Proposal for Extension of Java™ Floating Point Semantics, Revision 1, May 1998

Introduction and Scope

Prompted by feedback from several partners in the industry, Sun is proposing a change to the specification of floating-point in the Java programming language.

The current Java programming language and virtual machine specifications require that all single and double precision floating-point calculations must round their results to the IEEE 754 single and double precision formats, respectively. The intent of this proposed change is to permit additional floating-point calculations to be done using IEEE 754 extended precision formats. Informally, extended precision means precision that is at least as great as that required by Java programming language types.

By making this change, processors that more naturally and efficiently support extended precision formats and floating-point operations on extended precision formats can deliver better performance for floating-point calculations. Processors that naturally and efficiently implement IEEE 754 single and double precision operations as previously mandated by the specifications may continue to do so. In particular, this proposal does not invalidate the `class` file format. Any Java virtual machine implementation that conforms to the current specification conforms to the proposed new specification.

The proposal affects both the definition of the Java programming language, in *The Java Language Specification*, by Gosling, Joy, and Steele, and the definition of the Java virtual machine, in *The Java Virtual Machine Specification*, by Lindholm and Yellin. In addition, the proposal implies changes for Java platform compatibility testing and for compilers for the Java programming language.

Summary of Changes in Revision 1 of Proposal

The present document is a first revision to a proposal originally released for licensee review on March 2, 1998. The changes incorporated in this revision are in response to feedback gathered during the review and at JavaSoft's March 23, 1998 Licensee Day.

In broad terms, the licensees responding to the original proposal argued that it did not extend Java floating-point far enough to regain expected performance on common cases like loops, method invocations, or inlined code. They also demanded continued access to bit-for-bit write-once-run-anywhere floating-point semantics.

In response to this feedback, two new keywords have been introduced to indicate whether a class or method is to be treated as *FP-wide* or *FP-strict*. For code that is FP-wide implementations are permitted, but are not required, to use extended floating-point formats as described in the IEEE 754 standard. Implementations that use extended floating-point formats are permitted to convert values freely between extended and non-extended formats, in certain defined contexts. Code that is FP-strict must conform to the floating-point semantics historically required of all Java programs. Code that is not declared to be either FP-wide or FP-strict, including legacy code, are interpreted to be implicitly FP-wide.

Code that is FP-wide and code that is FP-strict may be freely mixed. An FP-wide method can be invoked from FP-strict code and vice versa; an FP-wide method can override an FP-strict method and vice versa; and so on.

The revised proposal requires that an implementation provide an FP-strict floating-point mode. Future floating-point compatibility testing will become increasingly stringent. Existing implementations that pass current conformance tests but that do not correctly implement the specifications may fail these more stringent tests. Notes to implementors targeting Intel Architecture CPUs are provided in this revision to help ensure that implementations on that platform are conformant.

This revision permits additional use of the extended floating-point formats. Where the previous proposal only permitted extended floating-point formats to be used for intermediate values of expressions, this revision also permits values in extended formats to be stored in local variables and passed as method parameters. Extended formats cannot be used for fields or array components.

IEEE 754 Floating-point Formats

The IEEE 754 standard defines four different floating-point formats: single, double, single-extended, and double-extended. Single precision occupies a single 32-bit word, double precision two consecutive 32-bit words. An extended format offers extra precision and exponent range over the associated non-extended format. The IEEE standard only specifies a lower bound on how many extra bits extended precision provides. The details of the IEEE 754 floating-point formats are as specified in the following table:

	Format			
Parameter	Single	Single-extended	Double	Double-extended
Number of significant bits	24	≥ 32	53	≥ 64
Maximum exponent	$+127$	$\geq +1023$	$+1023$	$\geq +16383$
Minimum exponent	-126	≤ -1022	-1022	≤ -16382
Exponent width in bits	8	≥ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

Where applied to the Java programming language or the Java virtual machine in this document, IEEE 754 single format will be referred to as *float format*, and IEEE 754 single-extended format will be referred to as *float-extended format,* to emphasize the relationship between the format and the Java programming language or Java virtual machine primitive data type. IEEE 754 double and double-extended formats will be referred to using the IEEE 754 names. To avoid confusion between the data type and the format of the same name, the data type will always be written in `code font`, as it is in *The Java Language Specification* and *The Java Virtual Machine Specification*. Floating-point formats will always be written in normal font.

Proposed Changes to the Java Language Specification

Section 3.9, Keywords

Two keywords are added: `widefp` and `strictfp`. This is an incompatible change in that any Java program that uses either `widefp` or `strictfp` as an identifier will no longer be supported. Likewise, any Java program that uses these new keywords will not be supported by older compilers.

Section 4.2.3, Floating-Point Types and Values

The Java programming language requires every implementation to support two floating-point formats, called float and double, which are documented in Section 4.2.3 of *The*

Java Language Specification to be identical to IEEE 754 single and double formats, respectively.

An implementation of the Java programming language may, at its option, support two additional floating-point formats, called float-extended and double-extended. If it does, then for that implementation there are specific constants fp, femax, femin, dp, demax, and demin such that:

$\mathrm{fp} \geq 32$	$\mathrm{dp} \geq 64$
$\mathrm{femax} \geq 1023$	$\mathrm{demax} \geq 16383$
$\mathrm{femin} \leq -1022$	$\mathrm{demin} \leq -16382$

These constraints are identical to those specified by IEEE 754 for single-extended and double-extended formats.

The finite nonzero values of the float-extended format, if it is supported by an implementation, are of the form $s \cdot m \cdot 2^{e}$, where s is $+1$ or -1, m is a positive integer less than 2^{fp}, and e is an integer between $\mathrm{femin}-\mathrm{fp}+1$ and $\mathrm{femax}-\mathrm{fp}+1$, inclusive. Values of that form such that m is positive but less than 2^{fp} and e is equal to $\mathrm{femin}-\mathrm{fp}+1$ are said to be denormalized.

The finite nonzero values of the double-extended format, if it is supported by an implementation, are of the form $s \cdot m \cdot 2^{e}$, where s is $+1$ or -1, m is a positive integer less than 2^{dp}, and e is an integer between $\mathrm{demin}-\mathrm{dp}+1$ and $\mathrm{demax}-\mathrm{dp}+1$, inclusive. Values of that form such that m is positive but less than 2^{dp} and e is equal to $\mathrm{demin}-\mathrm{dp}+1$ are said to be denormalized.

