Introduction à la sécurité – Cours 4
Java et sécurité

Catalin Dima

Java : modes d'utilisation

- Applications : programmation traditionnelle.
- Applets : interactions avec les web-browsers.
- Beans : programmation modulaire (façon OLE).
- Servlets : applications coté serveur.
- Aglets : agents intelligents.
- Doclets :générateurs de docs configurables.
- Systèmes embarqués.
- "JavaCard" : cartes à puce.

Java et sécurité

- Compliateur du langage dans un format intermédiaire = *bytecode*, indépendent de machine.
- *Machine virtuelle* (VM) pour exécuter le bytecode.
- Environnement d'exécution tournant au-dessus de la VM et fournissant un certain nombre de classes de base.
- Distinction entre **privé** et **public** (avec les variantes *protected* et paquetage).
- Les trois composants de la VM :
  1. **Bytecode Verifier** : vérifie si le code à exécuter est correct.
  2. **Class Loader** : détermine quand et comment des nouvelles classes doivent être ajoutées à l'environnement Java pour continuer le travail.
  3. **Security Manager** et **Access Controller** : restrictionnent l'utilisation des interfaces API par des classes dans l'environnement de travail.
- Rajouts ultérieurs
  - Paquetage de fournisseurs de services cryptographiques (mal-nommé *Security API*) : JCA/JCE/JSSE.
  - Paquetages d'authentification et d'autorisation : JAAS.

Tâches du bytecode verifier

- Tester du bon format et de la bonne longueur de la classe, ainsi que de la présence du *numéro magique* $0xCAFEBABE$.
- Tester de l'impossibilité de provoquer des débordements de pile (dans les deux sens : *overflow* et *underflow*)
  - Ne peut tester que des débordements de la pile d'opérands, pas de la pile de données.
- Refuser les conversions illégales (casts) – certaines tests de conversions differées à l'exécution.
- Tester de l'accès correcte aux objets (privé/protégé/public).
- Tester de l'emploi correcte des paramètres dans les méthodes tant en nombre qu'en types.
- S'assurer que les références à d'autres classes sont légales.

Le bytecode verifier est employé à la fois avant le chargeur de classe, et pendant l'exécution.

Vérifier le vérificateur ?

Tâches du ClassLoader

- Responsable du chargement des classes :
  - Donné un nom de classe, localiser et générer sa définition.
  - Toujours regarder d'abord aux classes "standard".
  - Toute classe possède une référence à l'instance du ClassLoader qui l'a défini.
- Responsable de la gestion des namespaces :
  - Vue différente de l'environnement, selon origine (e.g. applets provenant des sources différentes, à différents niveaux de confiance).
  - Un namespace par instance de ClassLoader.
  - Classes se trouvant dans `CLASSPATH` supposées de confiance, autres classes doivent passer par le vérificateur de bytecode.
- ClassLoader primordial, partie essentielle de la JVM.
- Autres ClassLoaders héritiers : `java.lang.ClassLoader`, `java.security.SecureClassLoader`, `java.net.URLClassLoader`, ...
  - Groupés en ClassLoaders de type Applet, de type RMI et de type Sécurité.
  - Certaines applications de confiance (e.g. browser) définissent eux-mêmes leurs ClassLoaders.

Tâches du SecurityManager et AccessController

- Définissent les restrictions d'utilisation des interfaces API Java par les applications.
  - En général, applications cibles = applets.
  - Mais ne pas oublier les RMI et la reflexion !
- Doivent gérer les requêtes d'opérations dangereuses du code à qui on ne fait pas confiance.
  - Prévenir l'installation de nouveaux ClassLoaders.
  - Protéger les (groupes de) threads les uns contre les autres.
  - Contrôler l'abilité d'éteindre la JVM.
  - Contrôler l'accès aux ressources de tout sorte : queues d'imprimante ou d'événements, propriétés de système, fenêtres, etc.
  - Contrôler l'accès aux fichiers , surtout aux fichiers locaux.
  - Contrôler les opérations réseau (socket connect ou accept).
  - Contrôler l'accès aux paquetages Java, surtout l'accès aux classes renforçant la sécurité.

