Fachhochschule Nordwestschweiz
Hochschule für Technik

Java 2 security overview

Carlo U. Nicola, SGI FHNW

with extract from publications of

Jonathan Katz, U. of Maryland, A. Kaminsky, Rochester Institute of Technology

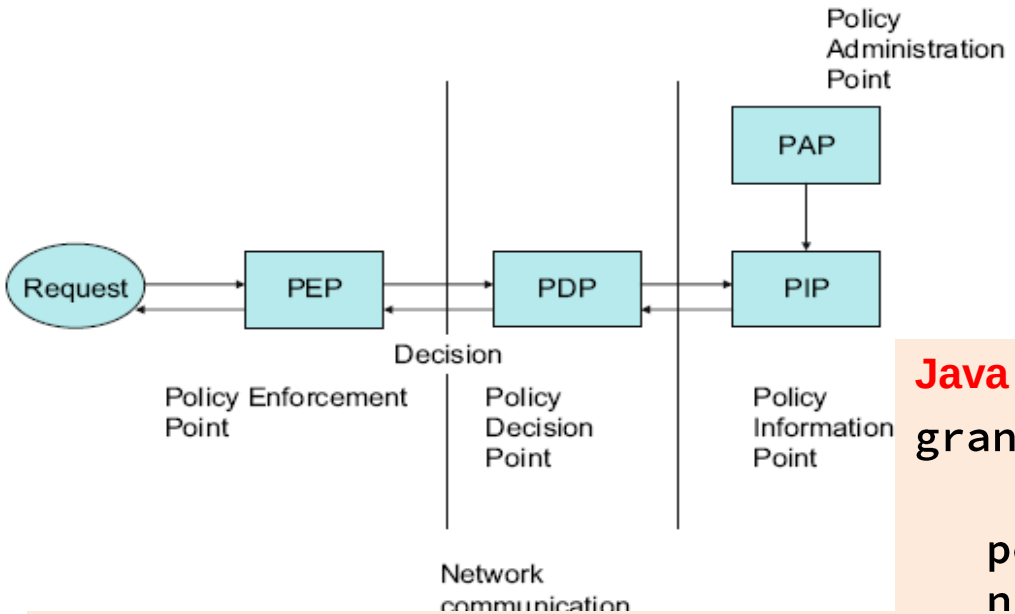
Topics

1. The Java 2 security model
2. How Java 2 defines and guards the protections domains
3. How Java 2 breaks the rules: local amplification of privileges
4. Java stack inspection and security
5. Java class format
6. The role of the class loader in Java security

Security Model: OpenXML and Java

Java PEP:

`SecurityManager, Permission, AccessController, AccessControlContext, ProtectionDomain.`



Java PAP → `File: java.security`

Java PIP → `File: java.policy`

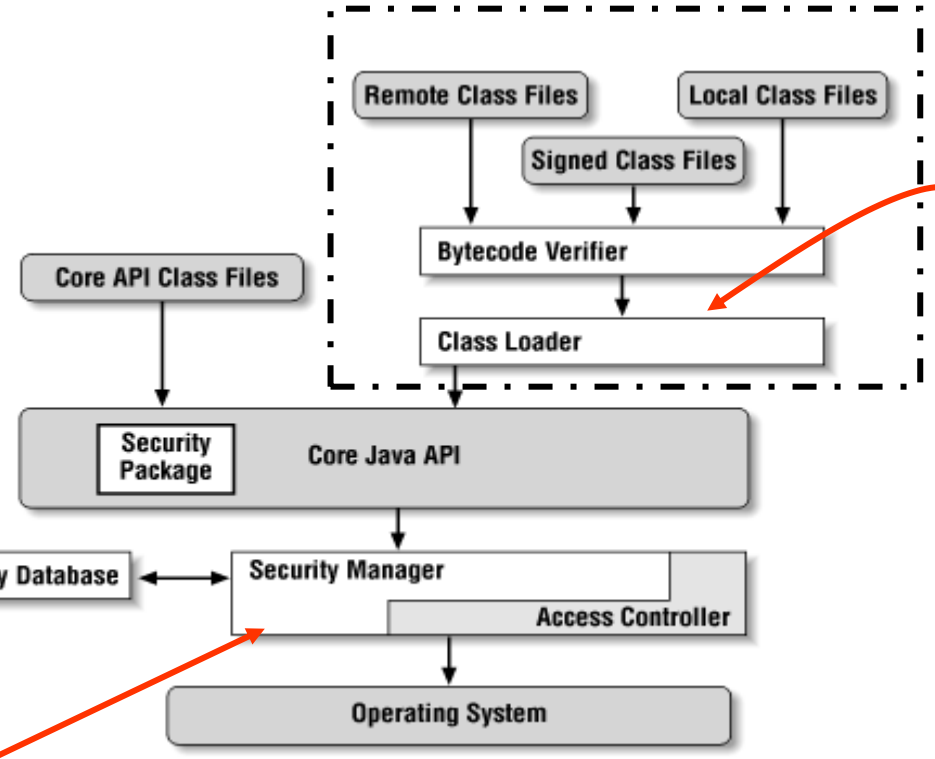
```
grant codeBase …{

   permission.java.security.AllPermissio
   ns:
```

Java PDP:

```
if (sm != null){
 context = sm.getSecurityContext();
 Filepermission p = new Fileprmission(filename, “read”);
 sm.checkPermission(p,context);
}
```

Java security in a nutshell (1)

Bytecode + Class loader security



`.keystore`, `.truststore`: set up with `keytool`

Java Core local libraries class-files are not Bytecode verified

Activated by: `-Djava.security.manager` or programmatically

Java security in a nutshell (2)

Repeat

Check if current method has the requested permission

If not, throw **`SecurityException`**.

