# Rozszerzenia architektury bezpieczeństwa

W ogólności bezpieczeństwo aplikacju Javy realizowane jest poprzez szereg mechanizmów.

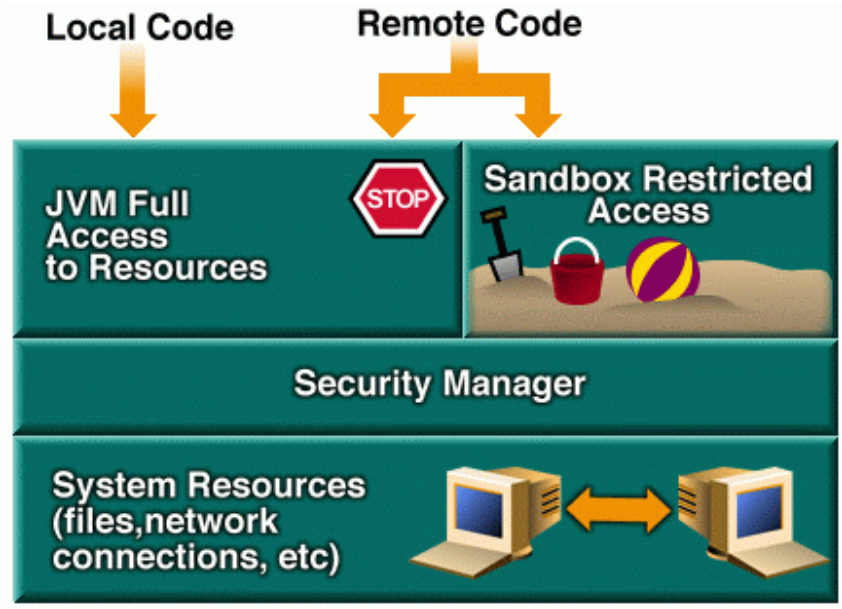
- W pierwszym rzędzie stoi sam język Javy (automatyczne zarządzanie pamięcią, odświeżanie, sprawdzanie zakresów dla ciągów oraz tablic.
- W drugim rzędzie mamy kompilator (który weryfikuje kody bajtowe, zapewniając wykonywalność tylko kodów Javy)
- Następnie bezpieczeństwo realizowane jest przez poprawne definicje ładowaczy klas.
- Na końcu mamy jeszcze klasę SecurityManager, która ogranicza wykonywanie akcji przez nieznany kod Javy.
- Możliwa jest również weryfikacja kodów za pomocą podpisów cyfrowych.

# Piaskownica (sand box)

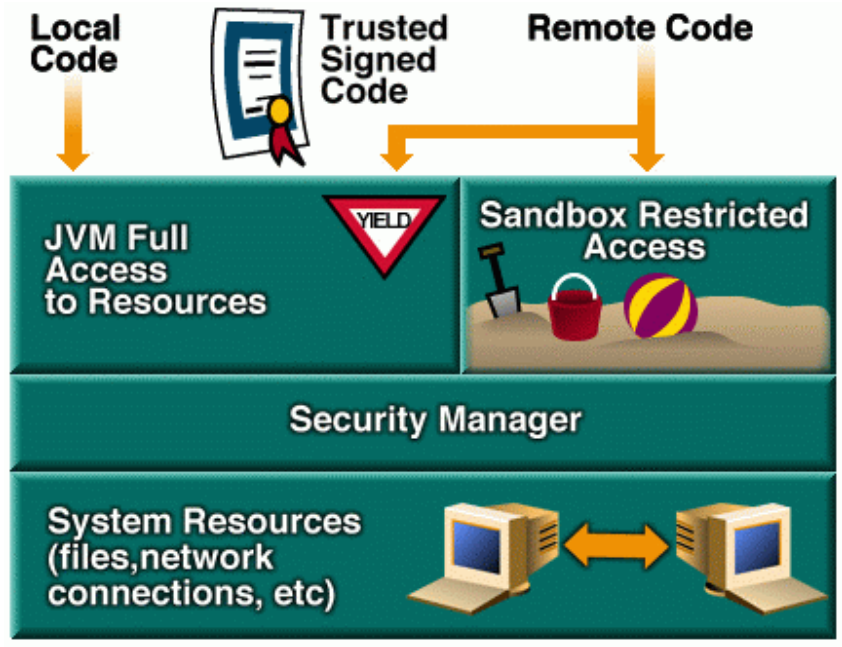
W modelu bezpieczeństwa Javy zajęto się zabezpieczeniem użytkowników przed wrogimi programami ściągniętymi z niepewnych miejsc sieci Internet. Zabezpiecznie to zrealizowano za pomocą koncepcji piaskownicy ("sandbox"), w której działają programy Javy. W piaskownicy program może robić wszystko, co mieści się w jej granicach. Nie może natomiast podejmować akcji wykraczających poza te granice. Dla przykładu ograniczenia nałożone przez piaskownice, w której działają aplety niepewnego pochodzenia obejmują m.in.:

- czytanie i pisanie na lokalnym dysku
- tworzenie połączeń sieciowych do hostów innych niż host, z którego aplet pochodzi
- tworzenie nowych procesów
- ładowanie nowej dynamicznej biblioteki i bezpośrednie wywołania metod natywnych

**JDK 1.0 Security Model:**

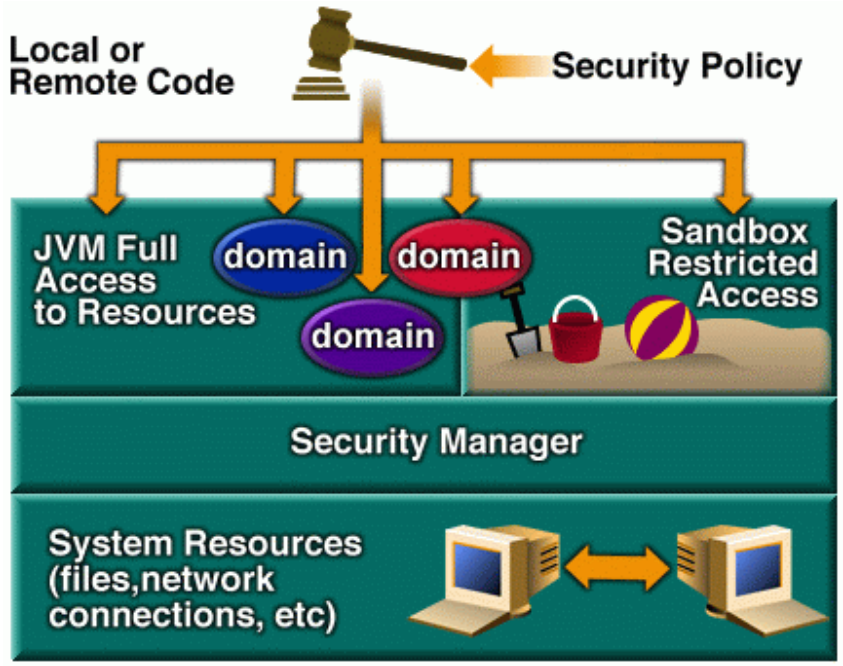



**JDK 1.1 Security Model:**




JDK 1.1 wprowadził koncept podpisanego apletu. Aplety takie dostarczane są w cyfrowo podpisanych plikach jar. Aplet taki może być traktowany jako kod lokalny (ze wszystkimi uprawnieniami), jeśli weryfikacja kluczy zakończyła się poprawnie. Aplety niepodpisane pracują zaś w trybie z brakiem zaufania (siedzą w piaskownicy)

