11. Java SE 7, le projet Coin

Niveau : Elémentaire

e projet Coin propose des améliorations au langage Java pour augmenter la productivité des développeurs et simplifier certaines tâches de programmation courantes. Il s'agit d'une part de réduire la quantité de code nécessaire et d'autre part de rendre ce code plus facile à lire en utilisant une syntaxe plus claire.

Le projet Coin a été développé par le JCP sous la JSR 334: "Small Enhancements to the Java Programming Language".

Le projet Coin a été développé et implémenté dans le cadre de l'open JDK, tout d'abord sous la forme d'un appel à contribution d'idées de la part de la communauté. Toutes les idées retenues ne sont pas proposées dans Java SE 7, certaines seront implémentées dans Java SE 8. Une première partie du projet Coin est incluse dans Java SE 7 l'autre partie sera intégrée dans Java SE 8.

Les fonctionnalités du projet Coin incluses dans Java 7 peuvent être regroupées en trois parties :

1) Simplifier l'utilisation des generics

- l'opérateur diamant
- la suppression possible des avertissements lors de l'utilisation des varargs

2) Simplifier la gestion des erreurs

- la prise en compte de plusieurs exceptions dans la clause catch
- l'opérateur try-with-resources

3) Simplifier l'écriture du code

- l'utilisation des d'objets de type String dans l'opérateur switch
- faciliter la lecture des valeurs littérales

Le but du projet Coin est de proposer quelques améliorations au niveau du langage Java :

- Les entiers exprimés en binaire (Binary Literals) : les types entiers (byte, short, int, et long) peuvent être exprimés dans le système binaire en utilisant le préfixe 0b ou 0B

Exemple (code Java 7) :

```
1. int valeurInt = 0b1000;
```

- Utilisation des underscores dans les entiers littéraux : il est possible d'utiliser le caractère tiret bas (underscore) entre les chiffres qui composent un entier littéral. Ceci permet de faire des groupes de chiffres pour par exemple séparer les milliers, les millions, les milliards, ... afin d'améliorer la lisibilité du code

Exemple (code Java 7) :

```
1. int maValeur = 123_1456_789;
```

- Utilisation d'objets de type Strings dans l'instruction Switch : il est possible d'utiliser un objet de type String dans l'expression d'une instruction Switch
- L'opérateur diamant (diamond operator) : lors de l'instanciation d'un type utilisant les generics, il n'est plus obligatoire de repréciser le type generics au niveau du constructeur mais de simplement utiliser l'opérateur diamant « <> » tant que le compilateur est en mesure de déterminer le type generic
- Le mot clé try-with-resources : il permet de déclarer une ou plusieurs ressources. Une ressource est un objet qui a besoin d'être fermé lorsqu'il n'est plus utilisé. Le mot clé try-with-resources garantit que chaque ressource sera fermée lorsqu'elle n'est plus utilisée. Une ressource et un objet qui implémente l'interface java.lang.AutoCloseable. Plusieurs classes du JDK implémentent l'interface AutoClosable : java.io.InputStream, OutputStream, Reader, Writer, java.sql.Connection, Statement, et ResultSet. Il est donc possible d'utiliser une instance de ces interfaces avec le mot clé try-with-resources.
- Il est possible de capturer plusieurs exceptions dans une même clause catch.
- Le compilateur de Java 7 détermine plus précisément les exceptions qui peuvent être levées dans le bloc try. Il est capable de vérifier la clause throws lorsque ces exceptions sont propagées dans un catch et ce, indépendamment du type utilisé pour les capturer.
- Il est possible de demander au compilateur de ne plus émettre de warnings lors de l'utilisation de varargs generic en utilisant l'option -Xlint:varargs ou les annotations @SafeVarags ou @SuppressWarnings({"unchecked", "varargs"})

Ce chapitre contient plusieurs sections :

11.1. Les entiers exprimés en binaire (Binary Literals)

Avec Java 7, la valeur des types entiers (byte, short, int, et long) peut être exprimée dans le système binaire en utilisant le préfixe 0b ou 0B

Exemple (code Java 7) :

```
01. public static void testEntierBinaire() {
02.   byte valeurByte = (byte) 0b00010001;
03.   System.out.println("valeurByte = " + valeurByte);
04.   valeurByte = (byte) 0B10001;
05.   System.out.println("valeurByte = " + valeurByte);
06.   valeurByte = (byte) 0B11101111;
07.   System.out.println("valeurByte = " + valeurByte);
08.   short valeurShort = (short) 0b1001110111101;
09.   System.out.println("valeurShort = " + valeurShort);
10.   int valeurInt = 0b1000;
11.   System.out.println("valeurInt = " + valeurInt);
12.   valeurInt = 0b10011101000101101001101010000101;
13.   System.out.println("valeurInt = " + valeurInt);
14.   long valeurLong =
15.       0b0100001010001011010000101000101101000010100010110100000101000101L;
16.   System.out.println("valeurLong = " + valeurLong);
17. }
```

11.2. Utilisation des underscores dans les entiers littéraux

Il n'est pas facile de lire un nombre qui compte de nombreux chiffres : dès que le nombre de chiffres dépasse 9 ou 10 la lecture n'est plus triviale, ce qui peut engendrer des erreurs.

