JKU

JOHANNES KEPLER
UNIVERSITY LINZ

# JAVA SECURITY

PR SW2 S18
Dr. Prähofer
DI Leopoldseder

# AGENDA

1. Introduction
2. Class Loading
3. Security Manager and Permissions
4. Summary

# AGENDA

1. **Introduction**
2. Class Loading
3. Security Manager and Permissions
4. Summary

# DIFFERENT LEVELS OF SECURITY

- Language Level
  - Strong data typing
  - Automatic memory management
  - Defined Overflow Semantics
- Bytecode Level
  - Signed Code
- Virtual Machine Level
  - Checked Array Accesses
  - Secure Class Loading
    - Byte-Code Verification
  - Security Manager
    - Control Access & Execution Permissions

# SECURITY MECHANISM

- Virtual Machine
  - Array access checks
  - Forbidden casts
  - …
- Class loader
  - Loading of code
  - Bytecode verification
- Security Manager
  - Allowed and forbidden operations
- Encryption technologies
  - Code signing
  - authentication

# AGENDA

1. Introduction
2. **Class Loading**
3. Security Manager and Permissions
4. Summary

# CLASS LOADING

- Java source code is compiled by **javac** to bytecode
- Bytecode is a platform independent stack machine based assembly like code format
- The JVM loads bytecode on demand
  - What does that mean
    1. Class containing **main** method is loaded
    2. Super  classes and classes (transitive) of fields are loaded
    3. Static initializer is executed
    4. Main function is executed
    5. New types encountered during execution are loaded

# BYTECODE VERIFICATION

- 2 Level verification
  - **Javac** will not compile corrupt source code
  - **VM** will not load corrupt class files (except if specified with **–noverify**)
- Problem: Not all bytecode generated by javac
- What does the VM’s verifier check?
  - Variables are initialized before they are used
  - Method calls match types of object references
  - Access rules (protection) is not violated
  - Local variable access fall into runtime stack (stack is not corrupt)
  - Runtime stack does not overflow

# CLASS LOADERS

- Classes are loaded by **ClassLoader** objects
  - Accessing class loader objects

```
Class clazz = Class.forName(“MyProgram“);
ClassLoader loader = clazz.getClassLoader();
```

```
ClassLoader loader = ClassLoader.getSystemClassLoader();
```

Default Class Loader

- Class loader Support
  - Explicit loading

```
Class myClass = loader.loadClass("mypack.MyClass");
```

  - Defining new classes

```
byte[] classCode = ...;
loader.defineClass(“MyDefinedClass“, classCode, 0, classCode.length);
```

  - Class loaders can be specified

```
Thread thread = Thread.currentThread();
thread.setContextClassLoader(loader);
```

Set class loader for thread

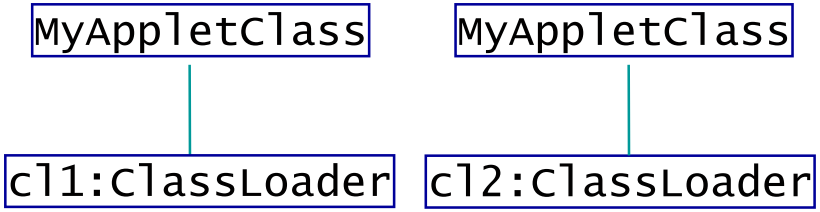
# KINDS OF CLASS LOADERS JDK8

- Class loaders are organized in a class hierarchy (tree)
  - Root class loader is called **Bootstrap class loader**
    - The bootstrap loader loads all classes from **rt.jar** (stands for runtime.jar, contains all classes of the JDK)
  - Extension class loader
    - Loads all extensions from jre/lib/ext
  - System class loader
    - Loads **CLASSPATH**
  - Special class loaders
    - Application specific class loader as extensions for System class loader

