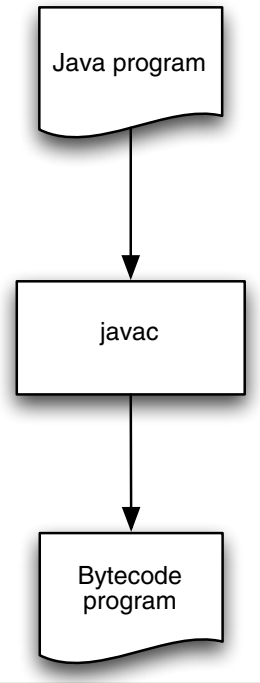
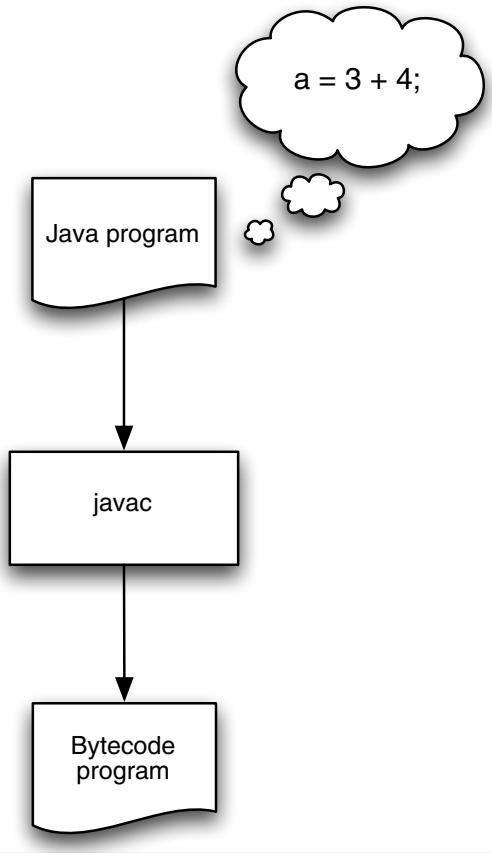
Java Virtual Machine

- The Java P-code is called "Bytecode".
- Bytecode instructions are executed by a "Java Virtual Machine".
- The JVM is a "stack machine" — that is a machine where all computations are performed on a stack.