A partir de Java 7, il est possible d'utiliser un ou plusieurs caractères tiret bas (underscore) entre les chiffres qui composent un entier littéral. Ceci permet de faire des groupes de chiffres pour par exemple séparer les milliers, les millions, les milliards, ... afin d'améliorer la lisibilité du code.

Exemple (code Java 7) :

```
01. int maValeur = 123_1456_789;
02. maValeur = 4_3;
03. maValeur = 4___3;
04. maValeur = 0x4_3;
05. maValeur = 0_43;
06. maValeur = 04_3;
07. maValeur = 0b1001110_10001011_01001101_01000101;
08. long creditCardNumber = 1234_5678_9012_3456L;
09. long numeroSecuriteSociale = 1_75_02_31_235_897L;
10. long octetsDebutFichierClass = 0xCAFE_BABE;
11. long maxLong = 0x7fff_ffff_ffff_ffffL;
12. float pi = 3.141_593f;
```

Un nombre quelconque de caractères de soulignement (underscore) peut apparaître n'importe où entre les chiffres d'un littéral numérique. Le caractère underscore doit être placé uniquement entre deux chiffres. Il n'est donc pas possible de l'utiliser :

- Au début ou à la fin d'un nombre
- Avant ou après le point de séparation de la partie décimale d'un nombre flottant
- Avant les suffixes F ou L

Exemple (code Java 7) :

```
01. // toutes ces expressions provoquent une erreur de compilation
02. int maValeur = _43;
03. int maValeur = 43_;
04. int x5 = 0_x43;
05. int x6 = 0x_43;
06. int x8 = 0x43_;
07. float pi1 = 3_.141593F;
08. float pi2 = 3._141593F;
09. long numeroSecuriteSociale = 1750231235897_L;
```

Le caractère underscore ne modifie pas la valeur mais facilite simplement sa lecture.

11.3. Utilisation des strings dans l'instruction switch

Avant Java 7, l'instruction switch ne peut être utilisée qu'avec des types primitifs ou des énumérations. L'utilisation d'une chaîne de caractères dans une instruction switch provoquait une erreur à la compilation "Incompatible Types. Require int instead of String".

Pour limiter l'utilisation d'instructions if/else utilisées avec des chaînes de caractères, il est possible d'utiliser l'instruction switch avec des énumérations.

A partir de Java SE 7, il est possible d'utiliser un objet de type String dans l'expression fournie à l'instruction Switch.

Exemple (code Java 7) :

```
01. public static Boolean getReponse(String reponse) {
02.   Boolean resultat = null;
03.   switch(reponse) {
04.     case "oui" :
05.     case "Oui" :
06.       resultat = true;
07.       break;
08.     case "non" :
09.     case "Non" :
10.       resultat = false;
11.       break;
12.     default:
13.       resultat = null;
14.       break;
15.   }
16.   return resultat;
17. }
```

L'instruction switch compare la valeur de la chaîne de caractères avec la valeur fournie à chaque instruction case comme si elle utilisait la méthode String.equals(). Dans les faits, le compilateur utilise la méthode String.hashCode() pour faire la comparaison. Le compilateur va ainsi générer le code qui soit plus optimisé que le code équivalent avec des instructions if/else.

Important : il est nécessaire de vérifier que la chaîne de caractères évaluée par l'instruction switch ne soit pas null sinon une exception de type NullPointerException est levée.

Le test réalisé par l'instruction switch est sensible à la casse : il faut donc en tenir compte si un test ne l'est pas.

Exemple (code Java 7) :

```
01.   public static Boolean getReponse(String reponse) {
02.   Boolean resultat = null;
03.
04.   switch (reponse.toLowerCase()) {
05.     case "oui":
06.       resultat = true;
07.       break;
08.     case "non":
09.       resultat = false;
10.       break;
11.     default:
12.       resultat = null;
13.       break;
14.   }
15.   return resultat;
16. }
```

L'instruction switch peut toujours être remplacée avantageusement par une utilisation du polymorphisme.

11.4. L'opérateur diamant

Avant Java 7, il était obligatoire, lors de l'instanciation d'une classe utilisant les generics, de préciser le type generic dans la déclaration de la variable et dans l'invocation du constructeur.

Exemple (code Java 5.0) :

```
1. Map<Integer, String> maMap = new HashMap<Integer, String>();
```

Avec Java 7, il est possible de remplacer les types generics utilisés lors de l'invocation du constructeur pour créer une instance par le simple opérateur <>, dit opérateur diamant (diamond operator), qui permet donc de réaliser une inférence de type.

Ceci est possible tant que le constructeur peut déterminer les arguments utilisés dans la déclaration du type generic à créer.

Exemple (code Java 7) :

```
1. Map<Integer, String> maMap = new HashMap<>();
```

L'utilisation de l'opérateur diamant n'est pas obligatoire. Si l'opérateur diamant est omis, le compilateur génère un warning de type unchecked conversion.

Exemple (code Java 7) :

```
1. Map<Integer, String> maMap = new HashMap();
2. // unchecked conversion warning
```

La déclaration et l'instanciation d'un type qui utilise les generics peuvent être verbeux. L'opérateur diamant est très pratique lorsque les types generics utilisés sont complexes : le code est moins verbeux et donc plus simple à lire

Exemple (code Java 5.0) :

```
1. Map<Integer, Map<String, List<String>>> maCollection = new HashMap<Integer,
2.   Map<String, List<String>>>();
```

L'inconvénient dans le code Java 5 ci-dessus est que le type generic utilisé doit être utilisé dans la déclaration et dans la création de l'instance : cette utilisation est redondante. Avec Java 7 et l'utilisation de l'opérateur diamant, le compilateur va automatiquement reporter le type utilisé dans la déclaration.