Gone in Java 9: Now called platform class loader

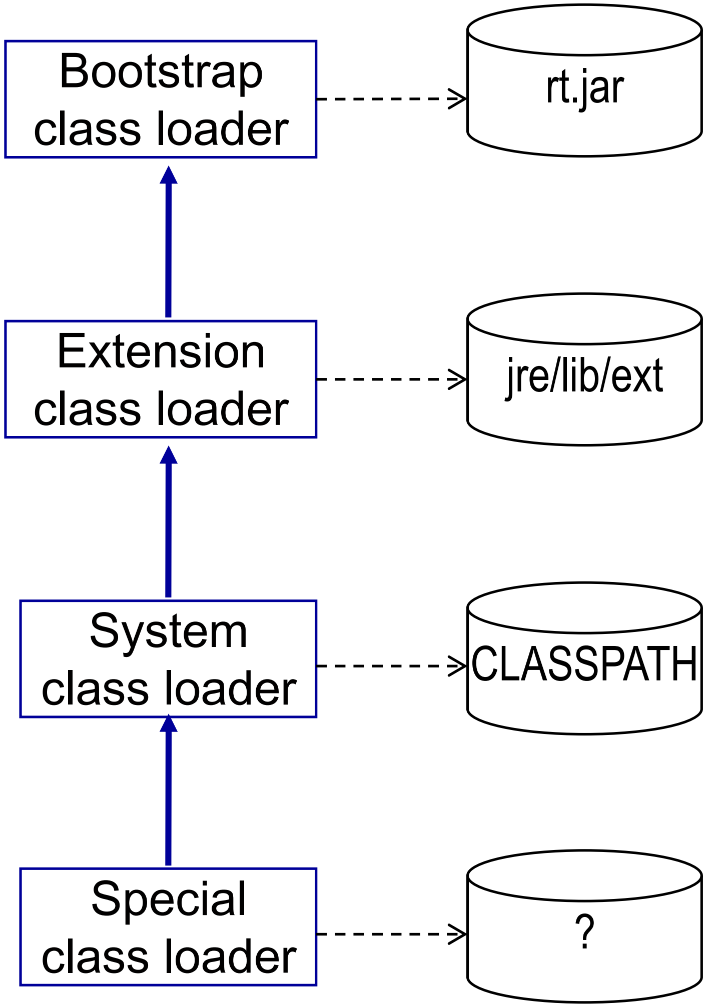
# CLASS LOADERS AND TYPE IDENTITY

- Every class in the VM (after loading) is associated with a class loader
- Can use user defined class loader to load own classes (or define them)
- Equality of classes is not only given by their name but also by the class loader that loaded the class so e.g. Foo.class can be loaded n times with n different class loaders
- Example: Applets loaded from different servers loaded by different class loaders