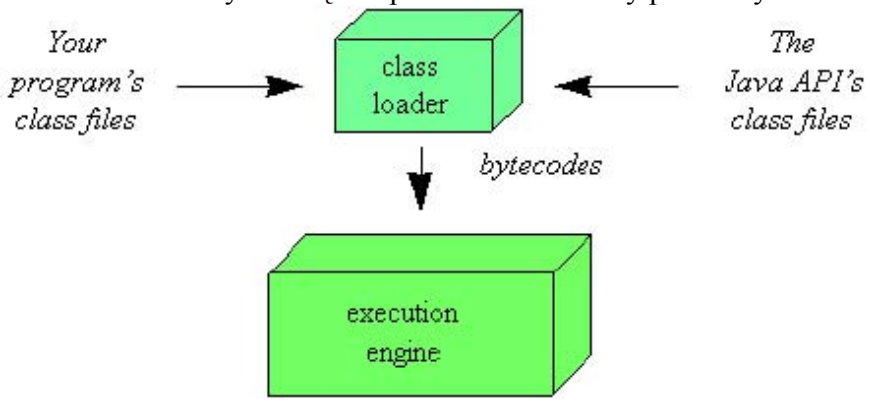
**JDK 1.2 Security Model:**




JDK 1.2 wprowadził szereg usprawnień (polityka bezpieczeństwa, zezwolenia, domeny)

# Architektura ładowacza klas

W JVM ładowacze klas odpowiedzialni są za ładowanie danych binarnych kodów klas i interfejsów, które wykorzystywane są w działającym programie. Na diagramie poniżej przedstawiono "the class loader", który w rzeczywistości może zawierać więcej niż jeden ładowaczy klas wewnątrz JVM. JVM posiada elastyczną architekturę ładowaczy klas, dzięki której ładowanie klas może odbywać się na sposób zdefiniowany przez użytkownika.




Architektura ładowaczy klas (funkcje)

Aplikacja Javy używa dwóch typów ładowaczy klas: pierwotny ładowacz klas ("primordial" class loader) oraz obiekty ładowaczy klas. Pierwotny ładowacz klas (jest tylko jeden) jest częścią implementacji JVM. Jeśli JVM zaimplementowano jako program w języku C w danym systemie operacyjnym, to pierwotny ładowacz klas jest częścią tego programu. Pierwotny ładowacz klas ładuje zaufane klasy, w tym klasy Java API rezydujące zazwyczaj na lokalnym dysku.

W czasie wykonywania programu Javy, aplikacja może zainstalować obiekty ładowaczy klas, które ładują klasy w sposób arbitralny, np. przez ściąganie plików klas z sieci. Klasy takich ładowaczy napisane są w Javie jak każde inne klasy. Po skompilowaniu i załadowaniu na JVM służą one do tworzenia instancji ładowaczy klas.

JVM traktuje wszystkie klasy ładowane przez pierwotnego ładowacza klas jako klasy zaufane, niezależnie od tego, czy są częścią Java API, czy też nie. Jednak klasy ładowane poprzez obiekty ładowaczy klas zawsze widziane są jako klasy podejrzane. Domyślnie klasy te rozważane są jako klasy bez kredytu zaufania.

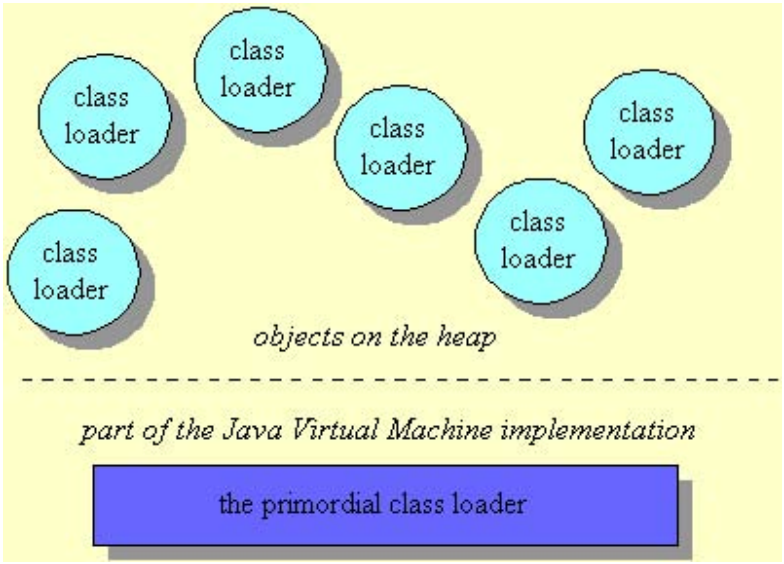



Figure 2. Architektura ładowaczy klas (pochodzenie)

