# *Java Card™ 2.1.2 Off-Card Verifier*

*White Paper*

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300

Sun Confidential Beta Release
Revision 1.0
January 31, 2001

# Contents

1

# *Java Card™ Off-Card Verifier*

## *Introduction*

This paper describes Sun's Java Card 2.1.2 Off-Card Verifier, Beta Release. The Off-Card Verifier provides functionality for verifying `CAP` files and `export` files. When applied to the set of `CAP` files that will reside on a Java Card-compliant smart card and the set of `export` files used to construct those `CAP` files, the Off-Card Verifier provides the means to assert that the content of the smart card has been verified.

Verification determines whether `CAP` and `export` files conform to the Java Card 2.1.1 specifications. As such it is an enforcer of the Java Card interoperability standards. The set of conformance checks provides that such files do not attempt to compromise the integrity of a Java Card virtual machine implementation, and hence other applets.

## *Purpose of Off-Card Verification*

As is well known, many of today's smart cards have extreme size limitations making it often infeasible to place verifiers onto Java Card-compliant smart cards. As larger smart cards continue to become cost-effective, the move to on-card verification will almost certainly occur. Both under today's circumstances and as on-card verification becomes available, Off-Card verification can play an integral role in the construction of secure Java Card-compliant smart cards.

Off-Card verification provides a means for evaluating `CAP` and `export` files in a desktop environment. Since debugging in a desktop environment is far easier than in a smart card emulator environment, it is advantageous to determine whether a `CAP` or `export` file is corrupt in a user-friendly environment. In this way the Off-Card Verifier serves as a gatekeeper, preventing a large set of complex problems from occurring when debugging on a smart card emulator.

Off-Card verification provides a complete solution for Java Card when additional security constructs are applied. Security constructs are beyond the scope of verification and are implementation dependent. These features of Java Card are described in more detail later in this paper.

## *Runtime Verification*

In a Java virtual machine implementation, a `class` file is loaded into a runtime environment, linked, initialized, and executed. Linking includes both verification and resolution. Verification confirms whether the content of a `class` file is structurally valid, while resolution matches symbolic references in a `class` file with the referenced items. In Java Card-compliant smart cards, installing a `CAP` file is equivalent to loading, resolving, and initializing. The smart card constitutes a runtime environment. In both platforms while a binary remains loaded it is maintained securely, preventing invalidation of verification that may have been performed.

Linking in a Java-compliant runtime environment typically follows an execution path, although there are many variations on timing. Resolution can be performed when an executable is verified, just before a reference-type instruction is executed, or at sometime in between. The common requirement for each variation is that nothing be executed before it is verified and resolved.

Following an execution path, verification and resolution begins with the entry method of an applet or application and proceeds into low-level, referenced libraries. This method can be characterized as *top-down*. If a referenced item is not located, resolution fails throwing a `ClassNotFoundException` or `NoSuchField/MethodException` exception. It is required that the full set of referenced binaries be present.

Figure 1-1 provides an example of this process. In the example the constructor method in a `MyApplet` class invokes the constructor method in the `Applet` class of the `javacard.framework` package. The constructor of the `Applet` class in turn invokes the constructor method in the `Object` class in the `java.lang` package.

Verification and resolution follows this chain of invocations, examining each class encountered. Assuming in the example that the constructor of the `MyApplet` class is the first element referenced, the process begins by loading linking (verifying and resolving), and initializing, the `MyApplet` class. While examining the constructor of `MyApplet`, the method invocation of the `javacard.framework.Applet` class is encountered. The same process of loading, linking, and initializing can then be applied to the `Applet` class. This process continues until java.lang.Object has been fully examined.