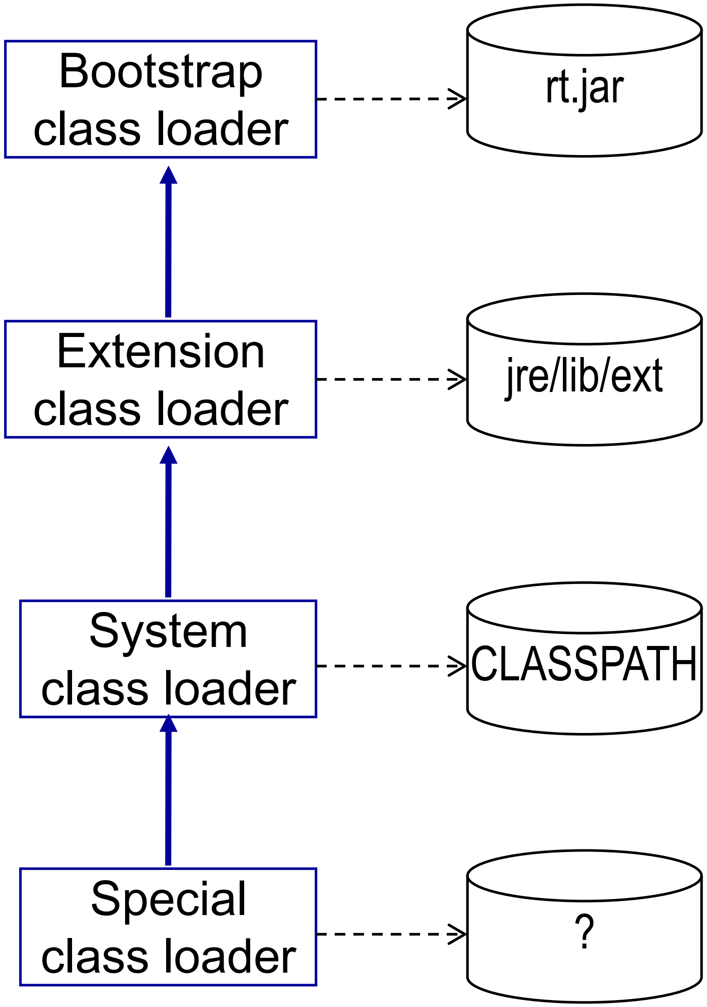


# CLASS LOADER HIERARCHY

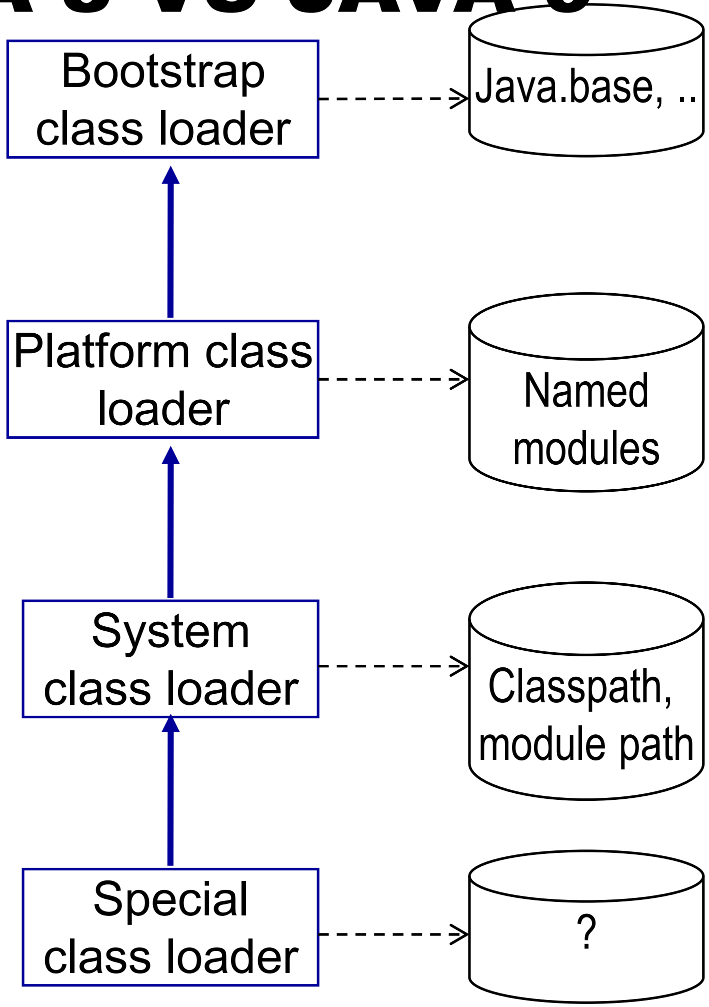
- Hierarchy with parent and child relation
- Determine priority of class loading
  1. Bootstrap class loader
  2. Extension class loader
  3. System class loader
  4. Specializations
- Class loaders typically first delegate loading of a class to its parent class loader
- Only if the parent fails then child tries



# CLASS LOADERS JAVA 8 VS JAVA 9



Java 8



Java 9

# SPECIAL CLASS LOADERS: EXAMPLE

- **URLClassLoader**
  - Loads classes from URLs

```
URL pluginUrl = new URL("file:c:/plugins.jar");
URLClassLoader pluginLoader = new URLClassLoader(
                                  new URL[] { pluginUrl },
                                  ClassLoader.getSystemClassLoader()
                              );

Class<?> cl = pluginLoader.loadClass("plugin1.PluginClass");
cl.getMethod("test").invoke(null);
```

Plugin-JAR **not** in CLASSPATH

Superior Class Loader

# SAMPLE IMPLEMENTATION OF SPECIAL CLASS LOADER

- Extend **ClassLoader** to define new semantics e.g.
  - □ Decryption of encrypted bytecode files
  - □ Overriding the **findClass** method

```java
public class CryptoClassLoader extends ClassLoader {
    private final String path;
    private final int key;

    public CryptoClassLoader(String path, int key) { ... }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classBytes = null;
        try {
            classBytes = loadAndDecryptClassBytes(name);
        } catch (IOException e) { throw new ClassNotFoundException(name); }
        Class<?> clazz = defineClass(name, classBytes, 0, classBytes.length);
        if (clazz == null ) throw new ClassNotFoundException(name);
        return clazz;
    }

    private byte[] loadAndDecryptClassBytes(String name) throws IOException {
        ...
    }
```

**findClass** called by **loadClass**

# AGENDA

1. Introduction
2. Class Loading
3. **Security Manager and Permissions**
4. Summary

# SECURITY MANAGER

- Although Java is considered to be a “safe” language based on executing bytecode in a “sandbox” hiding system and implementation details there are operations **breaking** that paradigm as they are inherently unsafe
- **java.lang.SecurityManager** allows the programmer to programmatically allow and permit (potentially untrusted) code to access certain resources and perform (potentially dangerous) operations

- **Which operations can you think of?**

# SECURITY MANAGER OPERATIONS AND PERMISSIONS

- What can be regulated by the **SecurityManager**
  - File Access
  - Opening Sockets
  - Accessing System Properties
  - Application Termination
  - Class loader creation and setting
  - AWT event queue access
  - Top-Level Window Opening(Frame)
  - Installing other security managers
  - ….

