
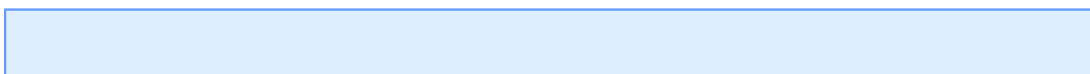


Développement sécurisé

Par Jean-Francois Lalande 

Date de publication : 2 août 2013

Dernière mise à jour : 15 août 2013



I - Introduction.....	3
I-A - 1.1 Module développement sécurisé.....	3
II - Exceptions.....	3
II-A - Rappels.....	3
II-A-1 - Exemple try.....	3
II-A-2 - Exemple catch.....	4
II-A-3 - Spécifier une exception.....	4
II-A-4 - Lever une exception.....	4
II-B - Hiérarchie des exceptions.....	5
II-B-1 - Filtrer les exceptions.....	5
II-B-2 - Chaîne d'exception.....	6
II-C - Créer une classe d'exception.....	6
III - Test unitaire.....	7
III-A - Introduction.....	7
III-A-1 - Principes.....	7
III-A-2 - Avantages.....	7
III-B - Junit.....	7
III-B-1 - Préparation de tests.....	8
III-C - Doctest.....	8
III-D - Unittest.....	8
III-D-1 - Exemple doctest sur la fonction factorielle.....	9
III-D-2 - Exécution du test.....	9
III-D-3 - Exemple de tests unitaires sur factorielle.....	9
III-D-4 - Test externalisé dans un fichier.....	10
III-D-5 - Règles d'analyse de l'output.....	10
III-E - Gestion des exceptions.....	10
III-F - Unittest.....	11
III-F-1 - Les concepts d'Unittest.....	11
IV - La sécurité dans la machine virtuelle Java.....	12
IV-A - Besoin : le sandboxing.....	12
IV-B - Introduction : les composants de sécurité la VM.....	13
IV-B-1 - Architecture de la sécurité dans Java.....	13
IV-C - Politiques de sécurité pour la VM.....	14
IV-C-1 - Permissions des politiques (1).....	14
IV-C-2 - Permissions des politiques (2).....	14
IV-C-3 - Domaines.....	15
IV-C-4 - Exemple complet de fichier de politique.....	15
IV-C-5 - Invocation du security manager.....	15
IV-D - Le security manager.....	16
IV-D-1 - Mettre en place un security manager.....	16
IV-D-2 - Méthodes du security manager : io.....	16
IV-D-3 - Méthodes du security manager : réseau.....	17
IV-D-4 - Méthodes du security manager : vm.....	17
IV-E - Le Bytecode verifier.....	18
IV-E-1 - Attaque par définition de classe.....	18
IV-E-2 - Attaque par substitution de classe.....	19
IV-E-3 - Vérifications effectuées par le bytecode verifier.....	19
IV-F - La sérialisation.....	20
IV-G - Le Keystore.....	20
V - La sécurité dans Python.....	20
V-A - Les vieilleries : rexec.....	20
V-B - mxProxy - Proxy-Access to Python Objects.....	21
V-B-1 - Types de proxy.....	21
V-B-2 - Méthodes sur un proxy : récupérer les attributs.....	21
V-B-3 - Méthodes sur un proxy : récupérer les méthodes.....	22
V-B-4 - Substitution de classe par son proxy.....	22
VI - Bibliographie.....	23

I - Introduction

I-A - 1.1 Module développement sécurisé

Objectifs du module :

- comprendre les exceptions pour améliorer la robustesse de codes Java ;
- savoir introduire les tests unitaires et comprendre leurs limites.

II - Exceptions

II-A - Rappels

- try/catch : handler qui traite l'exception.
- Une méthode peut :
 - récupérer l'exception (**catch** dans la fonction) ;
 - faire suivre l'exception (**throws** dans la déclaration).
- Types d'exceptions :
 - checked exception : toutes, sauf... ;
 - error : **Error** (e.g. **java.io.IOException**) ;
 - runtime : **RuntimeException** (e.g. **NullPointerException**).

Un bloc try/catch permet de « tenter » d'exécuter des instructions qui sont susceptibles de lever des exceptions. Le bloc **catch(X e)** récupère une exception levée si la classe de l'exception est une sous-classe de X.

Certaines exceptions sont obligatoirement protégées (« cachées ») par l'utilisateur : ce sont les checked exceptions. Cependant, d'autres exceptions peuvent ne pas être protégées par un bloc try/catch étant donné leur caractère imprévisible et pour éviter de surcharger le code.

II-A-1 - Exemple try





Exemple tiré de [JSE] : ouverture et écriture dans un fichier texte :

```
private Vector vector;
private static final int SIZE = 10;

PrintWriter out = null;

try
{
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));

    for (int i = 0; i < SIZE; i++)
    {
        out.println("Value at: "+i+" = "+vector.elementAt(i));
    }
}
catch ...
finally ...
```

-  **FileWriter** peut lever une exception de type  **IOException**.
-  **PrintWriter** peut lever une exception de type  **FileNotFoundException**.
- L'appel à **println** ne lève jamais d'exception, comme indiqué dans la Javadoc :

Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking `checkError()`.