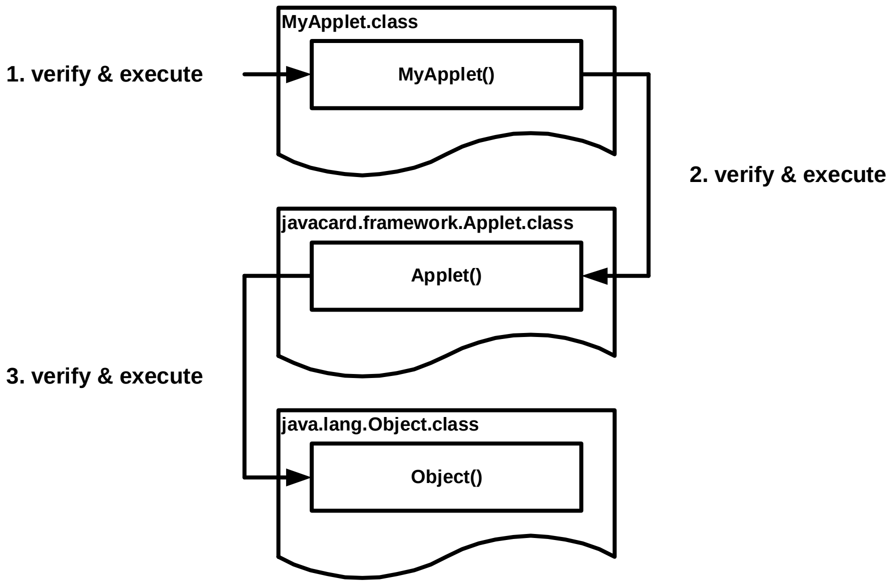

Figure 1-1 Verification Coupled with Execution

## Remote Verification

Like runtime linking, verification and resolution that is decoupled from a runtime environment also has numerous solutions, but all of the solutions have the same fundamental requirement. Each binary unit must be examined to determine not only whether it is internally consistent (verified), but also consistent with the context of other binary units it references (resolved). In the Java platform, remote verification of a class entails examining the `class` file and the set of directly and indirectly referenced `class` files. In the Java Card platform, remote verification entails examining both a CAP file and the set of directly and indirectly referenced CAP files.

Referenced classes and CAP files provide the context in which a respective class of a CAP file will be executed. Consider an example. A method can be examined to determine whether it is internally consistent by checking that it does not contain a branch instruction that attempts to jump outside of the method, or does not attempt to perform an arithmetic operation on a reference-type. If the method invokes another method, checks must be performed to determine whether the parameters passed during the invocation match the parameters expected by the invoked method, among other things. The referenced method, which may reside in a different CAP file, provides the appropriate context.

## *Limitations of Non-Incremental Remote Verification*

One solution for Java Card remote verification is to collect all of the CAP files that will be installed on a particular Java Card-compliant device and verify them together. This non-incremental solution has limitations.

This solution does not readily avail itself to incremental installations. If a CAP file is to be installed on a Java Card-compliant device that has already been populated with CAP files, the set of resident CAP files must be made available to an Off-Card Verifier. Since some CAP file installations are expected to occur post-issuance, this is undesirable.

A further limitation of non-incremental remote verification is potential disclosure of proprietary functionality. It is often the case that multiple vendors contribute CAP files that will reside on the same Java Card-compliant device. Since a CAP file may contain an implementation of a proprietary algorithm, it may not be desirable to ship the implementation in the form of a CAP file to a second vendor. Instead it is preferable to place the CAP file onto a smart card and only publish its Application Programming Interface (API) to a second vendor.

Finally, non-incremental remote verification is dependent upon particular implementations of libraries. When a CAP file is installed, an On-Card Installer confirms whether compatible versions of the CAP files it references are available on the card. The installer does not confirm whether particular implementations of the referenced CAP files are available. Off-Card verification and resolution performed with a particular implementation of a library is insufficient for affirming that verification and resolution is valid with different implementations of the same library.

To overcome these limitations, Sun's Off-Card Verifier implements a more flexible solution.

## *Incremental Remote Verification*

Sun's Off-Card Verifier supports incremental verification and resolution of the set of `CAP` files that will be installed on a Java Card-compliant device. The unit of verification is a single `CAP` file. The context in which a `CAP` file can be executed is provided through the Application Programming Interface (API) of referenced packages as defined in their `export` files. Resolution is validated off-card by examining the `export` files of referenced packages.

