiquement la variable dans son type enrobé.

Les itérations simplifiées : la nouvelle boucle `for`.

En java, une tâche récurrente et fastidieuse est l'itération sur une collection (au sens large c'est-à-dire autant sur un objet `Collection` que sur un tableau).
En effet, pour parcourir une collection, il faut instancier un itérateur sur celle-ci et pour chaque itération récupérer l'élément dans un objet conteneur. De plus, chaque élément doit être transtypé dans son type d'origine pour être utilisé en tant que tel.
L'expression actuelle du `for` est assez puissante et peut être utilisée afin de parcourir des collections ou des tableaux. Cependant, elle n'est pas optimisée pour une simple itération, car l'itérateur ne sert qu'à obtenir les éléments.

```
List stringList = new ArrayList();
stringList.add("foo");
stringList.add("bar");
…
for(Iterator it = stringList.iterator(); it.hasNext(); )
{
 String currentString = (String)it.next();

 // Do something with the string...
}
```

Tiger introduit une manière supplémentaire d'utiliser le mot clé `for`, qui s'avère être beaucoup plus légère et facile à mettre en œuvre.

Sa syntaxe est la suivante :

```
for(FormalParameter : Expression) Statement
```

`Expression` doit être un tableau ou une instance de la nouvelle interface `java.lang.Iterable`. L'interface `java.util.Collection` étend donc maintenant `Iterable`.

Avec cette nouvelle manière d'utiliser le `for`, nul besoin d'instancier explicitement un itérateur sur la collection.
De plus, le type des éléments contenus dans la collection est défini dans le corps de la boucle, évitant ainsi le recours au transtypage explicite.

```
for(String currentString: stringlist)
{
 // Do something with the string...
}
```

On le voit, cette nouvelle forme simplifie le développement et améliore la lisibilité des programmes.

Il existe cependant une limitation à celle-ci, il n'est pas possible de l'utiliser lorsque l'on souhaite modifier la collection, puisque aucun itérateur n'est accessible.

On peut aussi déplorer l'utilisation d'un mot clé ayant déjà une utilisation propre. Sun rétorque à cela que l'introduction d'un nouveau mot clé (`foreach` ?) est très (trop ?) coûteuse.

Les types énumérés type-safe : le nouveau mot clé enum.

Un type énuméré est un type dont la valeur fait partie d'un jeu de constantes définies. Jusqu'ici, le type `enum` du C avait été omit en Java.

Une façon simple de créer un type énuméré est de définir une classe ne contenant que des constantes de type `int` :

```
public class Numbers
{
 public static final int ZERO  = 0;
 public static final int ONE   = 1;
 public static final int TWO   = 2;
 public static final int THREE = 3;
 …
}
```

Cependant, celle-ci comporte plusieurs désavantages :

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

Un type énuméré « type-safe » est un type énuméré qui provoque des erreurs de compilation si l'on tente de lui affecter des valeurs non permises. Il existe bien entendu des patterns permettant de créer ces types énumérés, qui de plus contiennent des méthodes de traitement et leur associe des descriptions.

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

```
public enum Numbers {ONE, TWO, THREE, …};
```

Pour chaque type décrit dans l'énumération, une classe complète (valeur et méthodes utilitaires) est générée (à la différence des énumérations en C/C++ qui ne sont que des entiers).

De plus une classe supplémentaire est générée (`Numbers.class` ici), qui implémente l'interface `Comparable` et `Serializable`, qui contient des variables statiques correspondant

aux valeurs, et qui définit ou redéfinit un certain nombre de méthodes (`values()`, `toString()`, `valueOf(String s)`, `hashCode()`, `compareTo()`, …).

Ainsi, nous utilisons les énumérations Java de la manière suivante :

```
Numbers num = Numbers.ONE;
```

Au lieu de :

```
int num = Numbers.ONE;
```

Cette construction présente les avantages suivants :

- Elle est « *type-safe* ».
- Elle fournit un espace de nommage pour chaque énumération, évitant le recours à l'utilisation d'un préfixe dans les noms des constantes.
- Etant des objets à part entière, chaque constante peut changer son implémentation sans obliger à changer le code source des classes qui l'utilisent et donc, sans obliger à recompiler.
- Les valeurs, au lieu d'être de simples nombres, peuvent être des messages explicites.

De plus, le mot clé `enum` remplaçant le mot clé `class`, nous pouvons imaginer tout type de constructeur, permettant ainsi de créer des énumérations complexes :

```
public enum Numbers
{
 ONE("this is the first occurrence", 1),
 TWO("this is the second occurrence", 2),
 THREE("this is the third occurrence", 3);

 Numbers(String description, int value)
 {
  this.description = description;
  this.value = value;
 }

 private string description;
 private int  value;


 public String getDescription()
 {
  return this.description;
 }

 public int getValue()
 {
  return this.value;
 }
}
```

Les méthodes à arguments variables : l'ellipse.

Quel développeur n'a pas été un jour confronté au problème du traitement d'un nombre variable de paramètres ?
Ce problème récurrent, bien que facilement contournable par le passage en paramètre d'un tableau de données dans les signatures des méthodes, nous oblige à spécialiser celles-ci. Ce qui n'est pas sans enlever une certaine généricité ou oblige à utiliser le polymorphisme, tout en compliquant les interfaces.

Par exemple :

```
void doSomething(Object[] args)
{
 // do something
}

// invoke the method
doSomething(new Object[]{"arg1", "arg2", "arg3"});
```

Tiger introduit l'ellipse "…" pour remédier à cela. Il s'agit d'un opérateur qui informe le compilateur que le nombre d'arguments peut être variable.

La précédente signature devient alors :

```
void doSomething(Object… args)
{
 // do something
}

// invoke the method
doSomething("arg1", "arg2", "arg3");

// invoke the method in an other way
doSomething("arg1", "arg2");
```

Cette notation, en plus d'apporter de la généricité, permet d'alléger légèrement le travail du développeur.

Attention : l'ellipse de Tiger ne s'utilise pas de la même manière que l'ellipse du *C++*.
En effet, l'ellipse du *C++* prévoit que l'on peut passer n'importe quel type d'argument dans la signature. En Java, le type précédant directement l'ellipse définit le type des arguments susceptibles d'être ajoutés.

Nous allons en montrer un exemple concret dans le chapitre suivant qui concerne la nouvelle méthode de sortie standard formatée : le très connu `printf(…)`.

La sortie standard formatée : la (nouvelle) méthode `printf(…)`.

Tiger introduit une nouvelle méthode permettant de produire des messages formatés sur la sortie standard. Celle-ci vient en complément de l'actuelle méthode `System.out.print()`.

Cette méthode s'appuie directement sur le mécanisme d'ellipse vu précédemment et fonctionne comme la très connue fonction `printf(…)` en C.

Avec l'actuelle méthode `println()`, nous avons :

```
double a = 5.6d ;
double b = 2d ;

double mult = a * b ;

System.out.println(a + " mult by " + b + " equals " + mult);
```

Avec le nouveau `printf(…)`, nous aurons :

```
double a = 5.6d ;
double b = 2d ;

double mult = a * b ;

System.out.printf("%lf mult by %lf equals %lf \n", a, b , mult);
```

Les deux méthodes produisent le même message sur la sortie standard :

```
> 5.6 mult by 2 equals 11.2
>
```

Mais ce n'est pas tout. Cette nouvelle méthode permet bien plus, comme des formatages avancés :

```
double a = 5.6d ;
double b = 2d ;

double mult = a * b ;

System.out.printf("%3.2lf mult by %3.2lf equals %3.2lf\n", a, b , mult);
```

Le message standard sera dans ce cas :

```
> 005.60 mult by 002.00 equals 011.20
>
```

La gestion des flux standards : le Scanner.

L'API `Scanner` fournit des fonctionnalités de base pour lire tous les flux standards.
Cette classe peut être utilisée pour convertir du texte en primitives ou en `Strings`.
De plus, étant donné qu'elle est basée sur le package `java.util.regex`, elle offre un moyen simple d'appliquer des expressions régulières sur des flux, des fichiers de données, des `Strings`, ou tout autre objet implémentant la nouvelle interface `Readable`.