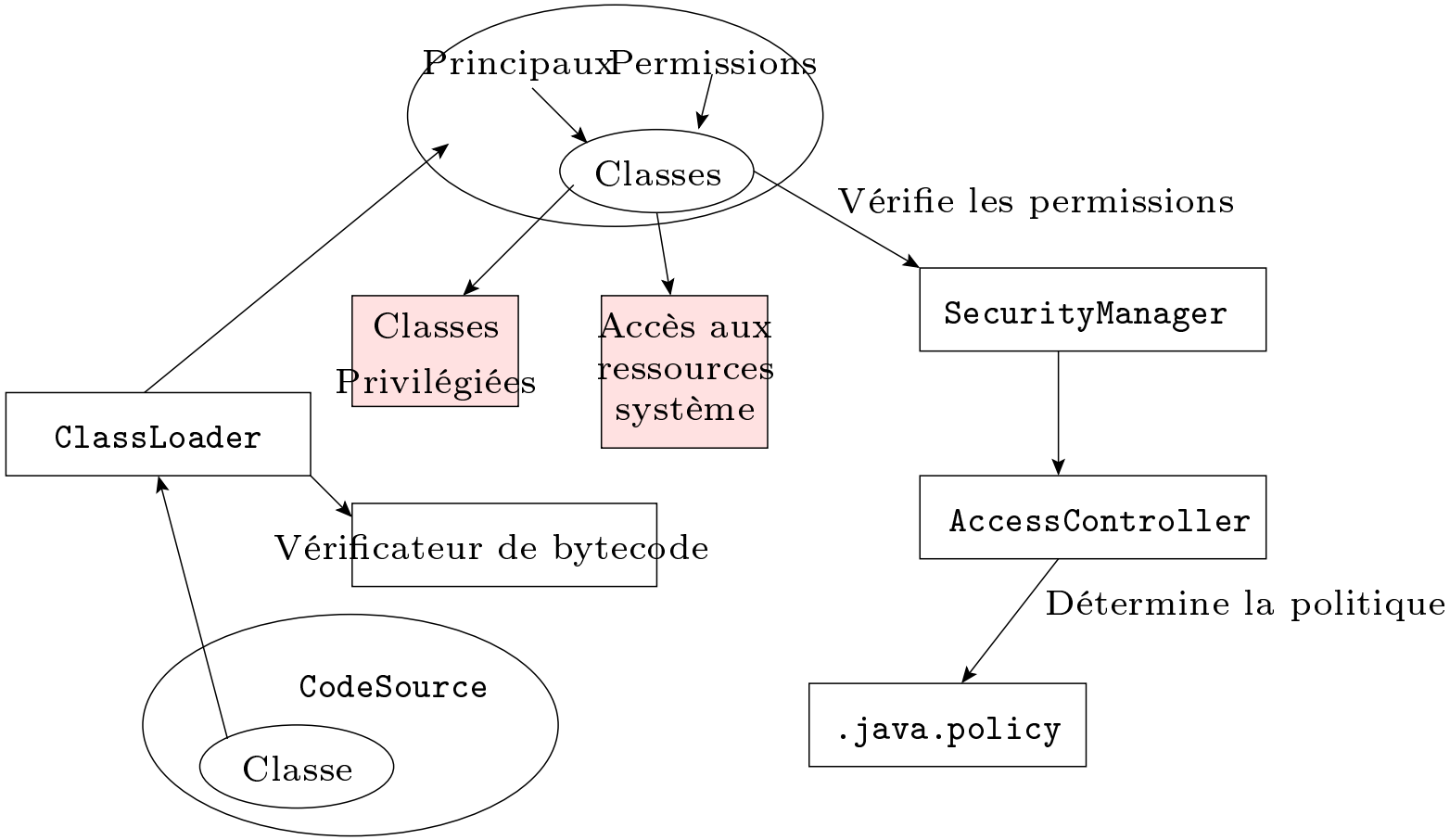
Architecture de sécurité (= controle d'accès) Java