An example is shown in Figure 1-2 of an applet package that references a library package. The `CAP` file of the applet package is verified and resolved in conjunction with the `export` file of a library package. Furthermore, the `CAP` file of the library package is verified and resolved in conjunction with its `export` file. After both `CAP` files pass verification and resolution with the common `export` file, they are considered to constitute a verified set of binary units.

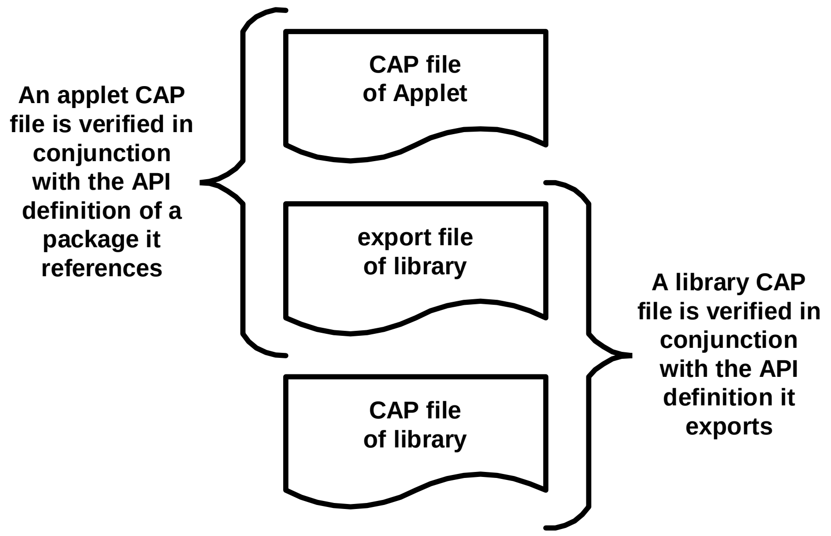
An applet CAP file is verified in conjunction with the API definition of a package it references
CAP file of Applet
export file of library
CAP file of library
A library CAP file is verified in conjunction with the API definition it exports

Figure 1-2 `export` Files Provide Context in CAP file Verification

The use of `export` files in remote verification renders results equivalent to those in runtime verification and resolution. At runtime when a method or other element in a different binary unit is referenced, the referenced binary unit is loaded, verified, resolved, initialized, and in the case of a method, executed. In the process of resolution, the features checked are those confirming that the manner in which the item is referenced matches the declaration of the item. An example of matching a method invocation with the declared method signature was described above. When examining an API definition of the referenced binary unit, sufficient information is available to validate resolution. The process can stop after this step, assuming that the implementation of the API has already been determined to be consistent with the API definition.