Check if current method has amplified privileges.

If so, grant permission.

Consider calling method (move up call stack)

Until call stack is empty

From: `java.policy`

```
AccessController.
   doPrivileged(new
   privilegedAction()
   {..}
```

Check if thread inherited the requested permission.

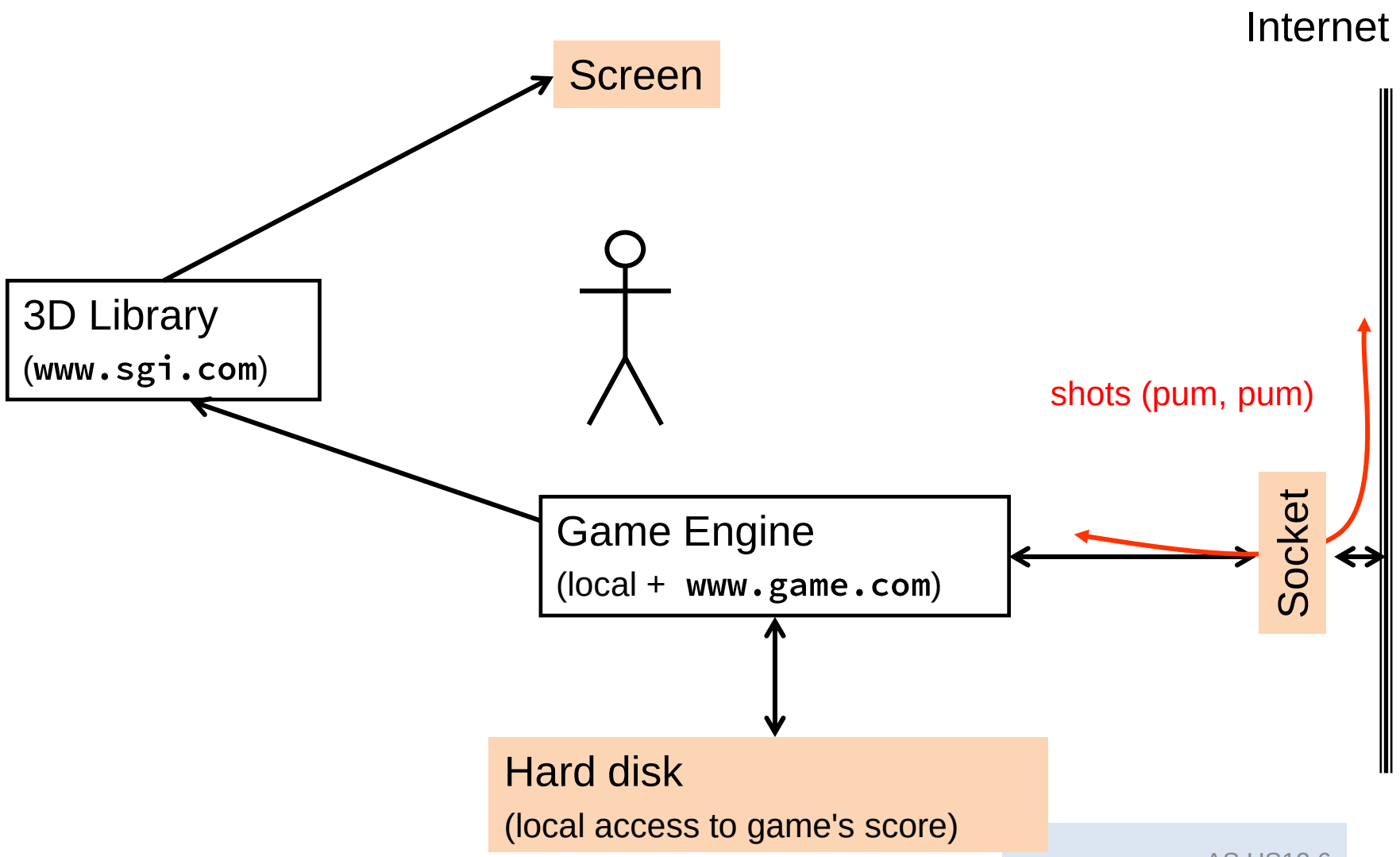
If not, throw **`SecurityException`**.

If yes, grant permission.

