ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

# Problems of Bytecode Verification

**Robert F. Stärk**

Department of Computer Science

ETH Zürich

E-mail: staerk@inf.ethz.ch

http://www.inf.ethz.ch/~staerk

Robert Stärk, Joachim Schmid, Egon Börger

**Java and the**
**Java Virtual Machine**

Definition, Verification, Validation

Springer-Verlag 2001 (to appear)

## Most important theorems in the book

**Theorem 1:** Java is type safe.

The run-time types of local variables are compatible to the declared types.

**Theorem 2:** The Java compiler is correct.

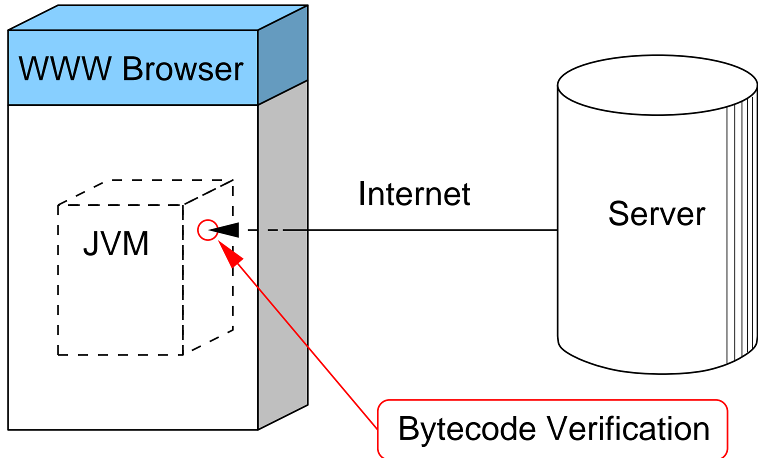
The program semantics is preserved by the compilation scheme.

**Theorem 3:** The Java bytecode verifier is correct.

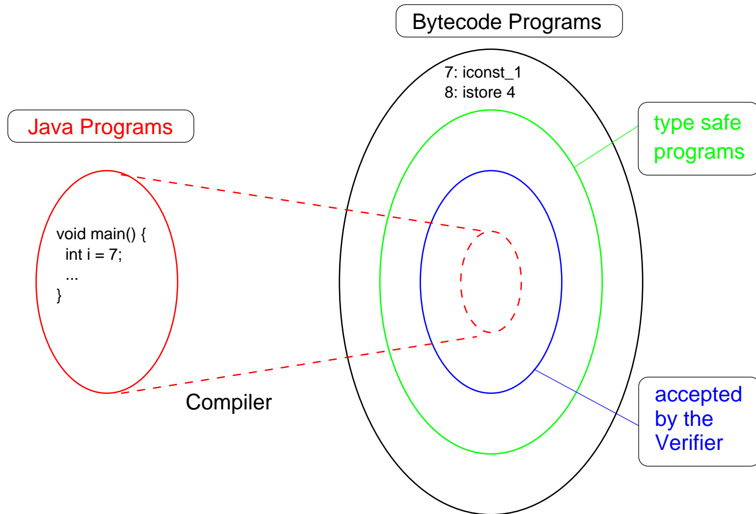
Bytecode which is accepted by the verifier does not violate run-time checks
(no type errors, no under- or overflow, no access to private fields, etc.)

**Theorem 4:** The Java compiler generates verifiable bytecode.

Bytecode from a legal Java program is accepted by the verifier.


WWW Browser
JVM
Internet
Server
Bytecode Verification

# Java Bytecode Verification



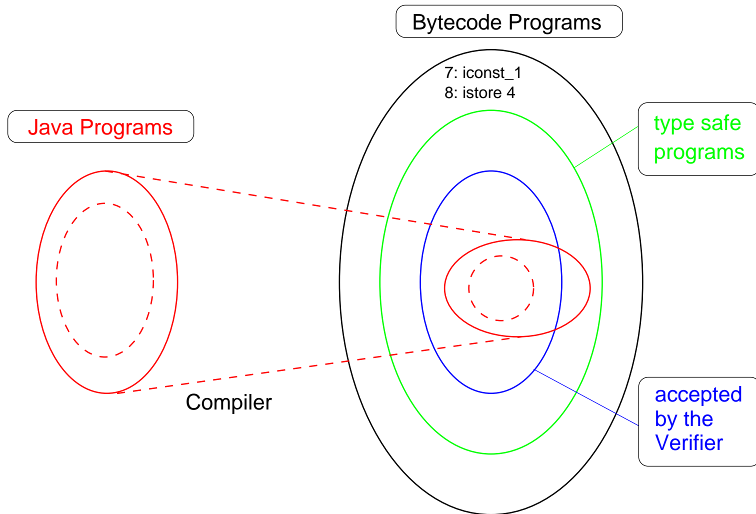
## What means “type-safe”?