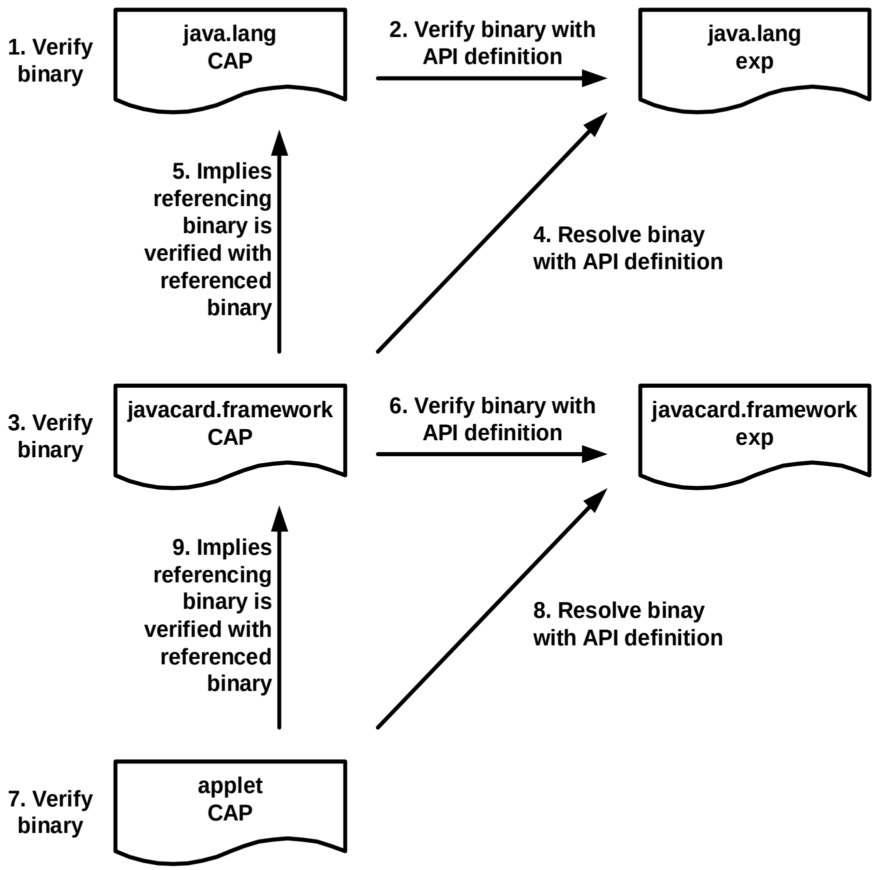
1. Verify binary
java.lang CAP
2. Verify binary with API definition
java.lang exp
5. Implies referencing binary is verified with referenced binary
4. Resolve binay with API definition
3. Verify binary
javacard.framework CAP
6. Verify binary with API definition
javacard.framework exp
9. Implies referencing binary is verified with referenced binary
8. Resolve binay with API definition
7. Verify binary
applet CAP

Figure 1-3 Incrementally Establishing a Verified Set of CAP files

Figure 1-3 shows a more detailed example of establishing a verified set of CAP files. In this example an applet package references the `javacard.framework` package, and the `javacard.framework` package references `java.lang`. First, the CAP file of `java.lang` is verified and resolved in conjunction with the `export` file of `java.lang`. Next, the CAP file of `javacard.framework` is verified and resolved in conjunction with the `export` file of `java.lang`. This establishes that `javacard.framework` and `java.lang` constitute a verified set. Verification of the CAP file of the applet package is the same. If the applet package also references `java.lang` directly, it is also resolved in conjunction with the `export` file of `java.lang`.

Unlike runtime verification described above, Sun's remote verification process begins with low-level libraries. This can be characterized as *bottom-up* verification. In the example of Figure 1-3, `java.lang` is verified first, and in subsequent verifications, `java.lang` is not verified again. This is analogous to the process performed by an optimized Java virtual machine where once `java.lang` has been loaded, verified, resolved, and initialized, it is not examined the succeeding times it is referenced. The same is the case for a Java Card-compliant device.

A Java Card-enabled device is a secure environment. At present the specifications do not explicitly support removing or upgrading a referenced library. Additional security measures, such as the firewall, prevent a library from being corrupted. Once a verified CAP file has been installed on a Java Card-compliant device its state cannot be changed. This includes both its internal state and its context.

Using this process of Off-Card verification and resolution, one can always assert that the content of a Java Card device is verified. Each library that has been placed on a card is verified, and subsequent installations are resolved with the API definitions of the content of the target card. The Java Card On-Card Installer acts as a marshal in this process. As indicated in the *Java Card™ 2.1.1 Runtime Environment (JCRE) Specification*, the On-Card Installer only accepts a CAP file when the CAP files it references are available and have versions compatible with the versions of those CAP files used during preparation.

An important advantage of incremental remote verification using `export` files is that the results are implementation independent. Since a referencing package's CAP is resolved with the API of a library, it is considered to be resolved with any implementation of that library. This assertion holds provided that each implementation of the library has been verified with the same `export` file.

