

An Introduction to JVM bytecode verifier

with an emphasis on Java platform safety

Hanbing Liu
April 24th, 2007

Java platform safety?

*Java programming language is created for **highly reliable software** With security features designed into the **language and run-time system**, Java technology lets you construct applications that **can't be invaded from outside**.*

---- Java Language Environment White Paper [1]

Compare with the C and C++?

[1] <http://java.sun.com/docs/white/langenv/>

- Reliable/safe software
 - Executions conform to some ground rules
 - Example: objects are always used in a type-safe way;
 - Method returns always return to the caller.
- To support writing reliable software, we have
 - Java language: object oriented, “final” modifier,
 - Run-time system: safe type cast, safe array access, access control,



Requirements on the JVM: safety vs. efficiency

- Ensure the executions of Java bytecode programs conform to some ground rules.

The naïve way is to always check no ground rules are broken.

However, we also want a JVM that can

- Execute these programs efficiently.
-
-

Meet the safety and the efficiency requirements:

- The simple idea:
 - Only execute safe programs that we know are safe
- This is worth implementing only when
 - We can efficiently identify a large subset of safe programs.

In many cases, we can. For example

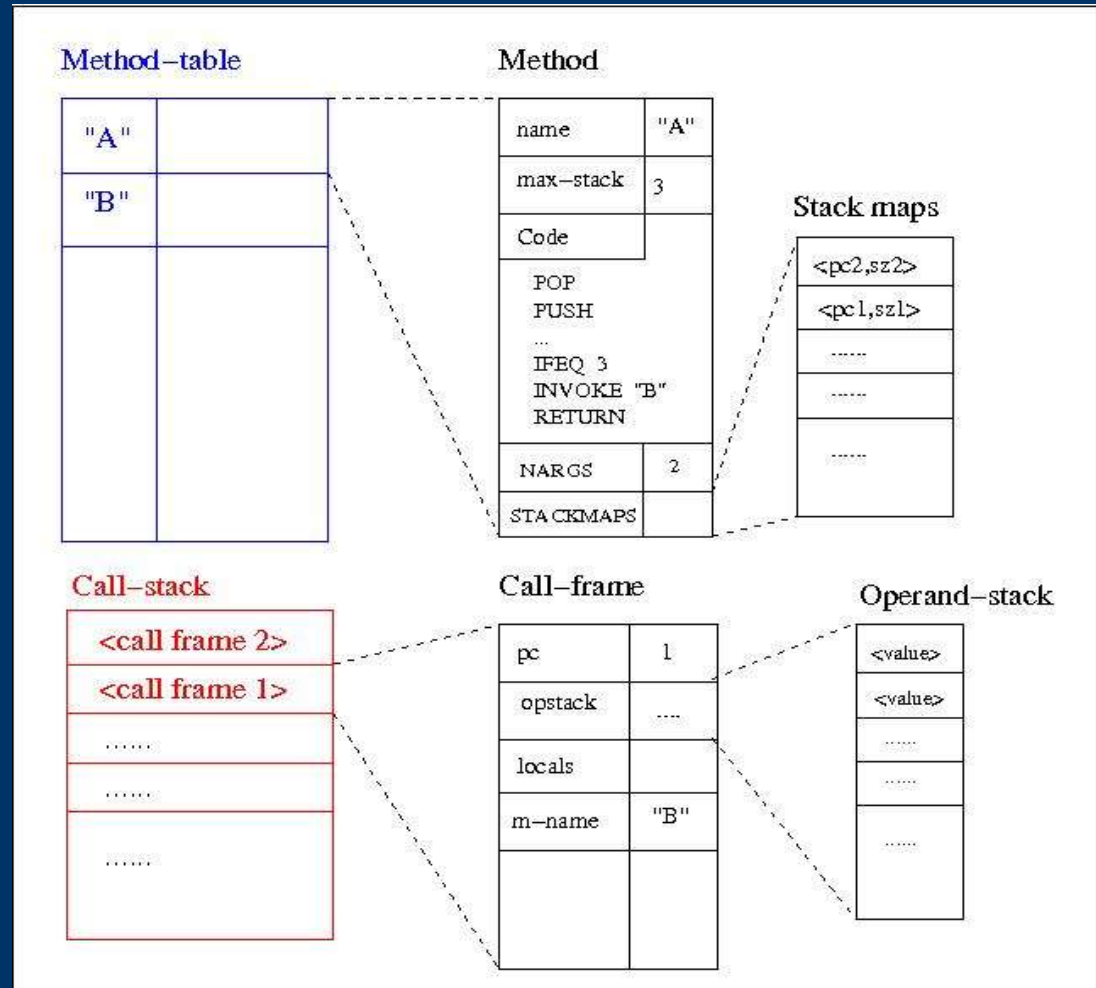
- Imagine a machine that computes product: $45 * 23 * -1 * 12$
 - Safety program: those produce positive result
-
-