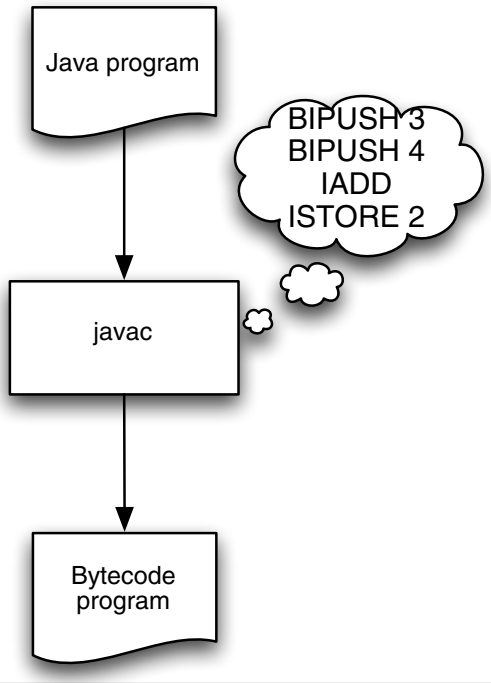
Java compilation



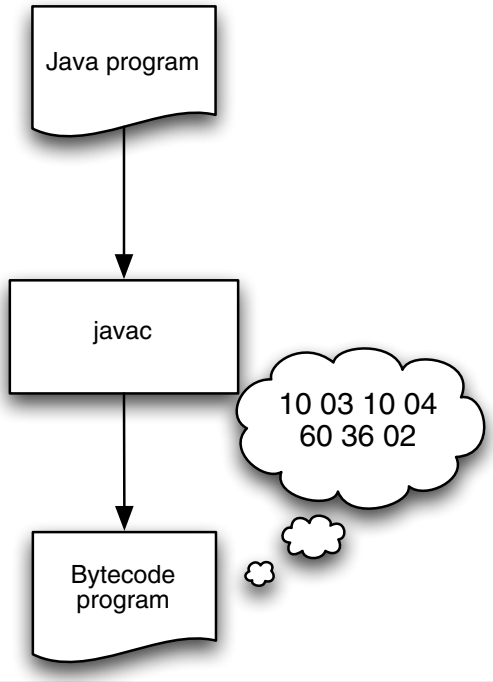
Java compilation



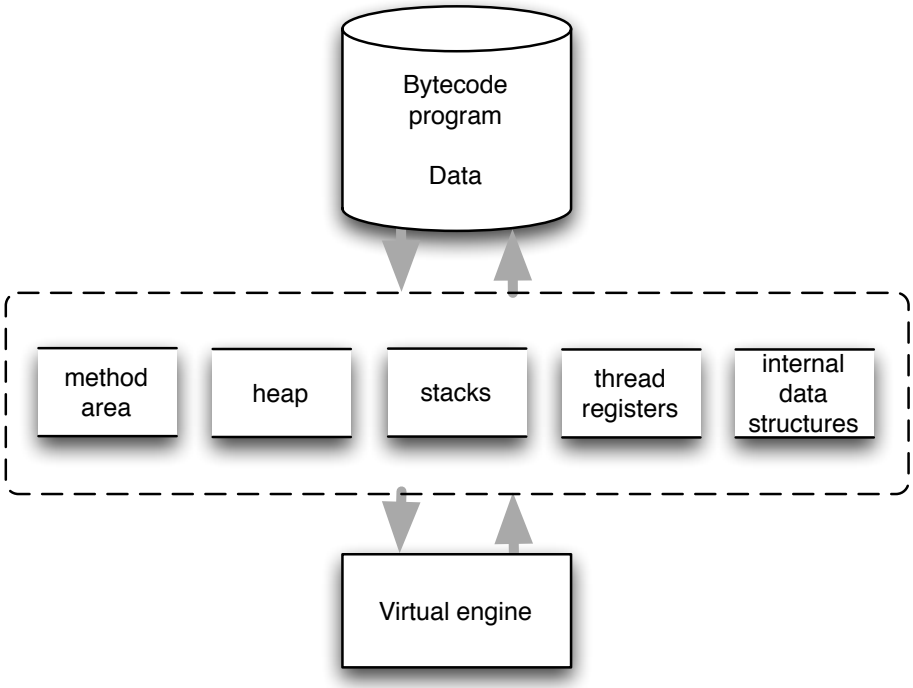
Java compilation



Java compilation



JVM architecture



Data areas

A method area containing:

Constants, Methods, Class variables.

A number of stacks (one for each thread) each containing:

A number of frames (one for each running method) each containing:

Local variables, Method parameters, Operand stack.

Administrative information allowing a called method to return to where it was called from, returning a return value.

A number of program counters (one for each thread).

A heap containing:

Instance information (referenced class and instance variables).

Internal data structures:

Used internally by the virtual machine.

Java sikkerhed

Java betragtes som en platform på linie med et operativsystem, da oversatte Javaprogrammer kan udføres på alle systemer med en kompatibel Java Virtual Machine (JVM).

Som ved operativsystemer har sikkerhed været en væsentlig faktor ved designet af platformen.

Der er nogle forskelle mellem JVM og et traditionelt operativsystem:

- Det er overladt til det underliggende operativsystem at håndtere brugerautentifikationen.
- Adgang til det underliggende operativsystem og de faciliteter det leverer kan pålægges visse restriktioner af den virtuelle maskine.

Det vil sige at der er lagt mindre vægt på brugervalidering og brugerautorisation, og mere vægt på kodevalidering, kodeautentificering og kodeautorisation.

Betroet og ikke betroet kode

Java håndterer **betroet** og ikke betroet kode forskelligt.

Hvad der er betroet og hvad der ikke er betroet har været forskelligt i de tidlige versioner af Java.

Her anvendes følgende begreber:

- Lokale og "*remote*" programmer.
- En **sandkasse**, hvori ikke betroet kode udføres.
- Mulighed for at **signere** kode.
- Mulighed for at opstille **sikkerhedspolitikker**.

Lokale og *remote* programmer

Java kan hente kode via:

- Sin lokale *classpath*.
- URL der refererer kode placeret på en anden maskine.

Som udgangspunkt er lokal kode betroet, og remote kode ikke betroet.

Java sandbox

Her forhindres koden som udgangspunkt i at:

- Læse og skrive til disk.
- Forbinde via netværk til andre maskiner end der hvor koden er hentet fra.
- Oprette nye processer.
- Dynamisk loading af biblioteker.

Som udgangspunkt er ikke betroet kode underlagt restriktionerne i sandkassen.

Signeret kode

Som udgangspunkt er signeret kode betroet — også når koden er remote.

Dette giver en "alt eller intet" adgang.

Sikkerhedspolitikker

Det er muligt at tildele ikke betroet kode yderligere rettigheder, således at kode kan tildeles netop de rettigheder der er behov for — og ikke andre unødvendige rettigheder.

Kode kan underlægges sikkerhedspolitikker med forskellig "scope":

- Sikkerhedspolitik der gælder for al Java kode der udføres på maskinen.
- SIkkerhedspolitik der gælder for al Java kode der udføres af en specifik bruger på maskinen.
- Sikkerhedspolitik der gælder for udførslen af en enkelt Java proces.

Disse sikkerhedspolitikker kan justere rettighederne for Javakode afhængigt af blandt andet:

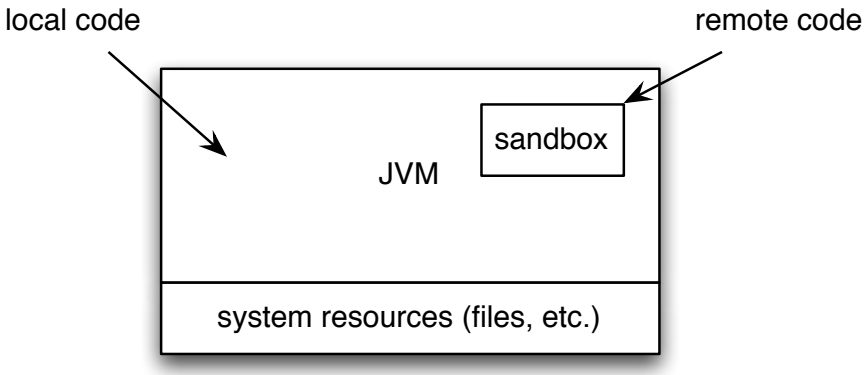
- hvorfra koden loades (via en bestemt URL).
- hvilke ressourcer der ønskes anvendt.