# SECURITY CHECK CONCEPT

- Class library implements security checks in potentially dangerous program paths
- E.g. Security Check in **System.exit**

```
public void exit(int status) {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        checkExit(status);
    }
    Shutdown.exit(status);
}
```

Can throw **SecurityException**

# INSTALLATION OF A SECURITY MANAGER

- Per default no security manager is installed
  - Therefore **no checks are performed**

- Application must define and install a security manager itself: 2 ways to do so
  - Programmatically

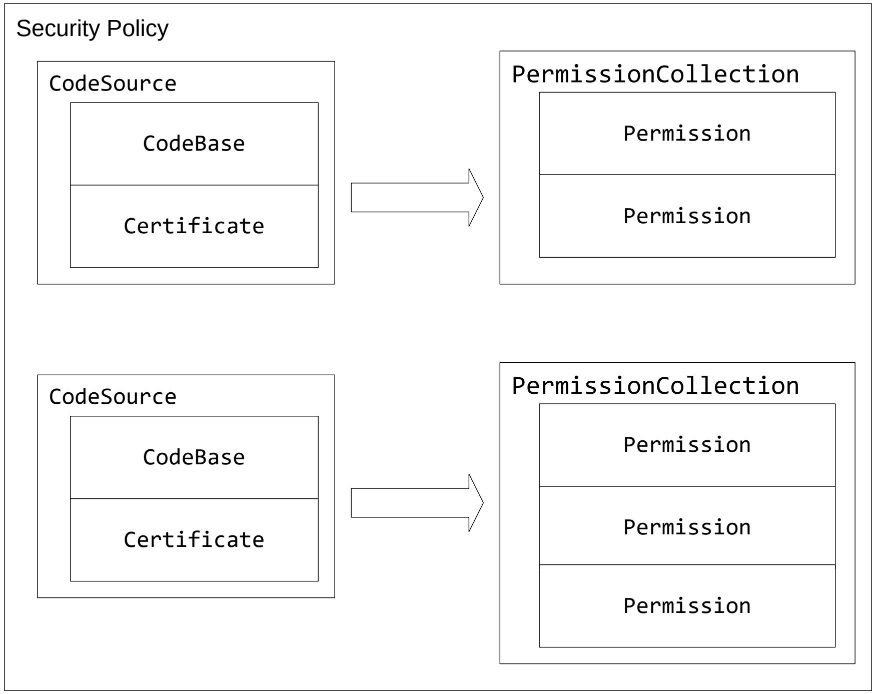
```
System.setSecurityManager(SecurityManager sm)
```

