Java security
(in a nutshell)

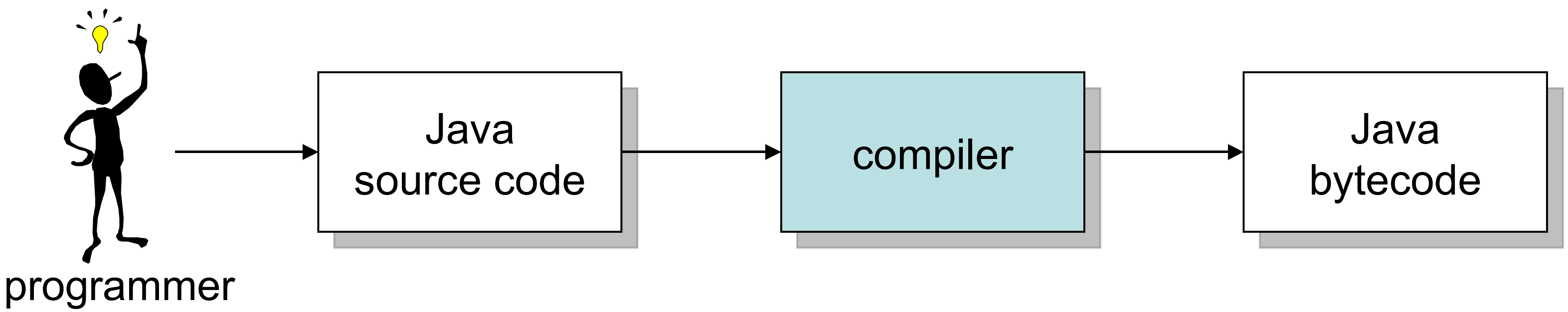
Outline

- components of Java
- Java security models
- main components of the Java security architecture
  - class loaders
  - byte code verification
  - the Security Manager

Components of Java

- the development environment
  - development lifecycle
  - Java language features
  - class files and bytecode
- the execution environment
  - the Java Virtual Machine (JVM)
- interfaces and architectures
  - e.g., Java beans, RMI, JDBC, etc.

Development lifecycle



notes

- Java is a high-level programming language
  → source code is English-like (syntax is similar to C)
- Java is *compiled* and *interpreted*
  - source code is compiled into bytecode (low-level, platform independent code)
  - bytecode is interpreted (real machine code produced at run time)

  → fast and portable ("write once run anywhere")
- dynamic linking (no link phase at compile time)
  - program consists of class definitions
  - each class is compiled into a separate class file
  - classes may refer to each other, references are resolved at run-time