Par exemple, pour lire l'entrée standard, il suffit d'invoquer la méthode `next()` du `Scanner` :

```
Scanner scanner = Scanner.create(System.in);

try
{
 String s1 = scanner.next(Pattern.compile("[Yy]"));
 // only match Y or y
}
catch(InputMismatchException e)
{
 System.out.println("Expected Y or y");
}

scanner.close();
```

Ce bout de code simple permet de capter toutes les frappes de `Y` ou `y` sur le clavier si celui-ci est l'entrée standard.
Il est à noter que la méthode `next()` est bloquante si aucune donnée n'est disponible dans le flux.

Le `Scanner` peut aussi être utilisé à la place du classique `StringTokenizer`. Il peut utiliser tous types de délimiteurs grâce à l'adjonction des expressions régulières :

```
String input = "1 test 2 test red test blue test ";

Scanner s = Scanner.create(input).useDelimiter("\\s*test\\s*");

System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());

s.close();
```

Ces instructions produisent la sortie suivante:

```
>1
>2
>red
>blue
```

Quiconque a déjà utilisé les `StringTokenizer` pour découper ou parser un fichier se rend tout de suite compte de la puissance du `Scanner`.

Les imports statiques.

Toujours dans leur volonté de simplifier les codes source écrits en Java, les personnes en charge des spécifications de Tiger ont retenu celle des imports statiques.

La proposition vise à introduire une variante de l'import pour rendre visibles des méthodes et des variables statiques de la même manière que l'on importe des classes ou des interfaces. Cette fonctionnalité permet de passer outre l'anti-pattern de l'interface constante (voir *Effective Java* de Joshua Bloch, le *lead manager* de Tiger).

La forme retenue pour la déclaration d'un tel import est la suivante :

```
import static TypeName . Identifier ;

import static TypeName . * ;
```

Son but principal, outre simplifier le code source, est de pallier au problème de l'espace de nommage déjà évoqué dans le chapitre sur les types énumérés (préfixage obligatoire) et de permettre d'ajouter des constantes et méthodes de portée globale (quasiment l'équivalent des `#define` en C, avec des différences tout de même).
Jusqu'ici la seule façon de faire était de les définir en tant qu'attributs statiques d'une classe conteneur, et de faire ensuite explicitement référence à cette classe en utilisant le nom de ce conteneur comme espace de nommage.

L'exemple typique est la classe `Math` de l'API standard dans laquelle toutes les méthodes et tous les attributs sont statiques:

```
Math.PI ;
Math.sin(x) ;
Math.cos(x) ;
…
```

Tiger assouplit l'utilisation de ces classes en autorisant les imports statiques. Cela permet de traiter les membres des classes statiques en tant que classes globales (pour le fichier courant) de la même manière que nous le faisons pour les classes classiques.
La conséquence directe est la suppression des préfixes obligatoires lorsque l'on fait référence aux membres statiques.

On peut ainsi écrire :

```
import static java.lang.Math.PI ;
import static java.lang.Math.sin();

// ou pour importer toutes les références statiques
import static java.lang.Math.* ;

PI ; // fait référence à Math.PI
sin(x) ; // fait référence à Math.sin()
cos(x) ; // fait référence à Math.cos()
…
```

Un bémol est toutefois à apporter à cette amélioration.
Si nous utilisons cette facilité sans prendre garde, nous pouvons arriver à des situations de conflits d'espaces de nommage. C'est pourquoi je recommande de ne pas utiliser cette fonctionnalité qui n'apporte pas de révolution et qui est sujet à introduire des erreurs (régressions en cas de reprise d'un code pré-existant).
D'autant plus que les éditeurs actuels savent très bien gérer cela.

Les imports statiques ne devraient être utilisés que pour des petites classes où il n'y a pas de conflit d'espace de nommage.

Les nouvelles APIs majeures de Java 5.

Chaque nouvelle version de J2SE apporte son lot de nouvelles API intégrées.
J2SE 1.4 nous avait ainsi gratifié d'une API permettant de gérer les entrées/sorties, aussi appelée `NIO` (*New Input Outpout*) ou encore de `JAXP` (*Java API for XML Parsing*).

Tiger n'est pas en reste avec notamment l'ajout majeur d'une API très complète de gestion de la concurrence à haut niveau, ou encore l'intégration de JMX pour le management et de JVMTI pour la supervision.

La synchronisation de haut niveau : l'API de concurrence.

La plate forme Java fournit des primitives basiques et de bas niveau pour écrire des programmes concurrents et synchronisés, mais celles-ci sont difficiles à utiliser correctement. La plupart des programmes deviennent plus clairs, plus rapides, plus faciles à écrire et plus fiables si une synchronisation de haut niveau est utilisée.

Une librairie étendue d'utilitaires de gestion de la concurrence a été intégrée dans le J2SE 5.0. Cette API, connue comme le package `java.util.concurrent`, contient des pools de threads, des queues, des collections synchronisées, des verrous spéciaux (atomiques ou pas), des barrières (ou barrage), et bon nombre d'autres utilitaires, comme un framework d'exécution de taches. L'ajout à la plateforme Java de cette librairie est un apport substantiel qui va bouleverser la façon dont nous écrivons la gestion de la concurrence dans les applications Java.

En fait cette API n'est autre que celle développée par Doug Léa, professeur au State University of New York (SUNY) College d'Oswego et qui est devenue le JSR-166 (anciennement dans le package edu.oswego.cs.dl.util.concurrent).

Nous allons voir par la suite une vue rapide des principaux éléments de cette énorme librairie.

Les exécuteurs de tâches.

L'interface `Executor` est une super-interface simple et standard qui permet de contrôler des sous-systèmes d'exécution de tâches (en fait des `Runnables`), comme des pools de threads, des entrées / sorties asynchrones et des frameworks légers de gestion de tâches.
Elle permet de découpler les appels d'exécution des exécutions elles-mêmes, en précisant l'utilisation de chaque thread, l'ordonnancement, etc.
En fonction du type d'exécuteur utilisé, la tache peut être exécutée par un nouveau thread, un thread existant ou même par le thread qui demande l'exécution de celle-ci, et ce de manière séquentielle ou concurrente :

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

L'interface `ExecutorService` fournit un framework d'exécution de tâches entièrement asynchrone. Celui-ci gère la mise en queue, l'ordonnancement et la terminaison des tâches. Il fournit à cet effet des méthodes de terminaison de processus et d'autres de monitoring de la progression d'une ou plusieurs tâches asynchrones (via les `Future`, retournant l'état du traitement ou thread voulu).

Tiger offre en standard deux implémentations concrètes, très flexibles et hautement configurables, de cette interface : la classe `ThreadPoolExecutor` et la classe `ScheduledThreadPoolExecutor`.

D'autres part, une classe utilitaire (helper), `Executors`, fournit des fabriques et des moyens de configurer les principaux `Executors`, ainsi que nombre de méthodes utilitaires.

L'exemple de référence crée un serveur en quelques lignes :

```
class NetworkService
{
 private final ServerSocket serverSocket;
 private final ExecutorService pool;

 public NetworkService(int port, int poolSize) throws IOException
 {
  serverSocket = new ServerSocket(port);
  pool = Executors.newFixedThreadPool(poolSize);
 }

 public void serve()
 {
  try
  {
   for (;;)
   {
    pool.execute(new Handler(serverSocket.accept()));
   }
  }
  catch (IOException ex)
  {
   pool.shutdown();
  }
 }
}

class Handler implements Runnable
{
 private final Socket socket;
 Handler(Socket socket) { this.socket = socket; }
 public void run()
 {
   // read and service request
 }
}
```

Les queues.

Tiger introduit une nouvelle interface pour les collections : `java.util.Queue`.
Cette nouvelle interface comporte un certain nombre de signatures supplémentaires pour ajouter, supprimer des éléments, et parcourir la collection:

```
public boolean offer(Object element)
public Object remove()
public Object poll()
public Object element()
public Object peek()
```

Une queue n'est rien d'autre qu'une structure de données de type FIFO (First In, First Out). La valeur ajoutée de Tiger est ici sur les moyens mis à disposition pour agir sur la queue.

Par exemple, à la différence de la méthode `add()`, la méthode `offer()` ne lève plus d'exception de type unchecked lorsqu'une erreur se produit. Elle retourne simplement `false`. De la même manière, la méthode `poll()` retourne `null` si nous tentons de supprimer un élément d'une collection vide, à la différence de la méthode `remove()` que nous connaissons.