Meet the safety and the efficiency requirements: simplified JVM

- Consider a M3-like machine:
 - *call stack, call frame, operand stack, locals,*
 - Safe program: those programs, during their executions, no operand stack holds more than “max-stack” values
 - *Each method specifies its own “max-stack” value.*
 - Bytecode verification algorithm
 - *Static checking algorithm; abstract execution*
-
-

M3-like machine

- State:
- Call frame:
- Method:



Difference with M3

- (PUSH V):
 - push value V onto the current operand stack
 - Effects are undefined:
 - if it will overflow the operand stack
 - if the current frame does not exist
 - if the method referred by the current frame does not exist
- (POP V):
- (INVOKE method-name):
 - Effects are undefined
 - if there are not enough operands (less than nargs)

Ground rules:

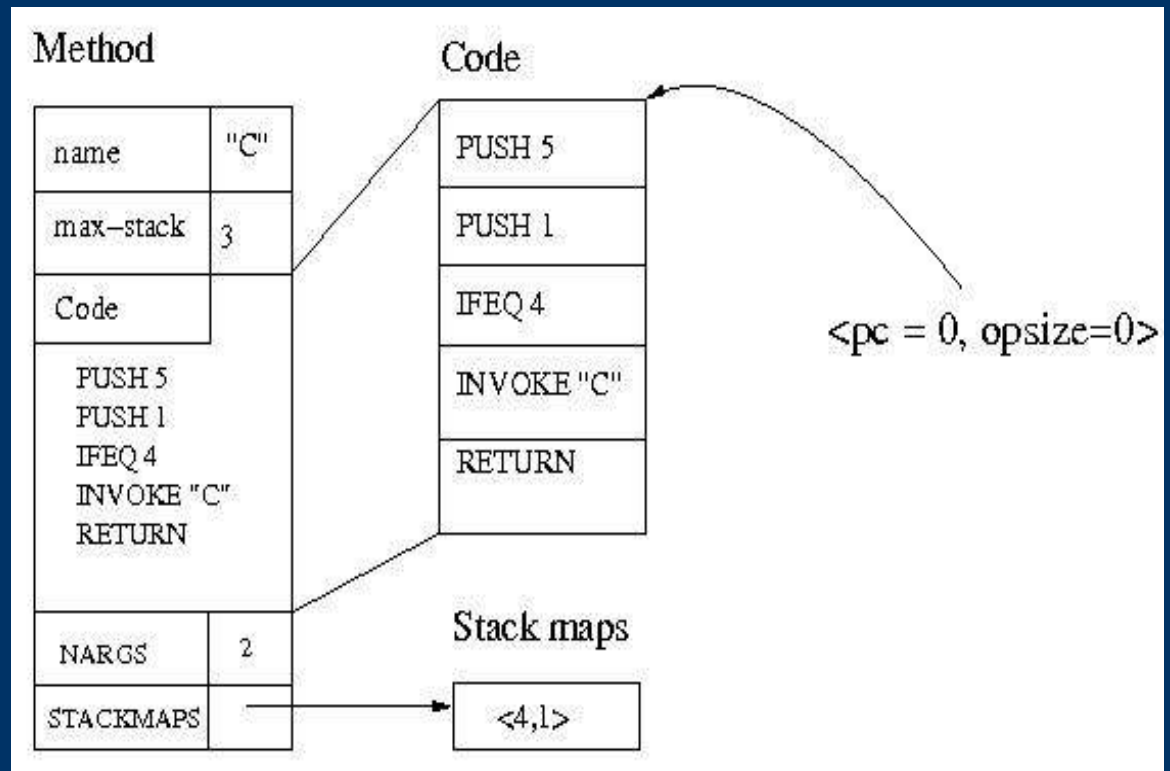
- In short,
never executes an operation that has undefined behavior
 - Specifically:
 - Don't overflow the operand stack
 - Don't underflow the operand stack
 - Don't branch to nonexistent locations
 -
 - Implications:
 - Safe programs can be easily execute efficiently
consider the implementation with current computer.
 - Non-trivial to identify the safe programs
-
-