Wykonywanie aplikacji Javy może odbywać się w sposób dynamiczny. W chwili pisania danej aplikacji mogą jeszcze nie istnieć klasy, które później, podczas wykonywania aplikacji staną się dostępne i zostaną załadowane. Ładowanie takich klas może odbywać się właśnie przez obiekty ładowaczy klas. Klasy ładowaczy klas zaimplementowane w Javie przez twórcę aplikacji, mogą działać w sposób praktycznie dowolny. Można na przykład napisać takiego ładowacza klas, który klasy ładowane wyszukiwał będzie w locie w zdalnym repozytorium klas.

## Ładowacze klas i przestrzenie nazw

Klasy ładowane są na skutek:

- rozwiązania pewnej klasy, która jest od nich zależna
- wywołania Class.forName(className); bez określenia procedury ładującej
- wwołanie określonej procedury ładującej loader.loadClass(className);

Każda klasa załadowana przez JVM pamiętana jest z własną nazwą oraz nazwą ładowacza, który ją załadował. Kiedy klasa już załadowana odwołuje się pierwszy raz do innej klasy, JVM sprawdza, czy ta inna klasa została załadowana przez tego samego ładowacza klas, co ładowacz klasy tworzącej odwołanie. Na przykład, jeśli JVM ładuje klasę `Volcano` wykorzystując konkretny ładowacz klas, to każda inna klasa, do której `Volcano` się odwołuje, również będzie ładowana tym samym ładowaczem klas. Jeśli `Volcano` odwołuje się do klasyu `Lava` (np. przez wywołanie jednej z metod klasy `Lava`), to JVM spróbuje załadować klasę `Lava` tym samym ładowaczem, którym załadowano klasę `Volcano`. Klasa `Lava` zwrócona przez ładowacza klas dynamicznie będzie zlinkowana z klasą `Volcano`.
Konsekwencją takiej strategii jest to, że dane klasy domyślnie widzą tylko te klasy, które załadowane zostały tym samym ładowaczem. W ten sposób wewnątrz pojedynczej aplikacji Javy tworzy się przestrzeń nazw klas (unikalne nazwy klas załadowane przez ten sam ładowacz). W

JVM dla każdego ładowacza klas utrzymywana jest przestrzeń nazw załadowanych przez niego klas.
Gdy JVM załaduje już jakąś klasę w konkretną przestrzeń nazw, kolejne ładowanie klasy o tej samej nazwie w tą samą przestrzeń staje się niemożliwe. Jednak ładowanie klas o tej samej nazwie jest możliwe, tylko że muszą one być załadowane do innych przestrzeni nazw (a więc przez innych ładowaczy klas).
Aplikacja Javy może tworzyć instancje ładowaczy klas tej samej klasy lub wielu różnych klas. Klasy załadowane przez różnych ładowaczy klas nie mają prawa dostępu do swoich imienniczek ładowanych innym ładowaczen, jeśli aplikacja jawnie na to nie zezwoli.
Gdy pisze się aplikację Jawy, klasy wczytywane z różnych źródeł powinny być umieszczane w różnych przestrzeniach nazw, aby w pełni wykorzystać możliwości kontroli kodu pochodzącego o różnym pochodzeniu.