Nous distinguons deux types de queues : celles qui implémentent l'interface `BlockingQueue` et celles qui ne le font pas et qui implémentent uniquement `Queue`, comme la classe `LinkedList` existante, ou les nouvelles `PriorityQueue` (ordonnancement naturel des éléments) et `ConcurrentLinkedQueue` (thread-safe).

Les `BlockingQueue` sont des queues qui se bloquent lorsque l'on tente d'ajouter ou de retirer un élément tant que l'espace disponible n'est pas suffisant. Elles fonctionnent dans un mode producteur / consommateur. Il existe cinq implémentations de ce type de queue dans le package `java.util.concurrent` :

`ArrayBlockingQueue`:	Une queue avec des limites de taille minimale et maximale, dont la structure sous jacente est un tableau.
`LinkedBlockingQueue`:	Une queue, que l'on peut limiter, et dont la structure sous jacente est un ensemble de noeuds de liaisons.
`PriorityBlockingQueue`:	Une `PriorityQueue` implémentée comme un segment.
`DelayQueue`:	Une queue dont les éléments peuvent expirer, et implémentée comme un segment.
`SynchronousQueue`:	Une simple queue attendant qu'un consommateur demande son élément.

Les Map atomiques.

Tiger introduit l'interface `ConcurrentMap` et son implémentation `ConcurrentHashMap`.
Il s'agit de `Map` dont les opérations sont atomiques et concurrentes.

Elles sont à distinguer des collections « synchronisées » avec le framework `Collections` (comme `Hashtable` et `Collections.synchronizedMap(new HashMap())`).
Les collections de `java.util.concurrent` sont « concurrentes », c'est-à-dire qu'elles sont « thread safe », mais pas supervisées par un seul et unique verrou d'exclusion (comme le sont les collections « synchronisées »).

Par exemple, `ConcurrentHashMap` permet non seulement des lectures simultanées mais aussi de décider du nombre d'écritures simultanées, et ce de manière déterministe. Elle possède donc une meilleure capacité de monter en charge que les `Hashtables`, qui elles ne permettent qu'une lecture ou qu'une écriture simultané.

Elles apportent deux méthodes essentielles pour cela :

- `putIfAbsent(String key, Object value)` qui ajoute l'objet uniquement si la clé n'existe pas dans la structure et retourne la valeur insérée. Sinon elle préserve la valeur existante en la retournant.

- `remove(String key, Object Value)` qui ne supprime l'entrée que si la valeur de l'objet pour la clé désignée est égale à la valeur passée. Dans ce cas elle retourne `true`. Sinon elle retourne `false`.

La représentation du temps et de ses unités.

La classe `TimeUnit` représente une unité de durée, pouvant aller jusqu'à la nanoseconde, qui fournit différentes méthodes pour déterminer et contrôler des opérations de timing (comme un « ordonnanceur »), d'ajournement ou de durée de vie.

Par exemple, pour faire échoir au bout de 50 millisecondes une tentative de pose d'un verrou sur un bloc, en cas d'échec de celle-ci :

```
Lock lock = ...;
if ( lock.tryLock(50L, TimeUnit.MILLISECONDS) ) ...
```

Ainsi, si la pose du verrou échoue, le traitement ne se bloque pas au delà de cinquante milliseconde.

Il faut cependant noter que `TimeUnit` représente uniquement une granularité de temps et ne garantit pas la détection de l'événement dans le même ordre de granularité.
Autrement dit, le système n'impose pas de contrainte "dure" à l'application par l'utilisation de `TimeUnit`. Dans notre exemple précédent, l'échéance du `lock` peut intervenir après soixante millisecondes si le système est occupé à autre chose.
`TimeUnit` définit simplement une "grandeur" vers laquelle le système va tenter de tendre.

Les synchroniseurs (loquet, barrière, sémaphore et échangeur).

Les synchroniseurs sont des éléments essentiels pour la gestion de la concurrence. En effet, il s'agit d'idiomes de haut niveau mettant en oeuvre des mécanismes ou algorithmes particuliers qui apportent différents moyens de contrôler la mise en concurrence et/ou la synchronisation de différents processus.

Tiger fournit quatre types des synchroniseurs :

- Les loquets.
- Les barrières.
- Les sémaphores.
- Les échangeurs.

Les **loquets** (ou *latchers*) via la classe `CountDownLatch`, permettent de bloquer (avec la méthode `await()`) plusieurs threads jusqu'à ce qu'un certain nombre d'opérations soient finies.
Typiquement, chacune de ces opérations va décrémenter un compteur non altérable au sein du loquet en invoquant la méthode `countDown()`, jusqu'à zéro. A ce moment, le loquet est levé et chaque thread en attente est débloqué et peut donc reprendre son exécution.

Le cas le plus simple de loquet est réalisé par un compteur de valeur un. Il s'agit alors d'un interrupteur, qui lorsqu'il est fermé, débloque le courant (en l'occurrence ici, le traitement).

Un autre cas typique est le découpage d'un traitement en plusieurs micros traitements (le fameux « diviser pour régner »), comme par exemple le découpage en sous requêtes d'une requête vers une base de données :

```
class RequestProcessor
{
 // ...
 void main() throws InterruptedException
 {
  CountDownLatch latcher = new CountDownLatch(N);
  Executor e = ...

  for (int i = 0; i < N; ++i)
  {
   // create and start threads
   e.execute(new RequestRunnable(latcher, i));

   latcher.await(); // wait for all to finish
  }
 }
}

class RequestRunnable implements Runnable
{
 private final CountDownLatch localLatcher;
 private final int id;

 RequestRunnable(CountDownLatch latcher, int id)
 {
  this. localLatcher = latcher;
  this.id = id;
 }

 public void run()
 {
  try
  {
   doRequest(id);
   localLatcher.countDown();
  }
  catch (InterruptedException ex) {..};
 }

 void doRequest () { ... }
}
```

Les **barrières** permettent de définir un point d'attente commun à plusieurs processus. Lorsque tous les processus enregistrés par la barrière ont atteint ce point, ils sont débloqués.

Elles sont particulièrement efficaces pour des traitements divisibles et récursifs, car à la différence des loquets pour lesquels le compteur est fixe, une barrière est réutilisable immédiatement après l'opération de déblocage. Elles sont alors qualifiées de cycliques et Tiger en fournit une implémentation dans la classe `CyclicBarrier`.

De plus, il est possible de déclarer une tâche optionnelle dans la barrière qui sera à même de changer l'état de celle-ci, car exécutée juste avant le déblocage des processus.

On peut par exemple imaginer un traitement récursif sur des matrices, chaque résultat de traitement sur une matrice en créant une autre, jusqu'au résultat voulu.

Le **sémaphore** est sans doute le mécanisme de synchronisation le plus connu et le plus classique. C'est un mécanisme permettant d'assurer une forme d'exclusion mutuelle.
Plus précisément, un sémaphore est une valeur entière S associée à une file d'attente. Nous pouvons accéder au sémaphore par deux méthodes, la première P attend que S soit positif et décrémente S (le processus est mis dans la file d'attente si S n'est pas strictement positif), la deuxième V incrémente S ou libère un des processus en attente sur un P.

Pour schématiser, nous pouvons dire que le sémaphore dispose d'un nombre fini de jetons. Des processus ayant besoin d'une ressource tentent d'acquérir un de ces jetons. S'il n'y a plus de jetons, ils se mettent en attente. Lorsque le traitement est terminé, le processus détenteur d'un jeton relâche ce dernier dans le sémaphore. Le premier processus en attente peut alors acquérir à son tour le jeton ainsi relâché.

Ce pattern simple règle le problème de la section critique avec un seul jeton (sémaphore binaire) :

```
private Semaphore s = new Semaphore(1, true);

s.acquireUninterruptibly(); // bloquant
value = balance +1; // section critique exclusive
s.release();
```

Il peut aussi servir pour réaliser des buffers de 10 éléments partagés entre des producteurs et des consommateurs :

```
class Resources
{
 private static final RES_NUMBER = 10;
 private final Semaphore token = new Semaphore(RES_NUMBER, true);
 private List resourcesPool = new ArraylList(RES_NUMBER);

 public Object getResource()
   throws InterruptedException
 {
  token.acquire();
  return resourcesPool.get();
 }

 public void releaseResource(Object o)
 {
  resourcesPool.put(o);
  token.release();
 }
}
```

Bien entendu, cet exemple est ridicule. Il est là uniquement pour illustrer le mécanisme.