Bytecode verifier: *identify safe/unsafe programs*

- Bytecode verifier
 - Quickly identify potential unsafe programs
 - Algorithm (basic idea)
 - Abstract the **actual operand stacks** into their **length**.
 - Abstract **(Push V)** into its effects on length of operand stack, **plus 1**
 - Similarly, for **(POP V)** into _____
 - Execute the program abstractly, check whether operand stack is overflow or not.
 - What do we do for *Branch* and *Invoke*?
-
-

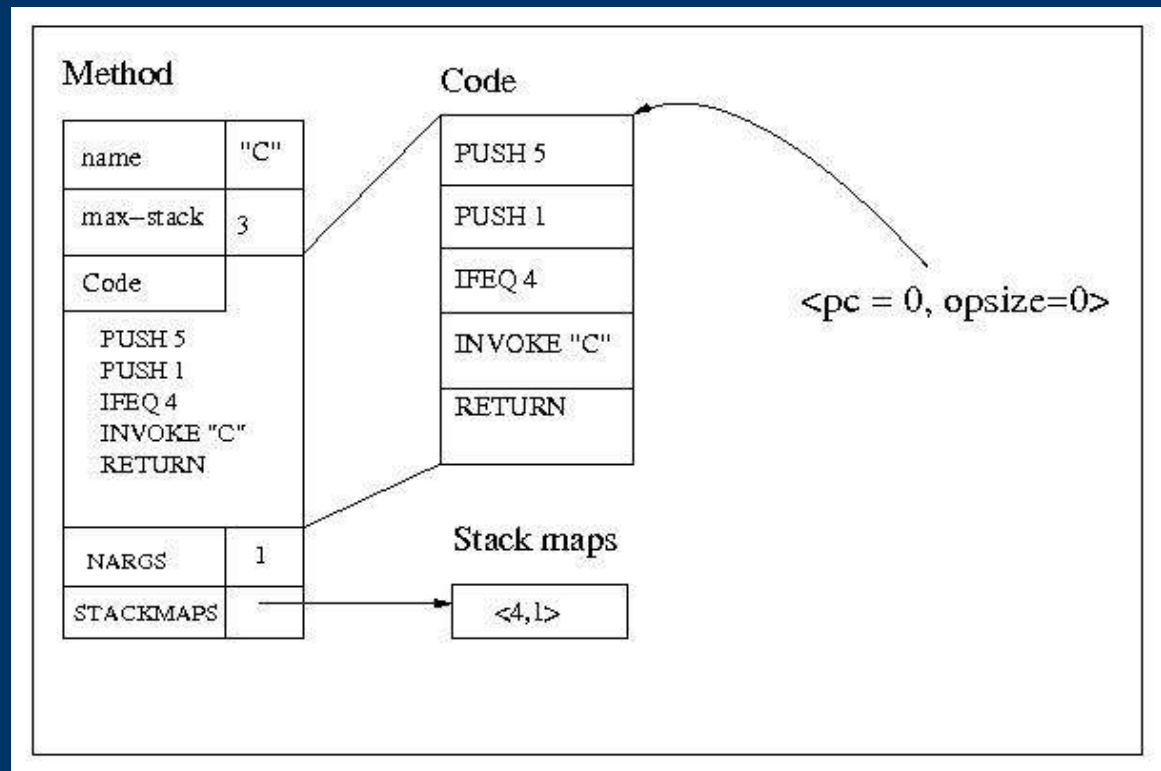
Example: verifying a method

- Execute the method symbolically, maintaining an abstract state: $\langle pc, opsize \rangle$
- Initially