Note that the constraints permit the float-extended format to be identical to the double format or to the double-extended format.

Note that float, float-extended, double, and double-extended are “formats” and not “types”. The float-extended format may be used instead of float format, and the double-extended format may be used instead of double format, according to rules described here and in Chapters 5, 14, and 15.

Every expression, parameter declaration, and local variable declaration is either *explicitly FP-wide*, *explicitly FP-strict*, or *implicitly FP-wide*.

Consider an expression, parameter declaration, or local variable declaration *E*; then consider the set *D* of all class declarations, interface declarations, and method declarations that contain *E*.

If no construct in the set *D* bears either the `widefp` modifier or the `strictfp` modifier, then *E* is implicitly FP-wide.

Otherwise, there must be one particular declaration *d* in *D* that bears either the `widefp` modifier or the `strictfp` modifier and is contained in every other declaration in *D* that bears either the `widefp` modifier or the `strictfp` modifier. If *d* bears the `widefp` modifier, then *E* is explicitly FP-wide; if *d* bears the `strictfp` modifier, then *E* is explicitly FP-strict.

An expression, parameter declaration, or local variable declaration that is explicitly FP-wide will be treated as FP-wide.

An expression, parameter declaration, or local variable declaration that is explicitly FP-strict will be treated as FP-strict.

An expression, parameter declaration, or local variable declaration that is implicitly FP-wide is normally treated as FP-wide, but in some programming or execution environments, it may be that a compiler command switch or a separate tool may be used to direct that every implicitly FP-wide expression, parameter declaration, and local variable declaration in a particular class, interface, or compilation unit be treated as FP-strict.

Less formally but more intuitively, we will refer to classes, methods, or bodies of code as being treated as FP-wide when all of the expressions, parameter declarations, or local variable declarations of the class, method or code are treated as FP-wide. Similarly, classes, methods, or code may be said to be treated as FP-strict.

Section 5.1.8, Format Conversion

A new kind of conversion is introduced that is not a type conversion, but a conversion between representations used for the same type.

Within an FP-wide expression, *format conversion* allows an implementation, at its option, to perform any of the following operations on a value:

- If the value is represented in float format, then the implementation may, at its option, convert the value from float format to float-extended format. (Note that converting from float format to float-extended format does not alter the mathematical value represented. In particular, NaNs remain NaNs and infinities remain infinities.)
- If the value is represented in float-extended format, then the implementation may, at its option, convert the value from float-extended format to float format (rounding, if necessary, to the nearest representable value in float format). Alternatively, the implementation may, at its option, keep the value in float-extended format but round the value to the nearest representable value in float format (this may also be regarded as converting the value to float format with rounding and then converting back to float-extended format).
- If the value is represented in double format, then the implementation may, at its option, convert the value from double format to double-extended format. (Note that

converting from double format to double-extended format does not alter the mathematical value represented. In particular, NaNs remain NaNs and infinities remain infinities.)

- If the value is represented in double-extended format, then the implementation may, at its option, convert the value from double-extended format to double format (rounding, if necessary, to the nearest representable value in double format). Alternatively, the implementation may, at its option, keep the value in double-extended format but round the value to the nearest representable value in double format (this may also be regarded as converting the value to double format with rounding and then converting back to double-extended format).

Within an FP-strict expression, format conversion always converts a value that is represented in float-extended format to float format (rounding, if necessary, to the nearest representable value in float format) and always converts a value that is represented in double-extended format to double format (rounding, if necessary, to the nearest representable value in double format). Such conversion is necessary only when the value of a local variable or method parameter is accessed, the declaration of that local variable or method parameter is FP-wide, and the implementation has chosen to represent the local variable or method parameter in an extended format; or when a method is invoked whose declaration is FP-wide and the implementation has chosen to represent the result of the method invocation in an extended format.

Format conversion leaves unchanged any value whose type is neither `float` nor `double`.

Section 5.2, Assignment Conversion

If a variable is represented in float-extended format, assignment conversion for that variable always automatically converts a value to be assigned that is represented in float format to float-extended format.

If a variable is represented in float format, assignment conversion for that variable always automatically converts a value to be assigned that is represented in float-extended format to float format (rounding, if necessary, to the nearest representable value in float format).

If a variable is represented in double-extended format, assignment conversion for that variable always automatically converts a value to be assigned that is represented in double format to double-extended format.

If a variable is represented in double format, assignment conversion for that variable always automatically converts a value to be assigned that is represented in double-extended format to double format (rounding, if necessary, to the nearest representable value in double format).

Section 5.3, Method Invocation Conversion

If a formal method parameter is represented in float-extended format, method invocation conversion for that method parameter always automatically converts a value to be passed as a parameter that is represented in float format to float-extended format.

If a formal method parameter is represented in float format, method invocation conversion for that method parameter always automatically converts a value to be passed as a parameter that is represented in float-extended format to float format (rounding, if necessary, to the nearest representable value in float format).

If a formal method parameter is represented in double-extended format, method invocation conversion for that method parameter always automatically converts a value to be passed as a parameter that is represented in double format to double-extended format.

If a formal method parameter is represented in double format, method invocation conversion for that method parameter always automatically converts a value to be passed as a parameter that is represented in double-extended format to double format (rounding, if necessary, to the nearest representable value in double format).

Section 5.6.1, Unary Numeric Promotion

Unary numeric promotion performs format conversion (section 5.1.8) on the operand.

Section 5.6.2, Binary Numeric Promotion

After performing binary numeric promotion where the two operands are converted to type `float` or to type `double` within an FP-wide expression, format conversion is applied separately to each operand, but subject to the constraint that the implementation must make its choices in such a way that the two operands, after format conversion, are represented in the same format.

Section 8.1.2, Class Modifiers

A *ClassModifier* may be either `widefp` or `strictfp`. A compile-time error occurs if both `widefp` and `strictfp` appear as class modifiers in the same class declaration.

Section 8.3, Field Declarations

A field of type `float` is always represented in float format. A field of type `double` is always represented in double format. It is not permitted to represent a field in float-extended format or double-extended format.