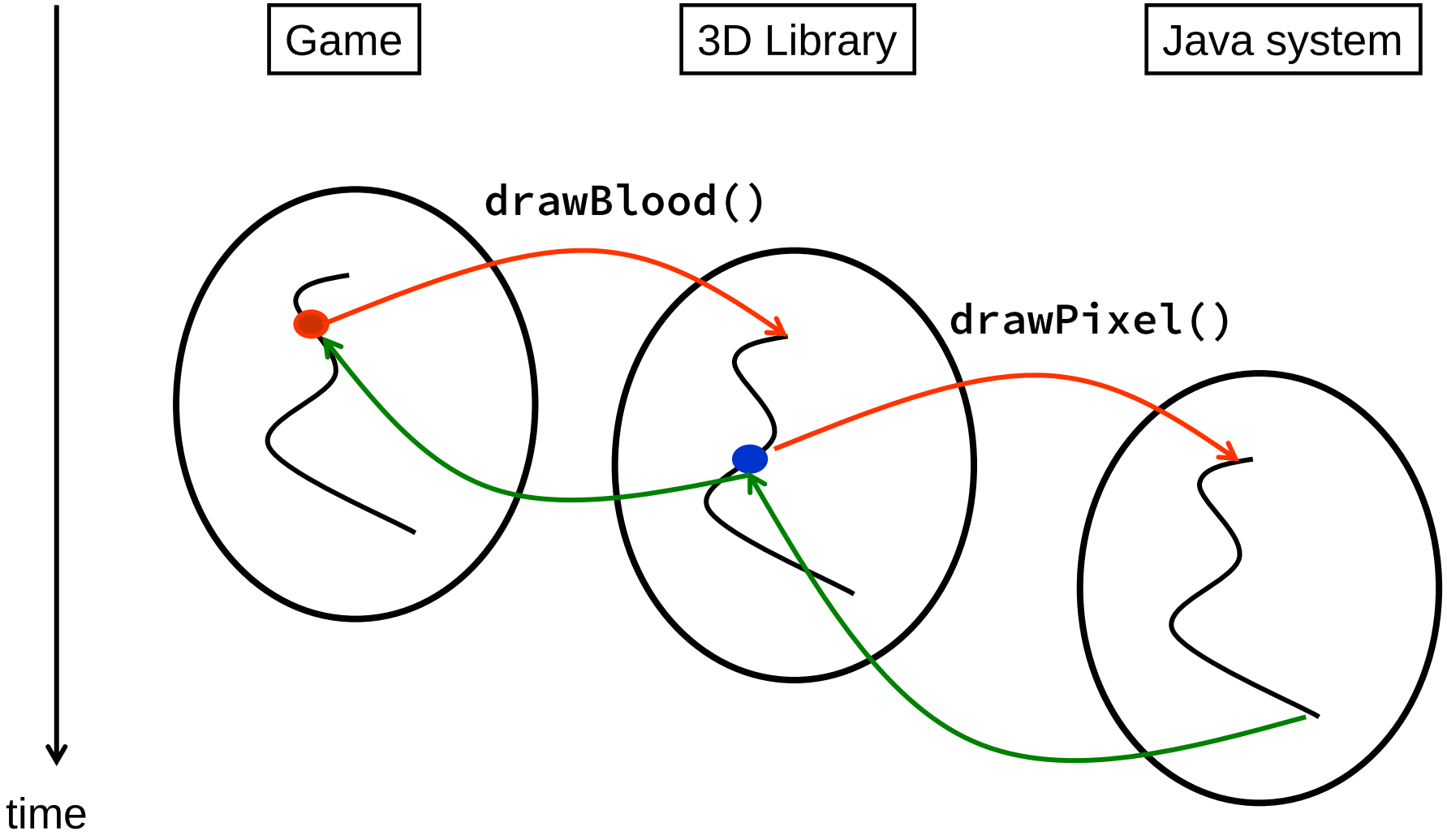
From: Protection domain

Protection Domain: Stores in a per thread variable the intersection (Å) of the static permissions of all methods invoked since its start, and grant permission on the result of that intersection operation.

The game


Internet
Screen
3D Library
(www.sgi.com)
shots (pum, pum)
Game Engine
(local + www.game.com)
Socket
Hard disk
(local access to game's score)

The problem (1)



The problem (2)

Which privileges are necessary in order to play the game?

- → Access to HW resource **screen**
- → Access to HW resource **hard disk** (to store the high score file)
- → Access to HW resource **socket** (communication with the game server)

We need a mechanism for allowing/disallowing access to local resources !

Java 2 defines different (code) domains based upon :

a) The **origin** of the code (i.e. `www.game.org` , `www.sgi.com`)

