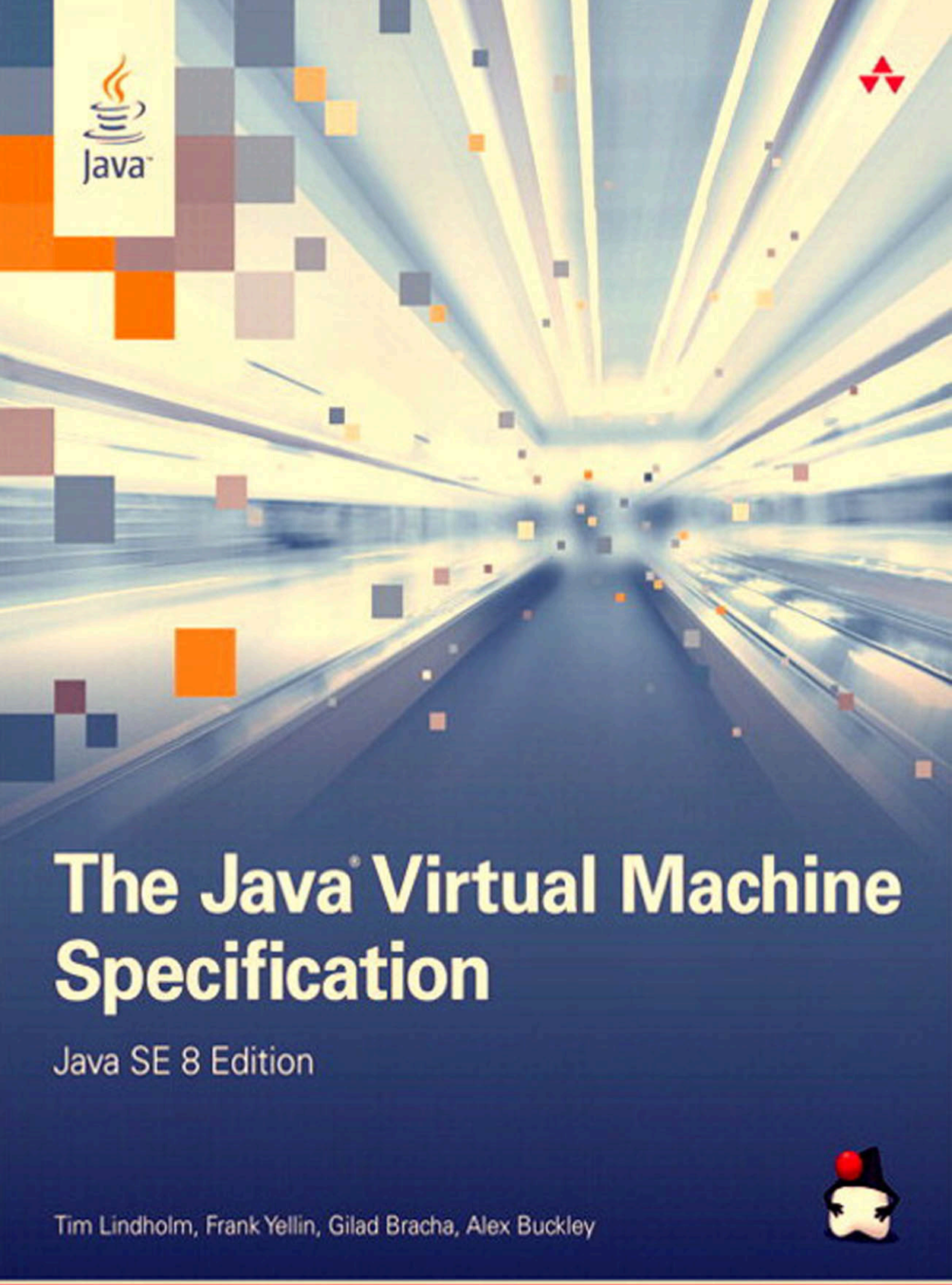

The Java® Virtual Machine Specification

Java SE 8 Edition

Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley

ORACLE

The Java® Virtual Machine Specification

Java SE 8 Edition

If an *invokespecial* instruction names an instance initialization method and the target reference on the operand stack is a class instance created by an earlier *new* instruction, then *invokespecial* must name an instance initialization method from the class of that class instance.

If an *invokespecial* instruction names a method which is not an instance initialization method, then the type of the target reference on the operand stack must be assignment compatible with the current class (JLS §5.2).

- Each instance initialization method, except for the instance initialization method derived from the constructor of class `Object`, must call either another instance initialization method of `this` or an instance initialization method of its direct superclass `super` before its instance members are *accessed*.

  However, instance fields of `this` that are declared in the current class may be *assigned* before calling any instance initialization method.

- When any instance method is invoked or when any instance variable is accessed, the class instance that contains the instance method or instance variable must already be initialized.

- There must never be an uninitialized class instance on the operand stack or in a local variable when a *jsr* or *jsr_w* instruction is executed.