II-A-2 - Exemple catch


Exemple tiré de [JSE] :

```
try
{
    ...
}

catch (FileNotFoundException e)
{
    System.err.println("FileNotFoundException: " + e.getMessage());
    throw new SampleException(e);
}

catch (IOException e)
{
    System.err.println("Caught IOException: IT " + e.getMessage());
}

finally
{
    System.out.println("PrintWriter not open");
}
```

-  **FileNotFoundException** hérite de  **IOException** : l'ordre des **catch** est important.
- **System.err** est la sortie standard d'erreur.
- La directive **finally** est exécutée dans tous les cas après un ou aucun **catch**.

II-A-3 - Spécifier une exception

Une méthode qui lève une exception la déclare dans son prototype à l'aide du mot-clé **throws** (à ne pas confondre avec **throw**):

- obligation pour les codes appelant de choisir entre :
 - récupérer l'exception (à l'aide d'un **try/catch**),
 - laisser remonter l'exception en la filtrant (et **filtrer les exceptions**),
 - laisser remonter l'exception en la retypant (et **chaînes d'exceptions**) ;
- plusieurs exceptions peuvent être déclarées (séparation par une virgule). Une méthode peut lever une ou plusieurs exceptions :

```
public void writeList() throws IOException
{ ... }
public void writeList() throws FileNotFoundException, EOFException
{ ... }
```

Le **main** est un cas particulier qui fait remonter l'exception jusqu'à la virtual machine qui arrête alors l'exécution :

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

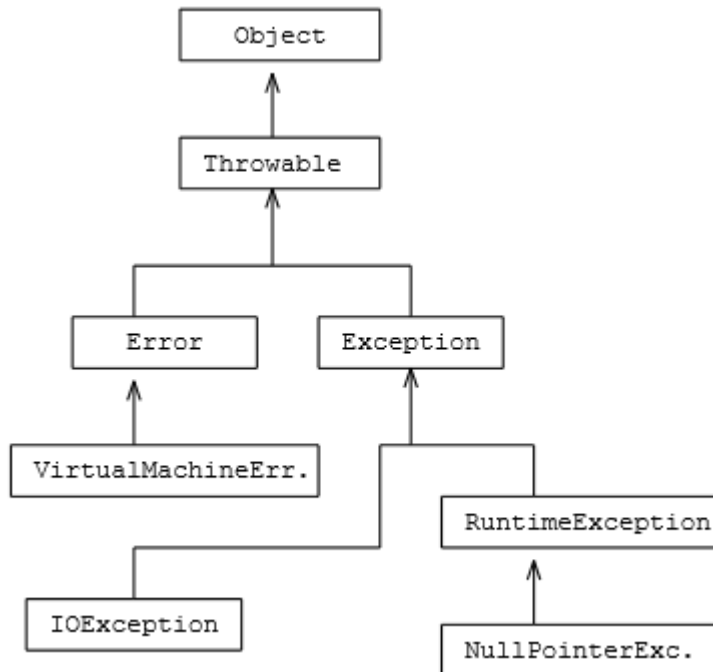
II-A-4 - Lever une exception

Lever une exception consiste à arrêter l'exécution normale du programme et à « lancer » une exception à travers un objet nouvellement instancié, héritant de **Throwable** qui va remonter la pile d'appels des fonctions.

- Lever une exception se fait à l'aide de **throw(e)**.
- L'objet créé est une instance d'une classe d'exception

```
public Object pop()
{
    if (size == 0)
    {
        throw new EmptyStackException();
    }
    ...
}
```

II-B - Hiérarchie des exceptions



II-B-1 - Filtrer les exceptions

- Un bloc **catch(X e)** capture une exception de type Y si Y est une sous-classe de X.
- On peut donc « filtrer » les exceptions par des catches successifs :
 - le premier catch filtre l'exception la plus typée ;
 - les **catch** suivants filtrent les exceptions de type plus général ;
 - c'est le principe du tamis filtrant du sable mélange à du gravier...

```
try
{
    ...
}
catch (FileNotFoundException e)
{
    System.Prr.println("Erreur d'ouverture de fichier.");
}
catch (IOException e)
{
    System.err.println("Erreur d#entrée/sortie.");
}
```

II-B-2 - Chaîne d'exception

Une chaîne d'exception permet de faire suivre une exception en la retypant.

- La récupération d'exception peut parfois ne pas être résolue : on la forward.
- Le changement de type permet de clarifier le type du problème :
 - **NullPointerException** : peu précis ;
 - **IOException** : mieux ;
 - **IOCarWriterException** : très précis

```
public class IOCarWriter
{
    public write()
    {
        try
        {
            ...
        }
        catch (IOException e)
        {
            throw new IOCarWriterException("Other IOException", e);
        }
    }
}

public class CarManager
{
    IOCarWriter writer = new IOCarWriter();
    try
    {
        writer.write();
    }
    catch (IOCarWriterException e)
    {
        System.err.println("Error when writing a car !");
    }
}
```

II-C - Créer une classe d'exception

- Choisir une classe mère en fonction du type d'exception.
- Créer la sous-classe par héritage.
- Ajouter du code informatif (optionnel).
- C'est tout.
- Exemple : [\[Rifflet\]](#)

```
class MonException extends Exception
{
    int valeur;
    MonException(int valeur)
    {
        this.valeur = valeur;
    }
    public String toString()
    {
        String str = super.toString();
        return new String(str + " --> " + valeur);
    }
}
```

Le code présenté ici n'est pas du tout obligatoire. En général, dans la plupart des classes d'exceptions, il n'y a aucun code. Cela n'empêche pas de multiplier les classes d'exception afin de permettre d'obtenir un nom d'exception pertinent dans la console et de pouvoir filtrer aisément les exceptions à l'aide des **catch**.