The program satisfies certain structural constraints (JVM Specification §4.8.2):

- The program counter is always a valid code index.
- No dangling pointers.
- No overflow nor underflow of the operand stack.
- Local variables are initialized before they are loaded.
- Primitive operations have enough operands.
- The operands are of the required type.
- Existing fields of objects are accessed only.
- The values of the fields of objects are compatible with the declared types.
- Arguments of method invocations are compatible with the declared types.
- No type violations at run-time.

Remark: No limits on memory and time consumption.

# Java Bytecode Verification (Reality)



## Example 1: Legal Java program rejected by all verifiers

```
public class Test1 {
  int test(boolean b) {
    int i;
    try {
      if (b) return 1;
      i = 2;
    } finally { if (b) i = 3; }
    return i;
  }
}
```

```
java version "1.3.0"
sun> javac Test1.java
sun> java Test1
java.lang.VerifyError: Register 2 contains wrong type
```

```
Kimera verifier: Security flaw: DFA_LOCVAR_WRONG_TYPE 684
in class file Test1
```

## Example 2: Legal Java program rejected by all verifiers

```
public class Test2 {
  int test(boolean b) {
  int i;
  L: { try {
         if (b) return 1;
         i = 2;
         if (b) break L;
       } finally { if (b) i = 3; }
       i = 4;
     }
     return i;
  }
}
```

```
java version "1.3.0"
sun> javac Test2.java
sun> java Test2
java.lang.VerifyError: Register 2 contains wrong type
```

## Publications on Java bytecode verification


- R. Stata and M. Abadi: *A type system for Java bytecode subroutines*. POPL 1998.
- S.N. Freund and J.C. Mitchell: *A formal framework for the Java bytecode language and verifier*. OOPSALA 1999.
- Z. Qian: *Constraint-based specification and dataflow analysis for Java bytecode verification*. 1998.
- R. O'Callahan: *A simple, comprehensive type system for Java bytecode subroutines*. POPL 1999.

## Java Bytecode Verification = Static Analysis + Type Inference

```
int m(boolean b) {
  int i;
  try {
    if (b)
      return 1;
    i = 2;
  } finally {
    if (b)
     i = 3;
  }
  return i;
}
```

```
    iload_1    ()        {1:int}
    ifeq A     (int)     {1:int}
    iconst_1   ()        {1:int}
    istore_3   (int)     {1:int}
    jsr S      ()        {1:int,3:int}
    iload_3    ()        {1:int,3:int}
    ireturn    (int)     {1:int,3:int}
A:  iconst_2   ()        {1:int}
    istore_2   (int)     {1:int}
    jsr S      ()        {1:int,2:int}
    goto C     ()        {1:int} // 2 mod. by S
S:  astore 4   (ra(S))   {1:int}
    iload_1    ()        {1:int,4:ra(S)}
    ifeq B     (int)     {1:int,4:ra(S)}
    iconst_3   ()        {1:int,4:ra(S)}
    istore_2   (int)     {1:int,4:ra(S)}
B:  ret 4      ()        {1:int,4:ra(S)}
C:  iload_2    ()        {1:int}
    ireturn    // 2 contains wrong type
```

## Static analysis on the source level: “The rules of definite assignment”

$x \in \mathit{after}(\mathit{stm}) \iff x$ has definitely a value after the execution of $\mathit{stm}$.

$x \in \mathit{before}(\mathit{stm}) \iff x$ has definitely a value before the execution of $\mathit{stm}$.

**SUN (Java Language Specification, 2nd ed.):**

$\mathit{after}(\texttt{try}\ \mathit{stm}\ \texttt{finally}\ \mathit{block}) :=$

$\quad \mathit{after}(\mathit{stm}) \cup \mathit{after}(\mathit{block})$

$\mathit{after}(\mathit{lab}\colon \mathit{stm}) :=$

$\quad \mathit{after}(\mathit{stm}) \cap \bigcap\{\mathit{before}(\texttt{break}\ \mathit{lab}) \mid \texttt{break}\ \mathit{lab}\ \text{can exit}\ \mathit{stm}\}$

**Stärk:**

$\mathit{after}(\texttt{try}\ \mathit{stm}\ \texttt{finally}\ \mathit{block}) :=$

$\quad \{x \in \mathit{after}(\mathit{stm}) \mid \text{there is no}\ x = \mathit{exp}\ \text{in}\ \mathit{block}\} \cup \mathit{after}(\mathit{block})$

$\mathit{after}(\mathit{lab}\colon \mathit{stm}) :=$

$\quad \mathit{after}(\mathit{stm}) \cap \bigcap\{\mathit{before}(\texttt{break}\ \mathit{lab}) \mid \texttt{break}\ \mathit{lab}\ \text{can exit}\ \mathit{stm}\} \cap$

$\quad \bigcap\{\mathit{after}(\texttt{try}\ s\ \texttt{finally}\ b) \mid \texttt{break}\ \mathit{lab} \in s,\ \texttt{try}\ s\ \texttt{finally}\ b \in \mathit{stm}\}$

## Main result

**Theorem (Stärk, 2000):**

For the restricted Java language, the compiler generates verifiable bytecode.