Dette giver mulighed for at finjustere rettigheder for specifik kode.

Original java security model

When running Java code the JVM distinguishes between:

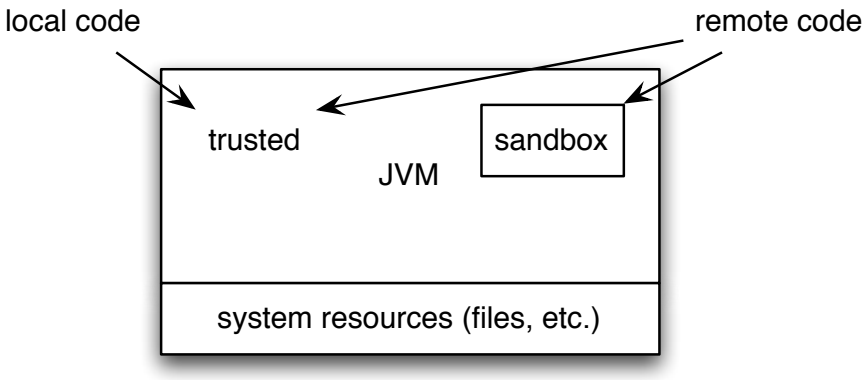
- Local code
- Remote code



Java 1.1 security model

When running Java code the JVM distinguishes between:

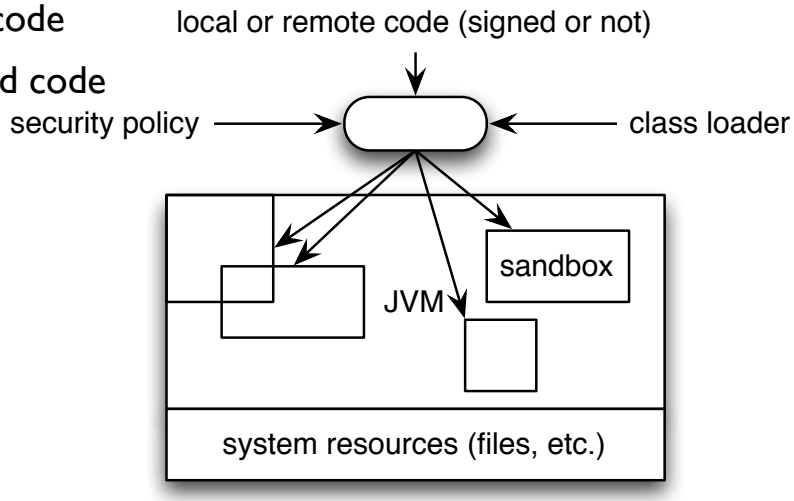
- Trusted code
- Untrusted code



Java 2 security model

When running Java code the JVM looks at the policy file for both local and remote code and determines if specific rights should be given to both signed and unsigned:

- Trusted code
- Untrusted code



Here the unmodified SecurityManager and ClassLoader uses the information in the security policy files to determine if a class can be loaded.

Java security

Without the sandbox, the original Java model would have allowed anybody to execute any code on the end users machine. As you way have heard quite a lot of malicious code circulates, such as:

- Vira
- Worms
- Backdoors
- Key/screen loggers
- Troyan horses
- Etc.

Java security

Such attacks can be categorised in different ways, such as:

- System Modification
- Privacy Invation
- Denial of Service
- Antagonism

Java sandbox

Sandkassen i Java styres af:

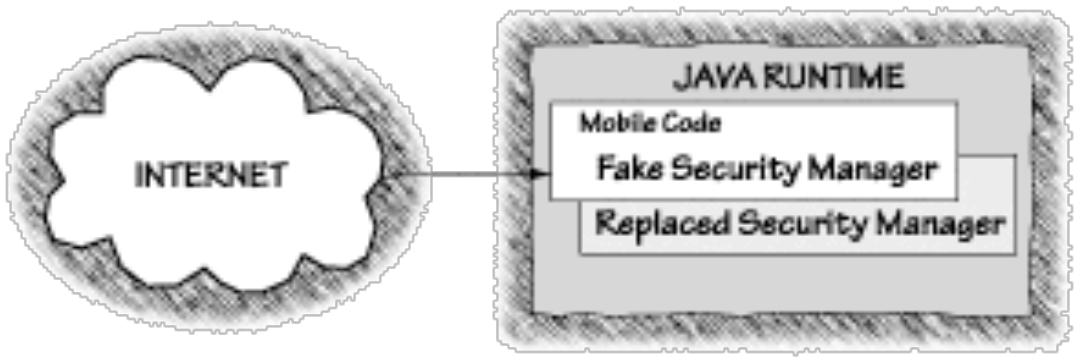
- En "*class loader*" arkitektur.
- En "*class file verifier*".
- Et antal sikkerhedsmekanismer indbygget i Java og JVM.
- En "*security manager*" og understøttelse heraf i Java API'et.