III - Test unitaire

III-A - Introduction

III-A-1 - Principes

- Qu'est-ce que le test ?
 - Ce n'est pas « exécuter une fois ».
 - C'est vérifier les sorties d'un programme pour une ou plusieurs exécutions dont on connaît les entrées.
- Qu'est-ce que le test unitaire ?
 - C'est tester toutes les parties d'un système compliqué (et non pas complexe).
 - C'est supposer que tester des parties permet au système de mieux fonctionner.
- Que doit-on/peut-on tester ?
 - Le comportement « normal ».
 - Le comportement aux bornes.
 - Le comportement en dehors d'une plage valide.

III-A-2 - Avantages

- Éviter les oublis : votre objectif est de passer le test unitaire.
- Réduire le nombre de bogues : chercher un bogue coûte cher.
- Éviter les régressions de code lors d'ajouts, refactoring, modifications.
- Faciliter le travail en équipe : le test d'autrui est testé et de confiance.

III-B - Junit

Junit est un framework de test unitaire. Il permet de réaliser proprement et d'automatiser l'exécution de tests unitaires pour :

- vérifier la conformité de l'implémentation par rapport à la spécification unitaire ;
- tester la non-régression du code au cours du temps.

Comment réaliser un test unitaire ?

- Utiliser le tag **@Test** devant une méthode ;
- Exécuter en tant que test Junit.

Exemple de test unitaire extrait de [JUCookbook] :

```
@Test
public void simpleAdd()
{
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

Exécution des tests contenus dans la classe TestClass1 :

```
java org.junit.runner.JUnitCore TestClass1.class
```

III-B-1 - Préparation de tests

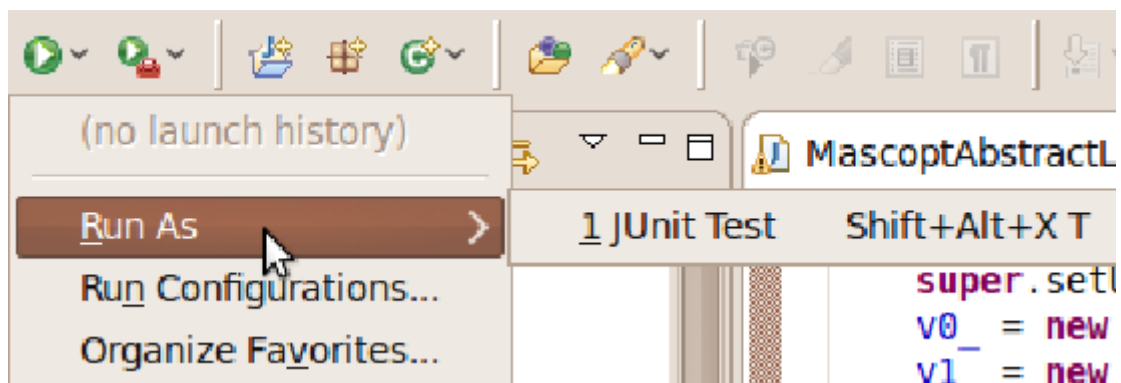
Avant une batterie de tests, il faut souvent faire des opérations pour instancier les objets, ouvrir les fichiers, ouvrir les sockets nécessaires aux tests. C'est l'installation du test (fixture en anglais). Deux tags Junit sont prévus pour cela :

- **@Before** : méthode appelée avant le test ;
- **@After** : méthode appelée après le test.

```
public class MoneyTest
{
    private Money f12CHF;
    private Money f14CHF;
    private Money f28USD;

    @Before public void setUp()
    {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }
}
```

Exécution des tests unitaires dans Eclipse :



III-C - Doctest

Doctest est un module Python permettant de réaliser du test unitaire sous la forme d'une sorte de documentation incluse au sein du code des classes/fonctions. Les idées principales sont :

- les tests doivent être écrits au sein du code testé :
 - meilleure maintenabilité,
 - les tests servent d'exemples d'entrées/sorties ;
- les tests permettent de faire des tests de non-régression ;
- ces tests sont une sorte de documentation.

Exemple de test unitaire pour la fonction factorielle.

III-D - Unittest

```
>>> factorial(3)
6
```

Exécution de(s) test(s) :


```
import doctest
doctest.testmod()
```

III-D-1 - Exemple doctest sur la fonction factorielle

```
def factorial (n):
    """
    >>> factorial (3)
    6
    """
    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n: # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result

if name == " main ":
    import doctest
    doctest.testmod()
```

III-D-2 - Exécution du test

```
Trying:
    factorial (3)
Expecting:
    6
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.factorial
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