  - On the command line at program start

```
java -Djava.security.manager ...
```

# JAVA SECURITY MODEL

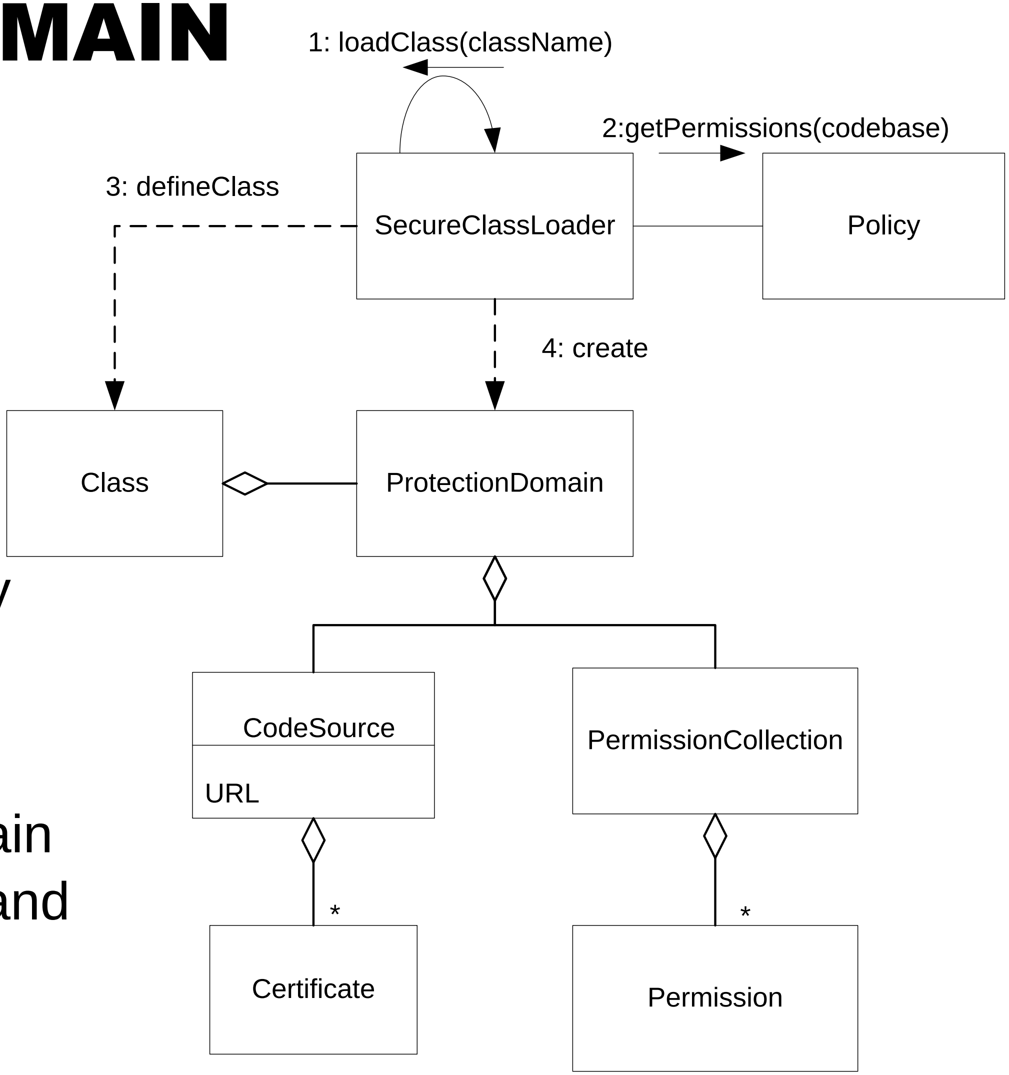
- Security policies which define mapping, i.e., code from specific source is given certain permissions
  - □ Mapping from **code source** to **permissions**
- Code sources and permissions represented by classes and objects
  - □ Class **CodeSource** with **CodeBase** and **Cerificate**
  - □ Class system of **Permissions**
  - □ **PermissionCollection** are collections of permissions
- Security policies are defined in policy files



# PROCESS OF CREATION OF A PROJECTION DOMAIN

Protection domain groups code source and permissions

1. Load class by **SecureClassLoader**
2. Get permissions from policy based on the **CodeBase**
3. Definition of class
4. Creation of protection domain for class with code source and permission collections



# OPERATIONS WITH SECURITY CHECKS

- Process of a security check
  - □ **System.getSecuirtyManager** is used
  - □ If installed (!=null) calling **checkPermission** on **SecurityManager**
  - □ If check is passed execution continues
  - □ If check fails **SecurityException** is **thrown**

```
public void <checkedOperation>(...) {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkPermission(new …Permission(…));
    }
    <uncheckedOperation>(...);
}
```

Can throw **SecurityException**

```
public void connect(SocketAddress endpoint, int timeout) throws IOException {
    …
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkPermission(new SocketPermission(host+":"+port,
                SecurityConstants.SOCKET_CONNECT_ACTION));
    …
}
```

Simplified

# CHECK PERMISSIONS

- Permissions are checked with **checkPermission** of the **SecurityManager**

```
public void checkPermission(Permission p)
```