De tre første forsøger at beskytte den virtuelle maskine, mens den fjerde forsøger at beskytte det underliggende operativsystem.

Class loader arkitektur

Den del af JVM der henter klassefiler ind i lageret således at metoder i disse kan udføres.

Det primære formål er at forhindre at der hentes kode ind der erstatter den kode der håndhæver sikkerheden.



http://www.securingjava.com/chapter-two/chapter-two-7.html

Class loader arkitektur

Dette formål kan udspecificeres som:

- **kodeisolation**: forhindrer skadelig kode i at påvirke anden Java kode under udførsel.
- **bibliotekisolation**: isolerer de medfølgende Java pakker fra anden kode.
- **beskyttelsesdomæner**: placerer kode i forskellige kategorier der styrer hvad koden har lov til at gøre.

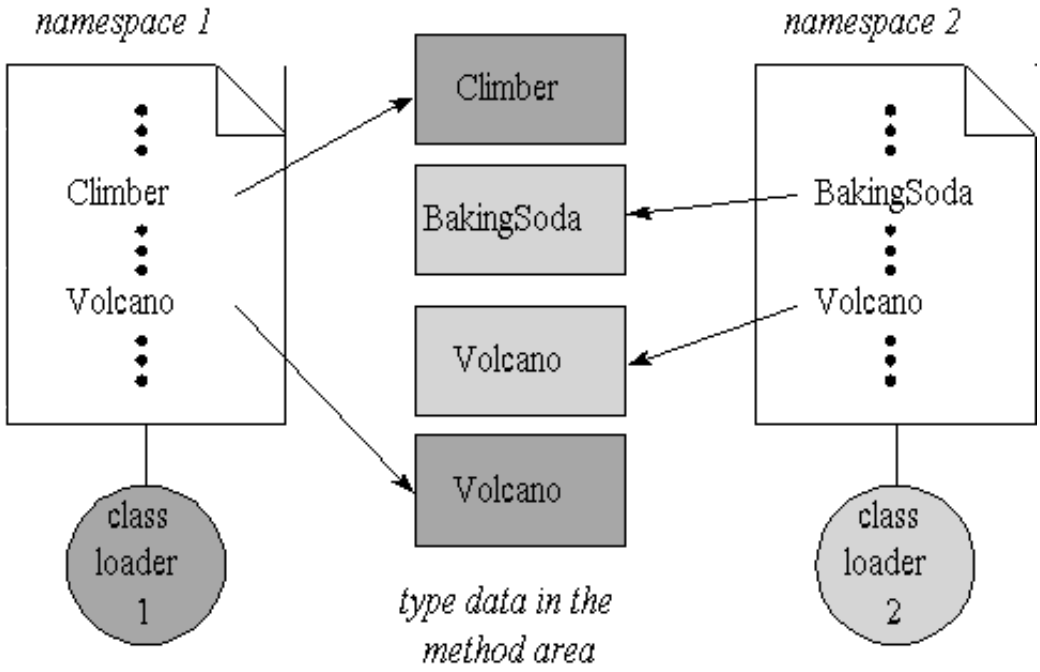
Class loaders

Java benytter sig af et antal class loaders, hvor hver class loader opretter et "*name-space*", hvori denne class loader placerer al kode den henter ind.

Det er kun muligt at hente en enkelt klasse med et givet navn ind i et enkelt name-space, men det er muligt at hente flere klasser med det samme navn ind i forskellige name-spaces.

Koden der hentes ind af en class loader kan starte yderligere class loaders, således at der opstår et hierarki af class loaders.

En class loader kan derfor enten selv hente en klassefil ind eller delegere denne opgave til en class loader den har startet.



http://www.artima.com/insidejvm/ed2/security.html

Class loader typer

To typer class loaders:

- En bootstrap class loader

  Startes af systemet, og henter de centrale dele af Java ind (core Java API).

  Har de adgangsrettigheder der sættes af operativsystemet ved start af programmet.

- Et antal user-defined class loaders

  Instans af en underklasse af den abstrakte klasse `ClassLoader`.

  Typisk begrænset i forhold til den klasse den nedarver fra, og kan ikke have færre begrænsninger i forhold til den startende class loader.

Ved start af et Java program sørger bootstrap class loaderen for at hente de centrale Java pakker, og starter derefter yderligere class loaders med stadig færre rettigheder.