Il permet surtout de mettre en attente une demande et de la débloquer automatiquement, sans aucune intervention ou test supplémentaires.

Le dernier type de synchroniseur est **l'échangeur** (`Exchanger`).
Un échangeur permet à deux processus de s'échanger mutuellement des données à un certain point dans le traitement. Nous l'appelons aussi un « rendez-vous ». Il définit un canal commun qui permet à deux objets de communiquer en s'échangeant d'autres objets.

On peut le voir comme un sablier qui lorsqu'il est prêt (ou arrive au « rendez-vous »), ouvre sa vanne de communication. Le contenu de la partie haute se déverse alors dans la partie basse :

```
class SandGlass
{
 Exchanger<Data> exchanger = new Exchanger();

 class SandGlassLowRoom implements Runnable
 {
  public void run()
  {
   Data content = computeContent();
   try
   {
    content = exchanger.exchange(content);
   }
   catch (InterruptedException ex){}
  }
 }

 class SandGlassHighRoom implements Runnable
 {
  public void run()
  {
   Data content = computeContent();
   try
   {
    content = exchanger.exchange(content);
   }
   catch (InterruptedException ex){}
  }
 }

 void openRooms()
 {
  new Thread(new SandGlassLowRoom()).start();
  new Thread(new SandGlassHighRoom()).start();
 }
}
```

Les traitements asynchrones anticipés.

Lorsqu'un traitement est long, il peut être intéressant de le déclencher de manière asynchrone et de n'en récupérer le résultat que plus tard, lorsque celui-ci est fini. De plus, il se peut que le développeur souhaite garder la main sur cette tâche, pour pouvoir par exemple l'interrompre à

tout moment, et ce sans créer un code bloquant, ou pour notifier un utilisateur de la fin de celle-ci. C'est ce que propose de définir la nouvelle interface `Future`.

`Future` représente le « résultat » d'un traitement asynchrone, c'est-à-dire un objet sur lequel nous conservons une référence, et que nous pouvons utiliser en lieu et place du véritable résultat.
De plus cette interface fournit des méthodes utilitaires permettant d'interroger l'état du traitement (`isDone()`), d'interrompre le traitement (`cancel()`) ou encore d'en récupérer le résultat (`get()`). Une dernière méthode permet de savoir si le traitement a été interrompu (`isCancelled()`), ce qui permet donc de vérifier la cohérence du résultat.

La méthode `get()` possède deux formes, dont l'une est bloquante tant que le traitement n'est pas terminé et donc le résultat non disponible, et une autre qui permet de définir un temps maximum d'attente avant de relâcher le verrou (exprimé grâce à un `TimeUnit`). Cette dernière peut être particulièrement intéressante pour gérer des problématiques d'IHM lourdes (chargement asynchrone d'image par exemple).

Voici un exemple permettant de construire un rapport depuis une base de données :

```
class ReportBuilder
{
 Future<ResultSet> result = null;
 SQLService sqlService = …;

 void build(String reportStatement)
  throws InterruptedException
 {
  result = executor.submit
           (
             new Callable<String>()
             {
              public String call()
              {
               return sqlService.query(reportStatement) ;
              }
             }
           )
 }
}
```

Combiné à la puissance des `Executors`, il est très aisé de lancer plusieurs traitements longs et de les mettre en queue pour une utilisation ultérieure, notamment avec les `ExecutorCompletionService` qui découplent la production de nouvelles tâches asynchrones de la consommation de leurs résultats. Les producteurs présentent (`submit()`) des tâches à exécuter et les consommateurs récupèrent (`take()`) les tâches accomplies et traitent leurs résultats.

De la même manière, les `Futures` peuvent servir à réaliser un cache hautement disponible, puisque la demande d'un élément non existant du cache, qui provoque en général son chargement au sein dudit cache, ne bloque pas l'accès aux autres éléments déjà cachés.

La classe `FutureTask` est une implémentation par défaut de `Future` et de `Runnable`, ce qui permet de simplifier grandement le développement avec des `Futures`.

Les variables atomiques.

Le package `java.util.concurrent.atomic` contient des classes « conteneurs » atomiques, parmi lesquelles nous trouvons `AtomicInteger`, `AtomicLong`, et `AtomicReference`, qui fournissent des mécanismes sécurisés d'opération basique de mise à jour ou d'interrogation, comme le sont les variables `volatiles`.

Toutes les opérations ne se font que si un certain nombre de conditions sont réunies et sans qu'un autre processus ne puisse agir sur le conteneur.

Elles sont très pratiques pour implémenter par exemple des compteurs sans recourir à des mécanismes de synchronisation particuliers, puisqu'elles le sont nativement.

De plus, elles sont beaucoup plus performantes que leurs équivalents dans `java.lang` utilisant la synchronisation, puisqu'elles s'appuient directement sur le nouveau mécanisme interne à la JVM 1.5, le « *compare-and-swap* ».

Les nouveaux verrous « haute performance ».

Le package `java.util.concurrent.locks` fournit de nouveaux mécanismes de verrouillage. Ils sont similaires aux verrous déjà présents dans l'API du J2SE 1.4, mais y ajoutent des fonctionnalités non supportés par le moniteur actuel comme la faculté d'interrompre un processus bloqué sur un verrou, de définir un temps limite de blocage sur un verrou, … .

De plus, ils sont beaucoup plus performants que la synchronisation actuelle, au prix il est vrai d'une rigueur dans la syntaxe beaucoup plus présente.

Le package comporte des implémentations de verrous et de conditions de verrouillage (dont se servent les fameux `wait()`,`notify()`, `notifyAll()`).

L'interface `Lock` supporte des méthodes de verrouillage fonctionnellement différentes (multi-entrées, propre, etc..). L'implémentation principale est la classe `ReentrantLock` qui est un verrou multi-entrées (qui peut être lu temporairement par plusieurs utilisateurs ou plusieurs fois par un seul utilisateur) exactement comme l'actuelle synchronisation (exclusion mutuelle). La classe `ReadWriteLock` permet d'implémenter des verrous pour des objets multi-utilisateurs, mono-producteur.

L'interface `Condition` permet de définir des variables conditionnelles que l'on associe à des verrous. Elles sont utilisées par le moniteur pour placer des conditions de blocage ou de déblocage. Plusieurs conditions peuvent être associées à un seul verrou (à la différence de l'ancien moniteur).

Voici l'exemple du *buffer* que fournit la documentation du JDK 5.0 :

```
class BoundedBuffer
{
 Lock lock = new ReentrantLock();
 final Condition notFull  = lock.newCondition();
 final Condition notEmpty = lock.newCondition();

 Object[] items = new Object[100];
 int putptr, takeptr, count;

 public void put(Object x)
    throws InterruptedException
 {
  lock.lock();
  try
  {
   while (count == items.length)
     notFull.await();

   items[putptr] = x;
   if (++putptr == items.length) putptr = 0;
     ++count;
   notEmpty.signal();
  }
  finally
  {
   lock.unlock();
  }
 }

 public Object take()
     throws InterruptedException
 {
  lock.lock();
  try
  {
   while (count == 0)
    notEmpty.await();
   Object x = items[takeptr];
   if (++takeptr == items.length) takeptr = 0;
    --count;
   notFull.signal();
   return x;
  }
  finally
  {
   lock.unlock();
  }
 }
}
```

La gestion et la supervision de la JVM : l'API de management.

Depuis bien longtemps, les utilisateurs d'applications écrites en Java demandaient une extension qui leur permettent de gérer facilement les événements de la JVM (*Java Virtual Machine*), et plus particulièrement l'état de la mémoire à bas niveau.

J2SE 5.0 fournit pour cela une extension permettant de supporter ces éléments clés garants de la fiabilité, de la disponibilité et du fonctionnement normal des applications Java.

Elle est composée de deux parties principales :

- L'API JMX (*Java Management Extensions*) bien connue en tant qu'API externe, qui consiste en un framework d'observation de la JVM. Les caractéristiques internes des données de celle-ci, comme les informations sur les processus et la mémoire sont désormais disponibles pour être publiées via l'interface de MBeans JMX (*JSR 003*), l'interface distante JMX (*JSR 160*) ou encore directement via l'espace d'adressage Java. Un ensemble d'outils et de protocoles standard d'accès (*JSR163*), comme le protocole SNMP (*Simple Network Management Protocol*), permettent la remontée d'informations vers un moniteur.
- Un outil de « profiling » natif appelé JVMTI, permettant de profiler les applications Java, mais aussi de les superviser, de les debugger voire de les manipuler directement au niveau du bytecode, c'est-à-dire à chaud, pendant leur exécution.

