Updates to the Java™ Language Specification for JDK™ Release 1.2 Floating Point

Changes and transformations every hour, every moment . . ..
—Walt Whitman, *Visor'd* (1860), in *Leaves of Grass*

THIS DOCUMENT contains all the portions of *The Java Language Specification*, first edition, that have been changed because of the introduction of the `strictfp` keyword. Lines containing changes are emphasized using change bars in the left-hand margin. Lines not marked as containing changes are as they appeared in *The Java Language Specification,* first edition, and may have themselves been modified or superseded by other updates. Sections are numbered as they appear *The Java Language Specification*. Where material from a given section was not necessary for context it has been omitted, and that omission represented by ellipsis (. . .).

3.9 Keywords

The following character sequences, formed from ASCII letters, are reserved for use as *keywords* and cannot be used as identifiers (§3.8):

Keyword: one of

```
abstract    default    if           private     this
boolean     do         implements   protected   throw
break       double     import       public      throws
byte        else       instanceof   return      transient
case        extends    int          short       try
catch       final      interface    static      void
char        finally    long         strictfp    volatile
class       float      native       super       while
const       for        new          switch
continue    goto       package      synchronized
```

The keywords `const` and `goto` are reserved by Java, even though they are not currently used in Java. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in Java programs.

While `true` and `false` might appear to be keywords, they are technically Boolean literals (§3.10.3). Similarly, while `null` might appear to be a keyword, it is technically the null literal (§3.10.7).

4.2.3 Floating-Point Types, Value Sets, and Values

The floating-point types are `float` and `double`, which are conceptually associated with the 32-bit single-precision and 64-bit double-precision format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also positive and negative zeros, positive and negative *infinities*, and a special *Not-a-Number* value (hereafter abbreviated as "NaN"). The NaN value is used to represent the result of certain operations such as dividing zero by zero. NaN constants of both `float` and `double` type are predefined as `Float.NaN` (§20.9.5) and `Double.NaN` (§20.10.5).

Every implementation of the Java programming language is required to support two standard sets of floating-point values, called the *float value set* and the *double value set*. In addition, an implementation of the Java programming language may, at its option, support either or both of two extended-exponent floating-point value sets, called the *float-extended-exponent value set* and the *double-extended-exponent value set*. These extended-exponent value sets may, under certain circumstances, be used instead of the standard value sets to represent the values of expressions of type `float` or `double` (§5.1.8, §15.28).

The finite nonzero values of any floating-point value set can all be expressed in the form $s \cdot m \cdot 2^{(e - N + 1)}$, where s is +1 or –1, m is a positive integer less than

2^N, and e is an integer between $Emin = -(2^{K-1} - 2)$ and $Emax = 2^{K-1} - 1$, inclusive, and where N and K are parameters that depend on the value set. Some values can be represented in this form in more than one way; for example, supposing that a value v in a value set might be represented in this form using certain values for s, m, and e, then if it happened that m were even and e were less than 2^{K-1}, one could halve m and increase e by 1 to produce a second representation for the same value v. A representation in this form is called *normalized* if $m \geq 2^{(N-1)}$; otherwise the representation is said to be *denormalized*. If a value in a value set cannot be represented in such a way that $m \geq 2^{(N-1)}$, then the value is said to be a *denormalized value*, because it has no normalized representation.

The constraints on the parameters N and K (and on the derived parameters *Emin* and *Emax*) for the two required and two optional floating-point value sets are summarized in Table 4.1.

Parameter	**float**	**float extended-exponent**	**double**	**double extended-exponent**
N	24	24	53	53
K	8	≥ 11	11	≥ 15
Emax	+127	≥ +1023	+1023	≥ +16383
Emin	–126	≤ –1022	–1022	≤ –16382

Table 4.1 Floating-point value set parameters

Where one or both extended-exponent value sets are supported by an implementation, then for each supported extended-exponent value set there is a specific implementation-dependent constant *K*, whose value is constrained by Table 4.1; this value *K* in turn dictates the values for *Emin* and *Emax*.

Each of the four value sets includes not only the finite nonzero values that are ascribed to it above, but also the five values positive zero, negative zero, positive infinity, negative infinity, and NaN.

Note that the constraints in Table 4.1 are designed so that every element of the float value set is necessarily also an element of the float-extended-exponent value set, the double value set, and the double-extended-exponent value set. Likewise, each element of the double value set is necessarily also an element of the double-extended-exponent value set. Each extended-exponent value set has a larger range of exponent values than the corresponding standard value set, but does not have more precision.

The elements of the float value set are exactly the values that can be represented using the single floating-point format defined in the IEEE 754 standard, except that there is only one NaN value (IEEE 754 specifies $2^{24} - 2$ distinct NaN values). The elements of the double value set are exactly the values that can be represented using the double floating-point format defined in the IEEE 754 standard, except that there is only one NaN value (IEEE 754 specifies $2^{53} - 2$ distinct NaN values). Note, however, that the elements of the float-extended-exponent and double-extended-exponent value sets defined here do *not* correspond to the values that be represented using IEEE 754 single extended and double extended formats, respectively.