En cas d'échec du test :

```
Failed example:
    factorial (3)
Expected:
    6
Got:
    7
```

III-D-3 - Exemple de tests unitaires sur factorielle

```
def factorial (n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int,
    return an int. Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial (30)
    26525285981219105863630848000000L
    >>> factorial (-1)
```

```
Traceback (most recent call last):
...
ValueError: n must be >= 0

Factorials of floats are OK, but the float must be integer: >>> factorial (30.1)
Traceback (most recent call last):
...
ValueError: n must be exact integer
"""
```

III-D-4 - Test externalisé dans un fichier

Le fichier contenant les tests peut être externe au programme. En fait, les tests peuvent être situés dans n'importe quel document texte

```
Ceci est un fichier texte banal. Il contient des tests,
mais il faut auparavant faire l'import du module concerné:
>>> from example import factorial
Le test en lui-même :
>>> factorial(6)
120
```

Le main consiste alors à exécuter les tests du fichier :

```
doctest.testfile("factoriel_test_externe.txt")
```

```
Failed example:
    factorial (6)
Expected:
    120
Got:
    720
```

III-D-5 - Règles d'analyse de l'output

Doctest se base sur la comparaison de l'output de la fonction testée avec l'output fourni dans le test. Certaines règles particulières sont à connaître :

- les lignes vides ne sont pas autorisées : elles servent de délimiteur pour signaler la fin de l'output (utiliser <BLANKLINE> à la place dans le test) ;
- stderr est ignoré ; seul stdout est capturé ;
- si un backslash est attendu en output, des raw strings doivent être utilisées

```
r"""
>>> machin(3)
6\n4
"""
```

```
Expected:
    6\n4
Got
    6
    4
```

III-E - Gestion des exceptions

Les exceptions sont traitées comme des chaînes standards par doctest, mais avec des traitements supplémentaires pour gérer la variabilité des traces de stack d'erreur :

- la ligne d'entête Traceback est détectée et comparée ;
- la ligne comportant la stack d'erreur est ignorée ;
- la ligne contenant le type de l'erreur et l'appel fautif est comparée.

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

Même si doctest fait le boulot proprement, un best practice est d'omettre la stack d'erreur explicitement, sauf s'il est souhaitable de la conserver à des fins de documentation :

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
...
ValueError: list.remove(x): x not in list
```

III-F - Unittest

Le test unitaire peut se faire de manière plus classique à l'aide du module unittest comme dans l'exemple ci-dessous :

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = range(10)

    def testshuffle(self):
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

    def testchoice(self):
        element = random.choice(self.seq)
        self.assertIn(element, self.seq)

if __name__ == '__main__':
    unittest.main()
```

III-F-1 - Les concepts d'Unittest

Unittest est un ensemble de classes permettant de développer des tests en utilisant les concepts suivants :

- test fixture : préparation du test (création d'objets, lancement de serveurs, proxys) ;
- test case : un test unitaire à proprement parler ;
- test suite : un ensemble de tests à exécuter successivement ;
- test runner : orchestration des tests et rapports à l'utilisateur (texte, graphique).

Unittest et doctest

Unittest est capable d'appeler des tests doctets :

```
import unittest
import doctest
import my_module_with_doctests, and_another

suite = unittest.TestSuite()

for mod in my_module_with_doctests, and_another:
```

```
suite.addTest(doctest DocTestSuite(mod))  
runner = unittest.TextTestRunner()  
runner.run(suite)
```

IV - La sécurité dans la machine virtuelle Java

Objectifs

- Comment définir des politiques de contrôle des applications Java ?
- Comment implémenter un composant de contrôle d'application Java ?
- Que vérifie la machine virtuelle lors de l'exécution du byte code ?

IV-A - Besoin : le sandboxing

Le sandboxing est le concept qui recouvre le contrôle d'une application. On souhaite contrôler ce qu'elle fait (les opérations autorisées) et à quoi l'application accède (les ressources).

Les ressources sont multiples :

- disques locaux, distants et leurs systèmes de fichier ;
- mémoire locale/distantes (RAM) ;
- serveurs (notamment dans le cas d'une applet).

Le sandboxing classique le plus restreint est de laisser l'application accéder au CPU, à l'écran, clavier, souris, et à une partie de la mémoire. Un degré de liberté supplémentaire consiste à donner accès au disque dur.

Deux modes classiques d'exécution existent :