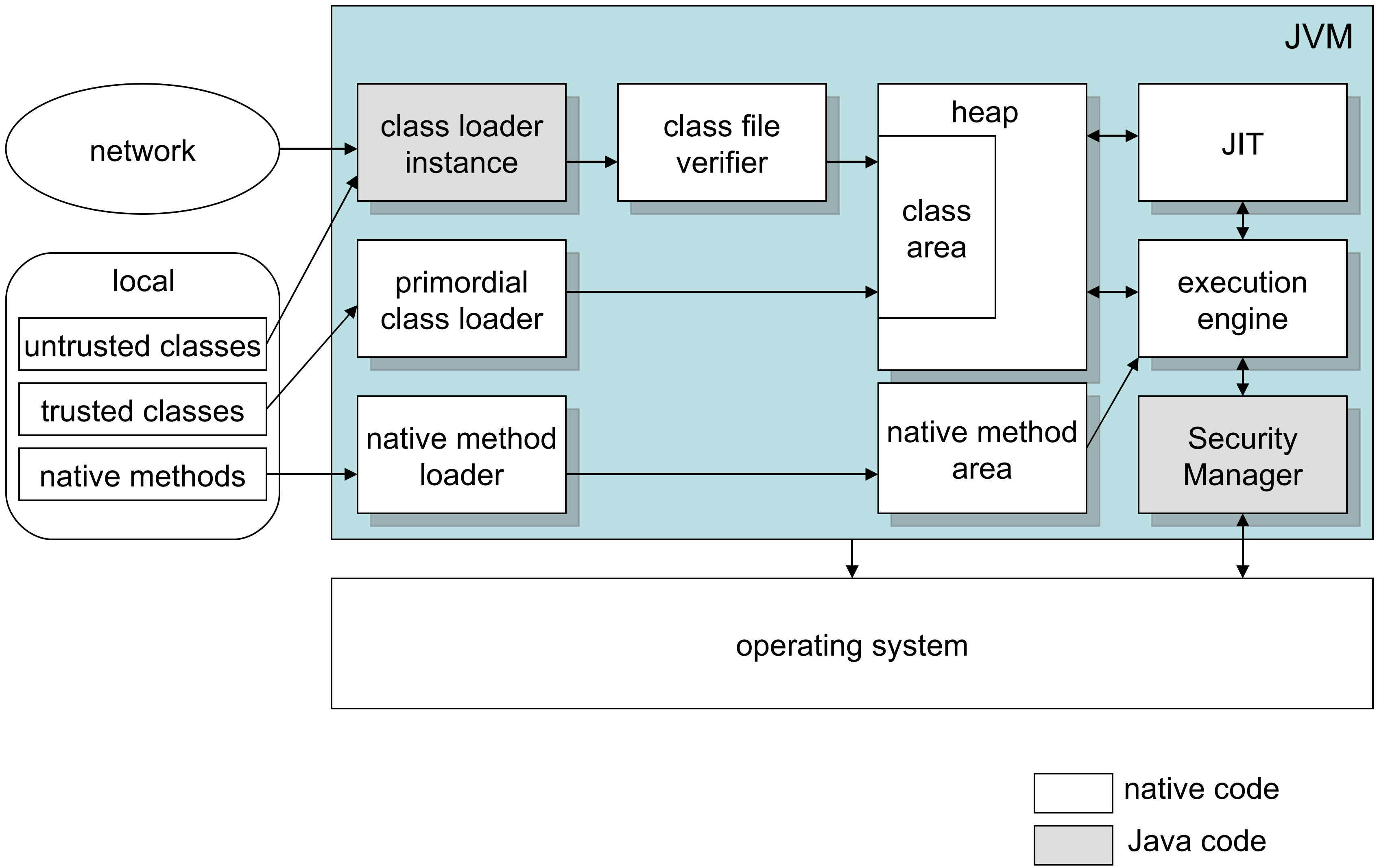
Java language features

- object-oriented
- multi-threaded
- strongly typed
- exception handling
- very similar to C/C++, but *cleaner* and *simpler*
  - no more struct and union
  - no more (stand alone) functions
  - no more multiple inheritance
  - no more operator overloading
  - no more pointers
- garbage collection
  - objects no longer in use are removed automatically from memory

Class files

- contain
  - magic number (0xCAFEBABE)
  - JVM major and minor version
  - constant pool
    - contains
      - constants (e.g., strings) used by this class
      - names of classes, fields, and methods that are referred to by this class
    - used as a symbol table for linking purposes
    - many bytecode instructions take as arguments numbers which are used as indexes into the constant pool
  - class information (e.g., name, super class, access flags, etc.)
  - description of interfaces, fields, and methods
  - attributes (name of the source file)
  - bytecode

The Java Virtual Machine (JVM)



JVM cont'd

- class loaders
  - locate and load classes into the JVM
  - primordial class loader
    - loads trusted classes (system classes found on the boot class path)
    - integral part of the JVM
  - class loader instances
    - instances of java.net.URLClassLoader (which extends SecureClassLoader)
    - load untrusted classes from the local file system or from the network and passes them to the class file verifier
    - application developers can implement their own class loaders
- class file verifier
  - checks untrusted class files
    - size and structure of the class file
    - bytecode integrity (references, illegal operations, …)
    - some run-time characteristics (e.g., stack overflow)
  - a class is accepted only if it passes the test

JVM cont'd

- native method loader
  - native methods are needed to access some of the underlying operating system functions (e.g., graphics and networking features)
  - once loaded, native code is stored in the native method area for easy access
- the heap
  - memory used to store objects during execution
  - how objects are stored is implementation specific
- execution engine
  - a virtual processor that executes bytecode
  - has virtual registers, stack, etc.
  - performs memory management, thread management, calls to native methods, etc.