Non seulement vous pouvez voir les informations de la JVM, mais le framework fournit les outils permettant d'agir sur celles-ci, d'essayer de nouvelles valeurs, comme changer le niveau de traces fonctionnelles (*log*) dynamiquement.

JPLIS (*Java Programming Language Instrumentation Services*) permet d'ajouter des points de contrôle ciblés dans le code, à la manière de la POA (*Programmation Orienté Aspect*), permettant ainsi une supervision plus fine de la JVM, et ce à chaud, au chargement de l'application, ou encore durant la compilation (opération de tissage).

L'autre fonctionnalité phare de cette API de management, est sans nul doute le détecteur de seuil de mémoire. Lorsque le seuil est atteint, une notification peut être envoyée par le bean JMX ou une « trap » SNMP peut être levée. Cela permet par exemple de prévoir une charge importante sur une application et de la gérer avec des machines supplémentaires en répartissant dynamiquement la charge.

La prise en main de cette API est très simple. Par exemple pour définir une supervision de la JVM via le protocole SNMP, on peut très simplement faire :

```
java -Dcom.sun.management.snmp.acl=false
     -Dcom.sun.management.snmp.port=161
     -Dcom.sun.management.snmp.trap=162
     -jar TestApplication.jar
```

Les ports utilisés ici sont les ports standard SNMP. De la même manière, il est possible de définir ces options dans un fichier de configuration (*management.properties*).

Associer un serveur de MBean JMX est tout aussi simple :

```
java -Dcom.sun.management.jmxremote.authenticate=false
     -Dcom.sun.management.jmxremote.port=5001
     -Dcom.sun.management.jmxremote.ssl=false
     -jar TestApplication.jar
```

L'exemple qui suit permet de suivre l'évolution de l'utilisation de la mémoire par la JVM. Comme nous allons pouvoir le constater, cette fonctionnalité est très simplement prise en charge par l'API de management :

```
import java.lang.management.*;
import java.util.*;

public class MemTest
{
 public static void main(String args[])
 {
  List<MemoryPoolMXBean> pools = ManagementFactory.getMemoryPoolMXBeans();
  for (MemoryPoolMXBean p: pools)
  {
   System.out.println("Memory type="+p.getType()
   System.out.print("Memory usage="+p.getUsage());
  }
 }
}
```

Voici le type de sortie que produit le code précédent:

```
Memory type=Non-heap memory Memory usage=initSize =163840, used =494144, committed =524288, maxSize =33554432
Memory type=Heap memory Memory usage=initSize =524288, used =166440, committed =524288, maxSize =-1
Memory type=Heap memory Memory usage=initSize =65536, used =0, committed =65536, maxSize =-1
Memory type=Heap memory Memory usage=initSize =65536, used =0, committed =65536, maxSize =0
Memory type=Heap memory Memory usage=initSize =1441792, used =0, committed =1441792, maxSize =61997056
Memory type=Non-heap memory Memory usage=initSize =8388608, used =84360, committed =8388608, maxSize =67108864
Memory type=Non-heap memory Memory usage=initSize =8388608, used =5844808, committed =8388608, maxSize =8388608
Memory type=Non-heap memory Memory usage=initSize =12582912, used =6010560, committed =12582912, maxSize =2582912
#
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  SIGSEGV (0xb) at pc=0x42073770, pid=24776, tid=1073993792
#
# Java VM: Java HotSpot(TM) Client VM (1.5.0-beta-b31 mixed mode)
# Problematic frame:
# C  [libc.so.6+0x73770]  __libc_free+0x70
#
# An error report file with more information is saved as /tmp/hs_err_pid24776.log
#
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
#
```

Une autre fonctionnalité très simple à mettre en œuvre, est la détection des crashs de la JVM HotSpot suite à une erreur fatale. Il est alors possible de déclencher une commande :

```
-XX:OnError="command"
```

Il est aussi possible de brancher un debugger ou un fichier système lors de cette détection :

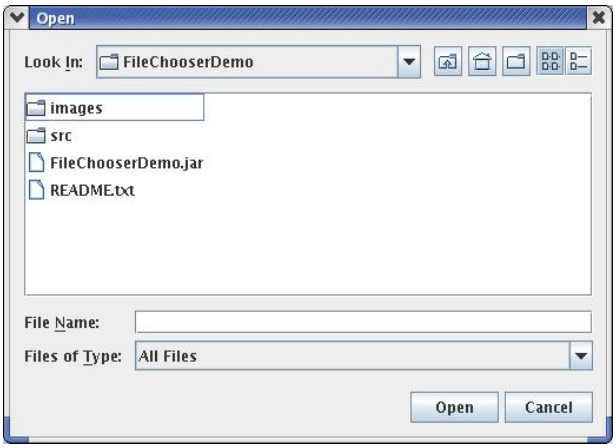
```
-XX:OnError="pmap %p" // %p représente le pid.
-XX:OnError="gdb %p"
```

Cette API étant très riche et les possibilités qu'elle offre très nombreuses, veuillez vous reporter au site de référence de JMX pour de plus amples informations :

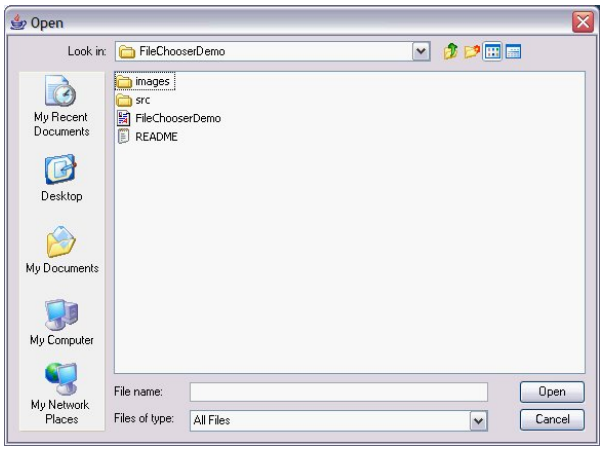
http://java.sun.com/products/JavaManagement/

Les nouveautés pour les clients lourds.

Tiger améliore grandement le temps de démarrage et la taille mémoire occupée par les applications développées en client lourd.
De plus il introduit un nouveau thème appelé « Ocean », en plus des thèmes « Windows XP » et « GTK » déjà ajoutés dans le J2SE 1.4.2.



L&F GTK



L&F Windows XP

L&F Ocean

Il faut aussi noter que les apparences graphiques sont désormais personnalisables (principe des skins), via des fichiers *XML*, ou de manière programmatique.

Tiger permet aussi d'utiliser les capacités d'accélération graphique avec *OpenGL* pour les applications Java développées en utilisant les composants *Java2D*.
Il suffit pour cela de le spécifier dans la ligne de commande de lancement de l'application :

```
java -Dsun.java2d.opengl=true -jar Java2DApplication.jar
```

Pour les utilisateurs du système d'exploitation *Linux*, il est désormais possible d'utiliser le système graphique X11, via XAWT (`sun.awt.X11.XToolkit`), une librairie graphique très rapide s'appuyant sur X11, et permettant d'utiliser le protocole *XDnD* (glisser / déposer) entre les applications Java et des applications comme *Mozilla* et *Star Office*.
Comme il est d'usage, cela se fait via la ligne de commande :

```
java -Dawt.toolkit=sun.awt.motif.MToolkit -jar X11Application.jar
```

La métaprogrammation par annotation.

La programmation déclarative qu'introduit Tiger prend des idées dans de nombreux domaines de l'informatique et plus particulièrement dans la métaprogrammation, la réflexivité et les protocoles à méta-objets, que nous devons à Mehmet Askit, Jean Pierre Briot, Shigeru Chiba, Pierre Cointe, Jacques Ferber, Patricia Maes et Brian Smith.
Elle se veut surtout être l'équivalent des « attributs » que propose aujourd'hui le framework .NET de Microsoft.

La JSR 175 à l'origine de celle-ci ajoute une nouvelle construction du langage par annotation dans Tiger, qui peut aider à simplifier et rationaliser les programmes, en fournissant un ensemble standard d'éléments d'enrichissement.