b) The **digital signature** of the code

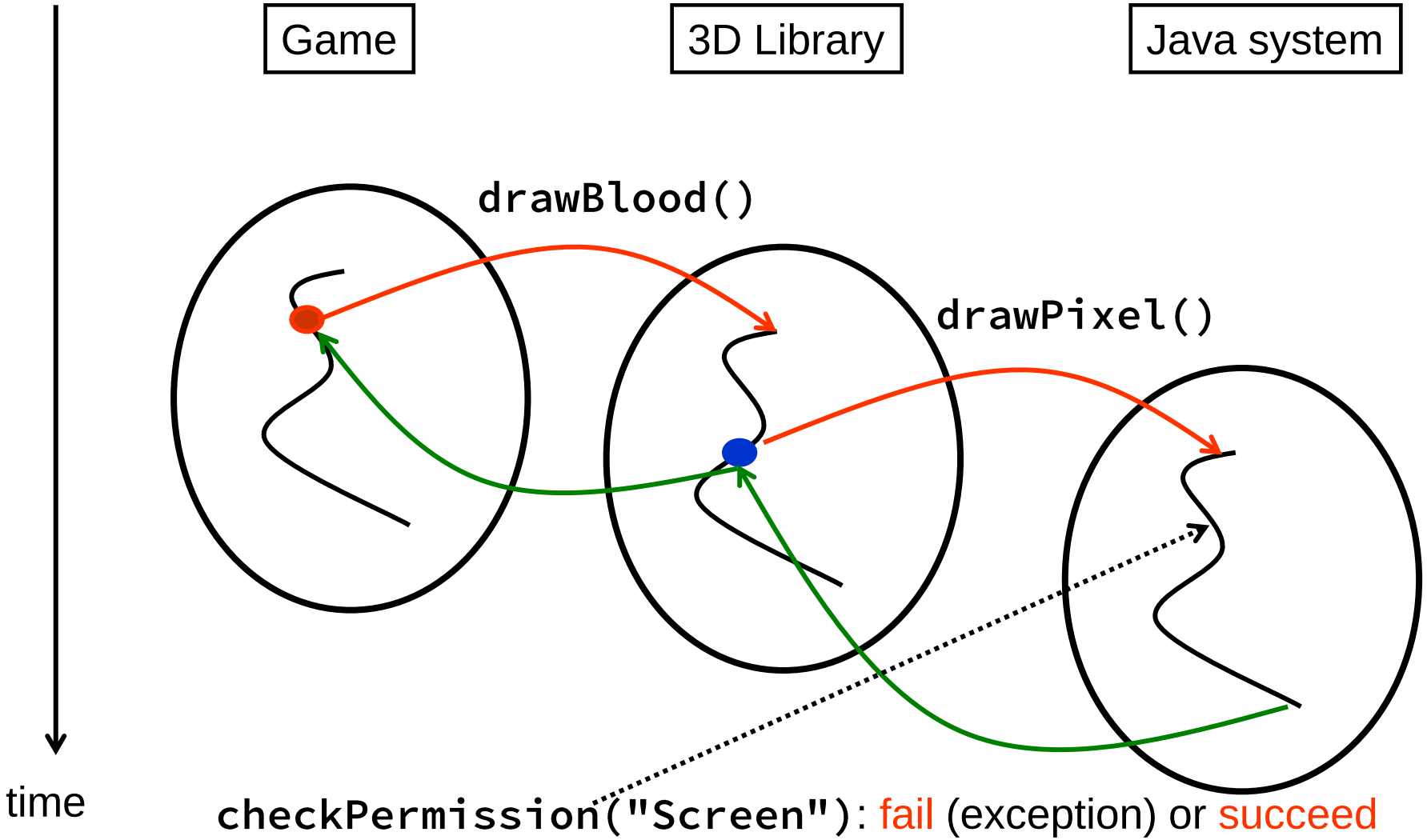
Java policy files for the game

```
grant CodeBase "http://www.game.org/-" {
  permission java.lang.RuntimePermission "Screen";
  permission java.net.SocketPermission "server.game.com";
  permission java.io.FilePermission "C:\\HighScore", "read", "write";
}
```

```
grant CodeBase "http://www.sgi.com/-" Signed by "SGI"{
  permission java.lang.RuntimePermission "Screen";
  permission java.io.FilePermission "C:\\lib\\3D_Data\\-", "read";
}
```

At run time these policies are enforced by the class **`AccessController.checkPermission(...)`** that is automatically called when the **`SecurityManager(…)`** is activated (either as option for the JVM or programmatically in **`main(…)`**.

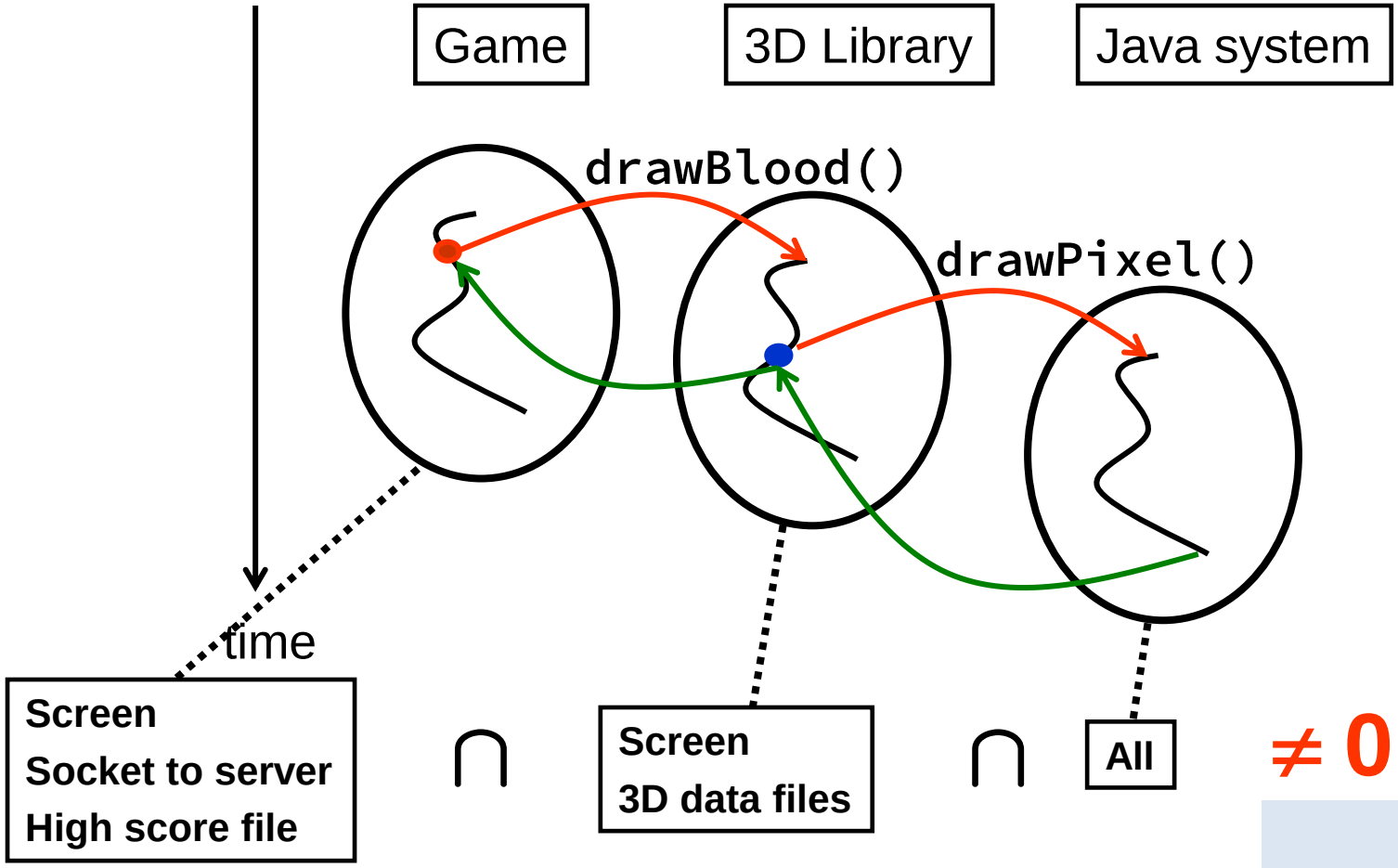
Protections' domains (1)



Protections' domains (2)

How do we guarantee that `checkPermission(…)` does not fail?