## CAP File Verification

Figure 1-4 shows the inputs to the Off-Card Verifier when verifying a CAP file. These include the CAP file, the corresponding `export` file of the CAP file, and the `export` files of the packages the CAP file references. If the CAP file does not `export` any items, as is the case of an applet package with no public sharable interfaces, then a corresponding `export` file is omitted. The `java.lang` package is the only case where `export` files of referenced packages are omitted.

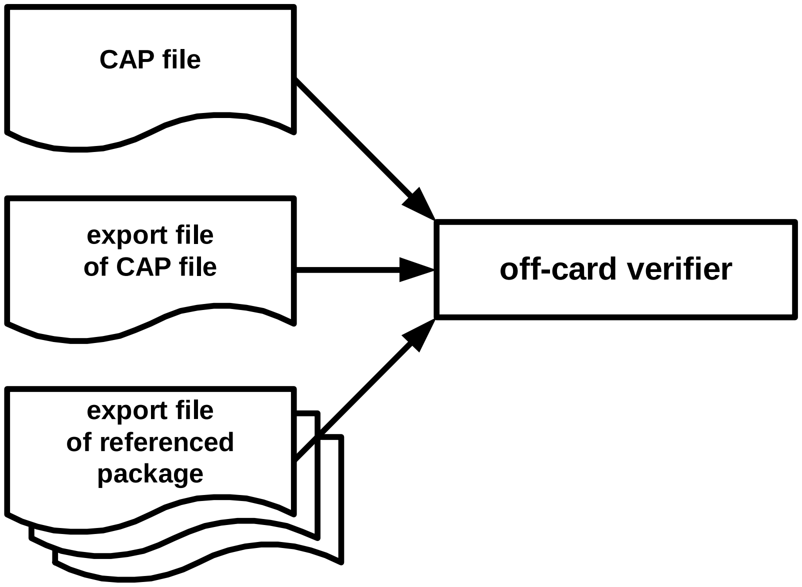

Figure 1-4 CAP file Verification

The process of CAP file verification includes three sets of checks: 1) verifying that a CAP file is internally consistent, 2) verifying that a CAP file is consistent with its corresponding `export` file, and 3) verifying that a CAP file is consistent with referenced `export` files. These checks are not necessarily performed sequentially. Each is described in general terms below.

Verifying that a CAP file is internally consistent and consistent with in the context of the referenced `export` file can be considered as a set of checks enumerated below.

1. Load: Entails reading a CAP file into the verifier, determining whether all required components are present and that each conforms to the general format of components.
2. Parse: Entails parsing each component to confirm that the basic syntax is correct, and that the values of individual fields are within appropriate ranges.
3. Link: Entails resolving inter- and intra-component references and confirming that referenced items are consistent with the references.
4. Verify CAP file Semantics: Entails confirming that class declarations and hierarchies are consistent with class representations.
5. Verify Byte Codes: Entails verifying each method. The byte code verification algorithm is described in "Byte Code Verification Algorithm" on page 13.

References in steps 3 through 5 are validated using the type information provided in the Descriptor Component of the CAP file. Reference to external items are further resolved by comparing their usage and type information in the CAP file to their declarations in the `export` file of the referenced CAP file.

Verifying that a CAP file is consistent with its corresponding `export` file entails matching all of the items that are be exported from a CAP file with those declared in its `export` file. The two sets must be exactly equal.

## *Export File Verification*

Each `export` file that is provided to the Off-Card Verifier during CAP file verification is also verified. Furthermore, the Off-Card Verifier can be invoked to verify a single `export` file. The later functionality is provided since `export` files play an integral role in both the construction of CAP files by the Java Card Converter and in Off-Card verification.

Verification of an `export` file is shallow, meaning that a single `export` file is examined at one time. The set of checks performed confirm that it is internally consistent and in conformance with the *Java Card™ 2.1.1 Virtual Machine*

*Specification*. Verification of the context of an `export` file is supported indirectly through `CAP` file verification.