Section 8.4.1, Formal Parameters

A method parameter of type `float` is always represented in float format if its declaration is FP-strict. A method parameter of type `float` may be represented in float format or in float-extended format, at the option of the implementation, if its declaration is FP-wide.

A method parameter of type `double` is always represented in double format if its declaration is FP-strict. A method parameter of type `double` may be represented in double format or in double-extended format, at the option of the implementation, if its declaration is FP-wide.

Section 8.4.3, Method Modifiers

A *MethodModifier* may be either `widefp` or `strictfp`. A compile-time error occurs if both `widefp` and `strictfp` appear as method modifiers in the same method declaration.

Section 8.3.3, Constructor Modifiers

A *ConstructorModifier* may *not* be `widefp` or `strictfp`. A compile-time error occurs if either `widefp` or `strictfp` appears as a constructor modifier. This difference between a *ConstructorModifier* and a *MethodModifier* is intentional.

Section 8.4.6.1, Overriding (By Instance Methods)

The presence or absence of `widefp` and `strictfp` modifiers has no effect whatsoever on the rules for overriding methods and implementing abstract methods.

Section 9.1.2, Interface Modifiers

An *InterfaceModifier* may be either `widefp` or `strictfp`. A compile-time error occurs if both `widefp` and `strictfp` appear as interface modifiers in the same interface declaration.

Section 10.1, Array Types

An array component of type `float` is always represented in float format. An array component of type `double` is always represented in double format. It is not permitted to represent an array component in float-extended format or double-extended format.

Section 14.3, Local Variable Declaration Statements

A local variable of type `float` is always represented in float format if its declaration is FP-strict. A local variable of type `float` may be represented in float format or in float-extended format, at the option of the implementation, if its declaration is FP-wide.

A local variable of type `double` is always represented in double format if its declaration is FP-strict. A local variable of type `double` may be represented in double format or in double-extended format, at the option of the implementation, if its declaration is FP-wide.

Section 15.1, Evaluation, Denotation, and Result

Format conversion (section 5.1.8) is applied to the result of *every* expression that produces a value.

Section 15.2, Variables as Values

If an expression denotes a variable, and a value is required for use in further evaluation, then the result of applying format conversion to the value of that variable is used.

Section 15.6, Evaluation Order

The rules for evaluation order are not changed.

Section 15.7.1, Literals

The value of a literal of type `float` is always represented in float format. The value of a literal of type `double` is always represented in double format.

Section 15.7.3, Parenthesized Expressions

Parentheses do not affect in any way the choice of format for the value of an expression.

Section 15.10, Field Access Expressions

The format conversion rule of section 15.2 applies to field access expressions.

Section 15.11, Method Invocation Expressions

Method invocation conversion (section 5.3) addresses format conversions for the values of argument expressions.

The fact that format conversion is applied to the result of every expression that returns a value (section 15.1) implies that when an FP-wide method is invoked from FP-strict code, if the value returned by the method is in an extended format, then the value must be converted to the corresponding non-extended format.

Section 15.12, Array Access Expressions

The format conversion rule of section 15.2 applies to array access expressions.

Sections 15.13.2, 15.13.3, 15.14.1, 15.14.2, Prefix and Postfix Operators

FP-wide prefix and postfix increment and decrement expressions behave with respect to format decisions exactly as if the expression had been written as `x=x+1` or `x=x-1` plus a variable access.

Section 15.14.4, Unary Minus Operator `-`

If, after unary numeric promotion, the operand is represented in float-extended format or double-extended format, then the unary negation operation is carried out in that format and the result is represented in that format. That result is then subject to further format conversion, by the general rule of section 15.1.

Section 15.15, Cast Expressions

A cast may convert a value of one numeric type to a similar value of a floating-point type, but it needs not have an effect on the choice of format for the result of the cast expression. Consequently, an FP-wide cast to type `float` does not necessarily cause its value to be rounded to the nearest representable value in float format, and an FP-wide cast to type `double` does not necessarily cause its value to be rounded to the nearest representable value in double format.

Section 15.16, Multiplicative Operators

If, after binary numeric promotion, the operands are represented in float-extended format or double-extended format, then the multiplication, division, or remainder operation is carried out in that format and the result is represented in that format. That result is then subject to further format conversion, by the general rule of section 15.1.

Section 15.17.2, Additive Operators (+ and -) for Numeric Types

If, after binary numeric promotion, the operands are represented in float-extended format or double-extended format, then the addition or subtraction operation is carried out in that format and the result is represented in that format. That result is then subject to further format conversion, by the general rule of section 15.1.

Section 15.19, Relational Operators

If, after binary numeric promotion, the operands are represented in float-extended format or double-extended format, then the comparison is carried out in that format.

Section 15.20.1, Numerical Equality Operators == and !=

If, after binary numeric promotion, the operands are represented in float-extended format or double-extended format, then the equality test is carried out in that format.

Section 15.24, Conditional Operator ? :

If, after binary numeric promotion, the second and third operands are represented in float-extended format or double-extended format, then the result is represented in that format. That result is then subject to further format conversion, by the general rule of section 15.1.

Sections 15.25.1, 15.25.2 Simple and Compound Assignment Operators

Assignment conversion (section 5.2) addresses format conversions for values that are to be assigned to variables.

Section 15.27, Constant Expression

A compile-time constant expression is always treated as though FP-strict.

Section 20.9, The Class `java.lang.Float`

```
public static final int WIDEFP_MAX_EXPONENT;
```

The constant value of this field is the largest exponent that can be represented in float-extended format. If the implementation does not use float-extended format, the constant value is +127. Otherwise, the constant value is femax as defined by Section 4.2.3.

```
public static final int WIDEFP_MIN_EXPONENT;
```

The constant value of this field is the smallest exponent that can be represented in float-extended format. If the implementation does not use float-extended format, the constant value is -126. Otherwise, the constant value is femin as defined by Section 4.2.3.

```
public static final int WIDEFP_SIGNIFICAND_BITS;
```

The constant value of this field is the number of bits of significand that can be represented in float-extended format. If the implementation does not use float-extended format, the constant value is 24. Otherwise, the constant value is fp, where fp is defined by Section 4.2.3.