Exemple (code Java 7) :

```
1. Map<Integer,Map<String, List<String>>> maCollection = new HashMap<>();
```

Cette inférence de type réalisée avec l'opérateur diamant n'est utilisable qu'avec un constructeur.

L'utilisation de l'opérateur est conditionnée par le fait que le compilateur puisse déterminer le type. Dans le cas contraire, une erreur de compilation est émise.

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test;
02.
03. import java.util.ArrayList;
04. import java.util.List;
05.
06. public class TestOperateurDiamant {
07.   public static void main(String[] args) {
08.     List<String> liste = new ArrayList<>();
09.     liste.add("element1");
10.     liste.addAll(new ArrayList<>());
11.   }
12. }
```

Résultat :

```
C:\eclipse helios\workspace\TestJava\src>javac com\jmdoudoux\test\TestOperateurD
iamant.java
com\jmdoudoux\test\TestOperateurDiamant.java:11: error: no suitable method found
 for addAll(ArrayList<Object>)
    liste.addAll(new ArrayList<>());
         ^
    method List.addAll(int,Collection<? extends String>) is not applicable
      (actual and formal argument lists differ in length)
    method List.addAll(Collection<? extends String>) is not applicable
      (actual argument ArrayList<Object> cannot be converted to Collection<? extends
String> by method invocation conversion)
1 error
```

La compilation de l'exemple ci-dessus échoue puisque la méthode addAll() attend en paramètre un objet de type Collection<String>.

L'exemple suivant compile car le compilateur peut explicitement déterminer le type à utiliser avec l'opérateur diamant.

Exemple (code Java 7) :

```
package com.jmdoudoux.test;

import java.util.ArrayList;
import java.util.List;

public class TestOperateurDiamant {
  public static void main(String[] args) {
    List<String> liste = new ArrayList<>();
    liste.add("element1");

    List<? extends String> liste2 = new ArrayList<>();
    liste2.add("element2");
    liste.addAll(liste2);
  }
}
```

L'opérateur diamant peut aussi être utilisé lors de la création d'une nouvelle instance dans une instruction return : le compilateur peut déterminer le type à utiliser par rapport à la valeur de retour de la méthode.

Exemple (code Java 7) :

```
public Map<String, List<String>> getParametres(String contenu) {
  if (contenu == null) {
    return new HashMap<>();
  }
  // ...
}
```

11.5. L'instruction try-with-resources

Des ressources comme des fichiers, des flux, des connections, ... doivent être fermées explicitement par le développeur pour libérer les ressources sous jacentes qu'elles utilisent. Généralement cela est fait en utilisant un bloc try / finally pour garantir leur fermeture dans la quasi-totalité des cas.

De plus, la nécessité de fermer explicitement la ressource implique un risque potentiel d'oubli de fermeture qui entraine généralement une fuite de ressources.

Avec Java 7, l'instruction try avec ressource permet de définir une ressource qui sera automatiquement fermée à la fin de l'exécution du bloc de code de l'instruction.

Ce mécanisme est aussi désigné par l'acronyme ARM (Automatic Ressource Management).

Avant Java 7, il était nécessaire d'utiliser un bloc finally pour s'assurer que le flux sera fermé même si une exception est levée durant les traitements. Ce type de traitement possède plusieurs inconvénients :

- La ressource utilisée doit être déclarée en dehors du bloc try pour pouvoir être utilisée dans le bloc finally
- L'invocation de la méthode close() sur la ressource peut aussi lever une exception de type IOException qu'il faut gérer en propageant cette exception ou en incluant l'invocation de cette méthode dans un bloc try/catch

Exemple :

```
package com.jmdoudoux.test.java7;

```

```
03. import java.io.BufferedReader;
04. import java.io.File;
05. import java.io.FileReader;
06. import java.io.IOException;
07.
08. public class TestCloseRessource {
09.
10.   public static void main(String[] args) throws IOException {
11.     System.out.println(lireContenu(new File("monfichier.txt")));
12.   }
13.
14.   static public String lireContenu(File fichier) {
15.     StringBuilder contenu = new StringBuilder();
16.     try {
17.       BufferedReader input = null;
18.       try {
19.         input = new BufferedReader(new FileReader(fichier));
20.         String ligne = null;
21.         while ((ligne = input.readLine()) != null) {
22.           contenu.append(ligne);
23.           contenu.append("\n");
24.         }
25.       } finally {
26.         if (input != null) {
27.           input.close();
28.         }
29.       }
30.     } catch (IOException ex) {
31.       ex.printStackTrace();
32.     }
33.     return contenu.toString();
34.   }
35. }
```

L'inconvénient de cette solution est que l'exception propagée serait celle de la méthode close() si elle lève une exception qui pourrait alors masquer une exception levée dans le bloc try. Il est possible de capturer l'exception de la méthode close().

Exemple :