- The type of every class instance that is the target of a method invocation instruction must be assignment compatible with the class or interface type specified in the instruction (JLS §5.2).

- The types of the arguments to each method invocation must be method invocation compatible with the method descriptor (JLS §5.3, §4.3.3).

- Each return instruction must match its method's return type:

  – If the method returns a `boolean`, `byte`, `char`, `short`, or `int`, only the *ireturn* instruction may be used.

  – If the method returns a `float`, `long`, or `double`, only an *freturn*, *lreturn*, or *dreturn* instruction, respectively, may be used.

  – If the method returns a `reference` type, only an *areturn* instruction may be used, and the type of the returned value must be assignment compatible with the return descriptor of the method (JLS §5.2, §4.3.3).

  – All instance initialization methods, class or interface initialization methods, and methods declared to return `void` must use only the *return* instruction.

- The type of every class instance accessed by a *getfield* instruction or modified by a *putfield* instruction must be assignment compatible with the class type specified in the instruction (JLS §5.2).
- The type of every value stored by a *putfield* or *putstatic* instruction must be compatible with the descriptor of the field (§4.3.2) of the class instance or class being stored into:
  - If the descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the value must be an `int`.
  - If the descriptor type is `float`, `long`, or `double`, then the value must be a `float`, `long`, or `double`, respectively.
  - If the descriptor type is a `reference` type, then the value must be of a type that is assignment compatible with the descriptor type (JLS §5.2).
- The type of every value stored into an array by an *aastore* instruction must be a `reference` type.

  The component type of the array being stored into by the *aastore* instruction must also be a `reference` type.
- Each *athrow* instruction must throw only values that are instances of class `Throwable` or of subclasses of `Throwable`.

  Each class mentioned in a `catch_type` item of the `exception_table` array of the method's `Code_attribute` structure must be `Throwable` or a subclass of `Throwable`.
- If *getfield* or *putfield* is used to access a `protected` field declared in a superclass that is a member of a different run-time package than the current class, then the type of the class instance being accessed must be the same as or a subclass of the current class.

  If *invokevirtual* or *invokespecial* is used to access a `protected` method declared in a superclass that is a member of a different run-time package than the current class, then the type of the class instance being accessed must be the same as or a subclass of the current class.
- Execution never falls off the bottom of the `code` array.
- No return address (a value of type `returnAddress`) may be loaded from a local variable.
- The instruction following each *jsr* or *jsr_w* instruction may be returned to only by a single *ret* instruction.

- No *jsr* or *jsr_w* instruction that is returned to may be used to recursively call a subroutine if that subroutine is already present in the subroutine call chain. (Subroutines can be nested when using `try-finally` constructs from within a `finally` clause.)
- Each instance of type `returnAddress` can be returned to at most once.

  If a *ret* instruction returns to a point in the subroutine call chain above the *ret* instruction corresponding to a given instance of type `returnAddress`, then that instance can never be used as a return address.

4.10 Verification of `class` Files

Even though a compiler for the Java programming language must only produce `class` files that satisfy all the static and structural constraints in the previous sections, the Java Virtual Machine has no guarantee that any file it is asked to load was generated by that compiler or is properly formed. Applications such as web browsers do not download source code, which they then compile; these applications download already-compiled `class` files. The browser needs to determine whether the `class` file was produced by a trustworthy compiler or by an adversary attempting to exploit the Java Virtual Machine.

> An additional problem with compile-time checking is version skew. A user may have successfully compiled a class, say `PurchaseStockOptions`, to be a subclass of `TradingClass`. But the definition of `TradingClass` might have changed since the time the class was compiled in a way that is not compatible with pre-existing binaries. Methods might have been deleted or had their return types or modifiers changed. Fields might have changed types or changed from instance variables to class variables. The access modifiers of a method or variable may have changed from `public` to `private`. For a discussion of these issues, see Chapter 13, "Binary Compatibility," in *The Java Language Specification, Java SE 8 Edition*.

Because of these potential problems, the Java Virtual Machine needs to verify for itself that the desired constraints are satisfied by the `class` files it attempts to incorporate. A Java Virtual Machine implementation verifies that each `class` file satisfies the necessary constraints at linking time (§5.4).

Link-time verification enhances the performance of the run-time interpreter. Expensive checks that would otherwise have to be performed to verify constraints at run time for each interpreted instruction can be eliminated. The Java Virtual Machine can assume that these checks have already been performed. For example, the Java Virtual Machine will already know the following:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java Virtual Machine instructions are of valid types.

There are two strategies that Java Virtual Machine implementations may use for verification:

- Verification by type checking must be used to verify `class` files whose version number is greater than or equal to 50.0.
- Verification by type inference must be supported by all Java Virtual Machine implementations, except those conforming to the Java ME CLDC and Java Card profiles, in order to verify `class` files whose version number is less than 50.0.

  Verification on Java Virtual Machine implementations supporting the Java ME CLDC and Java Card profiles is governed by their respective specifications.