- Distinction entre la *sécurité des applications* et la *sécurité système*.
  - Applications (applets !) téléchargées, pas totalement de confiance...
  - Nécessité de les faire tourner dans un "bac à sable", avec des permissions réduites – pas les mêmes que la VM !
  - Nécessité de permettre à la VM de gérer ces permissions.
  - Définition de *domaines de protections* associés aux objets.
  - Association de *permissions* aux domaines de protection.
  - Création de *politiques* en tant qu'objets, encapsulant les différentes règles de contrôle d'accès.

Architecture de sécurité Java

- *Domaine de protection* : groupement de composants ou ressources (dans la VM) qui possèdent un certain ensemble de permissions.
  - Classe `java.security.ProtectionDomain`.
  - Toute classe possède un `ProtectionDomain` : méthode `getProtectionDomain()` de `Class`.
  - Permissions accordées aux domaines de protection, et pas directement aux classes et objets !
- Une *politique de sécurité* définit un domaine de protection.
  - Identifié par un `CodeSource` = l'endroit d'ou les classes sont originaires (e.g. chargées) + une signature éventuelle (certificat).
  - Associé à une collection de *permissions* (`PermissionCollection`).
  - Associé aussi à un ensemble de principaux (JAAS).
  - Constructeur `ProtectionDomain(CodeSource codesource, PermissionCollection permissions)`.
  - Méthodes `getCodeSource()`, `getPermissions()`.
  - Méthode `implies(Permission p)` pour tester si *une permission spécifique est autorisée* pour les objets dans ce domaine de protection.
- Constructeur `CodeSource` : `CodeSource(URL url, Certificate[] certs)`
- Méthode : `boolean implies(CodeSource codesource)`.

Domaine de protection



Classes SecurityManager et AccessController

- Au plus un `SecurityManager` installé à tout instant.
  - Pas de constructeur publique !
  - `System.getSecurityManager()` et `System.setSecurityManager(nvSecMan)`.
- Prend les décisions d'autoriser ou non des opérations sensibles (en coopération avec `AccessController`)
- Méthodes `check`"Qqchose" appelées dans le code de l'API (liste à la fin).
- On peut créer notre propre `SecurityManager` héritant de la classe `SecurityManager`.

Version Java 2

- Toutes les décisions de `SecurityManager` passent par le `AccessController`.
- Par défaut, toutes les méthodes `check`Qqchose du SecurityManager appellent `checkPermission` de SecurityManager,...

– ...qui, à son tour, appelle `checkPermission` de `AccessController`.
– On définit toujours notre `SecurityManager` !

Algorithme de sécurité en Java 2

Permission refusée = **exception levée !**

– Donc terminaison de `checkPermission` = autorisation.