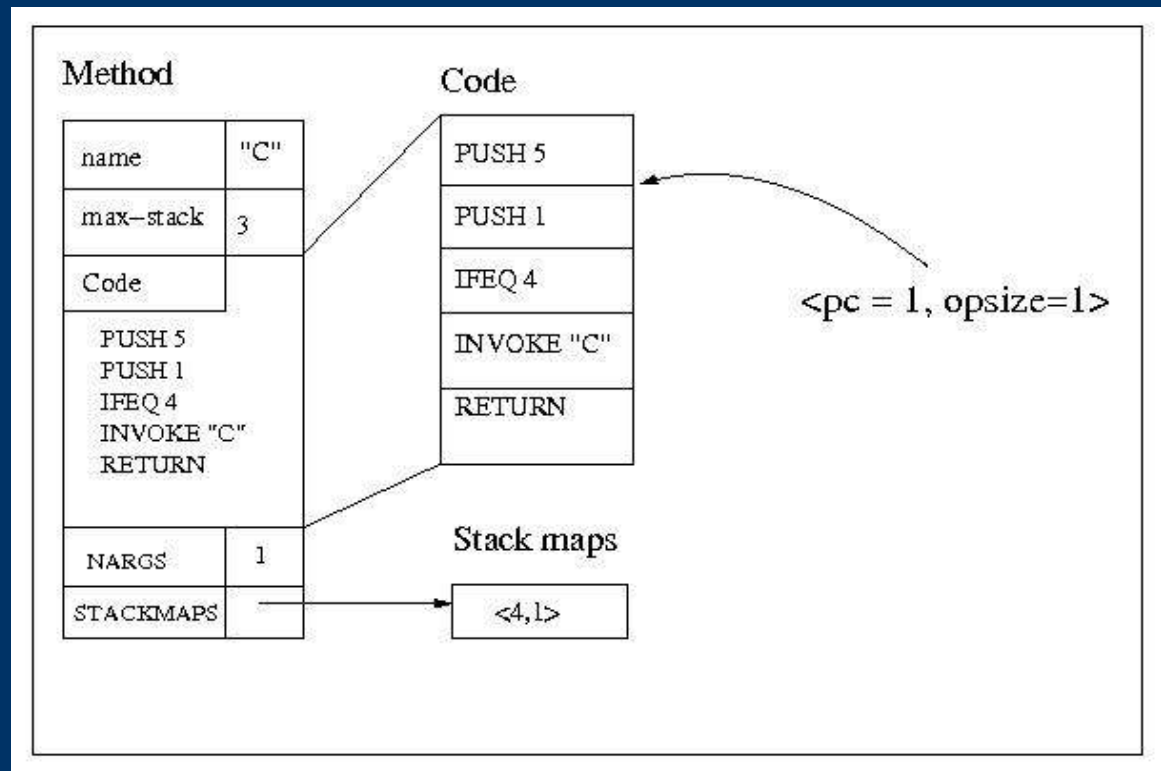
Example: verifying a method

- Check:
 - $ops\ size = 0 + 1 < 3 = \text{max-stack}$
 - pc in range



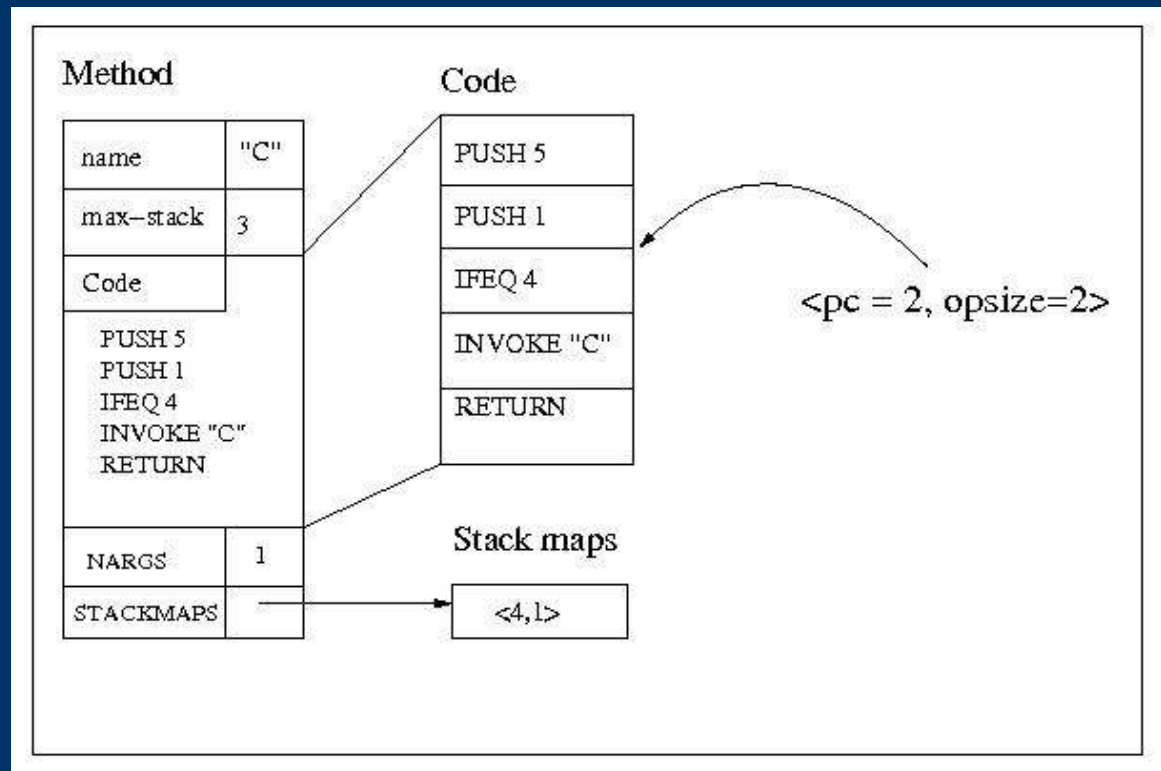
Example: verifying a method

- Check:
 - $opsize = 1 + 1 < 3 = \text{max-stack}$
 - pc in range



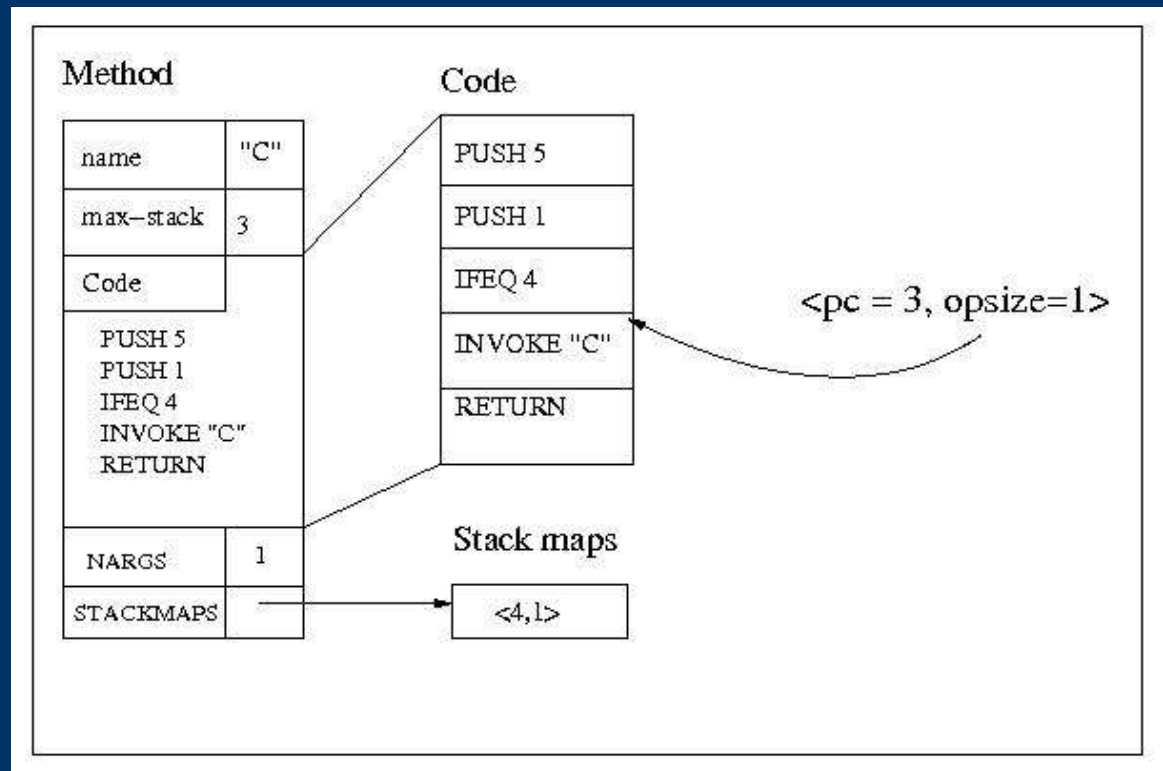
Example: verifying a method

- Check:
 - $\text{opsize} - 1 < 3 = \text{max-stack}$
 - IFEQ target
 - $\text{opsize} - 1 = ?$
 - next pc in range