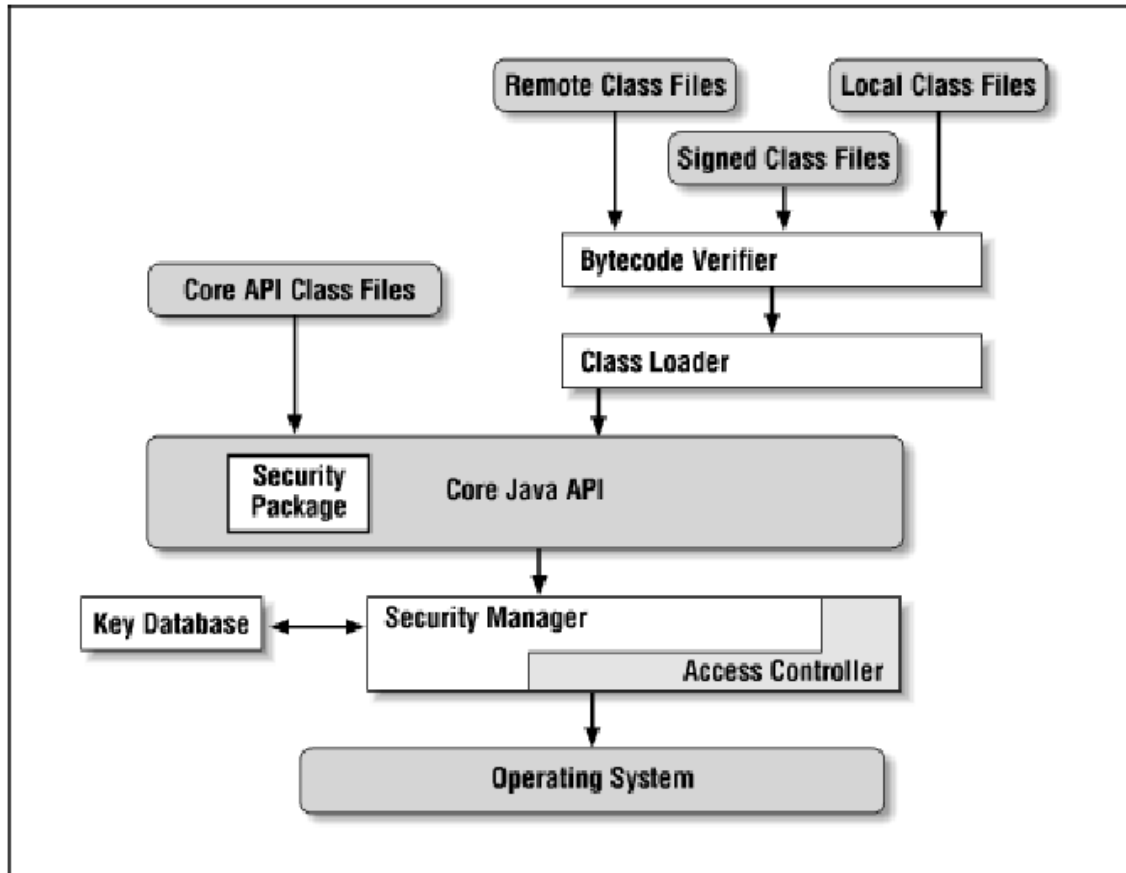
- applet : sandboxing le plus restreint ;
- application : aucun sand boxing.

Objectifs

Comment définir des politiques de sand boxing plus fines ?

Comment charger ces politiques dans la VM ?

IV-B - Introduction : les composants de sécurité la VM



Architecture de la sécurité dans Java, extrait de [JSEC].

IV-B-1 - Architecture de la sécurité dans Java

Le Bytecode verifier

Le Bytecode verifier s'assure que byte code ne viole pas des règles du langage Java (cf. [Le Bytecode verifier](#)). Toutes les classes ne sont pas vérifiées, notamment celle du *core API*.

Le class loader

Le class loader est responsable du chargement des classes qui ne sont pas trouvées dans le CLASSPATH.

L'access controller

Il autorise ou interdit les accès des classes du *core API* vers l'operating system. C'est une sorte de contrôle des appels système.

Le security manager

Prend les décisions d'accès aux ressources du système (fichiers, réseau, etc.) en fonction de la politique chargée. Il peut coopérer avec l'access controller pour une partie de ses décisions.

Le security package

Le package **java.security** est un ensemble d'outils dédiés aux mécanismes d'authentification des classes Java signées (clés, certificats, signatures, cryptage).

IV-C - Politiques de sécurité pour la VM

Une politique de sécurité s'écrit sous la forme suivante :

```
grant codeBase "domaine cible"
{
    permission X [...];
    permission Y [...];
    permission Z [...];
}
```

Le domaine cible est :

- des ressources locales (fichiers) ;
- des ressources distantes (serveurs).

Les permissions ont :

- un nom ;
- des paramètres optionnels (actions, types).

Exemple :

```
grant codeBase "http://www.ensi-bourges.fr/"
{
    permission java.security.AllPermission;
} ;
```

IV-C-1 - Permissions des politiques (1)

File permission : java.io.FilePermission

```
permission java.io.FilePermission "tmpFoo", "write";
permission java.io.FilePermission "<<ALL FILES>>",
    "read,write,delete,execute";
permission java.io.FilePermission "Sfuser.homed/-", "read";
```

Socket permission : java.net.SocketPermission

```
permission java.net.SocketPermission "www.ensi-bourges.fr:1-",
    "accept,listen,connect,resolve"
```

Runtime permission : java.lang.RuntimePermission

createClassLoader, getClassLoader, createSecurityManager, setIO, stopThread.

```
permission java.lang.RuntimePermission "accessClassInPackage.sdo.foo";
```

IV-C-2 - Permissions des politiques (2)

Security permission : java.io.SerializablePermission

`enableSubstitution` : autorise `enableReplaceObject()` de `ObjectInputStream`. `enableSubclassImplementation` : autorise la surcharge de `readObject()` et `writeObject()`.