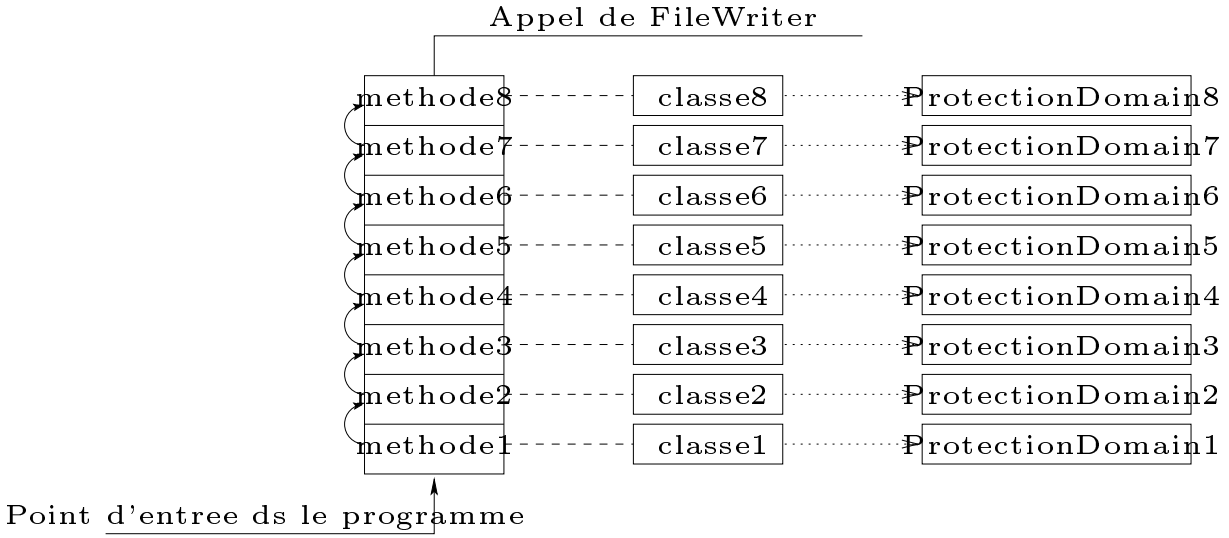
1. Méthode M demande la permission P :
   – L'implémentation de M appelle la méthode `checkPermission` du `SecurityManager` courant.
2. Par défaut, celle-ci appelle `checkPermission` de la classe `AccessController` :
3. Pour **toute entrée de la pile d'appels** :
   – Si le `ProtectionDomain` de l'objet (qui fait l'appel) manque la permission P, **exception levée**.
   – Si l'objet a appelé `doPrivileged` sans contexte, retour, donc **succès**.
   – Si l'objet a appelé `doPrivileged` avec contexte, tester le contexte pour la permission P et retour si contexte possède cette permission, **exception levée** sinon.

– Contexte d'appel = objet `AccessControlContext`.

– Constructeur `AccessControlContext(ProtectionDomain[] context)`.

Exception : `SecurityException` ou `AccessControlException`.

Mécanisme de fonctionnement de `AccessController`



- Une suite de méthodes s'appelle dans un certain ordre,
- ... et la dernière fait un appel sensible : `FileWriter(nom)`
- ... dans lequel est appelé `SecurityManager.checkWrite(nom)`,
- ... qui crée un objet (`perm`) `FilePermission(nom,"write")`
- ... qui appelle `SecurityManager.checkPermission(perm)`
- ... qui appelle `AccessController.checkPermission(perm)`
- ... qui teste si, pour toute classe (de 1 à 8), les permissions engendrées par le domaine de protection correspondant impliquent la permission `FilePermission(nom,"write")`.

Permissions en Java

- *Permissions* = privilèges accordés à un *principal*.
- Representés par des objets de classe `Permission` *abstraite*.
  - Sous-classes concrètes : (`BasicPermission`, `FilePermission`, `SocketPermission`...).
  - Permissions accordées à travers les implémentations de `SecurityManager`
  - Ou à travers des fichiers `java.policy`.
- Renforcées par la coordination entre le `AccessController` et le `SecurityManager`.
- `SecurityManager.checkPermission(Permission x)` est appelée par la plupart du code Java pour déterminer les droits d'accès aux ressources sensibles (fichiers, réseau...).
- Celle-ci délègue la vérification des permissions au `AccessController.checkPermission(Permission x)`.
  - Permission refusée : exception levée.
  - Permission accordée : pas d'exception levée.