JVM cont'd

- Security Manager
  - enforces access control at run-time (e.g., prevents applets from reading or writing to the file system, accessing the network, printing, ...)
  - application developers can implement their own Security Manager
  - or use the policy based SM implementation provided by the JDK
- JIT – Just In Time compiler
  - performance overhead due to interpreting bytecode
  - translates bytecode into native code on-the-fly
    - works on a method-by-method basis
    - the first time a method is called, it is translated into native code and stored in the class area
    - future calls to the same method run the native code
  - all this happens after the class has been loaded and verified

Java security models

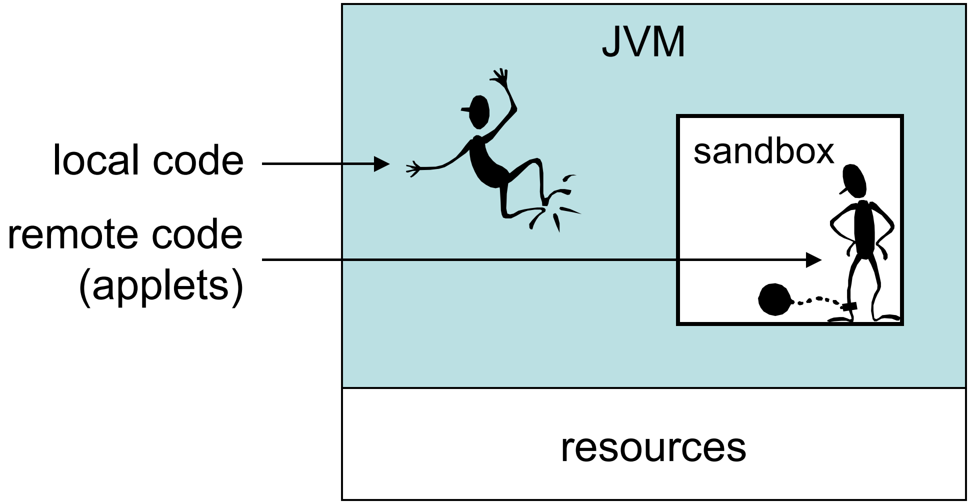
- the need for Java security
- the sandbox (Java 1.0)
- the concept of trusted code (Java 1.1)
- fine grained access control (Java 2)

The need for Java security

- code mobility can be useful (though not indispensable)
  - may reduce bandwidth requirements
  - improve functionality of web services
- but downloaded executable content is dangerous
  - the source may be unknown hence untrusted
  - hostile applets may modify or destroy data in your file system
  - hostile applets may read private data from your file system
  - hostile applets may install other hostile code on your system (e.g., virus, back-door, keyboard sniffer, …)
  - hostile applets may try to attack someone else from your system (making you appear as the responsible for the attack)
  - hostile applets may use (up) the resources of your system (DoS)
  - all this may happen without you knowing about it