Property permission : `java.util.PropertyPermission`

```
permission java.util.PropertyPermission "java.*", "read";
```

All permission : `java.security.AllPermission`.

IV-C-3 - Domaines

Fichiers :

```
file:/C:/files/truc.txt
file://files/truc.txt
file:${java.home}/files/truc.txt
```

URL :

```
// One jar filer
http://www.ensi-bourges.fr/files/truc.jar
// Class files:
http://www.ensi-bourges.fr/files/
// Class and jar files:
http://www.ensi-bourges.fr/files/*
// Class and jar files in any sub directories:
http://www.ensi-bourges.fr/files/-
```

IV-C-4 - Exemple complet de fichier de politique

```
keystore "Sfuser.home1${/}.keystore";

grant codeBase "file:${java.home}/lib/ext/-"
{
    permission java.security.AllPermission;
} ;

grant codeBase "http://www.ensi-bourges.fr/files/"
{
    permission java.io.FilePermission "/tmp", "read";
    permission java.lang.RuntimePermission "queuePrintJob";
} ;
```

IV-C-5 - Invocation du security manager

Permettre le sandboxing :

```
Java -Djava.security.manager TestEcriture
Exception in thread "main" java.security.AccessControlException:
access denied (java.io.FilePermission testEcriture.txt write)
```

Préciser une politique (URL du type HTTP ou file) :

```
java -Djava.security.manager -Djava.security.policy=URL TestEcriture
```

Préciser une et une seule politique :

```
java -Djava.security.manager -Djava.security.policy==URL TestEcriture
```

Appletviewer:

```
appletviewer -J-Djava.security.policy=URL urls
```

IV-D - Le security manager

Le security manager est mis en œuvre lorsqu'un appel vers l'API Java est fait depuis un objet du programmeur. Principes du security manager :

- requête du programme vers l'API Java ;
- l'API Java demande l'autorisation au security manager ;
- si refusée, une exception est levée ;
- sinon l'appel est honoré.

Il est possible de catcher l'exception. Par exemple, dans le cas particulier des applets, il est classique d'avoir l'exception **sun.appletAppletSecurityException: checkread**. Voici une meilleure façon de procéder, lorsque l'on est informé de l'exception, plutôt que de perdre la trace de l'exception :

```
OutputStream os;  
try  
{  
    os = new FileOutputStream("statefile");  
}  
catch (SecurityException e)  
{  
    os = new Socket webhost, webport).getOutputStream ();  
    // Ecriture dans os...  
}
```

IV-D-1 - Mettre en place un security manager

Changer de security manager :

- récupérer le security manager courant (ou null) ;
- Installer le security manager (ne peut plus être enlevé) (utile pour une applet qui ne peut plus le changer).

```
public static SecurityManager getSecurityManager()  
public static void setSecurityManager(SecurityManager sm)
```

Exemple :

```
public class TestSecurityManager  
{  
    public static void main(String args[])  
    {  
        System.setSecurityManager(new SecurityManagerImpl());  
        ...do the work of the application...  
    }  
}
```

IV-D-2 - Méthodes du security manager : io

Vérification de lecture/écriture/suppression de fichier :

```
public void checkRead(FileDescriptor fd)  
public void checkRead(String file)  
public void checkWrite(FileDescriptor fd)  
public void checkWrite(String file)
```



```
public void checkDelete(String file)
```

checkRead() est appelé pour :

```
File.canRead()  
File.isDirectory()  
File.isFile()  
File.lastModified()
```

Extrait Javadoc :

Check if the current thread is allowed to read the given file. This method is called from `FileInputStream`, `FileInputStreamO`, `RandomAccessFile`, `RandomAccessFile()`, `File.exists()`, `canReadO`, `isFile()`, `isDirectory()`, `lastModified()`, `length()` and `list`. The default implementation checks `FilePermission(filename, "read,.` If you override this, call `super.checkRead` rather than throwing an exception.

IV-D-3 - Méthodes du security manager : réseau

Vérifier l'ouverture d'une socket (appel pour **Socket()**) :

```
public void checkConnect(String host, int port)
```

Vérifier l'écoute d'un port et l'acceptation d'une requête (appel pour **ServerSocket.accept()**) :

```
public void checkListen(int port)  
public void checkAccept(String host, int port)
```

Changer l'implémentation par défaut d'une socket :

```
public void checkSetFactory()
```

Extrait Javadoc :

Check if the current thread is allowed to accept a connection from a particular host on a particular port. This method is called by `ServerSocketImplAccept()`. The default implementation checks `SocketPermission(host + ":", port, "accept,.` If you override this, call `super.checkAccept` rather than throwing an exception.