Rule: **All domains which the actual thread crosses must own the desired privilege** (in our example the right to write to the screen)



Stack introspection

The manner with which the privileges are inspected in Java 2 works as follows (pseudo Java code):

```
boolean checkPermission(Permission toCheck) {
  Stack domainStack = getDomainStack(); // Domain for current thread
  while (!domainStack.empty()) {
    Domain here = domainStack.pop();
    if (!here.implies(toCheck))
      // No such right: implies → ∩
      return false;
   }

  return true;
}
```

An old friend: ACM

Subject/Object	Screen	3D data fles	High score file	Socket to server
`game`	X		X	X
`3DLib`	X	X		
`system`	X	X	X	X
`Game+3DLib`	X			
`Game+3DLib+system`	X			

The final privilege is always calculated as follows:

$$Dom_1 \cap Dom_2 \cap \ldots \cap Dom_n \neq 0$$

Protection domain of the calling method

Protection domain of the method which access the resource

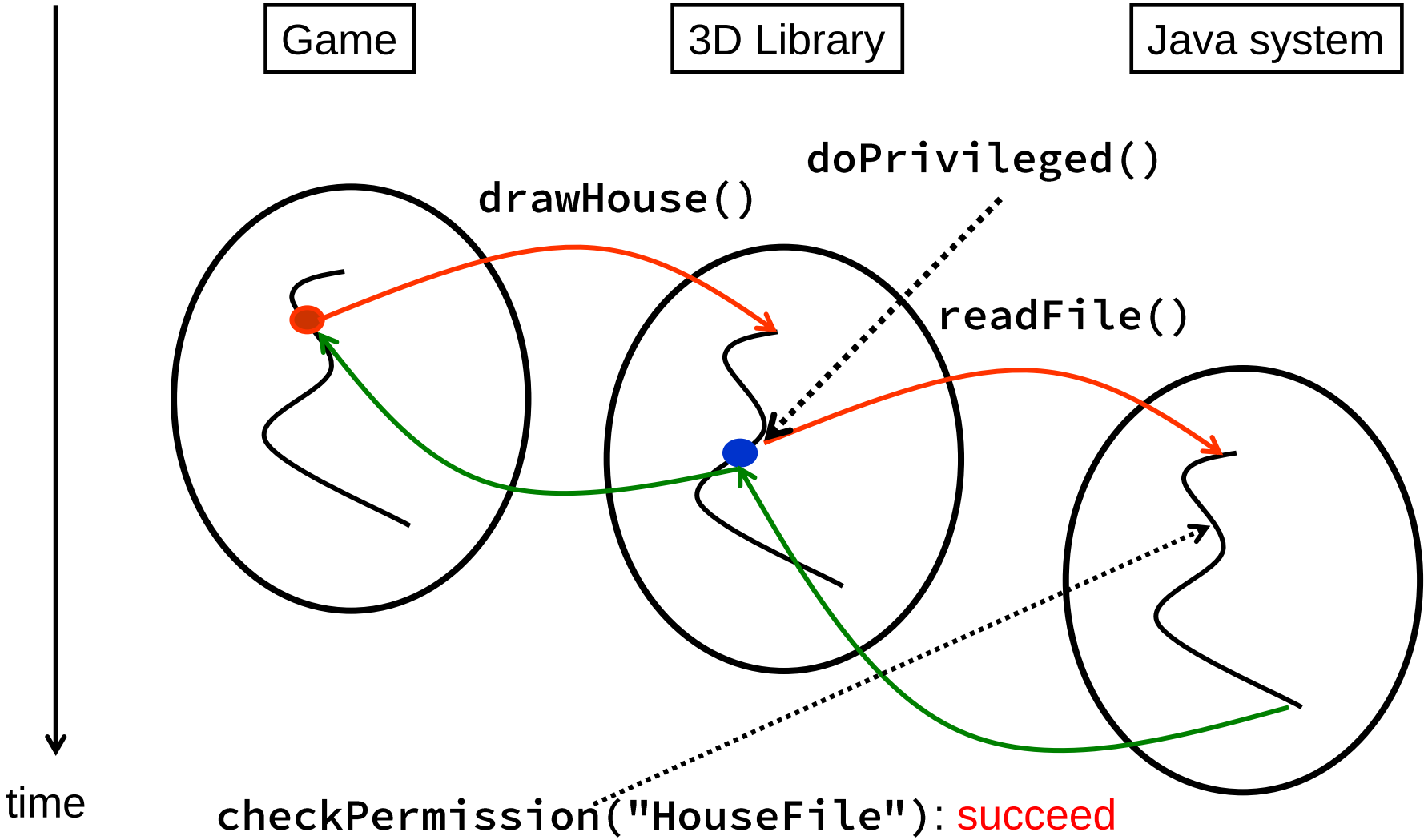
A new problem

How can the library **`3DLib`** access the 3D data files?
In fact the subject **`game`** (which uses the **`3DLib`**) has no right to access them (see the previous Access Control Matrix). Java 2 introduces here a well designed security hole, albeit half-respecting the **least privilege** principle. In our specific case that means:

> Only the subject **`game+3DLib`** may possess this right and not the subject **`game`** alone.