```
01. package com.jmdoudoux.test.java7;
02.
03. import java.io.BufferedReader;
04. import java.io.File;
05. import java.io.FileReader;
06. import java.io.IOException;
07.
08. public class TestCloseRessource {
09.
10.   public static void main(String[] args) throws IOException {
11.     System.out.println(lireContenu(new File("monfichier.txt")));
12.   }
13.
14.   static public String lireContenu(File fichier) {
15.     StringBuilder contenu = new StringBuilder();
16.     try {
17.       BufferedReader input = null;
18.       try {
19.         input = new BufferedReader(new FileReader(fichier));
20.         String ligne = null;
21.         while ((ligne = input.readLine()) != null) {
22.           contenu.append(ligne);
23.           contenu.append("\n");
24.         }
25.       } finally {
26.         if (input != null) {
27.
28.           try {
29.               input.close();
30.             } catch (IOException e) {
31.               e.printStackTrace();
32.             }
33.           }
34.       }
35.     } catch (IOException ex) {
36.       ex.printStackTrace();
37.     }
38.     return contenu.toString();
39.   }
40. }
```

L'inconvénient de cette solution est que l'exception qui peut être levée par la méthode close() n'est pas propagée. De plus la quantité de code produite devient plus importante.

Avec Java 7, le mot clé try peut être utilisé pour déclarer une ou plusieurs ressources.

Une ressource est un objet qui doit être fermé lorsque l'on a plus besoin de lui : généralement cette ressource encapsule ou utilise des ressources du système : fichiers, flux, connexions vers des serveurs, ...

Une nouvelle interface a été définie pour indiquer qu'une ressource peut être fermée automatiquement : java.lang.AutoCloseable.

Tous les objets qui implémentent l'interface java.lang.AutoCloseable peuvent être utilisés dans une instruction de type try-with-resources. L'instruction try avec des ressources garantit que chaque ressource déclarée sera fermée à la fin de l'exécution de son bloc de traitement.

L'interface java.lang.Autocloseable possède une unique méthode close() qui sera invoquée pour fermer automatiquement la ressource encapsulée par l'implémentation de l'interface.

L'interface java.io.Closable introduite par Java 5 hérite de l'interface AutoCloseable : ainsi toutes les classes qui implémentent l'interface Closable peuvent être utilisées comme ressource dans une instruction try-with-resource.

La méthode close() de l'interface Closeable lève une exception de type IOException alors que la méthode close() de l'interface AutoCloseable lève une exception de type Exception. Cela permet aux interfaces filles de AutoCloseable de redéfinir la méthode close() pour qu'elles puissent lever une exception plus spécifique ou aucune exception.

Contrairement à la méthode close() de l'interface Closeable, une implémentation de la méthode close() de l'interface AutoCloseable n'est pas supposée être idempotente : son invocation une seconde fois peut avoir des effets de bords.

Une implémentation de la méthode close() de l'interface AutoCloseable() devrait déclarer une exception plus précise que simplement Exception ou ne pas déclarer d'exception du tout si l'opération de fermeture ne peut échouer.

Il faut garder à l'esprit que l'exception levée sera masquée par l'instruction try-with-resource : l'implémentation de la méthode close() doit faire attention aux exceptions qu'elle peut lever (par exemple, comme le précise la Javadoc, elle ne doit pas lever une exception de type InterruptedException)

L'instruction try avec des ressources utilise le mot clé try avec une ou plusieurs ressources définies dans sa portée, chacune séparée par un point-virgule.

Exemple (code Java 7) :

```
01. try {
02.   try (BufferedReader bufferedReader = new BufferedReader(new
03.     FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt"))) {
04.     String ligne=null;
05.     while ((ligne = bufferedReader.readLine()) != null) {
06.       System.out.println(ligne);
07.     }
08.   }
09. } catch (IOException ioe) {
10.   ioe.printStackTrace();
11. }
```

Dans l'exemple ci-dessus, la ressource de type BufferedReader sera fermée proprement à la fin normale ou anormale des traitements.

Les ressources sont implicitement final : il n'est donc pas possible de leur affecter une nouvelle instance dans le bloc de l'instruction try.

Une instruction try avec ressources peut avoir des clauses catch et finally comme une instruction try classique. Avec l'instruction try avec ressources, les clauses catch et finally sont exécutées après que la ou les ressources ont été fermées.

Exemple (code Java 7) :

```
01. try (BufferedReader bufferedReader = new
02.   BufferedReader(new FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt"))) {
03.   String ligne=null;
04.   while ((ligne = bufferedReader.readLine()) != null) {
05.     System.out.println(ligne);
06.   }
07. } catch (IOException ioe) {
08.   ioe.printStackTrace();
09. }
```

Il est possible de déclarer plusieurs ressources dans une même instruction try avec ressources, chacune séparée par un caractère point-virgule. Dans ce cas, la méthode close() des ressources déclarées est invoquée dans l'ordre inverse de leur déclaration.

L'instruction try-with-resource présente un petit inconvénient : il est obligatoire de définir la variable qui encapsule la ressource entre les parenthèses qui suivent l'instruction try. Il n'est par exemple pas possible de fournir en paramètre de l'instruction try une instance déjà créé.

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test.java7;
02.
03. import java.io.FileNotFoundException;
04. import java.io.FileReader;
05. import java.io.IOException;
06. import java.io.InputStream;
07. import java.io.Reader;
08.
09. public class TestTryWithRessources {
10.
11.   public static void main(String[] args) {
12.     FileReader fr;
13.     try {
```

```
      fr = new FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
      afficherFichier(fr);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  public static void afficherFichier(Reader flux) throws IOException {
    try (flux) {
      int donnee;
      while ((donnee = flux.read()) >= 0) {
        System.out.print((char) donnee);
      }
    }
  }
}
```

Le compilateur génère une erreur lors de la compilation de ce code.