The classes and interfaces represented in `export` files contain sufficient information about their hierarchies to construct a `CAP` file of a package that references the package of the `export` file. When a class in an `export` file extends a class defined in another package, some items in the superclass are represented. These items include the set of superclasses, the set of virtual methods, and the set of implemented interfaces. Each of these sets must be consistent with those listed in the `export` file of the superclass. If an inconsistency exists between `export` files it will be detected during `CAP` file verification since a `CAP` file must be consistent with both the `export` file it imports and the `export` file it exports.

## Compatibility Verification

`CAP` files may rely on the accessible interfaces, classes, methods and fields of other `CAP` files.

In a widely distributed system, a certain `CAP` file might be used by a number of other `CAP` files. Thus, it is preferred that different versions of this `CAP` file should not change its external view. In other words, all the publicly accessible classes, interfaces, methods and fields should still be accessible in the same fashion without any changes.

The Off-card verifier can be invoked to check compatibility between two versions of a package by comparing the respective versions of the export files. This verification examines whether the Java Card version rules have been followed. This includes the rules imposed for binary compatibility as defined in *Java Card™ 2.1.1 Virtual Machine Specification* section 4.4, have been followed. The scenario is shown in Figure 1-5.

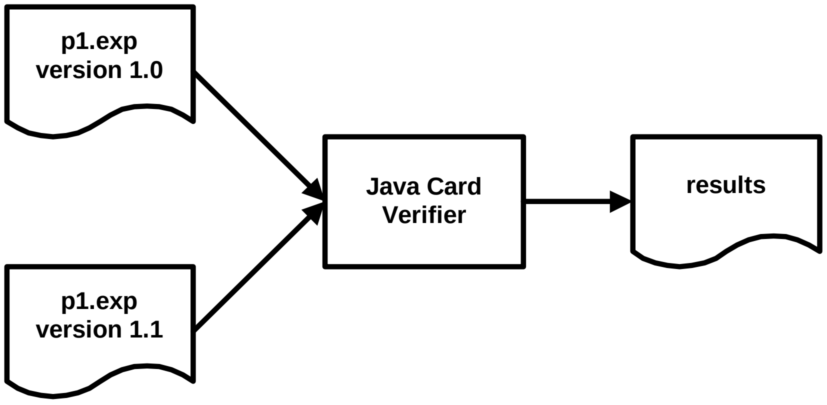

Figure 1-5 Compatibility Verification

Version comparison can only be made between packages that have the same major version number. This is because a change in the major version number means there has been a major change in the functionality. Such a change is considered a binary incompatibility. If the minor versions of the packages differ by more than one point, it may mean that the versions in between are not compatible. For example, version 1.1 and 1.9 may be binary-compatible, but 1.7 might not be compatible with 1.9. Therefore, if minor versions differ by more than one point, an implementation should raise a warning that the versions in between might not

be compatible. Such incompatibility is possible even if both minor versions are verified as binary-compatible. If the both the major and the minor versions of the packages are the same, the API should be checked for identical content.

## *Security Considerations*

Since verification is performed incrementally, it is required that all providers verify the set of `CAP` and `export` files they produce. Consumers of `CAP` and `export` files can add redundancy to the process by performing verification as well.

Off-Card verification is intended to be performed in a secure manner. Once a `CAP` file has passed verification, steps must be taken to help ensure that it is not corrupted before it is installed on a Java Card. These precautions are implementation dependent and may range from storing the verified CAP file in a trusted environment, encrypting the CAP file, signing the CAP file or some other security measure.

Since `export` files play an integral role in both the conversion and Off-Card verification processes, security measures must be applied to them as well. Precautions must be taken to ensure that an `export` file is not corrupted between the time it is created, verified with its corresponding CAP file, used by a Java Card Converter to create a referencing CAP file, and verified with that referencing CAP file. If an `export` file is corrupted after conversion of a referencing CAP file, verification will fail.

2

# *Byte Code Verification Algorithm*

## *Introduction*

This chapter describes the byte code verification algorithm in the Java Card Off-Card Verifier.

## *CAP File Format*

The Java Card `CAP` file format is analogous to the Java `class` file format, with some notable differences. A `CAP` file contains a representation of all the classes defined in a Java `package`, while a `class` file contains one Java `class`.