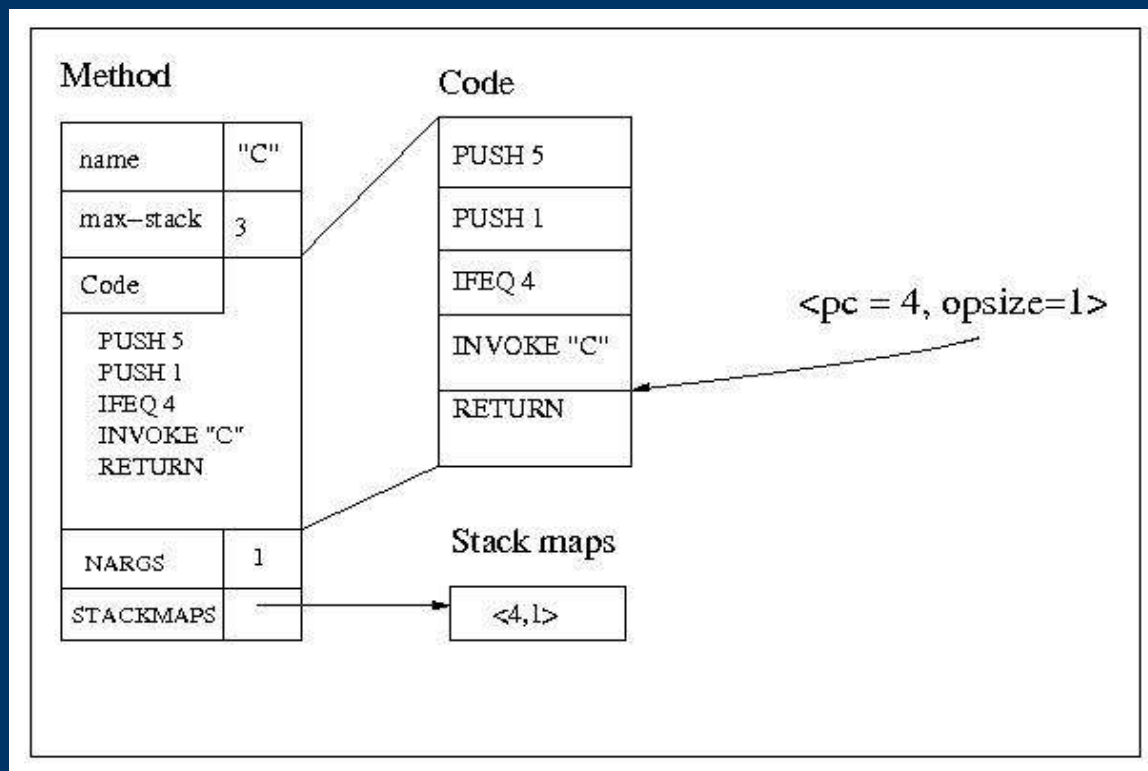
Example: verifying a method

- Check:
 - $opsize = 1 \geq 1 = nargs$
 - next pc in range
 - $opsize - 1 + 1 = ?$
 - $opsize - 1 + 1 \leq ?$