Permissions et politique

- checkPermission(Permission p) : l'objet p est la représentation de la permission *demandée*
  - qui peut correspondre à une permission d'accès au fichier, à une socket etc.
- L'`AccessController` va vérifier si, dans la politique courante, le code demandant est *associé à une permission* qui implique p.
- Donc on vérifie si la permission demandée est *impliquée* par une permission *associée* au code.
- Représentation de permissions (demandées ou associées par la politique !) :
  - Classe `Permission` (abstraite), contenant une méthode `implies`.
- Permissions reunies dans des collections homogènes de permissions :
  - Classe `PermissionCollection`, méthodes `add(Permission p)`, `implies(Permission p)`.
- Collections de permissions reunies dans des ensembles hétérogènes :
  - Classe finale `Permissions`, héritant de `PermissionCollection`

Permissions

- Caractérisées par
  - Le *type* de permission = **sous-classe de la classe** `Permission`.
  - La *cible* de la permission = un `String` identifiant cette cible.
  - L'action permise par la permission (optionnel).
- `FilePermission("/home/dima/*","read")`,
  `FilePermission("/home/dima/-","read,write")`
  - On aura donc `(new FilePermission("/home/dima/-","read,write")).implies(new FilePermission("/home/dima/fichier","read"))`
- `SocketPermission("www.hack.fr:1-23","resolve,listen")`,
  `SocketPermission("www.hack.org:1024-","connect,accept")`.
- `PropertyPermission("user.home","read")`,...
- `SerializablePermission("enableSubstitution")`
- `RuntimePermission("exitVM")`, `AWTPermission("accessClipboard")`,
  `NetPermission("requestPasswordAuthentication")`,
  `SecurityPermission("getFields")`,...
  `ReflectPermission("suppressAccessChecks")`, `AllPermission()`

Classe `Policy`

- Classe abstraite, représentation de la politique de sécurité globale agissant pour toute application tournant dans le SecurityManager courant.
- Implémentation par défaut : `sun.security.provider.PolicyFile`.
- Fichier `${java.home}/jre/lib/security/java.security` donne les endroits d'où `PolicyFile` charge la politique globale.
- Par défaut, ces fichiers sont deux :
  - `${java.home}/lib/security/java.policy`.
  - `${user.home}/.java.policy`.
- Politique configurable par les utilisateurs.
  - Mais la reconfiguration pendant l'exécution d'un programme n'est pas prise en compte automatiquement.
- Fournir à un programme un fichier de politique de sécurité particulier :
  - `java -Djava.security.policy=<monfichier>` ou
  - `java -Djava.security.policy==<monfichier>`.

java.policy

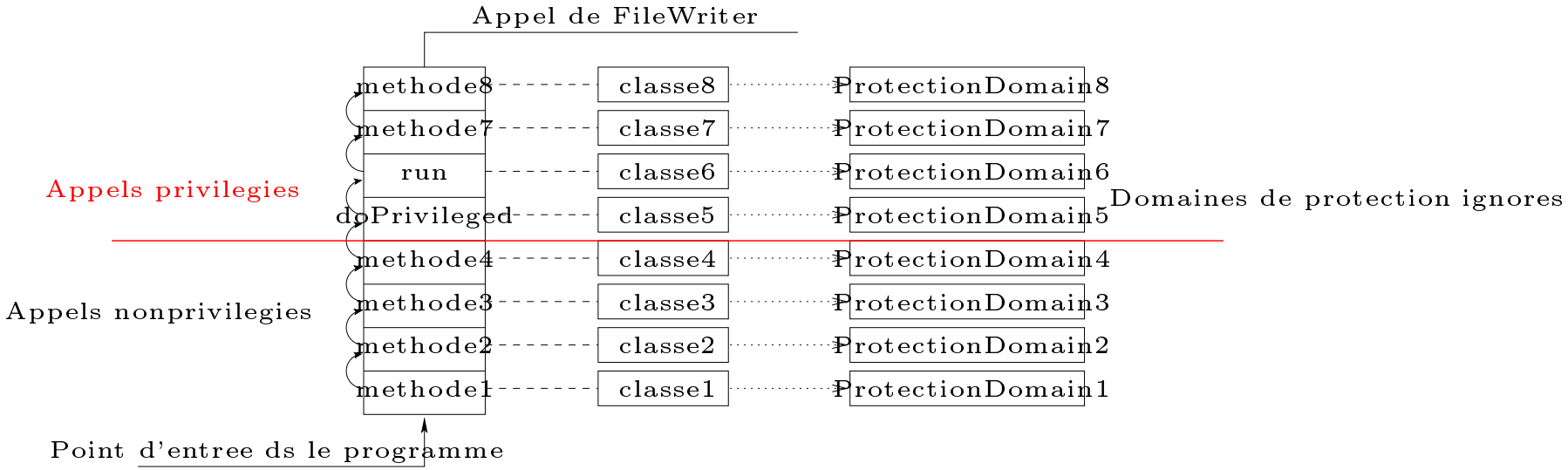
- Entrée unique indiquant l'endroit où trouver les certificats digitaux pour la politique (à voir plus tard) :
  `keystore "file :/..../repertoire", "JKS".`
- Entrées indiquant les permissions accordées :

```
grant [ SignedBy "nom"] , [ CodeBase "endroit" ],
      [ Principal [ nom_de_classe ] "nom" ]}
   { permission TypeDePermission "cible", "action",
     [ SignedBy "nom" ]
   }
```

