Módulo 7

Segurança

Lição 8

Class Loaders

Versão 1.0 - Jan/2008

Autor
Aldwin Lee
Cheryl Lorica

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Angelo de Oliveira
Bruno da Silva Bonfim
Denis Mitsuo Nakasaki
Emanoel Tadeu da Silva Freitas
Guilherme da Silveira Elias
Leandro Souza de Jesus
Lucas Vinícius Bibiano Thomé
Luiz Fernandes de Oliveira Junior
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach e Vinícius G. Ribeiro (Especialista em Segurança)
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Class Loader é o mecanismo pelo qual arquivos que contenham bytecodes Java são lidos na JVM e convertidos em definições de classes. Em qualquer situação, um ou mais *Class Loaders* estão prontamente disponíveis, pois o *Class Loader* do sistema (ou o *Class Loader* primordial ou o *Null Class Loader*) existe por padrão. Classes de sistema (são aquelas classes que residem no núcleo da API Java) são carregadas pelo *Class Loader* de sistema, ou, possivelmente, em alguma instância ou subclasse de *Class Loader*.

Ao final desta lição, o estudante será capaz de:

- Ter uma visão geral de funcionamento do *Class Loader*
- Compreender as considerações de segurança sobre o *Class Loader*
- Entender as classes do *Class Loader*

2. Visão Geral do *Class Loader*

O conceito de carregadores de classes também é um componente importante para determinar a política de segurança de um programa Java. Basicamente, existem três domínios em que o carregador de classes opera em um modelo de segurança:

1) O carregador de classes colabora com a máquina virtual para definir os *namespaces* para proteger a integridade dos recursos de segurança em Java.
2) Chama o gerenciador de segurança para garantir que a devida permissão foi dada a um código acessando e/ou definindo classes.
3) Cria o mapeamento de permissões para os objetos de classes permitindo ao controlador de acesso se manter-se a par das classes e suas respectivas permissões.

Como desenvolvedor, o terceiro domínio seria o mais importante. Isso ocorre porque a criação de um carregador de classes personalizado e a instituição de permissões de classes dentro deste carregador de classes seria mais conveniente na implementação de diferentes políticas de segurança dentro do seu aplicativo.

3. Arquitetura de Carregamento de Classes em Java

Antes de criar e usar *Class Loaders* personalizados, vamos analisar alguns de seus mecanismos fundamentais. Para compreender a mecânica dos *Class Loaders* devemos primeiro visualizar sua arquitetura.

Class Loaders são organizados como uma árvore hierárquica. Na raiz da árvore está o *Class Loader* de sistema que, como foi dito anteriormente, carrega o núcleo da API Java. O *Class Loader* de sistema, em seguida, tem pelo menos um filho. Tem pelo menos um filho, o *URL Class Loader* que é usado para carregar classes a partir do caminho da classe principal. O *Class Loader* de sistema pode ter qualquer outro filho direto, mas normalmente quaisquer outros descendentes seriam filhos de *URL Class Loader*.

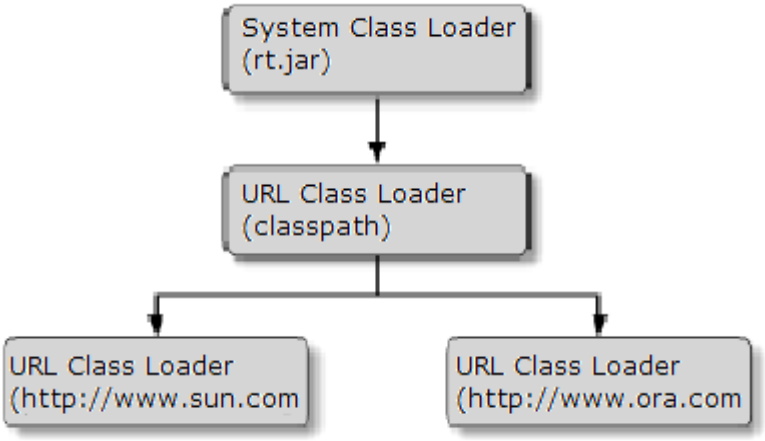

Figura 1: Hierarquia do Class Loader

Essa hierarquia seria a chave para compreender o que acontece quando uma classe é carregada. Uma classe pode ser carregada através de uma das seguintes maneiras:

1) Chamando, explicitamente, o método `loadClass()` de um carregador de classes
2) Chamando o método `Class.forName()`
3) Carregando implicitamente uma classe quando é referenciada por uma classe já carregada

Na primeira forma, o *Class Loader* é o objeto cujo o método *loadClass*() é chamado. Na segunda forma, ou o *Class Loader* de sistema ou outro filho deste que carregou a classe através do método estático *Class.forName*() será passado para o método (*Class.forName*()). Na terceira forma, o *Class Loader* que carregou a classe de origem será usada para carregar a classe referenciada.