IV-D-4 - Méthodes du security manager : vm

Protection de la virtual machine :

```
checkCreateClassLoader() --> ClassLoader() // Changement de class loader  
checkExit() --> Runtime.exit() // Arrêt de la virtual machine  
checkLink() --> Runtime.loadLibrary() // Chargement de code natif  
checkExec() --> Runtime.exec() // Execution de commande shell
```

Protection des threads avec **checkAccess(Thread g)** :

```
Thread.stop() Thread.interrupt()  
Thread.suspend() Thread.resume()  
Thread.setPriority0 Thread.setName()  
Thread.setDaemon()
```

Protection de l'inspection de classes avec **checkMemberAccess()** :

```
Class.getFields()
```

```
Class.getMethods()  
Class.getConstructors()  
Class.getField()  
Class.getMethod()  
Class.getConstructor()
```

IV-E - Le Bytecode verifier

L'expérience des langages dérivés du C a conduit les concepteurs de Java à définir un certain nombre de règles de sécurité lors de l'interprétation du bytecode verifier. La difficulté de garantir de la sécurité provient aussi de la **dynamisme du langage** : les classes peuvent être chargées à la volée, de manière distante, et peuvent être inspectées.

Par rapport à C++, on doit au moins garantir :

- l'encapsulation : private, protected, public ;
- la protection contre les accès mémoire arbitraires.
Java définit donc des règles qui garantissent que :
- l'attribut privé reste privé (sauf sérialisation) ;
- la mémoire ne peut pas être accédée arbitrairement ;
- entité ayant une méthode *final* ne peut-être change - variable public final - une sous#classe exemple **Thread.setPriority()** ;
- certaines entités sont *final* : exemple String (constante) ;
- une variable non initialisée ne peut être lue (sinon cela revient à lire la mémoire) ;
- Les bornes des tableaux sont vérifiées ;
- Les casts sont vérifiés.

Objectif : répondre à la question : ce byte code est-il légal ? (même s'il provient d'un autre langage que Java...).

IV-E-1 - Attaque par définition de classe

```
public class CreditCard  
{  
    public String acctNo = "0001 0002 0003 0004";  
}
```

```
public class TestCreditCard  
{  
    public static void main(String args[])  
    {  
        CreditCard cc = new CreditCard(); System.out.println("Your account number is " + cc.acctNo);  
    }  
}
```

On change public en private :

```
jf@linotte:security/code> javac CreditCard.java  
jf@linotte:security/code> javac TestCreditCard.java  
jf@linotte:security/code> java TestCreditCard  
Your account number is 0001 0002 0003 0004  
jf@linotte:security/code> gvim CreditCard.java  
jf@linotte:security/code> javac CreditCard.javaNTA  
jf@linotte:security/code> java TestCreditCard  
Exception in thread "main" java.lang.IllegalAccessError:  
    tried to access field CreditCard.acctNo from class  
    TestCreditCard at TestCreditCard.main(TestCreditCard.java:4)  
jf@linotte:security/code> javac TestCreditCard.java  
TestCreditCard.java:4: acctNo has private access in CreditCard  
System.out.println("Your account number is " + cc.acctNo);
```

IV-E-2 - Attaque par substitution de classe

Autre exemple tiré de [JSEC] :

- copy the java.lang.String source file into our CLASSPATH ;
- in the copy of the file, modify the definition of value--the private array that holds the actual characters of the string--to be public ;
- compile this modified class, and replace the String.class file in the JDK ;
- compile some new code against this modified version of the String class. The new code could include something like this :

```
public class CorruptString
{
    public static void modifyString(String src String dst)
    {
        for (int i = 0; i < src.length; i++)
        {
            if (i == dst.length)
                return;
            src.value[i]    dst.value[i];
        }
    }
}
```

Now any time you want to modify a string in place, simply call this modifyString() method with the string you want to corrupt (src) and the new string you want it to have (dst).

Remove the modified version of the String class.

IV-E-3 - Vérifications effectuées par le bytecode verifier

- Fichier .class : format correct ;
- Classes *final* : n'ont pas de sous-classe ;
- Méthodes *final* : non surchargées ;
- classe : une seule superclasse ;
- pas de cast illégal (ex. int vers object) ;
- vérification upcast/downcast ;
- data stack : débordement (récursion infinie) ;
- operand stack : no overflow/underflow.

Les vérifications sont faites avant l'exécution et/ou en temps réel :

- code vérifie avant l'exécution ;
- Remplace par un code spécial indiquant que les tests ont été faits ;
- Sinon: `IllegalAccessException` ;
- possibilité d'option : `-noverify` (ne marche plus en Java 2).

À l'exécution par exemple les casts, les bornes des tableaux :

```
void initArray(int a[], int nItems)
{
    for (int i = 0; i < nItems; i++)
        a[i] = 0;
```

IV-F - La sérialisation

La sérialisation est un mécanisme simple pour écrire des objets (et leurs dépendances pour les objets composites) dans des fichiers. Il est aussi utile par exemple pour l'API RMI pour la migration des objets sur le réseau. Pour rendre une classe sérialisable, on utilise un flag en implémentant l'interface **Serializable**.

Cependant, la sérialisation révèle les attributs privés :

- accessible en lecture sur le disque (ou autre buffer) ;
- peut même être édité.

Le langage Java fournit le mot-clé **Transient** pour spécifier qu'une variable ne doit pas être sérialisée.

```
Private transient String creditCardNumber ;
```

IV-G - Le Keystore

Code signé, cf. tool jarsigner.

- keystore : clés publiques des entités signées.
- À l'exécution :
 - récupération de la clé publique du code signé ;
 - installation dans le Keystore (.Keystore).
- Administration du keystore : keytool.