In both strategies, verification is mainly concerned with enforcing the static and structural constraints from §4.9 on the `code` array of the `Code` attribute (§4.7.3). However, there are three additional checks outside the `Code` attribute which must be performed during verification:

- Ensuring that `final` classes are not subclassed.
- Ensuring that `final` methods are not overridden (§5.4.5).
- Checking that every class (except `Object`) has a direct superclass.

4.10.1 Verification by Type Checking

A `class` file whose version number is 50.0 or above (§4.1) must be verified using the type checking rules given in this section.

If, and only if, a `class` file's version number equals 50.0, then if the type checking fails, a Java Virtual Machine implementation may choose to attempt to perform verification by type inference (§4.10.2).

> This is a pragmatic adjustment, designed to ease the transition to the new verification discipline. Many tools that manipulate `class` files may alter the bytecodes of a method in a manner that requires adjustment of the method's stack map frames. If a tool does not make the necessary adjustments to the stack map frames, type checking may fail even though the bytecode is in principle valid (and would consequently verify under the old type inference scheme). To allow implementors time to adapt their tools, Java Virtual Machine implementations may fall back to the older verification discipline, but only for a limited time.

> In cases where type checking fails but type inference is invoked and succeeds, a certain performance penalty is expected. Such a penalty is unavoidable. It also should serve as a signal to tool vendors that their output needs to be adjusted, and provides vendors with additional incentive to make these adjustments.
>
> In summary, failover to verification by type inference supports both the gradual addition of stack map frames to the Java SE platform (if they are not present in a version 50.0 `class` file, failover is allowed) and the gradual removal of the *jsr* and *jsr_w* instructions from the Java SE platform (if they are present in a version 50.0 `class` file, failover is allowed).

If a Java Virtual Machine implementation ever attempts to perform verification by type inference on version 50.0 class files, it must do so in all cases where verification by type checking fails.

> This means that a Java Virtual Machine implementation cannot choose to resort to type inference in once case and not in another. It must either reject `class` files that do not verify via type checking, or else consistently failover to the type inferencing verifier whenever type checking fails.

The type checker enforces type rules that are specified by means of Prolog clauses. English language text is used to describe the type rules in an informal way, while the Prolog clauses provide a formal specification.

The type checker requires a list of stack map frames for each method with a `Code` attribute (§4.7.3). A list of stack map frames is given by the `StackMapTable` attribute (§4.7.4) of a `Code` attribute. The intent is that a stack map frame must appear at the beginning of each basic block in a method. The stack map frame specifies the verification type of each operand stack entry and of each local variable at the start of each basic block. The type checker reads the stack map frames for each method with a `Code` attribute and uses these maps to generate a proof of the type safety of the instructions in the `Code` attribute.

A class is type safe if all its methods are type safe, and it does not subclass a `final` class.

```
classIsTypeSafe(Class) :-
    classClassName(Class, Name),
    classDefiningLoader(Class, L),
    superclassChain(Name, L, Chain),
    Chain \= [],
    classSuperClassName(Class, SuperclassName),
    loadedClass(SuperclassName, L, Superclass),
    classIsNotFinal(Superclass),
    classMethods(Class, Methods),
    checklist(methodIsTypeSafe(Class), Methods).
```

```
classIsTypeSafe(Class) :-
    classClassName(Class, 'java/lang/Object'),
    classDefiningLoader(Class, L),
    isBootstrapLoader(L),
    classMethods(Class, Methods),
    checklist(methodIsTypeSafe(Class), Methods).
```

The Prolog predicate `classIsTypeSafe` assumes that `Class` is a Prolog term representing a binary class that has been successfully parsed and loaded. This specification does not mandate the precise structure of this term, but does require that certain predicates be defined upon it.

> For example, we assume a predicate `classMethods(Class, Methods)` that, given a term representing a class as described above as its first argument, binds its second argument to a list comprising all the methods of the class, represented in a convenient form described later.

Iff the predicate `classIsTypeSafe` is not true, the type checker must throw the exception `VerifyError` to indicate that the `class` file is malformed. Otherwise, the `class` file has type checked successfully and bytecode verification has completed successfully.

The rest of this section explains the process of type checking in detail:

- First, we give Prolog predicates for core Java Virtual Machine artifacts like classes and methods (§4.10.1.1).
- Second, we specify the type system known to the type checker (§4.10.1.2).
- Third, we specify the Prolog representation of instructions and stack map frames (§4.10.1.3, §4.10.1.4).
- Fourth, we specify how a method is type checked, for methods without code (§4.10.1.5) and methods with code (§4.10.1.6).
- Fifth, we discuss type checking issues common to all load and store instructions (§4.10.1.7), and also issues of access to `protected` members (§4.10.1.8).
- Finally, we specify the rules to type check each instruction (§4.10.1.9).

4.10.1.1 *Accessors for Java Virtual Machine Artifacts*

We stipulate the existence of 26 Prolog predicates ("accessors") that have certain expected behavior but whose formal definitions are not given in this specification.

`classClassName(Class, ClassName)`

Extracts the name, `ClassName`, of the class `Class`.