Class Loaders são responsáveis por solicitar que as instâncias superiores carreguem uma classe. No entanto, se essa operação falhar, o *Class Loader* tentará definir esta classe. O efeito é que as classes de sistema serão sempre carregadas pelo *Class Loader* de sistema, classes no caminho da classe serão sempre carregadas pelo *Class Loader* que conhece o caminho da classe e, em geral, uma classe será carregada pela classe mais antiga na sua hierarquia que sabe onde encontrar outras classes.

4. Class Loaders e Namespaces

Uma das funções dos *Class Loaders* é cumprir determinadas regras relativas aos nomes usados pelas classes Java. Lembrando que o nome completamente qualificado de uma classe Java é a união do nome do pacote com o da classe. Por exemplo, não existe uma classe chamada *String* na API Java, existe a classe *java.lang.String*. Por outro lado, uma classe não deve, obrigatoriamente, fazer parte de um pacote, caso no qual o nome da classe representa o seu nome completo. É comum dizer que essas classes estão no pacote padrão, mas isso é um pouco enganador: existe um pacote padrão diferente para cada *Class Loader* em uso pela JVM.

Considere o seguinte exemplo.

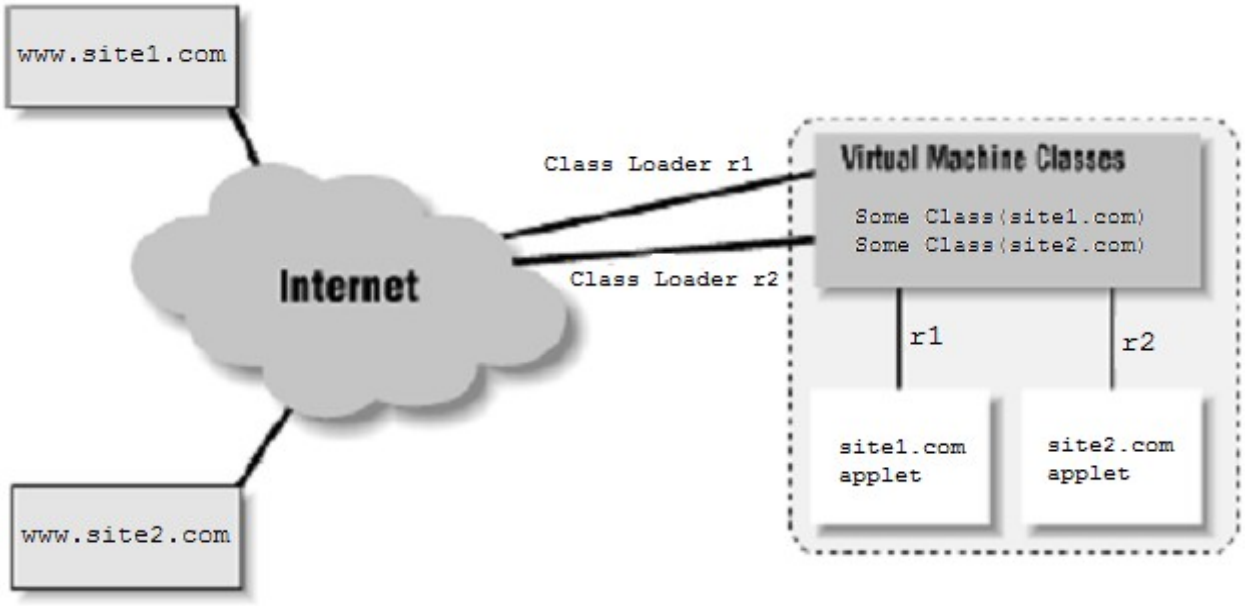

Figura 2: Acessos pelo Class Loader

Quando visitamos uma determinada página (*www.site1.com*), e carregamos uma *applet* que utiliza uma classe chamada *SomeClass* (sem o nome do pacote). Depois seguimos para outra página (*www.site2.com*) e carregamos outra aplicação que utiliza outra classe chamada *SomeClass* (também sem o nome do pacote), sabemos que estas são duas classes diferentes, contudo possuem o mesmo nome totalmente qualificado. Como a JVM conseguirá distinguir estas duas classes?