Nous allons voir dans ce chapitre ce qui se cache derrière cette nouvelle fonctionnalité.

Intérêts de la métaprogrammation.

La plupart des programmeurs Java connaissent les tags *Javadoc*. Nous savons que ces tags qui apparaissent dans les commentaires peuvent être traités par l'utilitaire *Javadoc* pour générer de la documentation.
Cette notation apporte du sens supplémentaire à un programme, comme par exemple le tag `@deprecated` qui indique souvent la méthode suppléante à utiliser.
L'information ajoutée, bien que non déterminante pour le programme, est souvent décisive pour la compréhension de celui-ci.

Pourtant, actuellement, le compilateur et la machine virtuelle ignorent ce type de tags, qui constituent un type d'annotations.

Certains développeurs ont intelligemment détournés leur utilisation pour leur faire exécuter des taches spécifiques via un interpréteur.
C'est par exemple le cas de l'outil *EJBGen* qui fournit un ensemble de tags propriétaires qui, une fois reconnus, génèrent un ensemble de classes et fichiers utiles pour construire des *EJBs*, comme les descripteurs de déploiement, les interfaces et le conteneur.
Dans ce type d'approche nous pouvons aussi citer *XDoclet*.

Nous le voyons, ce type de métaprogrammation peut apporter beaucoup dans la facilité de développement, la réduction du volume de code, la simplification et la rationalisation des programmes.

Définition des annotations.

Les annotations font partie d'un mécanisme générique qui associe des métadonnées (des informations déclaratives comme des tags) à des éléments d'un programme comme une classe, une méthode, un champ, un paramètre, des variables locales ou un package.

Les annotations s'inscrivent dans l'approche de la programmation déclarative, à opposer à la programmation impérative utilisée dans les programmes Java classiques, avec des instructions à exécuter dans un ordre précis.

La programmation déclarative définie des conditions et laisse le système déterminer comment les satisfaire. Elle décrit donc plus des prédicats que des instructions.

Dans Tiger, le compilateur agit comme un tisseur (*weaver*). Un tisseur est un compilateur qui compose des programmes à partir d'éléments externes et non connus de l'application tissée.
Le principe du tisseur est notamment très utilisé dans la Programmation Orientée Aspect (POA) utilisant des extensions du langage (par opposition aux *frameworks*).

Le compilateur de J2SE 5.0 peut stocker des métadonnées dans des classes. La machine virtuelle ou tout autre programme peuvent à tout moment venir lire ces métadonnées pour déterminer la manière d'interagir avec les éléments annotés du programme ou changer leur comportement. En ce sens, le tisseur de Tiger se rapproche d'un tisseur d'aspect.

Une annotation est un modificateur qui peut être utilisé sur n'importe quelle déclaration. Elle contient en général des couples membre/valeur (parfois optionnels, si des valeurs par défaut existent). Il existe trois types d'annotations : normale, repère et membre unique.
Nous allons voir par la suite comment les écrire et les utiliser.

Syntaxe et utilisation des annotations.

Pour créer une annotation, nous devons tout d'abord définir un type d'annotation.
Un type d'annotation est définit comme une interface avec un symbole « @ » avant celle-ci :

```
public @interface ExampleAnnotation
{
 String member1;
}
```

Une fois ce type crée nous pouvons l'utiliser comme ceci :

```
@ExampleAnnotation (member1="value1")
public void doSomething()
{
 …
}
```

Une annotation dîtes « normale » est utilisée pour annoter de manière ordinaire un élément du programme :

```
@NormalAnnotation(name1="value1", name2="value2")
public void doSomething()
{
 …
}
```

Une annotation dîtes « marqueur » ne contient pas de membre.
Une annotation dîtes « membre unique » contient un et un seul membre.

Les méta-annotations.

La spécification d'annotation de programme fournit la possibilité d'ajouter des méta-annotations, qui sont utilisées pour « annoter » des annotations. La classe `java.lang.Annnotation` en fournit les moyens.

Les méta-annotations sont prédéfinies et prennent la forme suivante :

```
@MetaTag(MetaValue)
public @interface WithMetaAnnotation
{
 String value;
}
```

Il existe cinq types de méta-annotations :

Meta-annotation	But
`@Documented`	Déclare que l'annotation **doit** être documentée par le biais de tag *Javadoc* ou autre. Elle indique aussi à l'utilitaire de documentation qu'il doit prendre en compte cette annotation.
`@Inherited`	Déclare que l'annotation est automatiquement héritée. Si un membre est ainsi annoté, et que ses descendants ne le sont pas, alors ils prendront automatiquement l'annotation de leur celui-ci.
`@Retention`	Déclare la politique d'usage de l'annotation. Elle définit dans quelle mesure l'annotation sera conservée (ou pas) à la fois dans le bytecode et dans la JVM. Les valeurs possibles sont `SOURCE`, `CLASS` et `RUNTIME`.
`@Target`	Déclare que l'annotation porte sur tel out tel type d'élement d'un programme. Les valeurs possibles sont `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `LOCAL_VARIABLE`, et `PACKAGE`.
`@SuppressWarnings`	Déclare que les avertissements du compilateur indiqué doivent être supprimés.

Voici un exemple plus complet :

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Documented
public @interface WithMetaAnnotation
{
 String value;
}
```

La méta annotation `Retention` déclare que `@WithMetaAnnotation` doit être stockée dans une classe et utilisée par la machine virtuelle à l'exécution par réflexion.
La méta annotation `Target` déclare que `@WithMetaAnnotation` peut être utilisée pour annoter des méthodes.

Les annotations standard.

Il existe deux types d'annotations standard dans le package `java.lang` :

Annotation	But
`@Deprecated`	Redéfinit le tag *Javadoc* du même nom comme une annotation. Le compilateur lancera un avertissement lorsque les membres annotés par ce tag sont utilisés.
`@Overrides`	Déclare que la déclaration de cette méthode redéfinit celle de sa classe ascendante. Si la méthode ainsi annotée ne redéfinit pas celle de sa super classe, alors un message d'erreur est lancé.

L'exemple suivant illustre l'utilisation des annotations standard:

```
public class Parent
{
 @Deprecated
 public void aMethod(int x)
 {
  System.out.println("Parent.aMethod(int x) called.");
 }
}
```

Supposons que nous souhaitions définir une classe étendant `Parent`, qui redéfinit la méthode `aMethod()`:

```
public class Child extends Parent
{
 @Overrides
 public void aMethod()
 {
  System.out.println("Child.aMethod() called.");
 }
}
```

Nous avons ici volontairement définit une méthode `aMethod()` avec une signature différente pour illustrer notre propos. Alors que cette erreur n'aurait auparavant été identifiée qu'à l'exécution et au prix d'une longue recherche, le compilateur du J2SE 5.0 va, à la compilation, lever une erreur :

```
javac -source 1.5 Parent.java Child.java
Child.java:3: method does not override
a method from its superclass
@Overrides
 ^
1 error
```

Si nous corrigeons cette erreur dans la signature, le compilateur va notifier que la méthode est marquée comme dépréciée, puisque dans la super classe `aMethod()` est annotée avec `@deprecated` :