In order to do that, Java 2 amplifies temporarily the rights of a subject in the actual protection domain.

Game example with `doPrivileged(…)`



The syntax of `doPrivileged(...)`

```
Object house = AccessController.doPrivileged(
          new PrivilegedAction() {
            public Object run() {
              return readObjectFromDataFiles("house");
            }
          }
)};
```

Stack inspection and `doPrivilege(...)`

The manner with which the privileges are inspected in Java 2 with the introduction of **`doPrivilege(…)`** works now as follows (pseudo Java code):

```
boolean checkPermission(Permission toCheck) {
  Stack domainStack = getDomainStack(); // Domain for current thread
  while (!domainStack.empty()) {
    Domain here = domainStack.pop();
    if (here.isPrivileged())
      return true ;
    if (!here.implies(toCheck))
      // No such right: implies → ∩
      return false;
  }
  return true;
}
```

Bytecode and class loader security

`.class` format

u4 magic; u2 minor_version;
u2 major_version;

u2 constant_pool__count;
cp_info constant_pool[constant_pool_count-1];

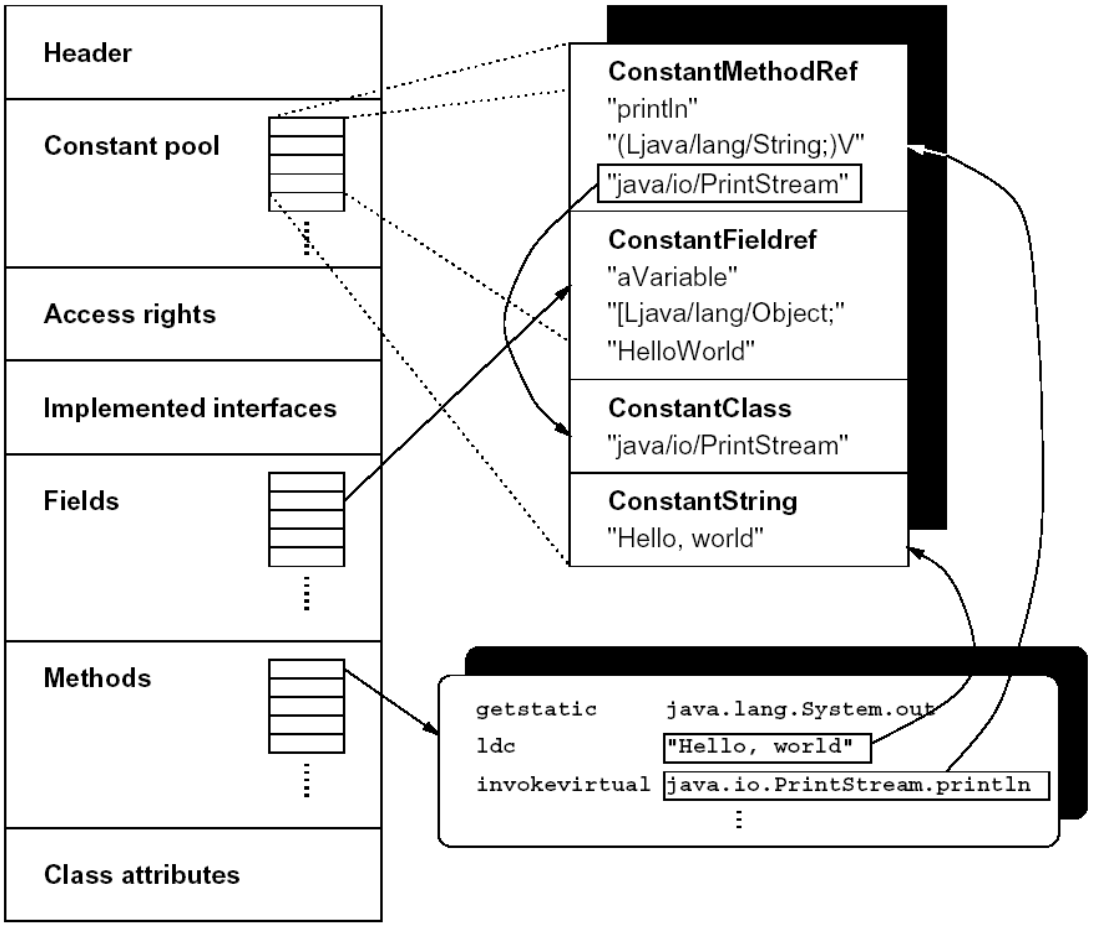
u2 access_flags; U2 this_class;
u2 super_class;

u2 interfaces_count;
attribute_info interfaces[interfaces_count];

u2 fields_count;
attribute_info fields[fields_count];

u2 metods_count;
attribute_info methods[methods_count];

u2 attributes_count;
attribute_info attributes[attributes_count];



HelloWorld.class

Byte code verifier

Goal: Prevent access to underlying physical machine via forged pointers, crashes, undefined states

Checks a `*.class` file for validity:

- → Code has only valid instructions and register use
- → Code does not overflow/underflow the stack
- → Code does not convert data types illegally or forge pointers
- → Code accesses objects as correct type (as given in the `constant_pool` header)
- → Method calls use correct number and types of arguments
- → References to other classes use legal names