Quando uma classe é carregada por um *Class Loader* é armazenada em uma referência interna a este *Class Loader*. Quando a máquina virtual precisa acessar uma determinada classe, a JVM pede esta ao *Class Loader* apropriado. Note que o *Class Loader* usado para carregar uma classe também faz parte do nome da mesma classe. Portanto, se dois *Class Loaders* carregam classes que têm idênticos nomes de pacote e de classe, estas ainda serão tratadas como classes diferentes.

5. Resource Loading e Naming

Além de carregar classes, os *Class Loaders* também são capazes de carregar recursos. Esses recursos, que podem ser qualquer seqüência de *bytes*, estão associados a um pacote. O carregamento de um recurso é feito invocando-se o método *getResource*() de um carregador de classes, ou *getSystemResource*(), pelo *Class Loader* primordial. Por exemplo, é possível carregar um arquivo de imagem .gif quando um programa é executado. Este caso pode ser tratado colocando o arquivo .gif no mesmo diretório da classe:

```
ClassLoader cl = this.getClass().getClassLoader();
URL url = null;

if (cl != null) {
  url = cl.getResource("someImage.gif");
} else {
  url = ClassLoader.getSystemResource("someImage.gif");
}
```

Outro método que pode ser usado para carregar recursos é o *getResourceAsStream*(), ou o método *getSystemResourceAsStream*(). Este método acessa o recurso como um *stream*. Por exemplo:

```
ClassLoader cl = this.getClass().getClassLoader();
InputStream in = null;
if (cl != null) {
  in = cl.getResourceAsStream("someImage.gif");
} else {
  in = ClassLoader.getSystemResourceAsStream("someImage.gif");
}
```

Também pode-se usar o método *findResource*() para encontrar um recurso especificado pelo nome. Além disso, pode-se usar os métodos *findResources*() e *getResources*() que retornam uma enumeração de todas as URLs que representam recursos através do nome.

No momento da nomeação de um recurso, os métodos *getResource*() devem refletir à nomeação do pacote. Por exemplo, se um arquivo *SomeResource.txt* está dentro do pacote *innerPackage* que, por sua vez, está dentro de outra pasta *somePackage*, então o recurso é referenciado como *somePackage.innerPackage.SomeResource.txt*.

6. Class Loader e Delegação

Que classes são carregadas pelos diferentes *Class Loaders*? Certas classes são carregadas por determinados *Class Loaders*, como a seguir:

- Classes de sistema – classes do núcleo da API Java – são carregadas pelo *Class Loaders* primordial.
- Extensões padrão – geralmente encontrados em jre diretório/lib/ext – são carregadas pela *ExtClassLoader*.
- Classes de aplicação – encontradas no caminho da classe, ou conforme especificado pela propriedade *java.class.path* ou a opção *-cp* – são carregadas pela *AppClassLoader*.

Para conferir estes *Class Loaders*, é possível escrever um aplicativo para invocar o método *getClassLoader*. Como resultado o *Class Loaders* primordial retorna *null*, o *ExtClassLoader* retorna *sun.misc.Launcher$ExtClassLoader* e o *AppClassLoader* retorna *sun.misc.Launcher $AppClassLoader*.

7. Considerações de Segurança sobre o Class Loader

Os aspectos de segurança da implementação de um *Class Loader* se dividem em duas grandes categorias: as características essenciais comuns a todos os *Class Loaders*, e funcionalidades adicionais que sejam adequadas para qualquer *Class Loader*.

Existem quatro aspectos que um *Class Loader* deve observar:

1) Verificar a validade do nome do pacote ou da classe
2) Verificar a cache antes de carregar uma classe
3) Verificar o caminho local (ou de sistema) da classe antes de tentar carregar uma classe remota
4) Executar o verificador

Além dos quatro aspectos mencionados, quaisquer restrições adicionais podem ser acrescentadas em um *Class Loader* para servir a qualquer propósito em particular. Isso pode incluir especificações de servidor ou restrições sobre quais servidores o *Class Loader* poderá carregar. Dois recursos são comumente implementados: restrição de certas hierarquias de pacote e manipulação de assinaturas.