Class loader klasser

Der er som standard defineret et antal varianter af user-defined class loaders:

- Generisk abstrakt Class Loader: java.lang.ClassLoader
- Secure Class Loader: java.security.SecureClassLoader

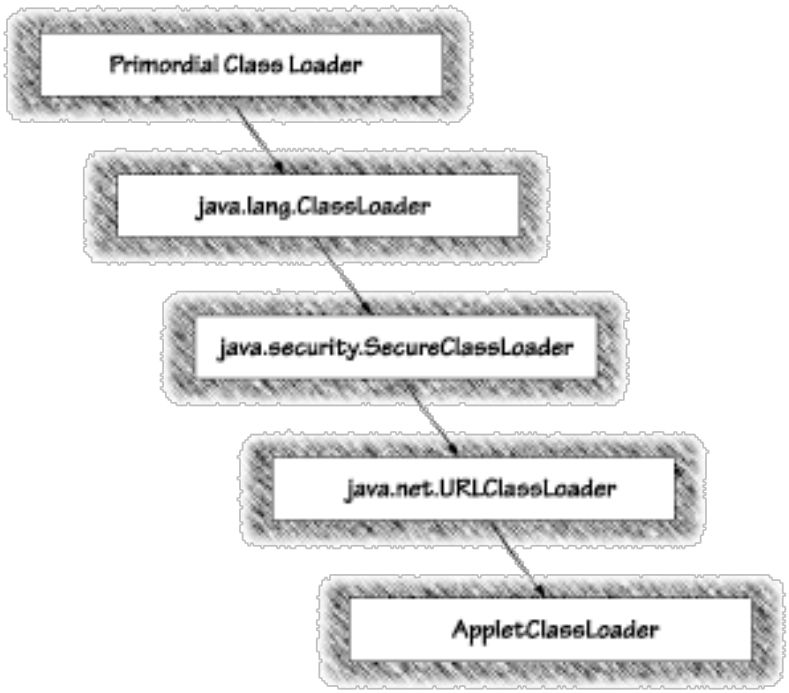
  Bruges til at hente klasser i programmets class-path (java.app.class.path).
- RMI Class Loader: java.rmi.server.RMIClassLoader

  Bruges til at hente klasser via RMI (angivet i rmi.server.codebase).
- URL Class Loader: java.net.URLClassLoader

  Bruges til at hente klasser via en URL.
- Applet Class Loader: underklasse af java.net.URLClassLoader

  Bruges til at hente (applet) klasser via en URL (angivet i CODEBASE i en <APPLET> tag).

Derudover kan en programmør selv definere yderligere class loaders ved nedarvning.

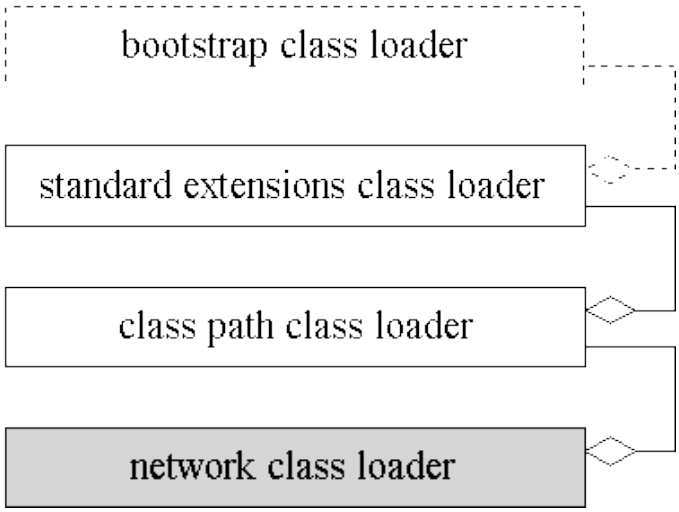


http://www.securingjava.com/chapter-two/chapter-two-7.html

Class loader ved start

Ved start af et Java program anvendes bootstrap class loaderen til at hente de centrale dele af Java ind.

Derefter startes yderligere class loaders med relevante restriktioner efter behov.



http://www.artima.com/insidejvm/ed2/security.html

Brug af klasser efter indhentning

Når en klasse skal bruges efter indhentning kan det risikeres, at en klasse med det angivne navn er hentet ind i flere name-spaces, hvorfor der altid gås op i hierarkiet af class loaders indtil bootstrap class loaderen nås.

Hvis denne har hentet en klasse med det angivne navn ind bruges denne.

Ellers gås tilbage mod det name-space hvorfra klassen blev forsøgt anvendt et skridt ad gangen indtil der findes en klasse med det angivne navn.

Det vil sige, at der altid anvendes klassen, der ligger højest i hierakiet af name-spaces.

På denne måde undgås at lavere rangerende class loaders kan hente mindre betroede klasser ind der erstatter mere betroede klasser.

Bibliotekisolation

Når man definerer nye klasser kan man vælge at tilføje disse til en pakke, hvorved man får adgang til visse interne dele af andre klasser i pakken.

Hvis man derfor definerer at en ny klasse skal tilhøre `java.lang` pakken skulle man kunne få adgang til visse interne faciliteter i Java.

Hvis man endvidere henter denne kode remote, kan man måske endda inficere en maskine ved at overrule almindeligt benyttede dele af `java.lang` pakken.

Ved at kræve at de centrale pakker kun kan hentes af bootstrap class loaderen, og at alle klasser der hentes via en URL skal hentes via en `URLClassLoader` (eller en class loader der er afledt deraf), kan man forhindre denne form for infektion.