Java class loaders

The role of the class loader in Java security

The class loader is the fountain head of Java security:

- → The loading and **verification** steps expel bogus class files before they even get into the JVM
- → **Protection domains** drive all later security decisions
- → **Namespaces** keep separate classes and packages coming from different places

The role of the class loader in Java security

Protection domains

→ Protection domain = **Code source + collection of permissions**
→ Code source = **Origin of class file (URL) + zero or more signers (certificates)**
→ A class's protection domain is established by the class loader when the class is loaded (only the class loader knows the class file's origin)
→ A class's protection domain is later used to make security decisions about what code in the class is or is not allowed to do

Namespaces

→ The JVM considers class `ch.fhnw.bar.Bar` loaded by class loader **X** to be different from class `ch.fhnw.foo.Bar` loaded by class loader **Y**, even though the class names are the same
→ The JVM considers class `ch.fhnw.foo.Bar` loaded by class loader **X** to be in a different package from class `ch.fhnw.foo.Xyz` loaded by class loader **Y**, even though the package names are the same. In other words, each class loader defines a separate *namespace* for classes and packages
→ This let same-named classes and packages from different sources coexist, but without having access to each other's protected and package-scoped members
→ Prevents *package insertion attacks* where an evil class claims to be part of a certain package in order to gain access to sensitive (protected) data

Java class loaders hierarchy

User-defined class loaders created by the program:

The system class loader is the parent class loader by default . Another parent class loader can be specified explicitly.

Each thread has an associated context class loader:

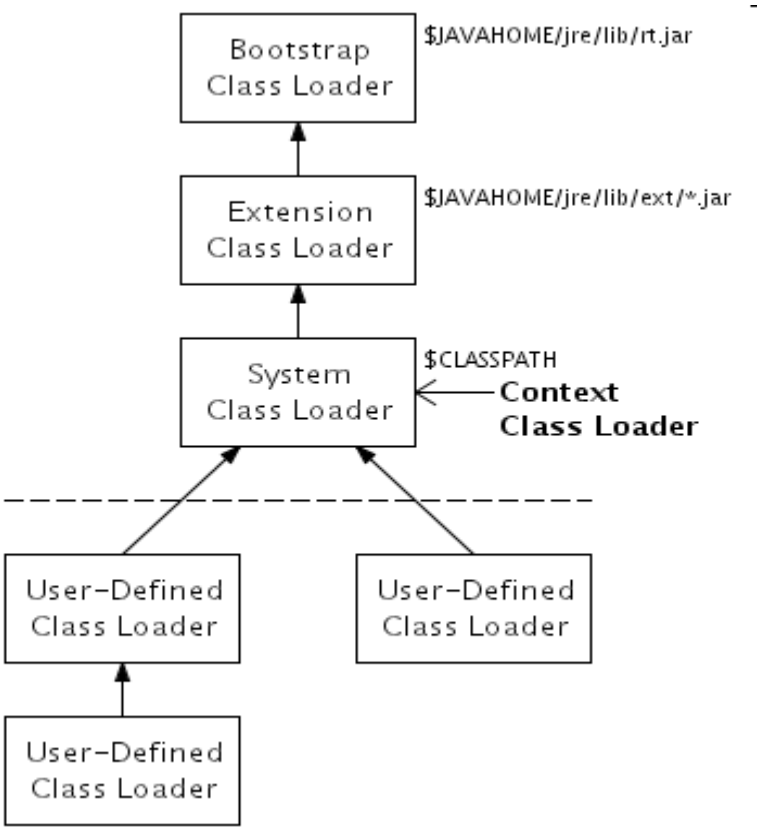
Default context class loader is the system class loader.
A different context class loader can be specified by calling `Thread.setContextClassLoader()`
Obtain a thread's context class loader by calling `Thread.getContextClassLoader()`

Delegation model:

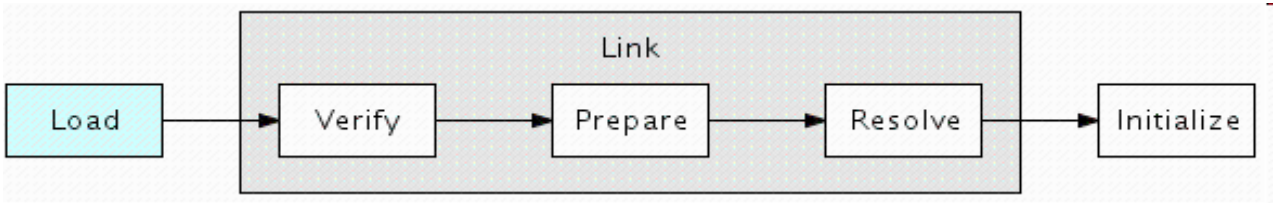
When asked to load a class, a class loader first asks its parent to load the class. If the parent succeeds, the class loader returns the class from the parent. If the parent fails, the class loader attempts to load the class itself . The class loader hierarchy is thus searched from the top back down to the starting point.

Which class loader is used?

You can explicitly load a class into a class loader by calling the class loader's `loadClass()` method. If a class needs to be loaded and a class loader is not explicitly specified , the class is loaded into the same class loader that loaded the class that contains the code that is executing.



The class loader security phases: load



→ **Given the type's fully-qualified name, obtain its class file (byte sequence)**
From a file in a directory, from a file in a JAR, from a URL, . . .