Résultat :

```
C:\Users\jm\java\JavaApplication1\src\com\jmdoudoux\test\java7>javac
TestTryWithRessources.java
TestTryWithRessources.java:22: error:
<identifier> expected
   try (flux) {
              ^
TestTryWithRessources.java:22: error: ')' expected
   try (flux) {
               ^
TestTryWithRessources.java:22: error: '{' expected
   try (flux) {
                 ^
TestTryWithRessources.java:23: error: not a statement
     int donnee;
         ^
4 errors
```

L'exemple ci-dessus génère une erreur à la compilation puisqu'aucune variable n'est définie entre les parenthèses de l'instruction try.

Pour pallier à ce petit inconvénient, il est possible de définir une variable et de l'initialiser avec l'instance existante.

Exemple (code Java 7) :

```
package com.jmdoudoux.test.java7;

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class TestTryWithRessources {

  public static void main(String[] args) {
    FileReader fr;
    try {
      fr = new FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
      afficherFichier(fr);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  public static void afficherFichier(Reader flux) throws IOException {
    try (Reader closeableReader = flux) {
      int donnee;
      while ((donnee = flux.read()) >= 0) {
        System.out.print((char) donnee);
      }
    }
  }
}
```

Dans l'exemple ci-dessus, comme la variable définie et celle existante pointent sur la même référence, les deux variables peuvent être utilisées indifféremment. L'instruction try-with-resource se charge de fermer automatiquement le flux.

Attention, seules les ressources déclarées dans l'instruction try seront fermées automatiquement. Si une ressource est explicitement instanciée dans le bloc try, la gestion de la fermeture et de l'exception qu'elle peut lever doit être gérée par le développeur.

Une exception peut être levée dans le bloc de l'instruction try mais aussi durant l'invocation de la méthode close() de la ou des ressources déclarées. La méthode close() pouvant lever une exception, celle-ci pourrait masquer une éventuelle exception levée dans le bloc de code de l'instruction try.

Il est obligatoire de gérer l'exception pouvant être levée par la méthode close() de la ressource soit en la capturant pour la traiter soit en propageant cette exception pour laisser le soin de son traitement à la méthode appelante.

Exemple (code Java 7) :

```
1. package com.jmdoudoux.test;
2.
3. public class MaRessource implements AutoCloseable {
4.   @Override
5.   public void close() throws MonException {
6.     throw new MonException("Erreur durant la fermeture");
7.   }
8. }
```

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test;
02.
03. public class TestMaRessource {
04.   public static void main(String[] args) {
05.     try (MaRessource res = new MaRessource()) {
06.       // utilisation da la ressource
07.     }
08.   }
09. }
```

Résultat :

```
01. C:\eclipse helios\workspace\TestJava\src>javac
02. com\jmdoudoux\test\TestMaRessource.java
03. com\jmdoudoux\test\TestMaRessource.java:6: error: unreported
04. exception MonException; must be caught or declared to be thrown
05.     try (MaRessource
06. res = new MaRessource()) {
07.                       ^
08.   exception thrown from implicit call to close() on resource variable 'res'
09. 1 error
```

Cette exemple ne se compile pas car l'exception pouvant être levée lors de l'invocation de la méthode close() n'est pas gérée.

Les exemples suivants utilisent deux exceptions personnalisées.

Exemple :

```
01. package com.jmdoudoux.test.java7;
02.
03. public class MonException1 extends Exception{
04.     public MonException1(String message){
05.         super(message);
06.     }
07. }
08.
09. package com.jmdoudoux.test.java7;
10.
11. public class MonException2 extends Exception{
12.     public MonException2(String message){
13.         super(message);
14.     }
15. }
```

Une ressource générique est définie : elle possède une méthode utiliser() et une redéfinition de la méthode close() car elle implémente l'interface AutoCloseable. Durant leur exécution, ces deux méthodes lèvent une exception.

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test.java7;
02.
03. public class MaRessource implements AutoCloseable {
04.   private String nom;
05.
06.   public MaRessource(String nom) {
07.     this.nom = nom;
08.   }
09.
10.   public String getNom() {return nom;}
11.
12.   public void utiliser() throws MonException1{
13.     System.out.println("Utilisation de la ressource "+nom);
14.     throw new MonException1("Erreur durant l'utilisation de la ressource "+nom);
15.   }
16.
17.   @Override
18.   public void close() throws MonException2{
19.     System.out.println("Fermeture de la ressource"+nom);
```

```
20.     throw new MonException2("Erreur durant la fermeture de la ressource "+nom);
21.   }
22. }
```

La ressource peut être utilisée dans du code compatible avec la version 6 de Java.