The sandbox

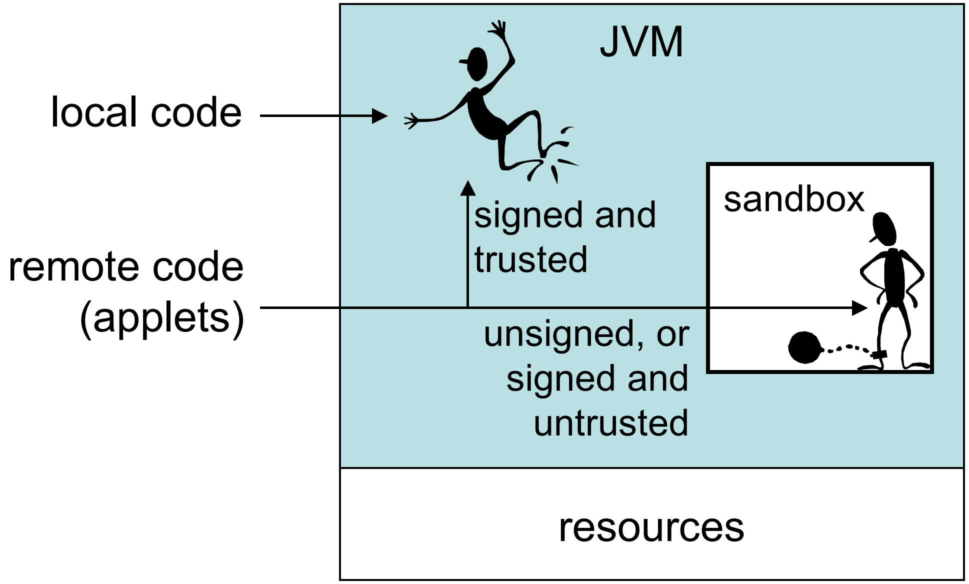
idea: limit the resources that can be accessed by applets



- **introduced in Java 1.0**
- **local code had unrestricted access to resources**
- **downloaded code (applet) was restricted to the sandbox**
  - **cannot access the local file system**
  - **cannot access system resources,**
  - **can establish a network connection only with its originating web server**

The concept of trusted code

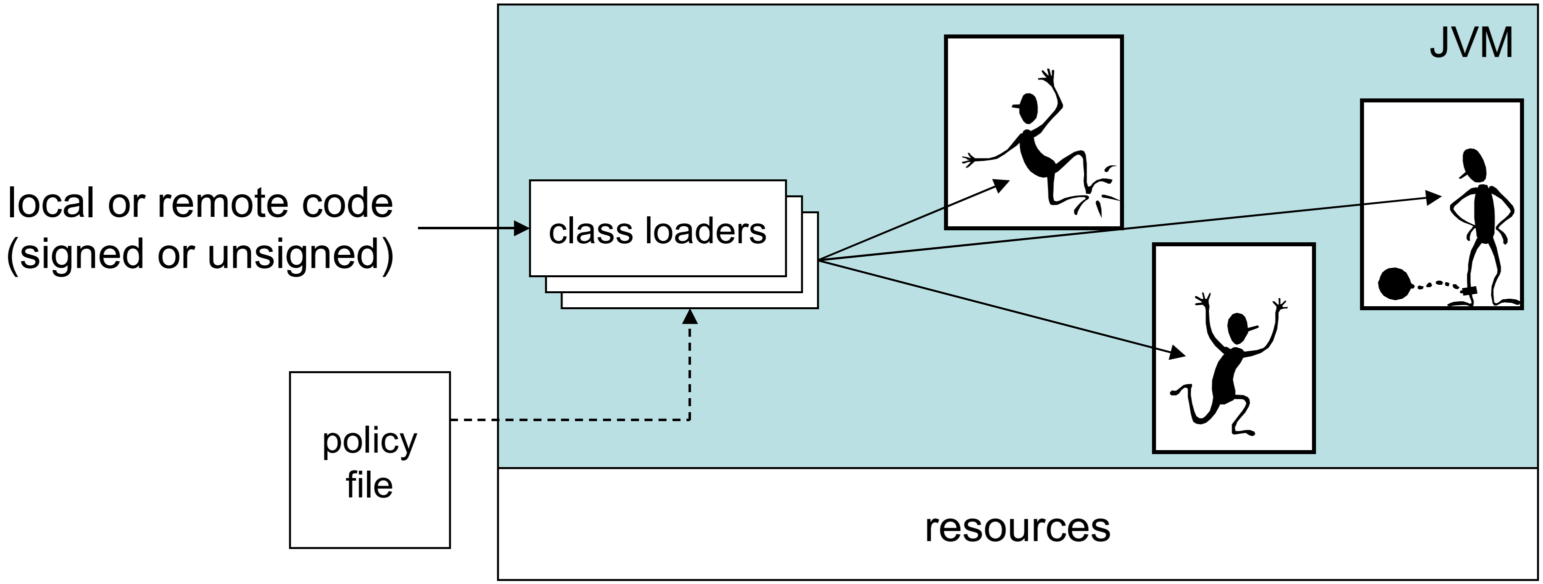
<u>idea</u>: applets that originate from a trusted source could be trusted