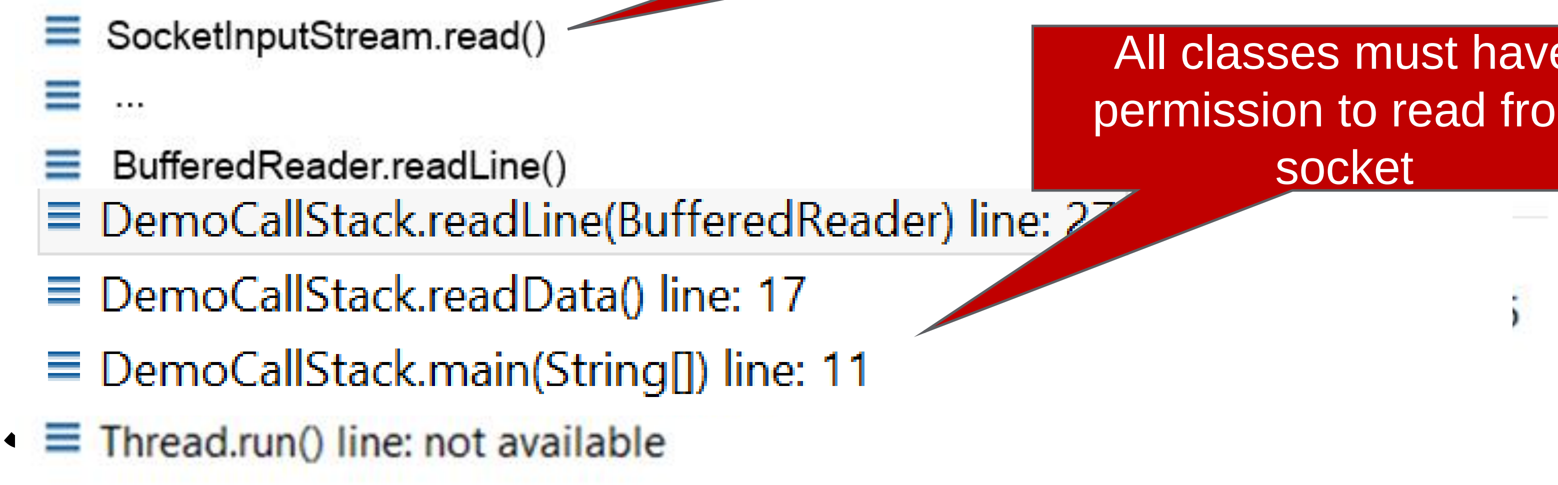
- Algorithm (simplified) to check for Permission p

```
checkPermission(Permission p) throws SecurityException {
   for all classes clazz of methods on call stack {
       permColl =  clazz.getProtectionDomain().getPermissions();
       if (! exists q in permColl with q.implies(p) )
            throw SecurityException(...);
   }
   permission p granted
}
```

# EXAMPLE: SOCKET

```
public class DemoCallStack {
    public static String readData() throws IOException {
        String fileName = "data.txt";
        try (BufferedReader r = new BufferedReader(new FileReader(fileName))) {
            String line = r.readLine();
            while (line != null) {
                …
                line = r.readLine();
            }
        }
    }
    public static void main(String[] args) throws IOException {
          readData();
    }
```

Stacktrace



```
SocketInputStream.read()
...
BufferedReader.readLine()
DemoCallStack.readLine(BufferedReader) line: 2
DemoCallStack.readData() line: 17
DemoCallStack.main(String[]) line: 11
Thread.run() line: not available
```

Needs permission

All classes must have permission to read from socket

# PERMISSION IMPLICIATIONS

- Permissions implements a method **implies** which checks if the given permission implies another permission

```
boolean implies(Permission permission)
```