## Ładowacze klas dla apletów

Przykładem użycia ładowaczy klas tworzonych dynamicznie są przeglądarki internetowe, które używają obiektów ładowaczy klas do ściągania plików klas dla apletów z różnych miejsc w sieci. Przeglądarka internetowa odpala aplikację Javy, która instaluje obiekt ładowacz klas – zazwyczaj jesto to *applet class loader* – który wie, jak zażądać pliku klasy z serwera HTTP.
Aplety są przykładem dynamicznego rozszerzenia programu. Gdy aplikacja Javy startuje, nie zna ona jeszcze, jakie pliki klas będą ściągnięte przez przeglądarkę z sieci. Okazuje się to dopiero podczas uruchamiania aplikacji, gdy przeglądarka zajdzie strony zawierające aplety.
Zazwyczaj aplikacja Javy wystartowana przez przeglądarkę tworzy różne obiekty ładowaczy klas apletów (applet class loader objects) dla każdej lokalizacji w sieci, z której ściągane są pliki klas. W efekcie, pliki klas z różnych źródeł ładowane są różnymi ładowaczami, a więc umieszczane są w różnych przestrzeniach nazw wewnątrz hosta, na którym aplikacja Javy działa. Dzięki temu klasy z różnych źródeł są izolowane i kod jakiegoś złośliwego apletu nie ma dostępu do innych klas ściągniętych z innych miejsc w sieci.

## Współpraca pomiędzy ładowaczami klas

W przypadku każdego programu Javy standardem jest wykorzystanie przynajmniej trzech procedur ładowania klas:

- początkowej (ładowanie klas z pliku rt.jar)
- rozszerzonej (ładowanie klas z katalogu jre/lib/ext)
- systemowej (lub aplikacji – ładowanie klas użytkownika ze ścieżek classpath).

Często zdarza się, że dany obiekt ładowacza klas korzysta z innych ładowaczy klas, tworząc w ten sposób hierarchię podlegania (gdzieś tam na końcu tego łańcucha jest pierwotny ładowacz klas).

Rozważmy przykład aplikacji Javy instalującej ładowacz klas, którego szczególny sposób ładowania klas polega na ściąganiu kodów klas z sieci. Załóżmy, że w czasie działania aplikacji zaszło żądanie załadowania klasy `Volcano`. Sposobem na napisanie takiego ładowacza klas jest zapewnienie, aby w pierwszej kolejności ładowacz poprosił pierwotnego ładowacza klas o znalezienie i załadowanie klasy z zaufanego repozytorium. W tym przypadku, ponieważ `Volcano` nie jest częścią Java API, można przypuszczać, że pierwotny ładowacz klas nie będzie mógł klasy `Volcano` odszukać. Gdy tak się stanie, ładowacz klas użytkownika może spróbować

załadować`Volcano` na swój własny sposób, tj. ściągnąć go z sieci. Zakładając, że takie ściąganie zakończyło się sukcesem, klasa `Volcano` będzie mogła pełnić swą rolę w kolejnych etapach wykonywania aplikacji.
Kontynuując przykład, załóżmy, że jakiś czas potem metoda klasy `Volcano` jest wywołana po raz pierwszy, i że metoda ta odwołuje się do klasy `String` z Java API. Ponieważ jest to pierwsze odwołanie w czasie działania programu, JVM poprosi ładowacza klas (tego, który załadował `Volcano`) o załadowanie klasy `String`. Jak wcześniej, ładowacz klas najpierw przekaże prośbę tą do pierwotnego ładowacza klas. Tym razem ładowacz ten będzie potrafił załadować klasę `String`, przekazując ją następnie do obiektu ładowacza klas, który załadował Volcano.
Pierwotny ładowacz klas najprawdopodobniej jednak nie będzie musiał ładować klasy `String`, gdyż `String`jest jedną z fundamentalnych klas używanych w programach Javy, ładowaną wcześniej niż klasa Volcano. Prawdopodobnie pierwotny ładowacz klas przekaże klasę `String` wczytaną już wcześniej z repozytorium. Ponieważ ładowanie klasy `String` kończy się sukcesem, klasa ta nie będzie poszukiwana w zasobach internetowych. Od chwili załadowania, kiedykolwiek klasa `Volcano` odwoływać się będzie do klasy `String`, użyta zostanie klasa wczytana.