Protection domains

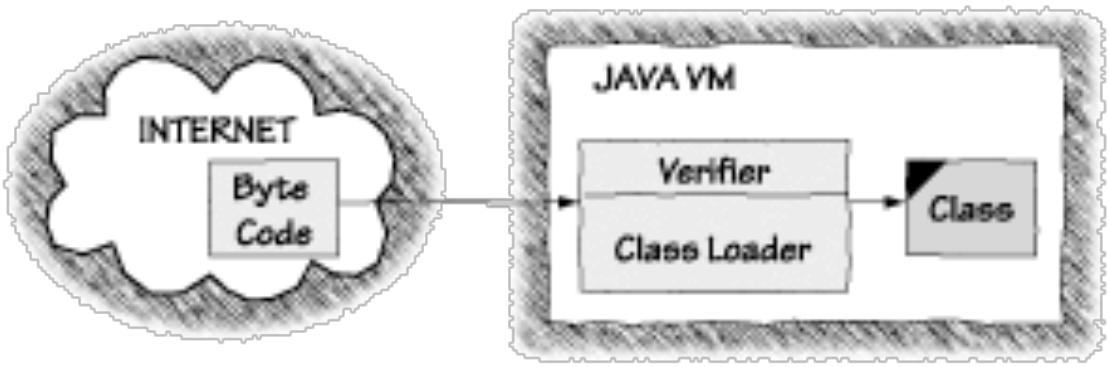
En class loader sørger endvidere for at placere de indhentede klasser i de relevante beskyttelsesdomæner, der definerer hvad koden har adgang til at gøre.

- Hvert beskyttelsesdomæne er styret af en sikkerhedspolitik (*security policy*), der håndhæves af en `SecurityManager`.

Class verifier

Ved indhentning af en klassefil bliver der anvendt en "*Class File Verifier*", der undersøger koden for at sikre at at den er udførbar, og at der overholdes et antal sikkerhedskrav.

Hvis klassefilen ikke består verifikationen genereres en undtagelse.



http://www.securingjava.com/chapter-two/chapter-two-7.html

Class verifier

Kodeverifikationen består af 4 gennemløb af koden:

1. Struktuel verifikation af klasse filen
2. Semantisk undersøgelse af erklæringer af metoder og data typer
3. Verifikation af byte koden
4. Verifikation af symbolske referencer

På trods af at mange problemer fanges af denne verifier er det stadigt muligt at snyde Javas verifier.

Struktuel verifikation af klasse filen

Sikrer at filen har det rette format ved at undersøge et antal forskellige ting, herunder blandt andet at:

- Filens første fire byte har det rette "*magic number*": 0xCAFEBABE
- Filen indeholder de rette angivelser af definitioner og sekvenser af byte kode.
- Filen har den rette længde (hverken for lang eller for kort ifølge oplysningerne i filen), og at den afsluttes korrekt.

Formålet er at sikre, at det er sikkert at forsøge at undersøge de enkelte erklæringer af data typer og metoder i klassefilen.

Semantisk undersøgelse af erklæringer

Her undersøges alt der kan undersøges uden at se på selve koden, herunder blandt andet at:

- `final` klasser ikke nedarves og at `final` metoder ikke overrides.
- Alle klasser har en superklasse (undtagen `java.lang.Object`).
- Restriktionerne på brug af constant pool overholdes.
- Alle referencer har lovlige navne, klasser og typeangivelser.

Nogle af disse undersøgelser synes overflødige, da disse foretages af Java oversætteren — men verifieren kan ikke vide om denne kode er blevet genereret af en Java oversætter — eller af et andet program med et uønsket formål.

Verifikation af bytecode

Her foretages en data-flow analyse af koden for at der for samtlige steder der kan nås i koden blandt andet gælder at:

- Indholdet af operantstakken har den rette størrelse og et lovligt indhold.
- Registrene bruges korrekt.
- Metoder kaldes med de rette parametre.
- Alle attributter tildeles værdier af kompatibel type.
- Alle bytecode operationer anvendes med de rette værdier i stak og registre.
- Attributter og variable initialiseres korrekt.
- Undtagelseshåndtering foretages korrekt.

Formålet er her at kunne konstatere så mange fejl som muligt før koden udføres.

Verifikation af symbolske referencer

Udføres i forbindelse med dynamisk loading og linking af klasser — det vil i forbindelse med Java sige når der bliver behov for klasserne, hvilket normalt er under udførslen af koden.

Dynamisk linking af en klasse består af to opgaver:
- Find den refererede klasse — hvilket kan medføre at den skal loades.
- Erstat (*link*) alle symbolske referencer (klasser, objekter, attributter og metoder) i den kaldende kode med de faktiske referencer i den netop loadede klasse.

Det vil sige at det fjerde gennemløb består af den kodeverifikation der først kan foretages under udførslen.

Ved ændring af en klasses grænseflade (offentlige metoder og attributters signaturer) er det nødvendigt af alle klasser der anvender denne klasse genoversættes. Dette gennemløb undersøger blandt andet om dette er sket.

Det vil sige at de forskellige klasser skal være binært kompatible (*binary campatible*).

JVM runtime monitorering

Javas virtuelle maskine har indbygget et antal monitoreringsmekanismer, der løbende overvåger det kørende program for at fange potentielle sikkerhedsproblemer.