- `SignedBy` : signature du certificateur de code.
- `Codebase` : endroit où chercher les classes.
- Types de permissions : `FilePermission, SocketPermission, RuntimePermission, AllPermission`...
- Plusieurs permissions possibles pour le même `Codebase`.
- Outil `policytool` pour créer des fichiers syntaxiquement corrects.
- Exemples...

Mécansimes alternatifs (1)

- Mécanisme basé sur les privilèges d'une autre méthode : AccessController.doPrivileged(PrivilegedAction pa)
- Interface `PrivilegedAction` contenant une seule méthode `Object run()`.
- On encapsule à l'intérieur d'une `PrivilegedAction` les accès sensibles mais à qui on délègue les privilèges.
- Tous les appels effectués *à partir de la méthode* `run` de (la classe implémentant) `PrivilegedAction` seront considérés de confiance.
- Le `AccessController` va vérifier seulement les permissions des méthodes *appelant* `run`.



PrivilegedAction exemple

```
unemethode() {
    ...code normal...
    String user = AccessController.doPrivileged(
        new PrivilegedAction<String>() {
        public String run() {
        ... faire qqchose de sensible...
        ... aussi avec les objets declares dans unemethode()...
        ... et renvoyer un String...
            }
        });
    ...encore du code normal...
}
```

- Utilité : court-circuiter le mécanisme de test de permissions.
- Ne provoque pas en soi-même une brèche de sécurité !
  - Le contexte de sécurité de `unemethode` sera toujours verifié !
- Mais à utiliser avec des précautions !

Mécanismes alternatifs (2)

- Situations précédentes : la décision d'accès pouvait être prise sur la base seule de la source du code,
  - et serait identique pour tout objet créé de la même classe.
- Parfois cela n'est pas le cas
  - Un thread (le demandeur) aurait besoin qu'un fichier soit ouvert.
  - Et c'est un autre thread qui peut faire l'action.
  - Les décisions de `AccessController` sont prises sur la base de la pile d'appels *du thread appelant* et pas sur la base des appels de tous les threads.
- Solution : le thread serveur crée l'objet demandé et l'encapsule dans un objet "gardien".
- Le thread demandeur peut accéder à l'objet demandé seulement par le biais du `AccessController` (ou autre mécanisme similaire).