## Ładowacze klas w piaskownicy

Architektura ładowaczy klas przyczynia się do utrzymania granic piaskownicy na dwa sposoby:

1. zabezpiecza „dobry" kod przed kodem „złym" (blokowane są odwołania do klas wewnątrz innych klas).
2. sprawuje straż nad granicą bibliotek zaufanych klas (blokowane są ładowania klas).

## Przestrzeń nazw i ochrona

Przestrzeń nazw jest zbiorem unikalnych nazw załadowanych klas, zarządzanym przez JVM. Ograniczenia wynikające z ram tej przestrzeni umożliwiają realizację pierwszego ze sposobów utrzymania granic piaskownicy.
Klasy z różnych przestrzeni nie mogą oddziaływać wzajemnie na siebie, jeśli jawnie nie dostarczy się im mechanizmów takiej iterakcji. Dlatego, aby zabezpieczyć kod klas zaufanych, warto jest utworzyć własnych ładowaczy klas.

## Zarządzanie pakietami z ograniczeniami

Java zezwala przydzielić każdej klasie z tego samego pakietu zezwolenia dostępu, które nie są zapewnione dla klas spoza pakietu. Dlatego, jeśli ładowacz klas otrzyma zlecenie załadowania klasy, której nazwa sugeruje, że jest to część Java API (np. klasa o nazwie `java.lang.Virus`), ładowacz klas musi zachować szczególną ostrożność. Jeśli taka klasa zostanie załadowana, może ona otrzymać specjalne zezwolenia dostępu do zaufanych klas z pakietu `java.lang`.
Wniosek: należy pisać ładowaczy klas w taki sposób, aby odrzucali oni żądania od klas deklarujących siebie jako część Java API (czy też część każdej innej zaufanej biblioteki), a jednak nie należących do lokalnego, zaufanego repozytorium. Innymi słowy, jeśli po przekazaniu przez ładowacza klas zlecenia ładowania klasy do pierwotnego ładowacza klas, ten drugi klasy nie znajdzie, ładowacz klas powinien sprawdzić, czy ładowana klasa nie deklaruje siebie jako część zaufanego pakietu. Jeśli okaże się, że tak właśnie jest, być może należałoby zgłosić wyjątek SecurityException.

## Dozór nad zabronionymi pakietami.

Może się zdarzyć, że część klas w zaufanym repozytorium powinna być ładowana pierwotnym ładowaczem klas, a nie żadnym innym. Np. załóżmy, że mamy stworzony pakiet `absolutepower` zainstalowany w lokalnym repozytorium, które osiąga pierwotny ładowacz klas. Załóżmy również, że klasy załadowane innymi ładowaczami klas nie powinny ładować klasy z pakietu `absolutepower`. W takim przypadku implementacja ładowacza klas w pierwszym kroku powinna sprawdzać, czy klasa zgłaszająca konieczność załadowania innej klasy nie deklaruje siebie jako elementem pakietu `absolutepower`. Jeśli klasa jest zadeklarowana jako element pakietu, powinno się wtedy zgłosić SecurityException, a nie przekazywać żądanie ładowania klasy do ładowacza pierwotnego.

Jedynym sposobem na rozpoznanie przynależności danej klasy do pakietu jest analiza jej nazwy. Dlatego bezpieczny ładowacz klas musi posiadać listę nazw pakietów oraz ocenę ich metod dostępu (restricted and forbidden packages). Wracając do przykładu, ponieważ nazwa `java.lang.Virus` sugeruje pakiet `java.lang`, a nazwa `java.lang` występuje na liście „restricted packages", dlatego więc ładowacz powinien zgłosicć wyjątek. Podobnie, jeśli nazwą potencjalnie ładowanej klasy będzie `absolutepower.FancyClassLoader` sugerująca pochodzenie klasy z pakietu `absolutepower`, oraz jeśli pakiet `absolutepower`

jest na liście „forbidden packages” ładowacz klas powinien zgłosić wyjątek SecurityException.