7.1. Restrição de hierarquias de pacote

Implementar este recurso implica em controlar o acesso a determinadas propriedades. Isso envolve conhecer sobre o domínio *RuntimePermission*, além de controlar o acesso às propriedades como *user.home* e *os.name.* O domínio *RuntimePermission* controla o acesso a pacotes especificados por meio da passagem dos argumentos *accessClassInPackage.package_name* e *defineClassInPackage.package_name* para o método. O primeiro argumento permite o acesso ao pacote especificado em "*package_name*" através do método *loadClass*() do *Class Loader*. O segundo permite a definição das classes no pacote especificado através do método *defineClass*() do *Class Loader*.

7.2. Manipular assinaturas digitais

A implementar deste recurso envolve a atribuição a uma classe de confiança baseada em assinatura. Isto pode ser feito através do método *setSigners* do *Class Loader*, que deve ser usado para definir uma classe como assinada. Este método é definido como *protected* e *final* e, como tal, não pode ser modificado. Recebe dois argumentos: um objeto Class a ser registrado como assinado e um *array* de *Object*, que geralmente são subclasses da *java.security.cert.Certificate*.

8. Classes do Class Loader

A classe base que define um *Class Loader* é a classe abstrata *java.lang.ClassLoader*:

```
public abstract class ClassLoader
```

Esta classe transforma uma série de *bytecodes* em uma definição de classe. Não define a forma como os *bytecodes* serão obtidos, mas prevê todas as outras funcionalidades necessárias para criar a definição da classe.

A classe preferida para ser usada como base para um carregador de classes é a classe *java.security.SecureClassLoader*, definida como:

```
public class SecureClassLoader extends ClassLoader
```

Esta classe transforma uma série de *bytecodes* em uma definição de classe. Esta classe acrescenta funcionalidade seguras à classe *ClassLoader*, mas não define como os *bytecodes* serão obtidos. Deve-se criar uma subclasse desta.

Há uma terceira classe nesta categoria, a classe *java.net.URL Class Loader*, definida como:

```
public class URL Class Loader extends SecureClassLoader
```

Esta classe carrega classes de maneira segura pela obtenção dos *bytecodes* em um conjunto de URLs. Se as classes forem carregadas através do sistema de arquivos ou a partir de um servidor HTTP, a classe *URL Class Loader* oferece uma definição completa de um *Class Loader*. Além disso, é possível substituir alguns de seus métodos ou modificar a política de segurança de classes que define.

8.1. Implementando Class Loaders

Pela implementação de um *Class Loader*, podemos estender as permissões que são concedidas através de arquivos de política de segurança, bem como utilizar certos recursos de segurança opcionais do *Class Loader*. Antes de criar um exemplo para implementar um *Class Loader*, veremos primeiro certos métodos essenciais na sua utilização e implementação:

```
public Class loadClass(String name)
```

Carregar uma classe identificada pelo argumento passado. Uma *ClassNotFoundException* pode ser lançada se a classe não for encontrada.

```
protected Class findClass(String name)
```

Carregar uma classe especificada pelo argumento passado. O nome será completamente qualificado do pacote da classe (por exemplo, *java.lang.String*).

```
protected final Class defineClass(
  String name,
  byte[] b,
  int off,
  int len) throws ClassFormatError
protected final Class defineClass(
  String name,
  byte[] b,
  int off,
  int len,
  ProtectionDomain protectionDomain) throws ClassFormatError
protected final Class defineClass(
  String name, byte[] b,
  int off,
  int len,
  CodeSource cs) throws ClassFormatError
```

Criar uma classe com base em *bytecodes* do *array* informado. A proteção do domínio associado à classe muda com base no método que está sendo utilizado.

O método a seguir está disponível dentro da classe *SecureClassLoader* e suas subclasses, incluindo a classe *URL Class Loader*.

```
protected PermissionCollection getPermissions(CodeSource cs)
```

Devolver as permissões que devem ser associadas a um determinado código fonte. A implementação padrão deste método chama o método *getPermissions*() da classe *Policy*.

Como exemplo, iremos realizar um exemplo quanto ao uso da classe *URL Class Loader*.

Etapa 1: Verificar se o programa pode acessar a classe em questão. Este é um passo opcional.