GuardedObject et Guard

- Interface contenant un constructeur et une méthode :
  - `public GuardedObject(Object obj_encapsule, Guard gardien)`
  - `public Object getObject()`
- Interface `Guard` contenant une seule méthode : `public void checkGuard(Object o)`.
  - Supposée se terminer correctement si l'accès à l'objet est permis.
  - Sinon – exception.
- Les classes `Permission` implémentent `Guard`.
  - L'implémentation de `checkGuard()` fait appel au `SecurityManager.checkPermission(this)` (si un `SecurityManager` est installé !).
  - ... donc on peut encapsuler un objet dans un `GuardedObject` dont le gardien est une `Permission`.
- `GuardedObject.getObject()` renvoie l'objet encapsulé, si le `Guard` associé l'autorise, sinon exception.

Mécanismes alternatifs (3)

- Mécanisme basé sur l'identité : classe `java.security.auth.Subject`
  - Groupement d'informations relatives à la sécurité d'une entité (personne).
  - Informations : mots de passe, clés de cryptage, etc.
- Chaque entité `Subject` se voit associer une ou plusieurs identités `java.security.Principal`.
  - C'est des identités à qui on peut attribuer des `Permissions` – revoir le `java.policy`.
- Utilisation : JAAS (authentification et autorisation), JavaCard, ...

Méthodes de SecurityManager : accès aux fichiers

`void checkRead(FileDescriptor fd)`
`void checkRead(String file)`
	`File.canRead(String file)`
	`FileInputStream`
	`RandomAccessFile`
	`File.isDirectory()`
	`File.isFile()`
	`File.lastModified()`
	`File.length()`
	`File.list()`

`void checkWrite(FileDescriptor fd)`
`void checkWrite(String file)`
	`File.canWrite()`
	`FileOutputStream()`
	`RandomAccessFile()`
	`File.mkdir()`
	`File.renameTo()`

`void checkDelete(String file)`	`File.delete()` `File.deleteOnExit()`

Méthodes relatives à l'accès réseau

`void checkConnect()`	`DatagramSocket.send()`
	`DatagramSocket.receive()`
	`MulticastSocket.send()`
	`Socket()`

`checkListen()`
 	
`checkAccept()`
	`DatagramSocket()`
	`MulticastSocket()`
	`ServerSocket()`
	`ServerSocket.accept()`
	`DatagramSocket.receive()`

Méthodes protégeant les aspects de sécurité

checkMemberAccess()	Class.getFields() Class.getMethods() Class.getConstructors() Class.getField() Class.getMethod() Class.getConstructor() Class.getDeclaredClasses() Class.getDeclaredFields() Class.getDeclaredMethods() Class.getDeclaredConstructors() Class.getDeclaredField() Class.getDeclaredMethod() Class.getDeclardConstructor()	checkSecurityAccess()	Identity.setPublicKey() Identity.setInfo() Identity.addCertificate() Identity.removeCertificate() IdentityScope.setSystemScope() Provider.clear() Provider.put() Provider.remove() Security.insertProviderAt() Security.removeProvider() Security.setProperty() Signer.getPrivateKey() Signer.setKeyPair() Identity.toString() Security.getProviders() Security.getProvider() Security.getProperty()

Méthodes protégeant les ressources du système

`checkSystemClipboardAccess()`	`Toolkit.getSystemClipboard()`
`checkAwtEventQueueAccess()`	`EventQueue.getEventQueue()`
`checkPropertiesAccess()`	`System.getProperties()` `System.setProperties()`

`System.getProperty()`	`checkPropertyAccess()`
	`Locale.setDefault()`
	`Font.getFont()`

`checkTopLevelWindow()`	`Window()`

Méthodes protégeant la VM

`checkCreateClassLoader()`	`ClassLoader()`
`checkExec()`	`Runtime.exec()`
`checkLink()`	`Runtime.load()` `Runtime.loadLibrary()`
`checkExit()`	`Runtime.exit()`
`checkPermission(Permission)`	Beaucoup de permissions !

- La méthode `checkPermission` est au fait au centre de la gestion des droits d'accès !
- Toute méthode de type `check<qqchose>` l'appelle !