- **introduced in Java 1.1**
- **applets could be digitally signed**
- **unsigned applets and applets signed by an untrusted principal were restricted to the sandbox**
- **local applications and applets signed by a trusted principal had unrestricted access to resources**

Fine grained access control

<u>idea</u>: every code (remote or local) has access to the system resources based on what is defined in a *policy file*



- introduced in Java 2
- a *protection domain* is an association of a code source and the permissions granted
- the code source consists of a URL and an optional signature
- permissions granted to a code source are specified in the policy file

```
grant CodeBase “http://java.sun.com”, SignedBy “Sun” {
   permission java.io.FilePermission “${user.home}${/}*”, “read, write”;
   permission java.net.SocketPermission “localhost:1024-”, “listen”;};
```

The three pillars of Java security

- the Security Manager
- class loaders
- the bytecode verifier

The Security Manager

- ensures that the permissions specified in the policy file are not overridden
- implements a checkPermission() method, which
  - takes a permission object as parameter, and
  - returns a yes or a no (based on the code source and the permissions granted for that code source in the policy file)
- checkPermission() is called from trusted system classes
  - e.g., if you want to open a socket you need to create a Socket object
  - the Socket class is a trusted system class that always invokes the checkPermission() method
- this requires that
  - all system resources are accessible only via trusted system classes
  - trusted system classes cannot be overwritten (ensured by the class loading mechanism)

The Security Manager cont'd

- the JVM allows only one SM to be active at a time
- there is a default SM provided by the JDK
- Java programs (applications, applets, beans, …) can replace the default SM by their own SM only if they have permission to do so
  - two permissions are needed:
    - create an instance of SecurityManager
    - set an SM instance as active
  - example:

```
grant CodeBase "…", SignedBy "…" {
    permission java.lang.RuntimePermission "createSecurityManager";
    permission java.lang.RuntimePermission "setSecurityManager";};
```

  - invoking the SecurityManager constructor or the setSecurityManager() method will call the checkPermissions() method of the current SM and verify if the caller has the needed permissions

Class loaders

- separate name spaces
  - classes loaded by a class loader instance belong to the same name space
  - since classes with the same name may exist on different Web sites, different Web sites are handled by different instances of the applet class loader
  - a class in one name space cannot access a class in another name space