Section 20.10, The Class `java.lang.Double`

```
public static final int WIDEFP_MAX_EXPONENT;
```

The constant value of this field is the largest exponent that can be represented in double-extended format. If the implementation does not use double-extended format, the constant value is 1023. Otherwise, the constant value is demax as defined by Section 4.2.3.

```
public static final int WIDEFP_MIN_EXPONENT;
```

The constant value of this field is the smallest exponent that can be represented in double-extended format. If the implementation does not use double-extended format, the constant value is -1022. Otherwise, the constant value is demin as defined by Section 4.2.3.

```
public static final int WIDEFP_SIGNIFICAND_BITS;
```

The constant value of this field is the number of bits of exponent that can be represented in double-extended format. If the implementation does not use float-extended format, the constant value is 53. Otherwise, the constant value is dp, where dp is defined by Section 4.2.3.

Proposed Changes to the Java Virtual Machine Specification

Chapter 2, Java Concepts

Changes must be made to this chapter tracking the changes to be made in *The Java Language Specification*, sketched above.

Section 3.2.2, Floating-Point Types and Values

The Java Virtual Machine Specification requires that every implementation support two floating-point formats, called float and double, which are documented in Section 3.2.2 to be identical to IEEE 754 single and double formats, respectively.

An implementation of the Java virtual machine may, at its option, support two additional floating-point formats, called float-extended and double-extended. If it does, the definition of these additional formats is identical to that given in Section 4.2.3 of Proposed Changes to the Java Language Specification, above.

Note that float, float-extended, double, and double-extended are “formats” and not Java virtual machine “types”. The float-extended format may be used instead of float format, and the double-extended format may be used instead of double format, according to rules described in Chapters 3 and 6 of *The Java Virtual Machine Specification*.

Every Java virtual machine instruction, local variable, or operand stack is either explicitly FP-wide, explicitly FP-strict, implicitly FP-wide, or implicitly FP-strict depending on the settings of the `ACC_STRICT` and `ACC_EXPLICIT` bits of the `access_flags` word of the `method_info` structure containing the instruction or defining the method that allocates the local variable or operand stack (see Section 4.6, Methods).

The encoding of floating-point modes in flag bit settings is as given in the following table:

Floating-point Mode	`ACC_STRICT` Flag	`ACC_EXPLICIT` Flag
explicitly FP-wide	unset	set
explicitly FP-strict	set	set
implicitly FP-wide	unset	unset
implicitly FP-strict	set	unset

Note that this mapping implies that Java virtual machine instructions in, or local variables or operand stacks allocated by, code compiled by a Java compiler that predates the proposed changes are implicitly FP-wide.

No means is provided to declare a method implicitly FP-strict in source code. That floating-point mode can only be produced by a tool such as a `class` file postprocessor or a user-defined class loader.

A Java virtual machine instruction, local variable, or operand stack that is either explicitly FP-wide or implicitly FP-wide will be treated as FP-wide.

A Java virtual machine instruction, local variable, or operand stack that is either explicitly FP-strict or implicitly FP-strict will be treated as FP-strict.

Less formally but more intuitively, we will refer to classes, methods, or bodies of code as being treated as FP-wide when all of the Java virtual machine instructions contained in, and local variables and operand stacks allocated by, the classes, methods, or bodies of code are treated as FP-wide. Similarly, classes, methods, or bodies of code may be said to be treated as FP-strict.

Section 3.4, Words

This section will be deleted. The current definitions of the Java virtual machine's local variables and operand stacks are in terms of an abstract word with an implementation-defined size. Such definitions are inconvenient if floating-point values using float-extended or double-extended formats may be stored in local variables or on an operand stack.

The proposed new specification defines local variables and operand stacks more abstractly than in the original specification, in terms of values rather than words.

Section 3.6.1, Local Variables

On each method invocation, the Java virtual machine allocates a Java frame which contains an array known as its local variables. A local variable can hold a value of any Java virtual machine data type, including a value of type `long` or of type `double`. Values of type `float` may be stored in a local variable in float format or, if the local variable is FP-wide, float-extended format. Values of type `double` may be stored in a local variable in double format or, if the local variable is FP-wide, double-extended format.

Individual local variables are addressed by indexing into the local variables array. Two consecutive local variable indices are reserved for each value of type `long` or `double`. Such a local variable is only addressed using the lesser index for the value.

For example, a local variable of type `double` reserves both indices *n* and *n*+1, although it can only be addressed using index *n*. The Java virtual machine specification does not require *n* to be even and does not require alignment of local variables.

Section 3.6.2, Operand Stacks

On each method invocation the Java virtual machine allocates a Java frame which contains an operand stack. Most Java virtual machine instructions take values from the operand stack of the current frame, operate on them, and return results to that same operand stack. The operand stack is also used to pass arguments to methods and to receive method results.

Each entry on the operand stack can hold a value of any Java virtual machine data type, including a value of type `long` or of type `double`. Values of type `float` may be represented in float or, if the operand stack is FP-wide, float-extended format. Values of type `double` may be represented in double or, if the operand stack is FP-wide, double-extended format. The Java virtual machine specification does not require alignment of values on the operand stack.

Values from the operand stack must be operated upon in ways appropriate to their types. It is incorrect, for example, to push two values of type `int` and then treat them as a value of type `long`, or to push two values of type `float` and then add them with an *iadd* instruction. A small number of Java virtual machine instructions (the *dup* instructions and *swap*) operate on runtime data areas without regard to the specific types of values; these instructions must not be used to break up or rearrange the words of values. These restrictions on operand stack manipulation are enforced through `class` file verification.

Section 4.3.3, Method Descriptors

A method descriptor is valid only if it represents method parameters with a total length of 255 or less, where that length includes the contribution for `this` in the case of instance method invocations. The total length is calculated by summing the lengths of the individual parameters, and the length of `this` if appropriate, where an item of type `long` or `double` contributes two units to the length and a value of any other type contributes one unit.

Section 4.4.4, `CONSTANT_Integer` and `CONSTANT_Float`

Floating-point values stored in `CONSTANT_Float_info` structures may only be represented using float format; values may not be represented using float-extended format.

Section 4.4.5, `CONSTANT_Long` and `CONSTANT_Double`

Floating-point values stored in `CONSTANT_Double_info` structures may only be represented using double format; values may not be represented using double-extended format.