```
javac -Xlint:deprecation -source 1.5 Child.java
Child.java:4: warning:
[deprecation] aMethod(int) in Parent has been deprecated
public void aMethod (int x) {
            ^
1 warning
```

Restrictions sur les annotations.

Certaines restrictions existent quant à l'utilisation et la déclaration des annotations :

- Les annotations sont déclarées comme des interfaces. Cependant il n'est pas possible de faire hériter une annotation d'une autre annotation en utilisant le mot clé `extends`. Il faut pour cela utiliser la méta-annotation `@Inherited`.

- Les méthodes déclarées dans une annotation ne doivent contenir ni paramètre, ni paramètre de type formel (voir les *generics*). Elles doivent en outre ne retourner que des types primitifs, des `String`s, des `Class`, des types énumérés, des types d'annotations ou des tableaux des précédents types.

- Les annotations ne doivent pas contenir de clause `throws`.

Les Generics.

Après plus de cinq ans d'attente depuis la première proposition, les *generics* vont finalement être intégrés au langage Java, dans le J2SE 5.0. Le chemin pour arriver à l'adoption de cette technologie pour le moins controversée mais sans doute essentielle fut long et douloureux.

Le manque des types génériques dans le langage Java est sujet à polémique quasiment depuis la sortie de ce langage de programmation.
En effet, cette fonctionnalité est une des extensions du langage les plus demandées par la communauté des développeurs Java (*JDC*). Elle arrive en seconde position pour les extensions du langage et en septième dans le bug parade.
Ainsi, nous comprenons mieux l'importance que prend cet ajout vis-à-vis du langage.

Le point de départ fut la conférence donnée par Gilad Bracha, théologiste chez Sun et aujourd'hui le leader technique de la JSR 014, « Add Generic Types to the Java Programming Language ».

L'implémentation retenue pour les generics en Java est basée sur celle de GJ (Generics Java), une extension du langage Java, développée par Philip Wadler, Martin Odersky, Gilad Bracha, et Dave Stoutamire.

Nous allons voir dans ce chapitre ce que sont les *generics* et ce qu'ils permettent.

Le principe des generics.

Pour les personnes familières avec les *templates* en C++, les *generics* prennent tout leurs sens, bien que des différences fondamentales existent.

Par type générique, on entend aussi polymorphisme paramétrique de type.

De manière concise, on peut dire que les types génériques permettent de développer un comportement unique pour des types polymorphes.

En d'autre terme, les *generics* permettent de s'abstraire du typage des objets lors de la conception et donc de définir des comportements communs quel que soit le type des objets manipulés.

Le plus simple pour comprendre les *generics* est d'analyser un exemple type :

```
List integerList = new ArrayList();
integerList.add(new Integer(1));
Integer i = (Integer)integerList.get(0);
```

Dans cet exemple, nous remarquons que nous sommes obligé de transtyper explicitement l'objet que l'on récupère de la collection, car la seule chose dont nous sommes sûrs, c'est que l'itérateur retourne un `Object`.

Ainsi, rien ne nous empêche d'insérer un objet de type `String` dans la liste :

```
integerList.add("exemple"); // (1)
```

Une telle construction, bien que fausse, ne déclenchera aucune erreur lors de la compilation.
C'est seulement à l'utilisation que le programme lèvera une exception de type `java.lang.ClassCastException`.

Avec Tiger, il est possible, via une syntaxe particulière, de spécifier explicitement et avant construction, quel sera le type des objets contenus dans la collection :

```
List integerList<Integer> = new ArrayList<Integer>();
integerList.add(new Integer(1));
Integer i = integerList.get(0);
```

La syntaxe `<Integer>` spécifie que le type des objets utilisés avec cette collection est `Integer`.

Ainsi, l'exemple précédent `(1)` fera échouer la compilation car les *generics* introduisent une vérification de type statique, c'est-à-dire durant la compilation.

En fait, dans Tiger, les collections sont désormais implémentées en utilisant les types génériques.

Quels sont les avantages des *generics*? C'est ce que nous allons voir dans le paragraphe suivant.

Pourquoi utiliser les generics ?

Le premier avantage des *generics* réside dans la suppression du contrôle de type à l'exécution. En effet, étant donné que les éventuelles erreurs de transtypage sont levées à la compilation, il n'est plus nécessaire de contrôler le type à l'exécution.
Ainsi, le code gagne en sécurité et le temps de maintenance éventuel en est très fortement réduit.

D'autre part, nul code supplémentaire à écrire (comme pour les *templates* C++) pour introduire les *generics*. La manière de développer ne change donc pas, si ce n'est des différences mineures dans la syntaxe (que nous verrons dans le paragraphe suivant).

Un avantage visible directement est une meilleure lisibilité et une plus grande robustesse du code.

Un autre aspect moins frappant mais tout aussi important, est la possibilité qu'offrent les types génériques de factoriser des comportements.
Les développeurs passent moins de temps à refaire sans cesse la même chose pour des types différents qu'ils doivent manipuler et peuvent ainsi se concentrer sur la logique fonctionnelle de leurs classes.
Le code en est d'autant plus concis et moins éparpillé et gagne ainsi en cohérence, en taille et en qualité.

La syntaxe des generics en Java.

Nous allons fixer dans ce chapitre un certain nombre d'appellations et de notations.

Comme nous l'avons déjà vu dans le premier exemple, les premiers symboles nouveaux avec les types génériques sont les chevrons < et >.
Ils permettent de définir les paramètres de types formels qui peuvent être utilisés dans toutes les déclarations génériques, et en particulier en lieu et place des types ordinaires (bien qu'il y ait d'importantes restrictions).
A l'invocation de l'objet, toutes les occurrences du paramètre de type formel sont remplacées par le type défini.

Ainsi, l'interface `List` est définie de la sorte :

```
public interface List<T>
{
 void add(T t);
 Iterator<T> iterator();
}
```

`T` est ici le paramètre de type formel de l'interface `List`.

J'insiste bien sur le fait que `T` est un type, et pas une valeur. Nous verrons pourquoi dans le chapitre suivant.

D'autre part, une convention de nommage semble être acceptée par la communauté pour la notation des paramètres de type formel.
Elle recommande l'utilisation d'un caractère unique en majuscule comme `E` (*Element*) ou `T` (*Type*).

L'autre symbole important pour les *generics* est le point d'interrogation (`?`), que l'on appelle aussi *wildcard* ou *inconnue*.

Ainsi nous aurions pu écrire :

```
List unKnownList<?> = new ArrayList<String>();
```

Il permet de définir la variance d'un type générique.
Il permet de spécifier que n'importe quel type peut convenir comme élément de la liste.

Nous reviendrons plus loin sur les différentes possibilités qu'offre le *wildcard*.

Les generics ne sont pas des templates !

Nous l'avions déjà évoqué dans l'introduction, les *generics* ont des différences majeures avec les *templates* tels que nous les connaissons en *C++* par exemple.

Nous avons vu que les occurrences d'un paramètre de type formel étaient remplacées à l'invocation.
A la lecture de ces mots, il est très facile d'extrapoler le code précédent d'une `List` qui aurait pour type `String` par exemple :

```
public interface StringList
{
 void add(String t);
 Iterator<String> iterator();
}
```

Ce mécanisme ne doit pas aller sans rappeler aux développeurs *C++* les *templates*.

Et bien cette extrapolation est **fausse**, et c'est en cela que les *templates* et les *generics* sont différents. En effet, il n'y a pas de multiples copies du code, en fonction du type, ni en mémoire, ni dans les sources, ni dans le *bytecode* généré.

Un *generics* est compilé une fois pour toute en une unique classe comme une classe ou une interface classique et l'instance de cette classe est partagée entre toutes les invocations :

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
boolean jvmIdentity = (l1.getClass() == l2.getClass());
```

Ainsi, la valeur booléenne `jvmIdentity` vaut `true`.

Les paramètres de type formel sont tout simplement remplacés à l'invocation comme les valeurs formelles le sont dans des classes ordinaires, avec la valeur courante du type donnée dans la construction du code (resp. la valeur donnée).

Les generics et l'héritage.

L'héritage avec les *generics* est une des choses les plus ardues à comprendre, car elle va à contresens des réflexions *a priori*.

En effet, si une classe `C'` hérite d'une classe `C`, et `G` est un *generics* de paramètre `T`, alors il est **faux** de dire que `G<C'>` hérite de `G<C>`. Pourtant cela semblerait être logique.

Voyons cela avec un exemple.
Supposons que nous disposions de la liste des habitants d'une ville et de la liste des clients d'un magasin de cette ville. Nous supposons aussi que la classe `Customer` hérite de la classe `Resident`.

```
List<Customer> customers = new ArrayList<Customer>;
```

Si nous supposons que tous les clients habitent la ville, on peut penser initialiser la liste des habitants comme ceci :

```
List<Resident> residents = customers;
```

Or cette initialisation est fausse; une liste de `Resident` n'est pas une liste de `Customer`. Nous allons voir pourquoi cette initialisation pose problème.
Supposons que l'initialisation de la liste des habitants se soit poursuivie comme ceci, sans l'erreur précédente :

```
residents.add(new Resident(…));
```

Si par la suite nous tentons de récupérer le premier client de la liste des clients, nous faisons naturellement :

```
Customer c = customers.get(0);
```

Or dans cette instruction nous tentons d'assigner à une instance de `Customer` une instance de `Resident`, puisque nous avons ajouté un habitant à la liste des habitants et que nous avons créé un alias entre les deux listes.

Cela ne poserait pas de problème si la liste des habitants était immutable. Or ce n'est pas le cas. En réalité, ce qui est passé à la liste des habitants est une copie de la liste des résidents.