Example: verifying a method

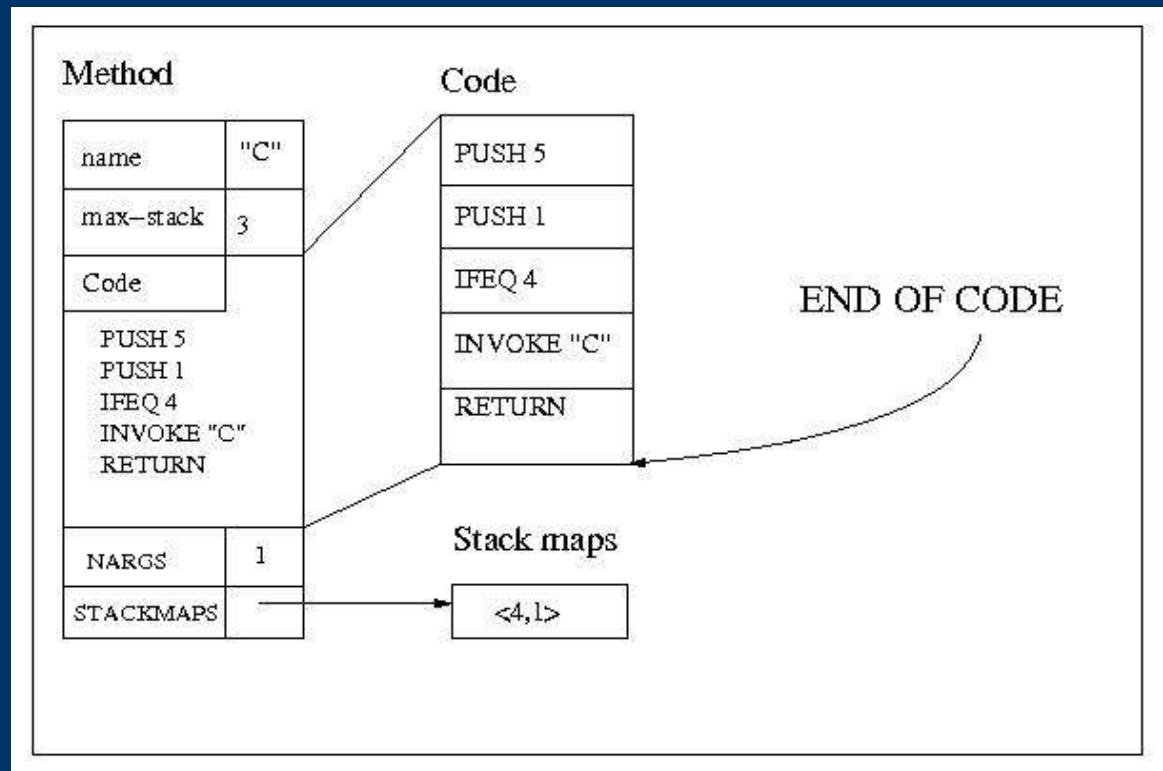
- Check:
 - opsize = 1 = stackmaps(4)
 - next pc



Example: verifying a method

- Succeed! (or are we sure?)
- Concerns:
 - Our handling of **invoke** and **branch** instruction

Note:
circular reasoning?



Small M3-like machine verified

```
(defthm verified-program-never-overflow-operand-stack-in-m
  (implies (and (consistent-state stx)
                (state-equiv st stx))
    (<= (len (g 'op-stack (topx (g 'call-stack (m-run st n))))))
      (max-stack (binding (g 'method-name
                            (topx (g 'call-stack (m-run st n))))
                        (g 'method-table st))))))
```

- <http://www.cs.utexas.edu/~hbl/dissertation/src/small/bcv-is-effective.lisp.html>

JVM bytecode verification

- Use abstract execution to identify safe programs.
Similar to our simple example with M3-like machine.
 - Challenge:
 - The control flow difference between abstract execution and concrete execution, *“circular reasoning”*
 - JVM BCV is more complicated:
 - Value of different types
 - Types have hierarchy: A is sub-type of B
 - Type hierarchy can be dynamically extended
-
-

Crack the Verifier Challenge

- The JVM bytecode verifier is described as part of JSR202 [2]
- There is a challenge for “crack the verifier” [3]

[2] <http://jcp.org/aboutJava/communityprocess/pr/jsr202/>

[3] <https://jdk.dev.java.net/CTV/challenge.html>

Using ACL2 to study the JVM

- Modeling the bytecode interpreter:
 - Precisely state how the bytecode interpreter behaves; under what conditions, effects are not defined.
- Modeling the bytecode verifier
- Prove that bytecode verified programs executes safely

See: <http://www.cs.utexas.edu/~hbl/dissertation/>

Questions: *GETFIELD*

- Resolve the symbolic reference to the field
 - Resolve the symbolic reference to the class
 - load the class if necessary
 - if the super class is not loaded, load it first
 - if the interfaces is not loaded, load them first
 - If the field is defined in the class, then return it
 - If not, search the super class chain to find where it is defined.
 - Check that the actual field is accessible to the current method.
- Check the object reference
 - If it is null, raise the exception
 - Effects undefined, if it is not null, but the object pointed to is not a compatible type.
- Access the field
 - if overflows the operand stack, effects not defined.

See: <http://www.cs.utexas.edu/~hbl/dissertation/src/DJVM/INST/>

Questions: class loading



Questions: exception handling