V - La sécurité dans Python

V-A - Les vieilleries : rexec

Le module **rexec** était fait pour exécuter du code dans lequel le programmeur n'a pas confiance dans un environnement préparé à cet effet : c'est l'idée classique du sandboxing.

Un code *non-trusted* peut :

- violer des restrictions du bac à sable : rexec lève une exception ;
- attaquer des ressources (cpu, mémoire) : non prévu, utiliser un process à part.
Constructeur : `class RExec(hooks=None, verbose=False)`

```
r = import rexec  
r = rexec.RExec()
```

Méthodes :

- `r.r_import(modname)` : importe le module dans le bac à sable ;
- `r.r_exec(code)` : exécute le code dans le bac à sable ;
- `r.r_execfile(filename)` : exécute le fichier dans le bac à sable ;
- `r.r_add_module(modname)` : ajoute et retourne un pointeur sur module.

Attributs de l'objet rexec :

- `nok_builtin_names` : Built-in functions not to be supplied in the sandbox ;
- `ok_builtin_modules` : Built-in modules that the sandbox can import ;
- `ok_path` : Used as `sys.path` for the sandbox's import statements.

V-B - mxProxy - Proxy-Access to Python Objects

mxProxy est une bibliothèque open source soutenue par eGenix.com. Son but est de fournir une méthodologie générale pour restreindre l'accès à des attributs ou des méthodes de classe.

Le proxy fournit une protection des attributs de classe en définissant une sorte d'interface : une liste des noms des attributs vers lesquels les accès sont autorisés.

Et les méthodes et attributs privés ?

- Ils existent et peuvent être utilisés à l'aide du `__` ;
- leur caractère privé est une convention, rien n'est garanti.

```
class Toto():
    def test(self):
        return 10;
    def test2(self):
        return 15;
if name == " main ":
    t = Toto()
    print t.test(); # Ok, that's public
    #print t.test2(); # Not ok, that's private
    print t._Toto.test2(); # Ok, too !
```

L'appel d'une fonction reste toujours possible en utilisant `_Classe` méthode.

V-B-1 - Types de proxy

Constructeur du proxy pour un objet :

```
Proxy(object, interface=None, passobj=None)
```

- `interface` : séquence de string autorisant l'accès aux attributs/méthodes ;
- `passobj` : permet d'autoriser la récupération de l'objet via `.proxy_object()`.

Proxy avec read-cache : met en cache les attributs de l'objet wrappé

```
CachingInstanceProxy(object, interface=None, passobj=None)
```

(attention aux remises à jour de l'objet *wrappé*) .

Proxy de type read-only :

```
ReadonlyInstanceProxy(object, interface=None, passobj=None)
```

Proxy factory : permet de créer des instances de Class wrappe dans un Proxy

```
ProxyFactory(Class, interface=None)
```

V-B-2 - Méthodes sur un proxy : récupérer les attributs

Méthode permettant de récupérer des attributs :

- `.proxy_getattr(name)` : récupère l'attribut `name` ;
- `.proxy_setattr(name)` : modifie l'attribut `name` ;

- `.proxy_defunct()` : retourne 1 si l'objet a déjà été garbage collecté.

```
from mx import Proxy
class DataRecord:
    a= 2
    b= 3
    # Make read-only:
    def public_setattr (self,what,to):
        raise Proxy.AccessError('read-only')
    # Cleanup protocol
    def cleanup (self):
        print 'cleaning up',self
o = DataRecord()
# wrap the instance:
p = Proxy.InstanceProxy(o, ('a',))
# Remove o from the accessible system:
del o

print p.proxy_getattr( 'a')
print p.a
```

V-B-3 - Méthodes sur un proxy : récupérer les méthodes

Pour récupérer des méthodes :

```
from mx import Proxy
class DataRecord:
    a= 2
    b= 3
    # Make read-only:
    def public_setattr (self,what,to):
        raise Proxy.AccessError('read-only')
    # Cleanup protocol
    def cleanup (self):
        print 'cleaning up',self
    def essai(self):
        print 'coucou'
o = DataRecord()
# Wrap the instance:
p = Proxy.InstanceProxy(o, ('essai',))
# Remove o from the accessible system:
del o

print p.essai()
print p.b # Ne marche pas
```

V-B-4 - Substitution de classe par son proxy

```
from mx import Proxy
class DataRecord:
    a = 2
    b = 3
    # Make read-only:
    def public_setattr (self,what,to):
        raise Proxy.AccessError('read-only')
    # Cleanup protocol
    def cleanup (self):
        print 'cleaning up',self
# If you want to have the wrapping done automatically, you can use
# the InstanceProxyFactory:
DataRecord = Proxy.InstanceProxyFactory(DataRecord, ('a',))
# This gives the same behaviour...
p = DataRecord()
print p.a
#p.a = 3 # interdit par le code de setattr
#print p.b # interdit par le proxy
```

VI - Bibliographie

JSE(1, 2)	The Java Tutorials : Exceptions .
Rifflet	Une présentation des principaux concepts objets dans Java 1.4.2, J.-M. Rifflet, 28 juin 2005.
JUCookbook	JUnit Cookbook , Kent Beck and Erich Gamma.
JSEC(1, 2)	Scott Oaks, Java Security, O'Reilly. Year={2001}.