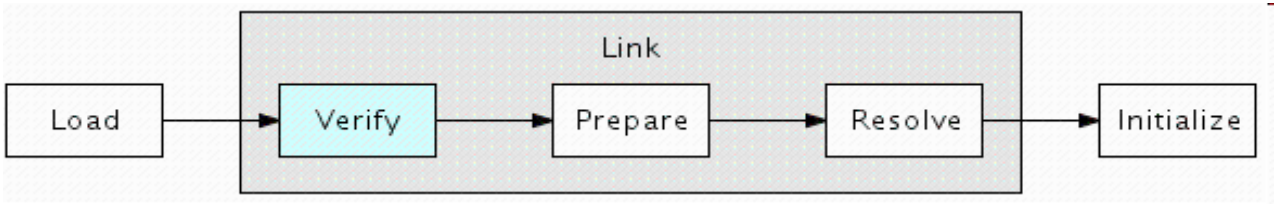
→ **Parse the class file (byte sequence) into implementation-dependent internal data structures in the method area**
Parsing problems cause various errors to be thrown (example: the first four bytes are not `0xCAFEBABE`

→ **Resolve the symbolic reference to the super class**
Resolution problems cause various errors to be thrown (example: the super class is not accessible to this class).

→ **Resolve the symbolic references to the interfaces if any**
Resolution problems cause various errors to be thrown (example: if an interface is not accessible to this class).

→ **Create an instance of class `Class` to represent the type**

The class loader security phases: verify



Checks a `.class` file for validity (Bytecode Verifier):

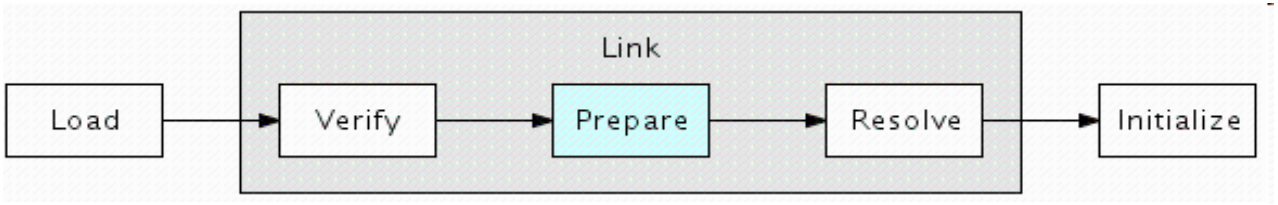
- → Final classes are not sub classed
- → Final methods are not overridden
- → Every class has a super class
- → Constant pool entries obey all specified constraints
- → All field and method references have valid classes, names, and type descriptors

1st part

- → Code has only valid instructions and register use;
- → Code does not overflow/underflow the stack;
- → Code does not convert data types illegally or forge pointers;
- → Code accesses objects as correct type;
- → Method calls use the correct number and types of arguments;
- → References to other classes use legal names.
- → Goal is to prevent access to the underlying machine:
  For example via forged pointers, crashes, or undefined states.

2nd part

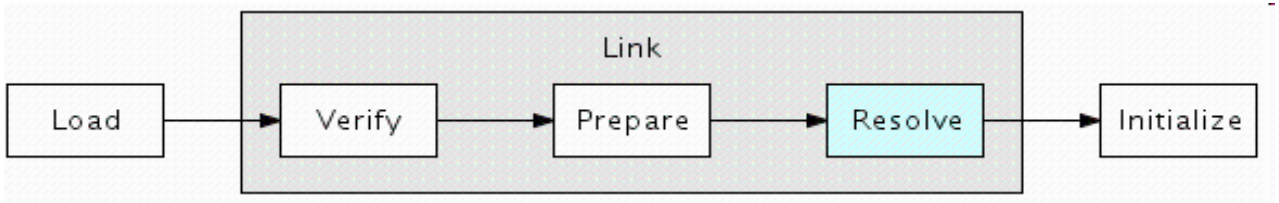
The class loader security phases: prepare



→ **Allocate storage for the class's static fields** (this is a shared memory region between all classes)

→ **Set the class's static fields to their default initial values**
Setting static fields to explicit initial values is performed later, in the Initialization phase.

The class loader security phases: resolve



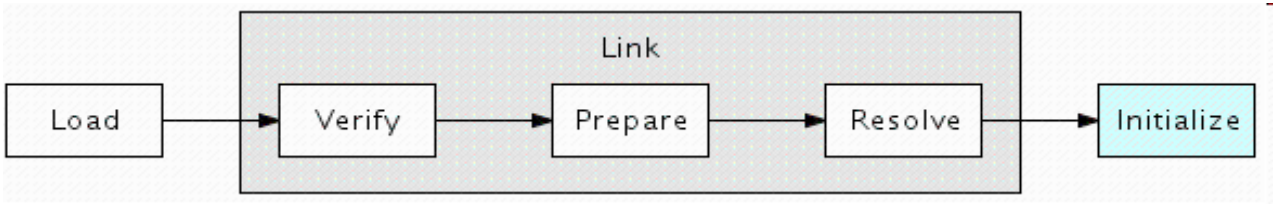
→ **Validate each symbolic class, field, and method reference**
Search up the inheritance hierarchy to find where the class, field, or method is declared
Verify that the class, field, or method are accessible to the class which uses them.

→ **Replace each symbolic reference with an (implementation-dependent) direct reference**

→ **Each symbolic reference can be resolved when a class is initially linked** (greedy resolution)

→ **Alternatively, each symbolic reference can be resolved later during execution, when the reference is actually used** (lazy resolution)

The class loader security phases: initialize



→ **Execute the class's static `clinit` method**

→ **The `clinit` method is compiler-generated and contains:**

- → Code for explicit initialization of static fields
- → Code from all static blocks: a typical covert channel a cracker can use to do mischief (see Java Anti-pattern lecture)