Udover verifikation af symbolske referencer involverer denne monitorering blandt andet:

- Sikring af type-sikker casting af referencer.
- Sikring af der kun er struktureret adgang til lageret.
- Automatisk garbage collection.
- Sikring af tabelstørrelse ved anvendelse.
- Sikring mod referering af null referencer.
- Struktureret undtagelseshåndtering.

Kan omgås ved brug af "native code" — kald af oversat kode skrevet i et andet sprog.

Security Manager

Formålet med en Security Manager er at beskytte det underliggende operativsystem og de faciliteter det udbyder mod dårlig Java kode kørende i en virtuel maskine.

Det vil her sige at det er muligt at begrænse adgangen til forskellig funktionalitet.

En *Security Manager* er kode der validerer om et program har adgang til specifik funktionalitet, og blokerer adgangen hvis der ikke er givet tilladelse.

Ved opstart af et Java program er der som udgangspunkt ingen security manager, men programmøren kan vælge at starte en ved at oprette en instans af `java.lang.SecurityManager`, eller ved at angive argumentet "`-Djava.security.manager`" ved starten af Java programmet.

Hvis et program har startet en security manager udføres programmet i en Java *sandbox*, med rettigheder bestemt af et `Policy` objekt.

Java sikkerhed

Da et Java program som udgangspunkt ikke starter en security manager betragtes det som betroet (*trusted*) og har adgang til al funktionalitet i programmeringssproget og alle ressourcer på maskinen.

Dette gælder også betroede appletter: lokalt gemte appletter (hentet fra et katalog i brugerens CLASSPATH) og signerede appletter.

Alle andre appletter (typisk hentet via en netværksforbindelse) starter en security manager, der begrænser applettens muligheder, hvorfor en applet normalt ikke er betroet (*untrusted*), og koden udføres derfor inde i en Java *sandbox*, der begrænser applettens muligheder for at gøre uønskede ting.

Untrusted appletter

Funktionen af en *untrusted* applet begrænses på et antal områder, så som:

- Udførsel af systemfaciliteter, såsom `System.exit()`, `Runtime.exit()`, `Runtime.exec()`, indhentning af biblioteker, definition og udførsel af kode i andre sprog en Java.
- Læsning og skrivning af filer på den udførende maskine.
- Netværksforbindelser til andre maskiner end den appletten er hentet fra.
- Netværksforbindelser til portnumre < 1024.
- Oprette tråde udenfor appletten selv.
- Oprette og kalde metoder på en `ClassLoader` eller `SecurityManager`.
- Anvende visse systemoplysninger.

Policy objekter og filer

Når en security manager startes oprettes et Policy objekt for programmet, hvor der indlæses rettigheder fra "policy" filer.

Ved at ændre en eksisterende policy fil — eller oprette en ny policy fil og anvende denne — på den udførende maskine, er det muligt at tildele specifikke rettigheder til et program eller en applet.

Dette gøres ved enten at ændre en "*system wide*" policy fil der gælder generelt på maskinen, ved at oprette en lokal policy fil i brugerens hjemmekatalog eller ved at angive stien til en policy fil som argument ved start af et Java program.

En lokal policy fil består af et antal "*grants*" der tilføjer rettigheder:

```
grant codeBase "http://localhost/sd/*" {
        permission java.lang.RuntimePermission "usePolicy";
        permission java.security.AllPermission;
};
```


Applet muligheder

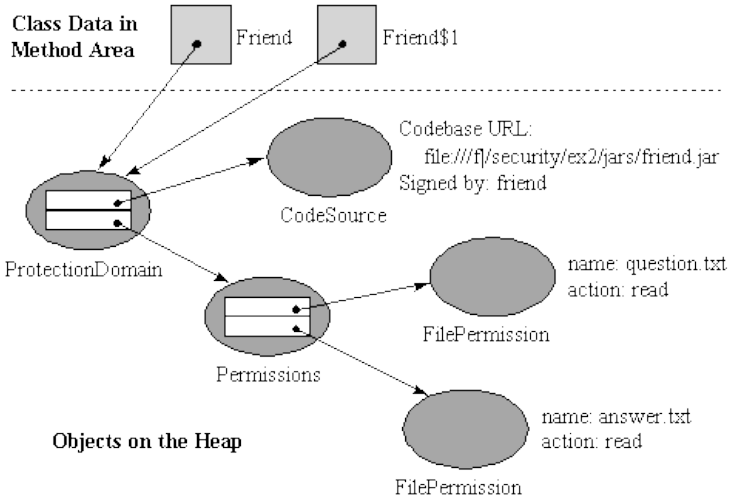
På trods af begrænsningerne kan en *untrusted* applet, der ikke har fået tildelt ekstra rettigheder udføre mange brugbare ting, så som:

- Lave netværksforbindelser til den maskine appletten blev hentet fra..
- Hvis appletten udføres inde fra en WWW browser, kan den medføre visning af HTML dokumenter.
- Kalde offentlige metoder på andre appletter der hentes ind af den samme HTML side.
- De fleste appletter afslutter udførslen når browseren skifter til visning af en anden side, men en applet kan også blive i lageret (*stay resident*) og fortsætte med at køre.
- Implementere ny brugergrænsefladefunktionalitet.