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test.java7;
02.
03. public class TestRessourceJ6 {
04.
05.   public static void  main(String[] args) {
06.     MaRessource res = null;
07.
08.     try {
09.       res = new MaRessource("Ressource1");
10.       res.utiliser();
11.     } catch (Exception e) {
12.        e.printStackTrace();
13.     } finally{
14.       try {
15.         res.close();
16.       } catch (Exception e) {
17.         e.printStackTrace();
18.       }
19.     }
20.   }
21. }
```

Résultat :

```
01. Utilisation de la ressource Ressource1
02. Fermeture de la ressourceRessource1
03. com.jmdoudoux.test.java7.MonException1:
04. Erreur durant l'utilisation de la ressource Ressource1
05.       at com.jmdoudoux.test.java7.MaRessource.utiliser(MaRessource.java:14)
06.       at com.jmdoudoux.test.java7.TestRessourceJ6.main(TestRessourceJ6.java:10)
07. com.jmdoudoux.test.java7.MonException2:
08. Erreur durant la fermeture de la ressource Ressource1
09.       at com.jmdoudoux.test.java7.MaRessource.close(MaRessource.java:20)
10.       at com.jmdoudoux.test.java7.TestRessourceJ6.main(TestRessourceJ6.java:15)
```

L'utilisation de la ressource avec l'instruction try-with-resource de Java 7 simplifie le code.

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test.java7;
02.
03. public class TestRessourceJ7 {
04.   public static void main(String[] args) {
05.     try(MaRessource res = new MaRessource("Ressource1")){
06.       res.utiliser();
07.     } catch(Exception e){
08.       e.printStackTrace();
09.     }
10.   }
11. }
```

Résultat :

```
01. Utilisation de la ressource Ressource1
02. com.jmdoudoux.test.java7.MonException1:
03. Erreur durant l'utilisation de la ressource Ressource1
04. Fermeture de la ressourceRessource1
05.       at com.jmdoudoux.test.java7.MaRessource.utiliser(MaRessource.java:14)
06.       at com.jmdoudoux.test.java7.TestRessourceJ7.main(TestRessourceJ7.java:6)
07.       Suppressed:
08. com.jmdoudoux.test.java7.MonException2:
09. Erreur durant la fermeture de la ressource Ressource1
10.             at com.jmdoudoux.test.java7.MaRessource.close(MaRessource.java:20)
11.             at com.jmdoudoux.test.java7.TestRessourceJ7.main(TestRessourceJ7.java:7)
```

Le résultat est aussi légèrement différent : c'est l'exception levée lors de l'utilisation de la ressource qui est propagée et non l'exception levée lors de la fermeture de la ressource.

Si une exception est levée dans le bloc try et lors de la fermeture de la ressource, c'est l'exception du bloc try qui est propagée et l'exception levée lors de la fermeture est masquée.

Pour obtenir l'exception masquée, il est possible d'invoquer la méthode getSuppressed() de la classe Throwable sur l'instance de l'exception qui est propagée.

L'ARM fonctionne aussi si plusieurs ressources sont utilisées dans plusieurs instructions try-with-resources imbriquées.

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test.java7;
02.
03. public class TestRessourcesJ7 {
04.   public static void main(String[] args) {
05.     try (MaRessource res1 = new MaRessource("Ressource1");
06.          MaRessource res2 = new MaRessource("Ressource2")) {
07.       try (MaRessource res3 = new MaRessource("Ressource3")) {
08.         res3.utiliser();
09.       } catch (Exception e) {
10.         e.printStackTrace();
11.       }
12.     } catch (Exception e) {
13.       e.printStackTrace();
14.     }
15.   }
16. }
```

Résultat :

```
01. Utilisation de la ressource Ressource3
02. Fermeture de la ressource Ressource3
03. Fermeture de la ressource Ressource2
04. Fermeture de la ressource Ressource1
05. com.jmdoudoux.test.java7.MonException1:
06. Erreur durant l'utilisation de la ressource Ressource3
07.       at com.jmdoudoux.test.java7.MaRessource.utiliser(MaRessource.java:14)
08.       at com.jmdoudoux.test.java7.TestRessourcesJ7.main(TestRessourcesJ7.java:8)
09.       Suppressed:
10. com.jmdoudoux.test.java7.MonException2: Erreur durant la fermeture de la
11. ressource Ressource3
12.             at com.jmdoudoux.test.java7.MaRessource.close(MaRessource.java:20)
13.             at com.jmdoudoux.test.java7.TestRessourcesJ7.main(TestRessourcesJ7.java:9)
14. com.jmdoudoux.test.java7.MonException2:
15. Erreur durant la fermeture de la ressource Ressource2
16.       at com.jmdoudoux.test.java7.MaRessource.close(MaRessource.java:20)
17.       at com.jmdoudoux.test.java7.TestRessourcesJ7.main(TestRessourcesJ7.java:12)
18.       Suppressed: com.jmdoudoux.test.java7.MonException2:
19. Erreur durant la fermeture de la ressource Ressource1
20.             ... 2 more
```

Toutes les exceptions levées lors de la fermeture des ressources sont inhibées et peuvent être obtenues en invoquant la méthode getSuppressed().

Exemple (code Java 7) :

```
01. package com.jmdoudoux.test.java7;
02.
03. public class TestRessourcesJ7 {
04.   public static void main(String[] args) {
05.     try (MaRessource res1 = new MaRessource("Ressource1");
06.          MaRessource res2 = new MaRessource("Ressource2")) {
07.       try (MaRessource res3 = new MaRessource("Ressource3")) {
08.         res3.utiliser();
09.       }
10.     } catch (Exception e) {
11.       System.out.println("Exception : " +
12.         e.getClass().getSimpleName() + " : " + e.getMessage());
13.       if (e.getSuppressed() != null) {
14.         for (Throwable t : e.getSuppressed()) {
15.           System.out.println(t.getClass().getSimpleName() +
16.             " : " + t.getMessage());
17.         }
18.       }
19.     }
20.   }
21. }
```

Résultat :