Section 4.6, Methods

Table 4.4, showing the `access_flags` modifiers of the `method_info` structure, defines two additional modifier bits:

Flag Name	Value	Meaning	Used By
`ACC_STRICT`	`0x0800`	Floating-point mode is FP-strict if set, is FP-wide if unset.	Any method
`ACC_EXPLICIT`	`0x1000`	Floating-point mode was specified in the source code.	Any method

Section 4.7.4, Code Attribute

The value of the `max_stack` item gives the maximum depth of the operand stack at any point during execution of this method, where a value of type `long` or `double` contributes two units to the depth and a value of any other type contributes one unit.

The value of the `max_locals` item gives the number of local variable array indices reserved for local variables used by this method, including the parameters passed in the local variable array to the method on invocation. The index of the first local variable is `0`. The greatest local variable index for a value of type `long` or double is `max_locals—2`; the greatest local variable index for a value of any other Java virtual machine type is `max_locals—1`.

Note that the proposed definition of the `max_stack` item may not be immediately useful to implementations which do not implement a fixed-width operand stack, for instance when determining where to push a stack frame. Naive implementations that cannot use `max_stack` directly can choose a conservative maximum stack size based on `max_stack`, or alternatively could calculate a more precise stack size using information derived during processing such as verification.

Similarly, the proposed definition of the `max_locals` item may not be immediately useful to implementations which do not implement local variables as a fixed-width array, for instance when determining the size of a Java frame.

Section 4.7.7, `LocalVariableTable` Attribute

The sentence "If the local variable at `index` is a two-word type (`double` or `long`), it occupies both `index` and `index+1`" will be deleted. It is currently found in the description of the `index` item of an entry in the `local_variable_table` array of the `LocalVariableTable` attribute.

Section 4.8.1, Static Constraints

The four constraints on the index operand of the Java virtual machine local variable *load* instructions still stand in the new proposal.

Section 4.8.2, Structural Constraints

The structural constraints on the code array rely on the abstract types of values on the operand stack, not on the format in which those values are literally stored. Constraints on a value of type `float` will hold whether the value is represented in float format or float-extended format. Constraints on a value of type `double` will hold whether the value is represented in double format or double-extended format.

The following four structural constraints, currently given in Section 4.8.2 of *The Java Virtual Machine Specification*, are affected by the change to a value-oriented operand stack:

- At no point during execution can the order of the words of a two-word type (`long` or `double`) be reversed or split up. At no point can the words of a two-word type be operated on individually.
- No local variable (or local variable pair, in the case of a two-word type) can be accessed before it is assigned a value.
- At no point during execution can the operand stack grow to contain more than `max_stack` words.
- At no point during execution can more words be popped from the operand stack than it contains.

The first of these constraints can be deleted. The new definition of the operand stack only permits values on the operand stack to be loaded and stored in their entirety. The second constraint can have the phrase “or local variable pair, in the case of a two-word type” deleted. The new definition of a local variable does not permit portions of a local variable to be addressed.

The third constraint can be restated as follows:

- At no point during execution can the operand stack grow to a depth greater than that implied by the `max_stack` item, where a value of type `long` or `double` contributes two units to the depth and a value of any other type contributes one unit.

The fourth constraint can be trivially restated:

- At no point during execution can more values be popped from the operand stack than it contains.

Finally, the following constraint on the operand stack can be retained unchanged where the notion of operand stack size is understood to reflect the number of values on the operand stack rather than the number of words:

- Where an instruction can be executed along several different execution paths, the operand stack must have the same size prior to the execution of the instruction, regardless of the path taken

Section 4.9.3, Long Integers and Doubles

Values of types `long` and `double` are treated specially by the verification process.

Whenever a value of type `long` or `double` is moved into a local variable addressed using index *n*, the index *n*+1 is specially marked to indicate that all references to the value of type `long` or `double` must be through the previous index *n*.

Whenever a value is moved to a local variable *n*, the preceding local variable *n*—1 is examined to see if it is the index of a value of type `long` or `double`. If so, that preceding local variable index is changed to indicate that it now contains an unusable value.

Dealing with values of type `long` or `double` on the operand stack is simpler; the verifier treats them as single units on the stack. For example, the verification code for the *dadd* instruction (add two values of type `double`) checks that the top two values on the operand stack are both of type `double`. When calculating operand stack length, values of types `long` and `double` each represent a single value.

Section 4.10, Limitations of the Java Virtual Machine and `class` File Format

The greatest index into the local variables array in a method is limited to 65534 by the size of the `max_locals` item of the `ClassFile` structure. Recall that values of types `long` and `double` are considered to reserve two local variable indices.

Section 5.1, The Runtime Constant Pool (in JVMS First Revision only)

Runtime constant values derived from `CONSTANT_Float_info` and `CONSTANT_Double_info` structures in the binary representation of a class or interface are always represented in float and double format, respectively, and cannot be represented in an extended format.

Chapter 6, Java Virtual Machine Instruction Set

The definition of each Java virtual machine instruction includes one or more “stack diagrams” that indicate the effect of the instruction on the operand stack. These diagrams are currently written in terms of abstract words, where each value on the operand stack consists of one or two words. The change from a word-oriented to a value-oriented operand stack requires that this word orientation be removed from the stack diagrams and associated text, including Section 6.4 where stack diagrams are first introduced and in descriptions of some instructions that do not operate on values of floating-point types.

Note that many Java virtual machine instructions are permitted to behave differently depending on whether they are treated as FP-wide or FP-strict.

Format Conversion

The format conversions introduced for the Java programming language in section 5.1.8 are reflected in the definition of the Java virtual machine. A format conversion is not a type conversion, but a conversion between representations used for the same type.

During the execution of an FP-wide Java virtual machine instruction, format conversion allows an implementation, at its option, to perform any of the following operations on a value:

- If the value is represented in float format, then the implementation may, at its option, convert the value from float format to float-extended format. (Note that converting from float format to float-extended format does not alter the mathematical value represented. In particular, NaNs remain NaNs and infinities remain infinities.)