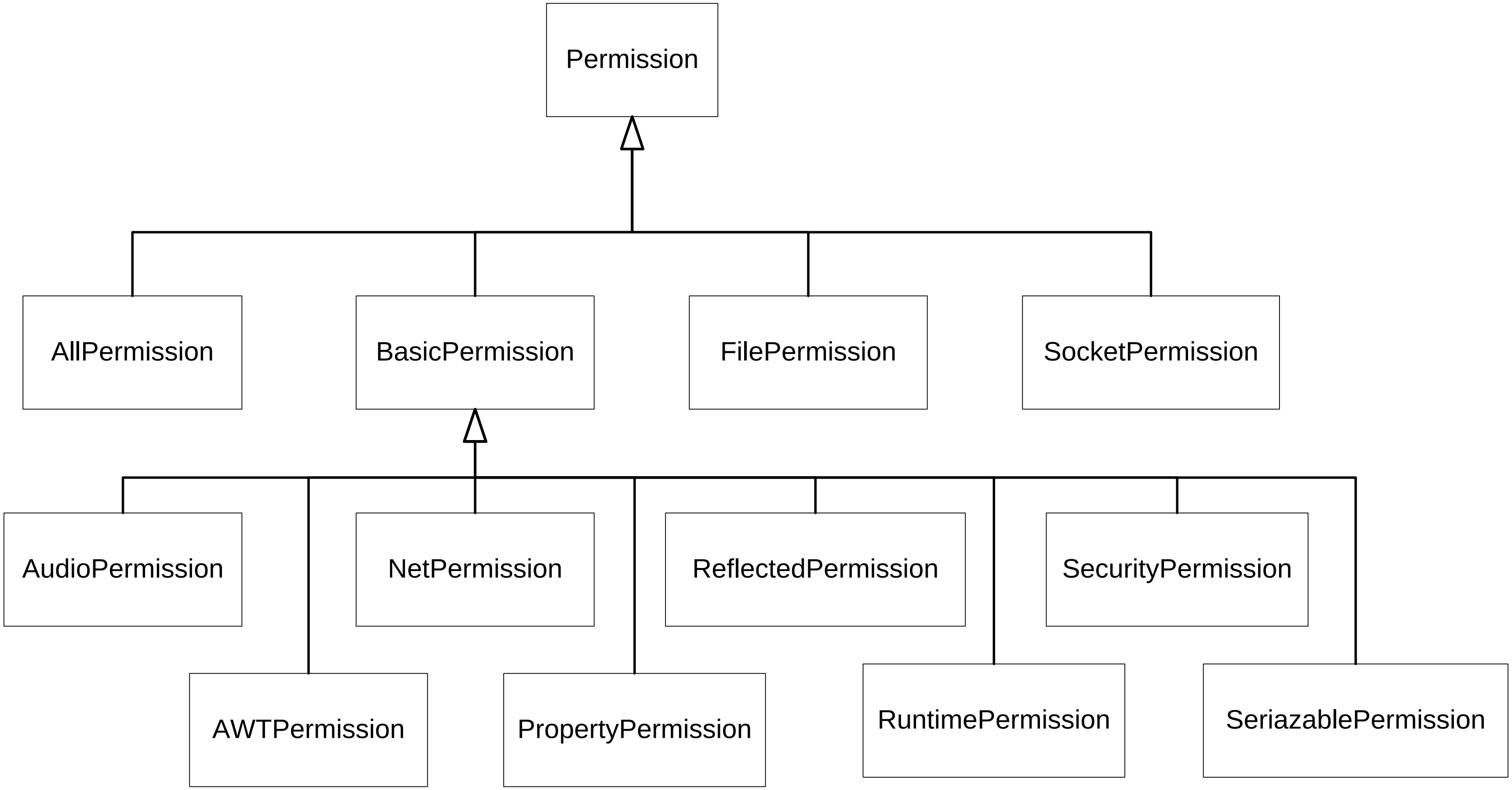
Examples

```
RuntimePermission("*")
   implies
RuntimePermission("ExitVM")

FilePermission(“C:\temp\*“, “read“)
   implies
FilePermission(“C:\temp\MyFile“, “read“)

SocketPermission( “*:1024-65535", “connect”)
   implies
SocketPermission “yourserver.com:1099", “connect“)
```

# PERMISSION CLASS HIERARCHY



Can be parameterized

# PERMISSION SPECIFICATION IN POLICY FILES

- Permissions are **usually** defined in **policy files** which contain a sequence of **grant entries**
  - Mapping from Code Source to Permissions

  - Code source consists of
    - URL for the code base
    - Name of trusted certifiers
  - Permissions in the form of
    - **permission** keyword
    - Class name of **permission** class
    - A permission-specific target for the permission (e.g. directory)
    - An optional list of permission-specific actions

```
grant Codesource {
  Permission_1;
  Permission_2;
};
```

```
grant
  codebase codebase-URL
  signedby certificate-name ... {
```

```
grant
  codebase codebase-URL
  signedby certificate-name ...
{
permission  permission-className
                 target
                 action1, ...;
...
};
```

# EXAMPLE POLICY FILE

All Code Sources can access properties and open sockets for reading

```
grant {
   permission java.lang.RuntimePermission "stopThread";
   permission java.net.SocketPermission "localhost:1024-", "listen";
   permission java.util.PropertyPermission "java.version", "read";
   permission java.util.PropertyPermission "java.vendor", "read";
        ...
};
```

JDK Extensions get all permissions

```
grant codeBase "file:${java.home}/lib/ext/*" {
   permission java.security.AllPermission;
};
```

SSW code base can only use certain files and sockets

```
grant codeBase “www.ssw.uni-linz.ac.at/classes/" {
   permission java.net.SocketPermission “*:1024-65535", “connect";
   permission java.io.FilePermission “${user.home}${/}-“,
                                            “read “, “ write “, “execute“;
   ...
};

...
```