```
01. Utilisation de la ressource Ressource3
02. Fermeture de la ressource Ressource3
03. Fermeture de la ressource Ressource2
04. Fermeture de la ressource Ressource1
05. Exception : MonException1 : Erreur durant l'utilisation de la ressource
06. Ressource3
07. MonException2 : Erreur durant la fermeture de la ressource Ressource3
08. MonException2 : Erreur durant la fermeture de la ressource Ressource2
09. MonException2 : Erreur durant la fermeture de la ressource Ressource1
```

La méthode getSuppressed() renvoie un tableau d'instances de Throwable qui contient les exceptions capturées lors de la fermeture des ressources et non propagées.

La classe Throwable est aussi enrichie d'un nouveau constructeur qui permet de prendre en compte ou non des exceptions supprimées. Si le booléen enableSuppression est à false, alors la méthode getSuppressed() renvoie un tableau vide et l'invocation de la méthode addSuppressed() n'aura aucun effet.

11.6. Des types plus précis lorsqu'une exception est relevée dans une clause catch

Il est possible de repropager une exception qui a été gérée par une instruction catch en utilisant le mot clé throw.

Avant Java 7, il n'était pas possible de relever une exception qui soit un super type de l'exception capturée dans une clause catch : dans ce cas, le compilateur émettait une erreur "unreported exception Exception; must be caught or declared to be thrown".

Dans l'exemple ci-dessous, l'exception MonExceptionFille hérite de l'exception MonExceptionMere.

Exemple :

```
1. public void maMethode() throws MonExceptionMere { <br>
2.   try {
3.     // traitement
4.     throw new MonExceptionFille();
5.   } catch(MonExceptionMere e) {
6.     throw e;
7.   }
8. }
```

Java 7 propose une analyse plus fine de la situation et permet de déclarer la levée d'une exception de type MonExceptionFille même si l'exception gérée et relevée est de type MonExceptionMere.

Exemple (code Java 7) :

```
1. public void maMethode() throws MonExceptionFille {
2.   try {
3.     // traitement
4.     throw new MonExceptionFille();
5.   } catch (MonExceptionMere e) {
6.     throw e;
7.   }
8. }
```

Avant Java 7, cette portion de code aurait provoqué une erreur de compilation « unreported exception MonExceptionMere ». Ceci s'applique aussi pour plusieurs exceptions.

Exemple :

```
01. public class MaClasse {
02.   public void maMethode(boolean valeur) throws MonExceptionA,
03.       MonExceptionB {
04.     try {
05.       if (valeur) {
06.         throw new MonExceptionA();
07.       } else {
08.         throw new MonExceptionB();
09.       }
10.     } catch (Exception e) {
11.       throw e;
12.     }
13.   }
14.
15.   static class MonExceptionA extends Exception { }
16.   static class MonExceptionB extends Exception { }
17. }
```

Résultat :

```
1. C:\eclipse helios\workspace\TestJava\src>javac MaClasse.java
2. MaClasse.java:11:
3. unreported exception java.lang.Exception; must be caught or declared to be thrown
4.       throw e;
```

```
      ^
1 error
```

Le compilateur vérifie si le type d'une exception levée dans un bloc catch correspond à un des types d'exceptions déclaré dans la clause throws de la méthode. Si le type de l'exception capturée par la clause catch est Exception alors la clause throws ne peut pas être d'un de ses sous-types.

Exemple :

```
01. public class MaClasse {
02.   public void maMethode(boolean valeur) throws Exception {
03.     try {
04.       if (valeur) {
05.         throw new MonExceptionA();
06.       } else {
07.         throw new MonExceptionB();
08.       }
09.     } catch (Exception e) {
10.       throw e;
11.     }
12.   }
13.
14.   static class MonExceptionA extends Exception { }
15.   static class MonExceptionB extends Exception { }
16. }
```

Pour déclarer dans la clause throws les exceptions précises, il faut les capturer individuellement dans des clauses catch dédiées.

Exemple :

```
01. public class MaClasse {
02.   public void maMethode(boolean valeur) throws MonExceptionA,
03.       MonExceptionB {
04.     try {
05.       if (valeur) {
06.         throw new MonExceptionA();
07.       } else {
08.         throw new MonExceptionB();
09.       }
10.     } catch (MonExceptionA e) {
11.       throw e;
12.     } catch (MonExceptionB e) {
13.       throw e;
14.     }
15.   }
16.
17.   static class MonExceptionA extends Exception { }
18.   static class MonExceptionB extends Exception { }
19. }
```

Le compilateur de Java 7 effectue une analyse plus précise qui lui permet de connaitre précisément les exceptions qui peuvent être relevées indépendamment du type déclaré dans la clause catch qui va les capturer. Il est ainsi possible de capturer un super type des exceptions qui seront relevées et déclarer le type précis des exceptions dans la clause throws.

Lorsqu'une clause catch déclare plusieurs types d'exceptions et relève l'exception dans son bloc de code, le compilateur vérifie :

- Que le code du bloc try peut lever les exceptions déclarées
- Qu'aucune autre clause catch ne déclare prendre en charge un des types d'exceptions
- Que l'exception relevée est du type ou un sous-type d'un des types d'exceptions déclaré dans la clause catch

Exemple (code Java 7) :