- If the value is represented in float-extended format, then the implementation may, at its option, convert the value from float-extended format to float format (rounding, if necessary, to the nearest representable value in float format). Alternatively, the implementation may, at its option, keep the value in float-extended format but round the value to the nearest representable value in float format (this may also be regarded as converting the value to float format with rounding and then converting back to float-extended format).
- If the value is represented in double format, then the implementation may, at its option, convert the value from double format to double-extended format. (Note that converting from double format to double-extended format does not alter the mathematical value represented. In particular, NaNs remain NaNs and infinities remain infinities.)
- If the value is represented in double-extended format, then the implementation may, at its option, convert the value from double-extended format to double format (rounding, if necessary, to the nearest representable value in double format). Alternatively, the implementation may, at its option, keep the value in double-extended format but round the value to the nearest representable value in double format (this may also be regarded as converting the value to double format with rounding and then converting back to double-extended format).

In addition, upon the execution of an FP-wide Java virtual machine instruction that implements a store into an FP-strict local variable or a push onto an FP-strict operand stack, format conversion always converts a value that is represented in float-extended format to float format (rounding, if necessary, to the nearest representable value in float format) and always converts a value that is represented in double-extended format to double format (rounding, if necessary, to the nearest representable value in double format). Such conversion is necessary only when the value is stored to a local variable that is FP-strict, and the implementation has chosen to represent the value in an extended format; or when an FP-strict method has invoked an FP-wide method and the implementation has chosen to represent the result of the method invocation, to be pushed onto the FP-strict operand stack of the invoker, in an extended format.

Upon the execution of an FP-strict Java virtual machine instruction, format conversion always converts an operand value that is represented in float-extended format to float format (rounding, if necessary, to the nearest representable value in float format) and always converts a value that is represented in double-extended format to double format (rounding, if necessary, to the nearest representable value in double format). Such conversion is necessary only when the value of a local variable is accessed, the local variable is FP-wide, and the implementation has chosen to represent the local variable in an extended format.

Note that the Java virtual machine's definition of format conversion is not identical to that for the Java programming language: the Java virtual machine's definition does not provide a catchall case for types other than `float` and `double`. While the catchall case is a useful convenience when describing the Java programming language, it would obscure the presentation of the Java virtual machine by increasing the number of Java virtual machine instructions subject to format conversion.

Load and Store Instructions

Execution of a Java virtual machine instruction implementing a load of a value of type `float` or `double` from a local variable onto the operand stack performs format conversion on the operand value before the value is loaded. The affected instructions are *fload*, *fload_<n>*, *dload*, and *dload_<n>*.

Execution of a Java virtual machine instruction implementing a load of a constant value of type `float` or `double` onto the operand stack performs format conversion on the constant value before the value is loaded. The affected instructions are *fconst_<f>*, *dconst_<d>*, and any *ldc*, *ldc_w*, or *ldc2_w* instruction with an operand representing a `float` or `double` constant.

Execution of a Java virtual machine instruction implementing a store of an operand of type `float` or `double` from the operand stack into a local variable performs format conversion on the operand value before the value is stored. The affected instructions are *fstore*, *fstore_<n>*, *dstore* and *dstore_<n>*.

Arithmetic Instructions

Execution of a Java virtual machine instruction implementing a unary arithmetic operation on an operand of type `float` or `double` first performs format conversion on its operand value. The affected instructions are *fneg* and *dneg*.

Execution of a Java virtual machine instruction implementing a binary arithmetic operation on two values of type `float` or on two values of type `double` first performs format conversion separately on its operand values, but subject to the additional constraint that the implementation must make its choices in such a way that the two operands, after format conversion, are represented in the same format. The affected instructions are *fadd*, *fsub*, *fmul*, *fdiv*, *frem*, *dadd*, *dsub*, *dmul*, *ddiv*, and *drem*.

Execution of a Java virtual machine instruction implementing an operation on one or more values of a floating-point type, having performed format conversion on the operand as permitted by the above rules, operates on the operand or operands exactly as specified by IEEE 754 for that operation and operand format. That is, the result must be computed exactly and then rounded to the nearest value of the operand format using round to even.

Execution of an Java virtual machine arithmetic instruction that produces a result which is a value of type `float` or `double` performs format conversion on its result value prior to pushing it onto the operand stack. The affected instructions are *fneg, fadd, fsub, fmul, fdiv, frem, dneg, dadd, dsub, dmul, ddiv,* and *drem*.

Type Conversion Instructions

Execution of a Java virtual machine type conversion instruction that operates on an operand of type `float` or `double` performs format conversion on the operand value prior to performing the type conversion. The affected instructions are *f2i, f2l, f2d, d2i, d2l,* and *d2f.*

Execution of a Java virtual machine type conversion instruction taking an operand of a floating-point type or producing a result of a floating-point type must, after any permitted format conversion on the operand value, convert between types exactly as specified by IEEE 754 for that conversion and operand format.

Execution of a Java virtual machine type conversion instruction that produces a result value of type `float` or `double` performs format conversion on the result value prior to pushing it onto the operand stack. The affected instructions are *i2f, l2f, d2f, i2d, l2d,* and *f2d*.

Object and Array Manipulation Instructions

Execution of a Java virtual machine instruction implementing a load of a value of type `float` or `double` from an array component, instance variable, or class (`static`) variable onto the operand stack performs format conversion on the value before it is loaded. The affected instructions are *faload, daload,* and *getfield* and *getstatic* where the referenced field is of type `float` or `double`.

Execution of a Java virtual machine instruction implementing a store of an operand of type `float` that is represented in float-extended format always converts the operand value to float format before it is stored.

Execution of a Java virtual machine instruction implementing a store of an operand of type `double` that is represented in double-extended format always converts the operand value to double format before it is stored.

The affected instructions are *faload, daload,* and *putfield* and *putstatic* where the referenced field is of type `float` or `double`.

Comparison Instructions

Execution of a Java virtual machine instruction implementing a comparison operation on two values of type `float` or on two values of type `double` first performs format conversion separately on its operand values, but subject to the additional constraint that the implementation must make its choices in such a way that the two operands, after format conversion, are represented in the same format. The affected instructions are *fcmpl*, *fcmpg*, *dcmpl*, and *dcmpg*.

Method Invocation Instructions

On every FP-wide execution of a Java virtual machine method invocation instruction, the operand stack must contain *nargs* values which are to be passed to the invoked method as local variables in the stack frame of the invoked method. The number of argument values and the type and order of the values must be consistent with the descriptor of the resolved method.