Para tanto, chamamos o método *checkPackageAccess*(). Se for preciso modificar outro comportamento do *URL Class Loader*, então não se pode usar o método *newInstance*(). Nesse caso, para utilizar o método *checkPackageAccess*(), deve anular o método *loadClass*() do seguinte modo:

```
public final synchronized Class loadClass(String name, boolean resolve)
   throws ClassNotFoundException {
  // Verificar a permissão para acessar o pacote.
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    int i = name.lastIndexOf('.');
    if (i != -1) {
      sm.checkPackageAccess(name.substring(0, i));
    }
  }
  return super.loadClass(name, resolve);
}
```

Etapa 2: Se houver uma classe previamente definida, o método *loadClass*() de *Class Loader* realiza esta operação. Se o *Class Loader* já carregou esta classe, então, encontra um objeto da classe previamente definido e retorna este.

Esta é a razão pela qual o método *super.loadClass*() é chamado no método *override*.

Etapa 3: Se não houver uma classe previamente definida, o *Class Loader* consulta a classe que este estendeu, para determinar como carregar a classe. Isto é feito através de uma operação recursiva. Deste modo, o *Class Loader* de sistema será chamado antes para carregar a classe.

No nosso exemplo, o carregamento da classe é feito de forma diferente para a superclasse. O método *loadClass*() da classe *ClassLoader* realiza esta operação.

Etapa 4: Consultar o *Security Manager* para saber se o programa está autorizado a criar a classe em questão. Este também é um passo opcional.

No exemplo mostrado, chamamos o método *checkPackageDefinition*(). Para tanto, deve-se fazer um *override* do método *findClass*():

```
protected Class findClass(final String name) throws ClassNotFoundException {
  // Primeiro verifica se tem permissão para acessar o pacote
  SecurityManager sm = System.getSecurityManager( );
  if (sm != null) {
    int i = name.lastIndexOf('.');
    if (i != -1) {
      sm.checkPackageDefinition(name.substring(0, i));
    }
  }
  return super.findClass(name);
}
```

Etapa 5: Ler o arquivo da classe através de um *array* de *bytes*. Isso ocorre no método *findClass*().

O *URL Class Loader* realiza esta operação através da consulta dos URLs que foram passados para o seu construtor. Se for preciso ajustar a forma de leitura dos *bytes* da classe, deve-se usar a classe *SecureClassLoader*.

Etapa 6: Criar o domínio de proteção adequado para a classe.

O *URL Class Loader* irá criar um código fonte para cada classe com base na URL a partir da qual a classe foi carregada e sua assinatura (se houver). As permissões associadas a essa classe serão obtidas através do método *getPermissions*() da classe *Policy*, que por padrão irá retornar as permissões lidas nos arquivos de política de segurança ativos. Além disso, o *URL Class Loader* irá adicionar outras permissões para esse conjunto.

Se a URL tem um protocolo de arquivo, este deve especificar uma autorização de arquivo que permita que todos os arquivos que descendem do caminho URL sejam lidos. Por exemplo, se a URL é *file://xyz/classes/* então, um arquivo com um nome de permissão */xyz/classes/* e uma lista de ação de leitura serão adicionados ao conjunto de permissões. Se a URL é um arquivo jar (*file://xyz/MyApp.jar*), o nome do arquivo de permissão será a própria URL.

Se a URL possui um protocolo HTTP, então será adicionada uma permissão *socket* para se conectar ou aceitar uma conexão da máquina. Por exemplo, se a URL é *http://someurl/classes/* então uma permissão *socket* com o nome *someurl:1* e uma lista de ação de conectar e aceitar serão adicionadas.

Caso desejamos associar diferentes permissões a uma classe, então devemos realizar um *override* com o método *getPermissions*(). Por exemplo, para que a regra anterior seja aplicada e fosse permitido que a classe finalizasse a máquina virtual, devemos usar o seguinte método:

```
protected PermissionCollection getPermissions(CodeSource codesource)   {
  PermissionCollection pc = super.getPermissions(codesource);
  pc.add(new RuntimePermission("exitVM"));
}
```

Podemos mudar completamente as permissões associadas à classe, substituindo completamente a classe *Policy*, construindo uma nova coleção de permissões neste método, ao invés de chamar *super.getPermissions*(). O *URL Class Loader* irá utilizar qualquer permissão retornada através do método *getPermissions*() para definir o domínio de proteção que será associado à classe.

Etapa 7: Definir a classe, verificá-la e resolvê-la.

Estes passos são tratados internamente pelo *URL Class Loader*.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.