Beskyttelsesdomæner

Når en klasse loader henter en klassefil ind i lageret placeres koden i et beskyttelsesdomæne, der definerer de adgangsrettigheder koden har.

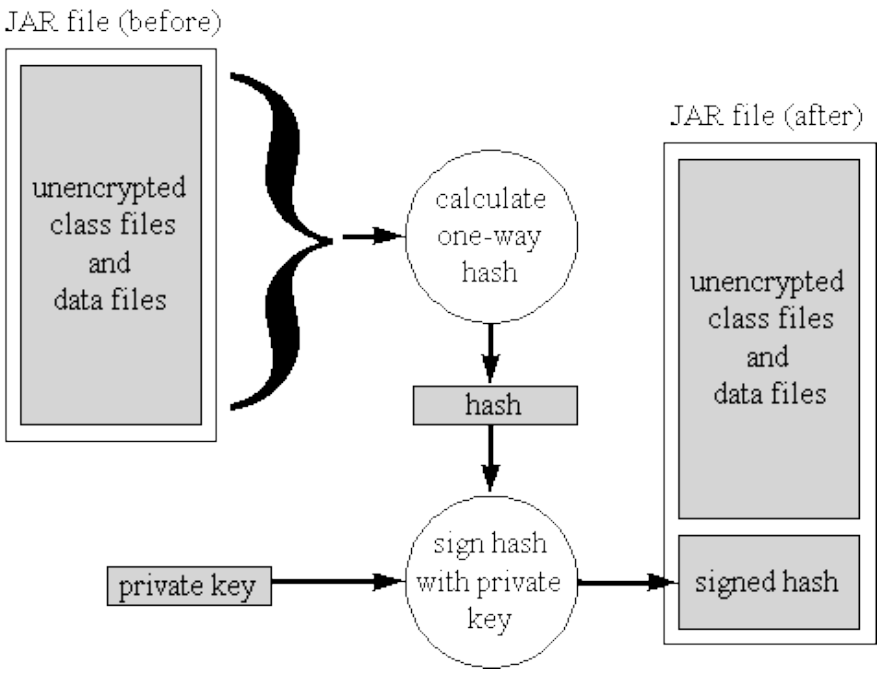
Klasse loaderen sørger for at det gældende `Policy` objekt knyttes til beskyttelsesdomænet, samt eventuelle tilføjelser der gælder specifikt for den hentede klasse.



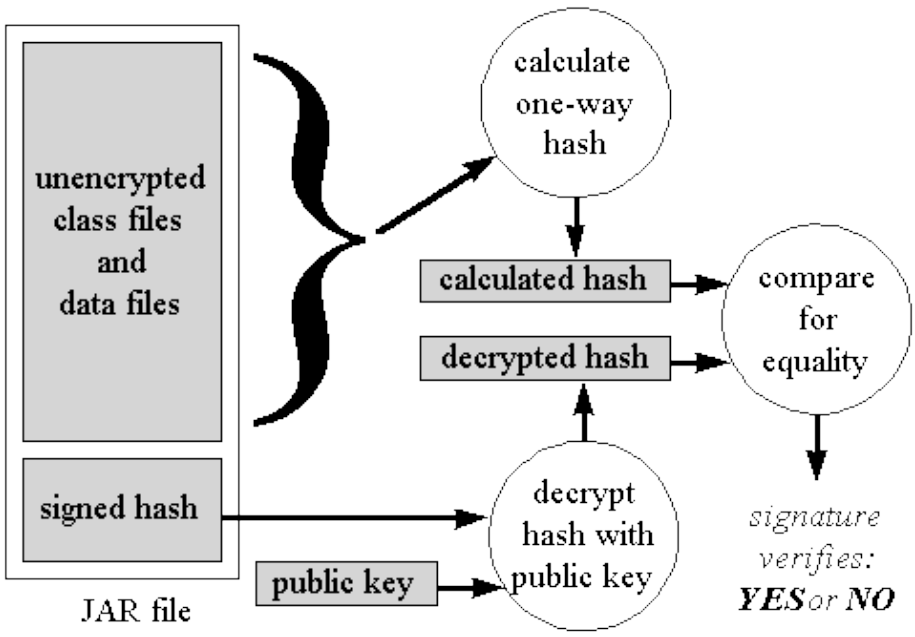
http://www.artima.com/insidejvm/ed2/security.html

Signerede klasser

Java giver mulighed for at der kan arbejdes med signerede klasser, således at det er muligt at verificere om indholdet af en .jar fil (typisk indeholdende et program eller en applet) er signeret af en bestemt person eller organisation.



http://www.artima.com/insidejvm/ed2/security.html



http://www.artima.com/insidejvm/ed2/security.html

Signerede klasser

Afhængigt af om verifikationen lykkes eller ej kan brugeren vælge om koden skal udføres eller ej.

En verifikation kan mislykkes af følgende årsager:

- Der haves ikke en offentlig nøgle til kontrol af signaturen.
- Den offentlige nøgle kan ikke verificeres at tilhøre den påståede underskriver.

Disse problemer kan løses på forskellig vis, typisk ved brug af rodcertifikater, eller andre betroede certifikater på brugerens maskine.