Each invocation of a non-`native` method creates a new stack frame for the method being invoked. Two cases must be considered:

- If this is a class (`static`) method invocation, the *nargs* argument values are popped from the operand stack. Format conversion is performed on each argument value of type `float` or type `double`, then the argument values are stored into the first *nargs* local variables of the new stack frame, with the first argument value in local variable 0, the second argument value in local variable 1, and so on.
- If this is not a class method invocation, the *nargs* argument values and *objectref* are popped from the operand stack. Format conversion is performed on each argument value of type `float` or type `double`, then the *objectref* and argument values are stored into the first *nargs*+1 local variables of the new stack frame, with *objectref* in local variable 0, the first argument value in local variable 1, and so on.

The *invokeinterface* instruction has an *nargs* operand (called *count* in the first revision of *The Java Virtual Machine Specification*) which is currently used to specify the number of words of arguments to be found on the operand stack. The *count* operand and its interpretation are retained for compatibility. The *count* operand is an unsigned byte which must not be zero. The *count* value must be consistent with the descriptor of the resolved interface method, where each method descriptor parameter of type `long` or `double` contributes two units to the *count* value, and each method descriptor parameter of any other type contributes one unit.

Note that implementations that use extended floating-point formats may have difficulty using *count* to derive the number of argument values to be pushed for the interface method invocation. This information may be derived from the interface method descriptor.

The Java virtual machine method invocation instructions are *invokeinterface*, *invokespecial*, *invokestatic*, and *invokevirtual*.

Return Instructions

Execution of a Java virtual machine instruction implementing a return operation on a return value of type `float` or `double` first performs format conversion on the return value. The affected instructions are *freturn* and *dreturn*.

Operand Stack Management Instructions

Java virtual machine instructions that manipulate untyped data on the operand stack (e.g. *dup*, *pop2*, *swap*) will be defined in terms of values rather than words. These instructions will be interpreted depending on the format used to represent the value they operate on, but not the type of those values. Some of these instructions can operate on values that may be in any one of several forms, and will behave in different ways depending on the forms of their operands. Each alternative behavior will be represented by a separate stack diagram in the instruction definition. For instance the *pop2* instruction will have two stack diagrams; the *dup2_x2* instruction will have four stack diagrams.

For the purposes of defining this class of instructions we will separate the possible formats of operands into two classes. Values represented using the floating-point formats float-extended, double, double-extended, as well as values of the integral type `long`, will be called “format class 2". Values of other types (`boolean`, `byte`, `char`, `short`, `int`, and `float` values represented using float format), which will be called “format class 1".

The instructions to be defined in this way, and the stack diagrams they will support, include:

dup2

Stack ..., *value2*, *value1* $\Rightarrow$
..., *value2*, *value1*, *value2*, *value1*

where *value1* and *value2* are of format class 1

OR

Stack ..., *value* $\Rightarrow$
..., *value*, *value*

where *value* is of format class 2

dup2_x1

Stack ..., *value3*, *value2*, *value1* ⇒
..., *value2*, *value1*, *value3*, *value2*, *value1*

where value1, value2, and value3 are of format class 1

OR

Stack ..., *value2*, *value1* ⇒
..., *value1*, *value2*, *value1*

where *value1* is of format class 2 and *value2* is of format class 1

dup2_x2

Stack ..., *value4*, *value3*, *value2*, *value1* ⇒
..., *value2*, *value1*, *value4*, *value3*, *value2*, *value1*

where *value1*, *value2*, *value3*, and *value4* are of format class 1

OR

Stack ..., *value3*, *value2*, *value1* ⇒
..., *value1*, *value3*, *value2*, *value1*

where *value1* is of format class 2 and *value2* and *value3* are of format class 1

OR

Stack ..., *value3*, *value2*, *value1* ⇒
..., *value2*, *value1*, *value3*, *value2*, *value1*

where *value3* is of format class 2 and *value1* and *value2* are of format class 1

OR

Stack ..., *value2*, *value1* ⇒
..., *value1*, *value2*, *value1*

where *value1* and *value2* are of format class 2

dup_x2

Stack ..., *value3*, *value2*, *value1* ⇒
..., *value1*, *value3*, *value2*, *value1*

where *value1*, *value2*, and *value3* are of format class 1

OR

Stack ..., *value2*, *value1* ⇒
..., *value1*, *value2*, *value1*

where *value2* is of format class 2 and *value1* is of format class 1 *pop2*

Stack ..., *value2*, *value1* $\Rightarrow$
...

where *value2* and *value2* are of format class 1

OR

Stack ..., *value* $\Rightarrow$
...

where *value2* is of format class 2

The following operand stack manipulation instructions are currently defined in terms of one word on the operand and need to be trivially modified to operate on one value instead. Each will have a single stack diagram:

dup

Stack ..., *value* $\Rightarrow$
..., *value*, *value*

where *value* is of format class 1

dup_x1

Stack ..., *value2*, *value1* $\Rightarrow$
..., *value1*, *value2*, *value1*

where *value1* and *value2* are of format class 1

pop

Stack ..., *value* $\Rightarrow$
...

where *value* is of format class 1

swap

Stack ..., *value2*, *value1* $\Rightarrow$
..., *value1*, *value2*

where *value1* and value2 are of format class 1

Chapter 7, Compiling for the Java Virtual Machine

This chapter requires minor changes to correct operand stack word orientation, e.g. on pp. 343-344 of the first printing.

Chapter 9, An Optimization

The definitions of the implementation-specific *_quick* instructions will be modified to be consistent with the Java virtual machine instruction set. The changes are not detailed in this proposal.

Implications for Compilers and Tools

Although the Java programming language permits classes as well as methods to be declared using the `widefp` or `strictfp` modifiers, the `class` file format only provides facilities to specify the floating-point mode of individual methods. Compilers for the Java programming language must propagate an explicitly specified floating-point mode from a class declaration to each method defined in that class unless the declaration of the method itself explicitly specifies a floating-point mode.

Compilers for the Java programming language must flag each method that has been declared to be FP-wide or FP-strict, whether the declaration was made for the specific method or for the class containing the method and propagated to the method. They do this by setting the `ACC_EXPLICIT` bit of the `access_flags` item of the `method_info` structure for each method with an explicitly declared floating-point mode, and leaving it cleared otherwise.