```
01. public class MaClasse {
02.
03.   public void maMethode(boolean valeur) throws MonExceptionA,
04.       MonExceptionB {
05.     try {
06.       if (valeur) {
07.         throw new MonExceptionA();
08.       } else {
09.         throw new MonExceptionB();
10.       }
11.     } catch (Exception e) {
12.       throw e;
13.     }
14.   }
15.
16.   static class MonExceptionA extends Exception { }
17.   static class MonExceptionB extends Exception { }
18. }
```

Attention cependant, il y a un cas ou la compatibilité du code antérieur n'est pas assurée avec Java 7 : ce cas concerne l'imbrication de deux try/catch quant le second bloc apparaît dans la clause catch du premier try. Le code du bloc try imbriqué lève une exception.

L'exemple ci-dessous ce compile sans problème avec Java 6 :

Exemple :

```
01. public void maMethode() throws MonExceptionMere {
02.   try {
03.   // traitement
04.   throw new MonExceptionFille();
05.   } catch (MonExceptionMere e) {
06.     try {
07.       // traitement
08.       throw e;
09.     } catch (MonExceptionFille2 mem) {
10.     }
11.   }
12. }
```

Ce même code ne se compile plus avec Java 7 car l'exception de type MonExceptionMere ne sera jamais traitée par la seconde clause catch.

Pour être compilé en Java 7, le code devra être modifié.

11.7. Multiples exceptions dans une clause catch

Java SE 7 propose une amélioration de la gestion des exceptions en permettant le traitement de plusieurs exceptions dans une même clause catch.

Il n'est pas rare d'avoir à dupliquer les mêmes lignes de code dans le bloc de code de plusieurs clauses catch().

Exemple :

```
01. try {
02.   // traitements pouvant lever les exceptions
03. } catch(ExceptionType1 e1) {
04.   // Traitement de l'exception
05. } catch(ExceptionType2 e2) {
06.   // Traitement de l'exception
07. } catch(ExceptionType3 e3) {
08.   // Traitement de l'exception
09. }
```

Avant Java 7, il était difficile d'éviter la duplication de code car chaque exception est de type différent.

Une solution utilisée pour éviter cette duplication est de catcher un super-type d'exception, généralement le type Exception. Cependant cette solution a plusieurs effets de bord, notamment le fait que le traitement s'appliquera à toutes les exceptions filles et englobera peut-être des exceptions qui auraient nécessité un traitement particulier. De plus, il ne sera pas possible de propager un autre type d'exception que celui capturé.

A partir de Java 7, la même portion de code est simplifiée : il suffit de déclarer les exceptions dans une même clause catch en les séparant par le caractère "|".

Exemple (code Java 7) :

```
1. try {
2.   // traitements pouvant lever les exceptions
3. } catch(ExceptionType1|ExceptionType2|ExceptionType3 ex) {
4.   // Traitement de l'exception
5. }
```

Il n'est plus nécessaire de définir un bloc catch pour chaque exception et de dupliquer le code du bloc si c'est le même pour tous.

La clause catch peut contenir plusieurs types d'exceptions qui provoqueront l'exécution du bloc de code associé, chaque type d'exception est séparé d'un autre en utilisant le caractère barre verticale.

Il est possible d'utiliser plusieurs blocs catch notamment si les traitements des exceptions sont différents selon leur type.

Exemple (code Java 7) :

```
1. try {
2.   // traitements pouvant lever les exceptions
3. } catch(ExceptionType1|ExceptionType2|ExceptionType3 ex) {
4.   // Traitement de l'exception
5. } catch(ExceptionType4|ExceptionType5 ex) {
6.   // Traitement de l'exception
```

```
7. }
```

Si plusieurs types d'exceptions sont déclarés dans une clause catch alors la variable qui permettra un accès à l'exception concernée est implicitement déclarée final.

Le paramètre de la clause catch étant implicitement final, il n'est pas possible de réaffecter sa valeur dans le bloc de code dans lequel il est défini.

Exemple (code Java 7) :

```
01. public class MaClasse {
02.
03.   public void rethrowException(boolean valeur) throws MonExceptionA,
04.       MonExceptionB {
05.     try {
06.       if (valeur) {
07.         throw new MonExceptionA();
08.       } else {
09.         throw new MonExceptionB();
10.       }
11.     } catch (MonExceptionA|MonExceptionB e) {
12.       e = new MonExceptionB();
13.       throw e;
14.     }
15.   }
16.
17.   static class MonExceptionA extends Exception { }
18.   static class MonExceptionB extends Exception { }
19. }
```

Résultat :

```
1. C:\eclipse helios\workspace\TestJava\src>javac MaClasse.java
2. MaClasse.java:11:
3. error: multi-catch parameter e may not be assigned
4.       e =
5. new MonExceptionB();
6.       ^
7. 1 error
```

C'est le compilateur qui prend en charge la génération du code correspondant au support multi exceptions de la clause catch sans duplication de code.

L'avantage de cette gestion de plusieurs exceptions dans une clause catch n'est pas seulement syntaxique car il réduit la quantité de code produite. Le bytecode généré par le compilateur est meilleur comparé à celui produit pour plusieurs clauses catch équivalentes.