## Wskazówki do tworzeniu ładowacza klas

A common way to write a security-minded class loader is to use the following four steps:

1. If packages exist that this class loader is not allowed to load from, the class loader checks whether the requested class is in one of those forbidden packages mentioned above. If so, it throws a security exception. If not, it continues on to step two.
2. The class loader passes the request to the primordial class loader. If the primordial class loader successfully returns the class, the class loader returns that same class. Otherwise it continues on to step three.
3. If trusted packages exist that this class loader is not allowed to add classes to, the class loader checks whether the requested class is in one of those restricted packages. If so, it throws a security exception. If not, it continues on to step four.
4. Finally, the class loader attempts to load the class in the custom way, such as by downloading it across a network. If successful, it returns the class. If unsuccessful, it throws a "no class definition found" error.

```
/*
 * SimpleClassLoader.java - a bare bones class loader.
 *
 * Copyright (c) 1996 Chuck McManis, All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies.
 *
 * CHUCK MCMANIS MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT
THE
 * SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED,
INCLUDING
 * BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. CHUCK
MCMANIS
 * SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A
RESULT
 * OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS
DERIVATIVES.
 */

import java.util.Hashtable;
import util.ClassFile;
import java.io.ByteArrayInputStream;
import java.io.FileInputStream;

public class SimpleClassLoader extends ClassLoader {
    private Hashtable classes = new Hashtable();

    public SimpleClassLoader() {
```

```
    }

    /**
     * This sample function for reading class implementations reads
     * them from the local file system
     */
    private byte getClassImplFromDataBase(String className)[] {
            System.out.println("        &gt;&gt;&gt;&gt;&gt;&gt; Fetching the implementation
of "+className);
            byte result[];
            try {
                FileInputStream fi = new FileInputStream("store\\"+className+".impl");
                result = new byte[fi.available()];
                fi.read(result);
                return result;
            } catch (Exception e) {

                /*
                 * If we caught an exception, either the class wasnt found or it
                 * was unreadable by our process.
                 */
                return null;
            }
    }

    /**
     * This is a simple version for external clients since they
     * will always want the class resolved before it is returned
     * to them.
     */
    public Class loadClass(String className) throws ClassNotFoundException {
        return (loadClass(className, true));
    }

    /**
     * This is the required version of loadClass which is called
     * both from loadClass above and from the internal function
     * FindClassFromClass.
     */
    public synchronized Class loadClass(String className, boolean resolveIt)
            throws ClassNotFoundException {
        Class result;
        byte  classData[];

        System.out.println("        &gt;&gt;&gt;&gt;&gt;&gt; Load class : "+className);

        /* Check our local cache of classes */
        result = (Class)classes.get(className);
        if (result != null) {
            System.out.println("        &gt;&gt;&gt;&gt;&gt;&gt; returning cached result.");
```

```
            return result;
        }

        /* Check with the primordial class loader */
        try {
            result = super.findSystemClass(className);
            System.out.println("        &gt;&gt;&gt;&gt;&gt;&gt; returning system class (in
CLASSPATH).");
            return result;
        } catch (ClassNotFoundException e) {
            System.out.println("        &gt;&gt;&gt;&gt;&gt;&gt; Not a system class.");
        }

        /* Try to load it from our repository */
        classData = getClassImplFromDataBase(className);
        if (classData == null) {
            throw new ClassNotFoundException();
        }

        /* Define it (parse the class file) */
        result = defineClass(classData, 0, classData.length);
        if (result == null) {
            throw new ClassFormatError();
        }

        if (resolveIt) {
            resolveClass(result);
        }

        classes.put(className, result);
        System.out.println("        &gt;&gt;&gt;&gt;&gt;&gt; Returning newly loaded
class.");
        return result;
    }
}
```