If a method is explicitly declared to be FP-wide, whether in its declaration or in the declaration of its class, or if no floating-point mode is explicitly declared for the method or its class, then a compiler must clear the `ACC_STRICT` bit and set the `ACC_EXPLICIT` bit of the `access_flags` item of the `method_info` structure of that method. If a method is explicitly declared to be FP-strict, whether in its declaration or in the declaration of its class, then a compiler must set the `ACC_STRICT` bit and the `ACC_EXPLICIT` of the `access_flags` item of the `method_info` structure of that method.

The floating-point mode of a class is used as the floating-point mode of its class initialization and instance initialization methods. The floating-point mode of constructors, static initializers, and instance initializers may not be set individually.

Constant expressions should always be treated as FP-strict. Compilers themselves implemented in the Java programming language must take care that the possible use of extended precision by the host Java platform on which the compiler is run does not cause compile-time constant expressions to be evaluated using extended formats.

Compilers and static optimizers must not inline if doing so would reduce the constraints on rounding mandated by the virtual machine specification. The constraints that apply to a given situation will vary depending on the floating-point mode in effect.

For example, a method declared using the `widefp` modifier such as

```
widefp private double add(double a, double b) { return a + b; }
```

could normally be inlined using a single *fadd* instruction. However, a variant of that method declared using the `strictfp` modifier cannot be trivially inlined:

```
strictfp private double add(double a, double b) { return a + b; }
```

Because the specification requires rounding of the method arguments, a compiler wishing to inline this method may have to insert additional instructions to make the inlining conformant.

Other compiler optimizations such as forward substitution may require similar introduction of additional instructions to make the optimization conformant.

Tools such as class-to-class transformers and user-defined class loaders may be defined to change the effective floating-point mode of implicitly FP-strict classes to FP-wide or implicitly FP-wide classes to FP-strict. Such tools may be useful for legacy classes where source is not available and you find different results on different platforms. Alternatively, you may wish to use such tools to convert classes to be FP-wide in order to detect numerically unstable algorithms.

Such tools should only modify the `ACC_STRICT` flag value of methods whose `ACC_EXPLICIT` bit is not set. Methods whose `ACC_EXPLICIT` bit is set should not be modified by such tools; these methods have declared a specific floating-point mode and may not operate properly if that declaration is not respected.

Neither the Java language specification nor the Java virtual machine specification mandates a tool or procedure to change the effective floating-point mode of implicitly FP-wide or implicitly FP-strict classes.

Alternatives for Java Virtual Machine Implementors

The proposed changes to the Java programming language and virtual machine specifications do not invalidate a currently conforming Java virtual machine implementation; any implementation conforming to the existing specification conforms to the proposed new specification. However, implementors targeting processors that can benefit by the proposed extensions may wish to modify their Java virtual machine implementations to take advantage of those extensions to provide better performance or increased precision.

A given implementation is free to offer alternatives, within the confines of the specifications. On platforms that naturally support floating-point calculations using IEEE 754 extended formats, the most likely implementation choices will be between using extended formats in the permitted contexts and exclusively using the more restrictive formats of earlier Java programming language and virtual machine specifications. The first choice is likely to yield improved speed for floating-point calculations. The second should guarantee identical results across a wider range of contemporary and future implementations and backwards compatibility with older implementations.

An implementation that supports floating-point calculations using extended formats may provide variants of that support. For instance, an implementation may let the user select whether to use float-extended format, double-extended format, or both. An implementation may also let the user select details about the extended formats to use, within the ranges permitted by IEEE 754.

Where an implementation provides a choice between alternative permitted behaviors, those behaviors must be chosen on Java virtual machine startup using an implementation-specific mechanism such as a command line flag.

Implications for Testing

The proposed changes have implications on Java platform compatibility testing. In particular, some small number of current tests that test for the original Java programming language floating-point behavior may be invalidated. A new suite of tests for floating-point conformance will be written. These tests will be written in such a way as to verify conformance of implementations that produce results within the range of values permitted by the extended Java programming language specification when running FP-wide code. Such tests would of course verify conformance of an implementation that chooses to treat FP-wide code as FP-strict.

In addition, tests verifying conformance of implementations when running FP-strict code will be written, and will require bit-for-bit compatibility for all implementations for such code. Note that such tests are likely to be more stringent than those performed in current compatibility testing; hence, an implementation that passes current floating-point compatibility tests, but does not exactly implement the Java programming language and virtual machine specifications for floating-point, may not pass the improved test suites.

A given implementation that may be run in several modes offering alternative floating-point semantics will be required to conform to the specification in all available floating-point modes. The proposed extended specification limits permissible floating-point behaviors. Implementations or modes of implementations that fall outside of the proposed new specification will not pass conformance testing.

Notes for Implementors

The existing Java programming language and virtual machine specifications specify that values of type `float` and operators on values of type `float` behave exactly as specified for IEEE 754 single format values. Values of type `double` and operators on values of type `double` behave exactly as specified for IEEE 754 double format values. In particular, the Java programming language and virtual machine currently require support of IEEE 754 denormalized floating-point numbers and gradual underflow. In the present proposal, these properties continue to apply to FP-strict code but not to FP-wide code.

Some CPU architectures may need to take special care to implement fully conforming FP-strict operations. For instance, CPUs with fused multiply and add instructions may be unable to use those instructions when implementing FP-strict operations.

In order to implement conforming FP-strict operations on floating-point values on Intel Architecture CPUs (e.g. 486 or Pentium), a Java virtual machine implementation has to implement an operation such as multiplication using a code sequence equivalent to the following:

```
  fld    qword ptr [dx]
  fclex
  fmul   qword ptr [dy]   // 53-bits of sign., 15-bits of exp.
  fstsw  word ptr  [sw]   // rounded-up in C1 and sticky
                          // in Precision(Inexact)
  fst    qword ptr [dtmp] // 53-bits of significand,
  fstsw  ax               // and 11-bits of exponent
  and    ax,0x30          // Precision/Inexact AND Underflow
  xor    ax,0x30          // set after fmul and store
  jne    skip             // if not then okay, continue
// subroutine call to fix-up:
// fix-up will use [sw] and top of x87 to round and clamp as
// required by strict Java floating-point
skip:    fstp  qword ptr [dz]
```

Note that the following code sequence:

```
  fld    qword ptr [dx]
  fmul   qword ptr [dy]
  fstp   qword ptr [dz]
```

is not sufficient and will not satisfy future conformance testing.