Sinon, le service commercial du magasin pourrait ajouter à sa liste de clients des personnes qui ne sont pas clientes.

C'est pour cela que le compilateur de J2SE 5.0 lève une erreur de compilation pour la seconde instruction.
Ceci est pour le moins restrictif, et pour palier à cela, Tiger comporte un mécanisme permettant de passer outre : il s'agit de l'utilisation du *wildcard*.

La variance dans les generics.

La variance désigne les limites de portée d'un type générique.
Elle peut se définir comme les limites de la portée d'un type classique par les mots clé `extends`, `super` et `implements` et s'utilise de la même manière.
Pour combiner deux limites sur un même type, il faut utiliser le caractère `&`.

La construction respecte la règle suivante :

```
TypeVariable keyword Bound1 & Bound2 & ... & Boundn
```

Voici un exemple de limite complexe :

```
final class Foo<A extends Comparable<A> & Cloneable<A>,
                B extends Comparable<B> & Cloneable<B>>
implements Comparable<Foo<A,B>>, Cloneable<Foo<A,B>>
{
 …
}
```

Le wildcard dans les generics.

Supposons que la mairie de la ville souhaite envoyer un courrier à tous ses habitants qui sont clients du magasin. Le service informatique de la mairie a une méthode permettant d'envoyer un courrier aux habitants de la ville.

Nous aurions une méthode qui ressemble à celle-ci :

```
void mailResidents(List<Resident> residents)
{
 for(Resident r : residents)
 {
  sendMail(r);
 }
}
```

Nous tenterions de l'utiliser comme ceci:

```
List<Customer> customers = …
…
mailResidents(customers);
```

Or nous avons vu dans le chapitre précédent qu'il est impossible d'assigner à une liste d'habitants une liste de clients.

Quelle est alors l'alternative ? Il est tout à fait normal de vouloir (et pouvoir…) faire ce type d'opération. Pour cela nous avons le *wildcard* (`?`).

Nous pouvons en effet modifier la méthode `mailResidents()` afin qu'elle accepte n'importe quel type d'individu :

```
void mailResidents(List<?> persons)
{
 for(Person p : persons)
 {
  sendMail(p);
 }
}
```

Si nous souhaitons que le système se limite à tout type d'individu, pour peu que ceux-ci soient des habitants de la ville, nous pouvons écrire :

```
void mailResidents(List<? extends Resident> residents)
{
 for(Resident r : residents)
 {
  sendMail(r);
 }
}
```

Ainsi, comme `Customer` hérite de `Resident`, l'utilisation de la méthode devient valide dans notre exemple précédent. Dans ce cas, la classe `Resident` est dite la classe « *upper bound* » de la classe `Customer`.

Cette construction à pourtant une limitation. Il est en effet impossible de modifier la liste dans le corps de la méthode, car comme nous ne connaissons pas le type effectif (si ce n'est que c'est un type inconnu dont le supertype est `Resident`). Toute tentative d'ajout d'élément nous confronterait à un problème de compatibilité de type entre le type effectif et le type de l'objet à ajouter.

Tiger encore une fois propose une possibilité pour outrepasser ceci. Il s'agit des méthodes génériques.

Les méthodes génériques.

Les méthodes génériques sont des méthodes qui sont paramétrées par des types génériques. Nous allons voir comment celles-ci se mettent en œuvre.

Supposons que le service du recensement fournisse à la mairie une liste de personnes nouvellement installées dans la commune. La mairie souhaite alors mettre à jour sa liste d'habitants. La classe `NewComer` hérite de la classe `Resident`.

La mairie dispose pour cela de la méthode suivante :

```
static void addResidents(List<? extends Resident> newComers,
                         List<Resident> residents)
{
 for(Resident n: newComers)
 {
  residents.add(n); // erreur de compilation
 }
}
```

Nous avons bien pris soin de spécifier que les éléments de la liste des résidents sont de tous types héritant de `Resident`. Cependant comme nous venons de le dire, il est impossible de modifier la liste passée en paramètre. La ligne 6 provoque donc une erreur de compilation.

Les méthodes génériques arrivent ici à point nommé en déduisant le type à assigner :

```
static <T extends Resident> void addResidents(List<T> newComers,
                                              List<T> residents)
{
 for(T n: newComers)
 {
  residents.add(n); // pas d'erreur ici.
 }
}
```

Mêler code classique et code générique.

Jusqu'ici nous avons toujours remplacé du code classique par du code générique. Or, dans une application existante, il ne sera sans doute pas possible de tout refaire en utilisant les *generics*.

On peut isoler deux cas principaux :

- Nous utilisons du code générique au sein d'un code ordinaire.
- Nous ajoutons du code existant ordinaire dans du code générique.

Dans le premier cas, le compilateur va lever des avertissements sur les blocs de code qui pourraient poser problème, comme par exemple des collections classiques qui utilisent des collections génériques.

Dans le second cas, le compilateur va convertir le code générique en code ordinaire. Ce processus est appelé « *erasure and translation* ». Il supprime toutes les informations génériques et transtype les objets dans un type compatible (voire `Object` si aucun type n'est compatible).

Ressources et liens divers.

Sun - J2SE 1.5 Main page:
http://java.sun.com/j2se/1.5/

JCP - JSR 176:
http://www.jcp.org/en/jsr/detail?id=176

Sun - J2SE 1.5 in a nutshell:
http://java.sun.com/developer/technicalArticles/releases/j2se15/

Sun - Programming with the New Language Features in J2SE 1.5:
http://java.sun.com/developer/technicalArticles/releases/j2se15langfeat/

JavaWorld - Taming Tiger:
http://www.javaworld.com/javaworld/jw-04-2004/jw-0426-tiger1.html

Oswego university- Doug Lea's concurrent Api:
http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

JavaPro - Fast Track:
http://www.ftponline.com/javapro/toc.aspx

O'Reilly – Les types énumérés:
http://www.oreilly.com/catalog/javaadn/chapter/ch03.pdf

106-IBM – Concurrent collections :
http://www-106.ibm.com/developerworks/java/library/j-tiger06164.html

GJ – Generics Java:
http://homepages.inf.ed.ac.uk/wadler/gj/

Sun – Generics in the Java programming Language:
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Eyrolles – Programmation Orientée Aspect pour Java/J2EE:
ISBN : 2-212-11408-7

Developper.com – Generics
http://www.developer.com/java/other/article.php/3112301

Raible Design – Struts annotations
http://raibledesigns.com/page/rd/Weblog/20040519

OnJava - Declarative
http://www.onjava.com/pub/a/onjava/2004/04/21/declarative.html

Glossaire et acronymes.

[a] Java 2 Standard Edition
[b] Java Specification Request
[c] Java Community Process

API : Application Programming Interface. Il s'agit d'un ensemble de classes offrant des fonctionnalités directement visibles et utilisables par un programme tiers. L'API définie comment ces fonctionnalités peuvent être invoquées.

AutoBoxing : *Boxing* (voir Boxing) automatique, sans intervention explicite du développeur.
Boxing : Action visant à transformer un type en un autre type représentant la même chose ou grandeur.

Design Pattern / Pattern : Les Design Pattern sont souvent désignés comme des solutions éprouvées à des problèmes récurrents. En informatique il existe plusieurs dizaines de ces modèles permettant de réaliser des logiciels fiables et robustes. Vous pouvez trouver beaucoup de littérature à ce sujet sur www.developpez.com.

Framework : Un framework est un cadre logiciel servant de base à des applications. Par exemple, le framework Struts peut servir de base au développement d'application Web. Le framework sert surtout à éviter de redéfinir sans cesse les mêmes choses dans chaque application.

J2EE : Java 2 Entreprise Edition.

J2SE : Java 2 Standard Edition.

JCP : Java Community Process.

JDK : Java Developper Kit.

JSR : Java Specification Request.

POA : Programmation Orientée Aspect. La POA est un nouveau paradigme de programmation qui étend les possibilités de la POO.

POO : Programmation Orientée Objet.

Unboxing : Action visant à défaire une action de boxing.

Wrapper : Aussi appelé enrobeur, le wrapper est généralement un type qui encapsule une autre type en lui ajoutant des fonctionnalités.

XML : eXtensible Markup Language. Il s'agit d'un langage de balises non statique comme peut l'être le HTML.