**Proof.**

Step 1: Extension of the compiler. Compiler generates *Stack Maps*.

Step 2: The generated *Stack Maps* represent a *Bytecode Type Assignment*.

Step 3: Bytecode with type assignment is type safe at run-time.

Step 4: Bytecode verifier computes principal type assignment. □

## Types and subtypes of Java

**Types of Java:**

- Primitive types: `byte`, `short`, `char`, `int`, `long`, `double`, `boolean`
- Reference types: *Classes*, *Interfaces*, *Arrays*, `Null`

**Subtype relation for reference types $A \preceq B$:**

- $A \preceq A$.
- If $A$ is a subclass/subinterface of $B$ or $A$ implements $B$, then $A \preceq B$.
- `Null` $\preceq A$ and $A \preceq$ `Object`.
- If $A$ is an array type, then $A \preceq$ `Clonable` and $A \preceq$ `Serializable`.
- If $A \preceq B$, then $A[] \preceq B[]$.

## Types and subtypes of the Bytecode Verifier

**Types of the Bytecode Verifier:**

- Primitive: `int`, `lowLong`, `highLong`, `float`, `lowDouble`, `highDouble`
- Reference types: finite sets of Java reference types
- Object initialization types: `InInit`, $(Class, Pc)_{\mathsf{new}}$,
- Return address types: `retAddr`$(Pc)$
- Topmost type: `unusable`

**Subtype relation:** $\sigma \sqsubseteq \tau \;:\Longleftrightarrow$ one of the following conditions is satisfied:

- $\sigma = \tau$
- $\tau =$ `unusable`
- $\sigma$ and $\tau$ are finite sets of reference types and $\forall A \in \sigma\, \exists B \in \tau\, (A \preceq B)$

## Bytecode type assignments

A type assignment with domain $\mathcal{D}$ for a method $C/M$ consists of two families $(regT_i)_{i\in\mathcal{D}}$ and $(opdT_i)_{i\in\mathcal{D}}$ of type assignments to local registers and the operand stack such that:

**T1.** $\mathcal{D}$ is a set of valid code indices of the method $C/M$.

**T2.** $0$ belongs to $\mathcal{D}$.

**T3.** $regT_0$ assigns the argument types of $C/M$ to the local registers.

**T4.** $opdT_0$ is the empty sequence (the empty operand stack).

**T5.** If $i \in \mathcal{D}$ and $code(i) = K$, then the constraint $\varphi_K(regT_i, opdT_i)$ is true.

**T6.** If $i \in \mathcal{D}$ and $\langle j, regS, opdS\rangle$ is a successor of $\langle i, regT_i, opdT_i\rangle$, then $j \in \mathcal{D}$, $regS \sqsubseteq_{\mathsf{reg}} regT_j$ and $opdS \sqsubseteq_{\mathsf{opd}} opdT_j$.

# Bytecode type assignments (continued)

**T7.** If $i \in \mathcal{D}$, $code(i) = Ret(x)$ and $regT_i(x) = \texttt{retAddr}(s)$, then for all reachable $j \in \mathcal{D}$ with $code(j) = Jsr(s)$:

(a) $j + 1 \in \mathcal{D}$,

(b) $regT_i \sqsubseteq_{\mathsf{reg}} (mod(s) \lhd regT_{j+1})$,

(c) $regT_j \sqsubseteq_{\mathsf{reg}} (mod(s) \ntriangleleft regT_{j+1})$,

(d) $opdT_i \sqsubseteq_{\mathsf{opd}} opdT_{j+1}$,

(e) if $\texttt{retAddr}(\ell)$ occurs in $mod(s) \ntriangleleft regT_{j+1}$, then each code index which belongs to $s$ belongs to $\ell$,

(f) neither $(_, _)_{\mathsf{new}}$ nor InInit occur in $mod(s) \ntriangleleft regT_{j+1}$.

**T8.** If $i \in \mathcal{D}$ and $\texttt{retAddr}(s)$ occurs in $regT_i$, then $i$ belongs to $s$.

If $i \in \mathcal{D}$ and $\texttt{retAddr}(s)$ occurs in $opdT_i$, then $i = s$.