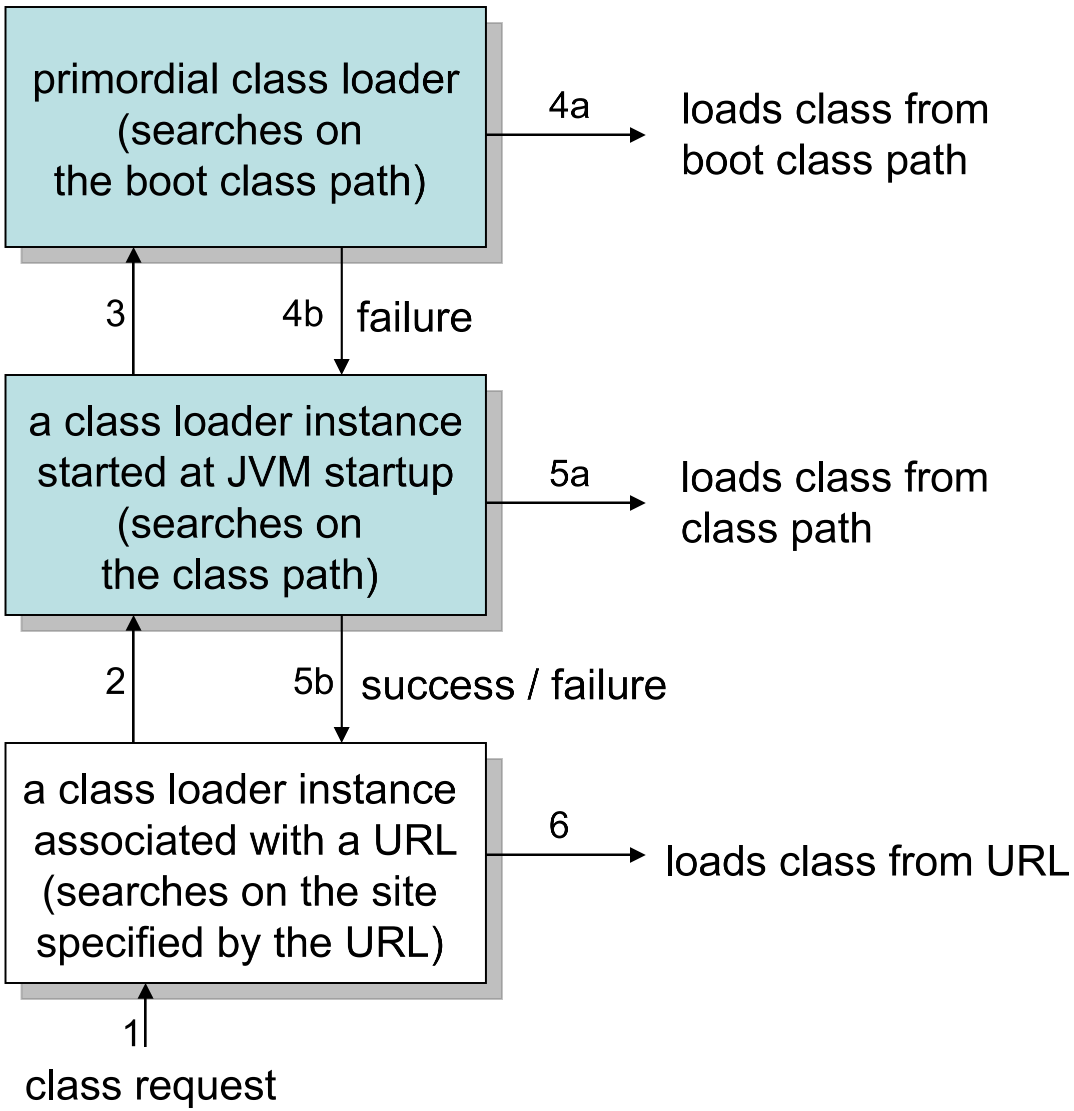
  → classes from different Web sites cannot access each other
- establish the protection domain (set of permissions) for a loaded class
- enforce a search order that prevents trusted system classes from being replaced by classes from less trusted sources
  - see next two slide …

Class loading process

when a class is referenced

- JVM: invokes the class loader associated with the requesting program
- class loader: has the class already been loaded?
  - yes:
    - does the program have permission to access the class?
      - yes: return object reference
      - no: security exception
  - no:
    - does the program have permission to create the requested class?
      - yes:
        - first delegate loading task to parent
        - if parent returns success, then return (class is loaded)
        - if parent returned failure, then load class and return
      - no: security exception

Class loading task delegation



Byte code verifier

- performs *static* analysis of the bytecode
  - syntactic analysis
    - all arguments to flow control instructions must cause branches to the start of a valid instruction
    - all references to local variables must be legal
    - all references to the constant pool must be to an entry of appropriate type
    - all opcodes must have the correct number of arguments
    - exception handlers must start at the beginning of a valid instruction
    - …
  - data flow analysis
    - attempts to reconstruct the behavior of the code at run time without actually running the code
    - keeps track only of types not the actual values in the stack and in local variables
  - it is theoretically impossible to identify all problems that may occur at run time with static analysis

Comparison with ActiveX

- ActiveX controls contain native code
- security is based on the concept of trusted code
  - ActiveX controls are signed
  - if signer is trusted, then the control is trusted too
  - once trusted, the control has full access to resources
- not suitable to run untrusted code
  - no sandbox mechanism