NUM53-J. Use the strictfp modifier for floating-point calculation consistency across platforms

The Java language allows platforms to use available floating-point hardware that can provide *extended floating-point support* with exponents that contain more bits than the standard Java primitive type `double` (in the absence of the `strictfp` modifier). Consequently, these platforms can represent a superset of the values that can be represented by the standard floating-point types. Floating-point computations on such platforms can produce different results than would be obtained if the floating-point computations were restricted to the standard representations of `float` and `double`. According to the JLS, §15.4, "FP-strict Expressions" [JLS 2005]:

> *The net effect [of non-fp-strict evaluation], roughly speaking, is that a calculation might produce "the correct answer" in situations where exclusive use of the float value set or double value set might result in overflow or underflow.*

Programs that require consistent results from floating-point operations across different JVMs and platforms must use the `strictfp` modifier. This modifier requires the JVM and the platform to behave as though all floating-point computations were performed using values limited to those that can be represented by a standard Java `float` or `double`, guaranteeing that the result of the computations will match exactly across all JVMs and platforms.

Using the `strictfp` modifier leaves execution unchanged on platforms that lack platform-specific, extended floating-point support. It can have substantial impact, however, on both the efficiency and the resulting values of floating-point computations when executing on platforms that provide extended floating-point support. On these platforms, using the `strictfp` modifier increases the likelihood that intermediate operations will overflow or underflow because it restricts the range of intermediate values that can be represented; it can also reduce computational efficiency. These issues are unavoidable when portability is the main concern.

The `strictfp` modifier can be used with a class, method, or interface:

Usage	Applies to
Class	All code in the class (instance, variable, static initializers), and code in nested classes
Method	All code within the method
Interface	All code in any class that implements the interface

An expression is *FP-strict* when any of the containing classes, methods, or interfaces is declared to be `strictfp`. Constant expressions containing floating-point operations are also evaluated strictly. All compile-time constant expressions are by default FP-strict.

Strict behavior is not inherited by a subclass that extends a FP-strict superclass. An overriding method can independently choose to be FP-strict when the overridden method is not, or vice versa.

Noncompliant Code Example

This noncompliant code example does not mandate FP-strict computation. `Double.MAX_VALUE` is multiplied by 1.1 and reduced back by dividing by 1.1, according to the evaluation order. If `Double.MAX_VALUE` is the maximum value permissible by the platform, the calculation will yield the result `infinity`.

However, if the platform provides extended floating-point support, this program might print a numeric result roughly equivalent to `Double.MAX_VALUE`.

The JVM may choose to treat this case as FP-strict; if it does so, overflow occurs. Because the expression is not FP-strict, an implementation may use an extended exponent range to represent intermediate results.

```
class Example {
  public static void main(String[] args) {
    double d = Double.MAX_VALUE;
    System.out.println("This value \"" + ((d * 1.1) / 1.1) + "\" cannot be represented as double.");
  }
}
```

Compliant Solution

For maximum portability, use the `strictfp` modifier within an expression (class, method, or interface) to guarantee that intermediate results do not vary because of implementation-defined behavior. The calculation in this compliant solution is guaranteed to produce `infinity` because of the intermediate overflow condition, regardless of what floating-point support is provided by the platform.

```
strictfp class Example {
  public static void main(String[] args) {
    double d = Double.MAX_VALUE;
    System.out.println("This value \"" + ((d * 1.1) / 1.1) + "\" cannot be represented as double.");
  }
}
```

Noncompliant Code Example

Native floating-point hardware provides greater range than `double`. On these platforms, the JIT is permitted to use floating-point registers to hold values of type `float` or type `double` (in the absence of the `strictfp` modifier), even though the registers support values with greater exponent range than that of the primitive types. Consequently, conversion from `float` to `double` can cause an *effective* loss of magnitude.

```
class Example {
  double d = 0.0;

  public void example() {
    float f = Float.MAX_VALUE;
    float g = Float.MAX_VALUE;
    this.d = f * g;
    System.out.println("d (" + this.d + ") might not be equal to " +
                       (f * g));
  }

  public static void main(String[] args) {
    Example ex = new Example();
    ex.example();
  }
}
```

Magnitude loss would also occur if the value were stored to memory – for example, to a field of type `float`.

Compliant Solution

This compliant solution uses the `strictfp` keyword to require exact conformance with standard Java floating-point. Consequently, the intermediate value of both computations of `f * g` is identical to the value stored in `this.d`, even on platforms that support extended range exponents.

```
strictfp class Example {
  double d = 0.0;

  public void example() {
    float f = Float.MAX_VALUE;
    float g = Float.MAX_VALUE;
    this.d = f * g;
    System.out.println("d (" + this.d + ") might not be equal to " +
                       (f * g));
  }

  public static void main(String[] args) {
    Example ex = new Example();
    ex.example();
  }
}
```

Exceptions

NUM06-J-EX0: This rule applies only to calculations that require consistent floating-point results on all platforms. Applications that lack this requirement need not comply.

Risk Assessment

Failure to use the `strictfp` modifier can result in nonportable, implementation-defined behavior with respect to the behavior of floating-point operations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
NUM06-J	low	unlikely	high	P1	L3

Related Guidelines

The CERT C Secure Coding Standard	FLP00-C. Understand the limitations of floating-point numbers
SEI CERT C++ Coding Standard	VOID FLP00-CPP. Understand the limitations of floating-point numbers

Bibliography

[Darwin 2004]	Ensuring the Accuracy of Floating-Point Numbers
[JLS 2005]	§15.4, FP-strict Expressions
[JPL 2006]	9.1.3, Strict and Non-Strict Floating-Point Arithmetic
[McCluskey 2001]	Making Deep Copies of Objects, Using strictfp, and Optimizing String Performance

---