# PERMISSIONS (EXCERPT)

| Permission | Target | Action |
|---|---|---|
| java.io.FilePermission | File | read, write, execute, delete |
| java.net.SocketPermission | Socket (host, port) | accept,connect, listen, resolve |
| java.util.PropertyPermission | Property I | read, write |
| java.lang.RuntimePermission | createClassLoader<br>createSecurityManager<br>exitVM<br>stopThread<br>queuePrintJob<br>... | |
| java.net.NetPermission | setDefaultAuthenticator<br>specifyStreamHandler<br>requestPassword-Authentication | |
| java.awt.AWTPermission | showWindowWithoutWarningBanner<br>accessClipboard<br>acccessEventQueue<br>listenToAllAWTEvents<br>readDisplayPixels | |
| java.security.SecurityPermission | getPolicy, setPolicy<br>... | |
| ... | | |

# SECURITY MANAGER EXAMPLE

- Installation

```
System.setSecurityManager(new SecurityManager());
```

- Then files cannot be read or written

```
try (BufferedReader reader = new BufferedReader(new InputStreamReader(
                                           new FileInputStream(filename)))) {
  for (String line = reader.readLine(); line != null; line = reader.readLine()) {
    System.out.println(line);
  }
} catch (Throwable t) { System.out.println("Unable to read file: " + t); }

try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
                                  new FileOutputStream(filename, true)))) {
  writer.append("Hello World!\n");
} catch (Throwable t) { System.out.println("Unable to write file: " + t); }
```

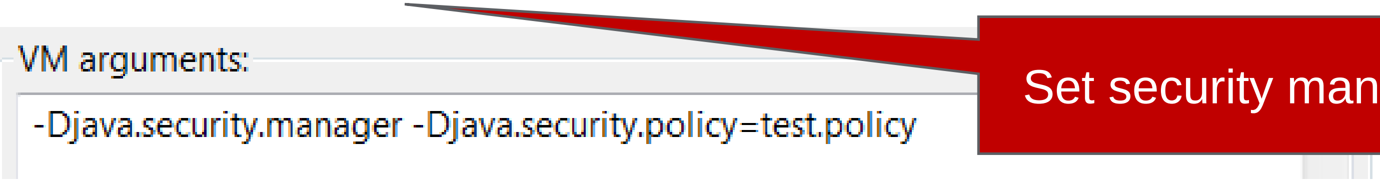
```
Unable to read file: java.security.AccessControlException: access denied
("java.io.FilePermission" "test.txt" "read")
Unable to write file: java.security.AccessControlException: access denied
("java.io.FilePermission" "test.txt" "write")
```

# EXAMPLE

- Policy file

```
grant codeBase "file:C:/.../UE03_WS/Security_Permissions_1/bin/-" {
   permission java.io.FilePermission "test.txt", "read";
   permission java.io.FilePermission "test.txt", "write";
};
```

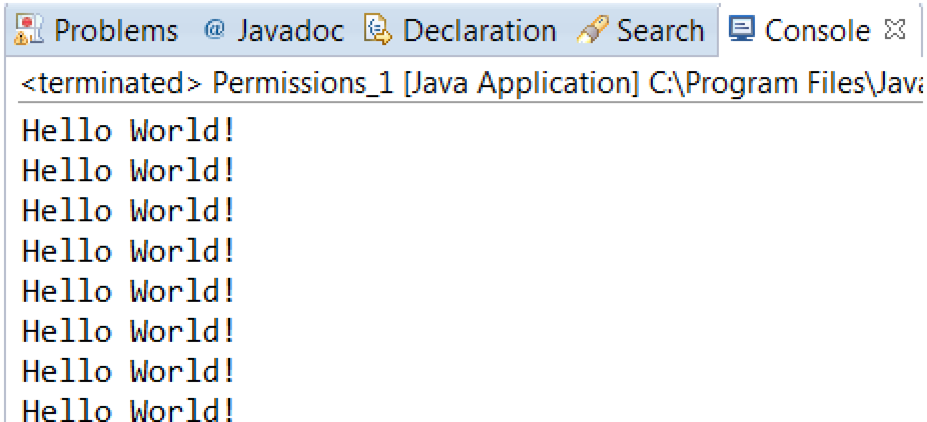
```
java -Djava.security.manager -Djava.security.policy=test.policy ...
```



VM arguments:

```
-Djava.security.manager -Djava.security.policy=test.policy
```

Set security manager



```
<terminated> Permissions_1 [Java Application] C:\Program Files\Javi
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

JOHANNES KEPLER
UNIVERSITY LINZ

# THANK YOU

**JOHANNES KEPLER**
**UNIVERSITY LINZ**
Altenberger Strasse 69
4040 Linz, Austria
www.jku.at

JKU

JOHANNES KEPLER
UNIVERSITY LINZ