The information in a `CAP` file is distributed across a number of individual components, where each component contains information about one aspect of the classes in a Java `package`. For example, a Class component contains information defining the set of classes in a Java `package`, and a Method component contains information defining the set of methods (byte codes) in a Java `package`. The information in these components is optimized for the small, smartcard platform.

Each individual component is stored in a separate file, and all of the component files are stored in one JAR file. All components have the same fundamental structure:

`tag` – identifies the type of the component

`size` – indicates the number of bytes in the `info[]`

`info[]` – contains the data

The format of the content of an `info[]` array is specific to a particular component.

## Context of the Algorithm

In the Off-Card Verifier the process of verification consists of two steps:

- for each component, read the data
- for each component, verify the data

When the data of each component is read, it is stored in an `info[]` array.

When the data in a component is verified, it may be parsed and stored in a set of data structures that simplify the process of examination. In the case of the Method component, the byte codes of each method are parsed and stored in an array of instructions.

## Entry Point

The byte code verification is triggered through the `verify` method of the `MethodComponent` class. For byte codes of each method, `verify` invokes `method_verify`, which performs some basic verification checks ensuring that boundaries, header, and associated exception handlers of the method are valid. `method_verify` then invokes the `typecheck` method in the `Checkcode` class. `typecheck` performs the following:

- parses the byte codes and stores them in an array of instructions.
- verifies static features of each instruction by calling the method `static_check`. The `static_check` method determines whether the operands of each instruction have appropriate values. For example, an operand of a branch instruction must contain a jump value to the beginning of an instruction contained within the method. The set of checks is fairly straightforward.
- verifies the context of each instruction by calling an abstract interpreter. The abstract interpreter simulates execution of each instruction, using types of the data being operated on instead of values. This functionality is implemented in the `AbstrInterp` class and is described further below.

## Definitions of States

Both an interpreter of a Java virtual machine and the abstract interpreter of the Off-Card Verifier maintain state information used when executing or verifying instructions. The elements of the states are described in the sections below.

### Java Virtual Machine State

When a Java virtual machine interpreter executes a method, it maintains a *program counter* which indicates the location of the instruction being executed. After an instruction is executed, the program counter is set to the location of the next instruction that will be executed.

The Java virtual machine also maintains a *operand stack* and a set of *local variables*. Both of these contain values. When an instruction is executed it may consume (pop) items on the operand stack, and/or place (push) items onto the operand stack and/or modify the values of local variables.

Finally, the Java virtual machine interpreter also maintains a *stack pointer* which points to the top entry on the operand stack. The stack pointer is increased and decreased as values are pushed onto and popped off of the operand stack.

In summary, the *state* of the Java virtual machine interpreter just before an instruction is executed is defined by the following set:

- program counter,
- stack pointer,
- operand stack, and
- local variables.

### Abstract Interpreter State

As is the case for all byte code verifiers, the abstract interpreter of the Off-Card Verifier simulates execution using a program counter, stack pointer, operand stack, and a set of local variables. However, entries in the operand stack and local variables contain the type of the value being operated on instead of the value. For each instruction, the state of the operand stack and local variables are compared to the type(s) required during execution, and then are updated according to the operation of the instruction. For example, during simulation of the `int`-type add instruction (`iadd`) the top two entries on the operand stack must be type `int`. The `iadd` instruction is then simulated by popping both of the `int`-type entries off of the operand stack and pushing an `int`-type onto the operand stack.

The abstract interpreter also maintains state information related to subroutines for each instruction. Subroutines are used to implement `finally` blocks in `try-catch-finally` constructs. At the byte code level, they are implemented using the `jsr` and `ret` instructions. The state information for subroutines is maintained in a class called `Contour`. The information is used to match invocation of (`jsr` instruction) and return from (`ret` instruction) a subroutine and to perform various other checks.

In summary, the *state* of the abstract interpreter just before an instruction is executed is defined by the following set:

- program counter,
- stack pointer,
- operand stack,
- local variables, and
- contour.

As each instruction is simulated, it has an associated *before-state* and *after-state*. A before-state contains the state used to locate (via the program counter) an instruction and to simulate that instruction. An after-state contains the set of values resulting from simulation of an instruction. While most instructions have exactly one after-state there are a few exceptions. The `return` instructions have a `null` after-state since execution does not continue in a method after a return statement. Conditional branch instruction have two after-states, one for each execution path resulting from each branch.

## The Algorithm

The abstract interpreter is defined in a class called `AbstrInterp`. This class defines three central methods: `simulate`, `transition` and `enter_state`. Each are described further below.

The abstract interpreter uses two data structures to record states: `seen_states`, and `pending_states`. `seen_states` contains the set of states that have been simulated or are about to be simulated. `pending_states` contains the set of states that have not been simulated. The algorithm continues until there are no more pending states.

## *The `simulate` method*

The `simulate` method is the entry point for simulating one method. The functionality is as follows:

- Initialize `seen_states` with the empty set.
- Initialize `pending_states` with the empty set.
- Construct the initial state of the method based on the signature and local variables defined. Initialize the contour to be empty. Set the program counter to the first instruction.
- Store the initial state in both `seen_states` and `pending_states`. (This is a call to `enter_state` described below.)
- While `pending_states` is not empty:
  - Remove a state from `pending_states` and name it *before-state*.
  - Call `transition` with *before-state*.

## *The `transition` method*

The `transition` method simulates a single instruction. It receives as a parameter *before-state*, and does the following:

- Locate the instruction to be simulated using the program counter of before-state.
- Simulate the instruction:
  - Confirm whether *before-state* contains the appropriate values in the operand stack, local variables, and contour.
  - Create the operand stack, local variables and contour of the *after-state*.
- For each instruction that can be executed after this instruction:
  - Set the value of program counter of the *after-state* to that instruction.
  - Call `enter_state` with *after-state*.

When an instruction is simulated, the Off-Card Verifier must know the required types on the operand stack and/or local variables of the *before-state* and *after-state*. This is accomplished in one of two ways. When possible the information is obtained from a table, while for more complex instructions the information is hard-coded.

The table used in this process is named `stack-effect`. `stack-effect` is an array containing an entry for each instruction. Each entry contains the type(s) of the entries required to be on the operand stack of the *before-state* and the type of the entry to be place on the operand stack of the *after-state*. The syntax is as follows:

```
ordered list of before types, null, ordered list of after types, null
```

Either list may be empty depending upon the instruction. Here are two examples of entries in the `stack-effect` table:

```
sadd: short, short, null, short, null
sconst_0: null, short, null
```

For instructions that do not modify the operand stack or that operate on a class type, the entry in the `stack-effect` table is `null`. The `stack-effect` table is not used when simulating these instructions but instead the required information is hard-coded.

## *The `enter_state` method*

The `enter_state` method receives a state and records it in `seen_states` and/or `pending_states` as appropriate:

- If a state with the same program counter value is not found in `seen_states`:
  - Add the state in `seen_states`
  - Add the state in `pending_states`
- If a state with the same program counter value is found in `seen_states`:
  - Compute the (Least Upper Bound) LUB of local variables of the two states
  - If the LUB of the local variables is different than the local variables in `seen_states`:

    Set the local variables in `seen_states` to LUB of the local variables

    Record that `seen_states` was changed
  - Compute the (Least Upper Bound) LUB of operand stack of the two states
  - If the LUB of the operand stack is different than the operand stack in `seen_states`:

    Set the operand stack in `seen_states` to LUB of the operand stacks

Record that `seen_states` was changed

- If `seen_states` was changed:
  - Add the LUB state to `pending_states`

Determining the LUB of entries in the operand stack and local variable is a well-known process. If the types of the two corresponding entries are the same, nothing is done. If the types are not the same, an attempt is made to determine a general type that applies to both entries. For example, if two classes directly extend Object their LUB is Object. If class A extends Object and class B extends A, their LUB is class A.