The float, float-extended-exponent, double, and double-extended-exponent value sets are not types. It is always correct for an implementation of the Java language to use an element of the float value set to represent a value of type `float`; however, it may be permissible in certain regions of code for an implementation to use an element of the float-extended-exponent value set instead. Similarly, it is always correct for an implementation to use an element of the double value set to represent a value of type `double`; however, it may be permissible in certain regions of code for an implementation to use an element of the double-extended-exponent value set instead.

Except for NaN, floating-point values are *ordered*; arranged from smallest to largest, they are negative infinity, negative finite nonzero values, positive and negative zero, positive finite nonzero values, and positive infinity.

Positive zero and negative zero compare equal; thus the result of the expression `0.0==-0.0` is `true` and the result of `0.0>-0.0` is `false`. But other operations can distinguish positive and negative zero; for example, `1.0/0.0` has the value positive infinity, while the value of `1.0/-0.0` is negative infinity. The operations `Math.min` and `Math.max` also distinguish positive zero and negative zero.

NaN is *unordered*, so the numerical comparison operators <, <=, >, and >= return `false` if either or both operands are NaN (§15.19.1). The equality operator == returns `false` if either operand is NaN, and the inequality operator `!=` returns `true` if either operand is NaN (§15.20.1). In particular, `x!=x` is `true` if and only if `x` is NaN, and `(x<y) == !(x>=y)` will be `false` if `x` or `y` is NaN.

Any value of a floating-point type may be cast to or from any numeric type. There are no casts between floating-point types and the type `boolean`.

Chapter 5

. . .

In every conversion context, only certain specific conversions are permitted. The specific conversions that are possible in Java are grouped for convenience of description into several broad categories:

- Identity conversions
- Widening primitive conversions
- Narrowing primitive conversions
- Widening reference conversions
- Narrowing reference conversions
- String conversions
- Value set conversions

There are five *conversion contexts* in which conversion of Java expressions may occur. Each context allows conversions in some of the categories named above but not others. The term "conversion" is also used to describe the process of choosing a specific conversion for such a context. For example, we say that an expression that is an actual argument in a method invocation is subject to "method invocation conversion," meaning that a specific conversion will be implicitly chosen for that expression according to the rules for the method invocation argument context.

One conversion context is the operand of a numeric operator such as + or *. The conversion process for such operands is called *numeric promotion*. Promotion is special in that, in the case of binary operators, the conversion chosen for one operand may depend in part on the type of the other operand expression.

This chapter first describes the seven categories of conversions (§5.1), including value set conversions and the special conversions to `String` allowed for the string concatenation operator +. Then the five conversion contexts are described:

- Assignment conversion (§5.2, §15.25) converts the type of an expression to the type of a specified variable. The conversions permitted for assignment are limited in such a way that assignment conversion never causes an exception.
- Method invocation conversion (§5.3, §15.8, §15.11) is applied to each argument in a method or constructor invocation and, except in one case, performs the same conversions that assignment conversion does. Method invocation conversion never causes an exception.
- Casting conversion (§5.4) converts the type of an expression to a type explicitly specified by a cast operator (§15.15). It is more inclusive than assignment or method invocation conversion, allowing any specific conversion other than a string conversion, but certain casts to a reference type may cause an exception at run time.

- String conversion (§5.4, §15.17.1) allows any type to be converted to type `String`.
- Numeric promotion (§5.6) brings the operands of a numeric operator to a common type so that an operation can be performed.

. . .

5.1 Kinds of Conversion

Specific type conversions in Java are divided into seven categories.

5.1.8 Value Set Conversion

Value set conversion is the process of mapping a floating-point value from one value set to another without changing its type.

Within an expression that is not FP-strict (§15.28), value set conversion provides choices to an implementation of the Java language:

- If the value is an element of the float-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the float value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).
- If the value is an element of the double-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the double value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).

Within an FP-strict expression (§15.28), value set conversion does not provide any choices; every implementation must behave in the same way:

- If the value is of type `float` and is not an element of the float value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If the value is of type `double` and is not an element of the double value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

Within an FP-strict expression, mapping values from the float-extended-exponent value set or double-extended-exponent value set is necessary only when a method is invoked whose declaration is not FP-strict and the implementation has chosen to represent the result of the method invocation as an element of an extended-exponent value set.

Whether in FP-strict code or code that is not FP-strict, value set conversion always leaves unchanged any value whose type is neither `float` nor `double`.

5.2 Assignment Conversion

Assignment conversion occurs when the value of an expression is assigned (§15.25) to a variable: the type of the expression must be converted to the type of the variable. Assignment contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4). In addition, a narrowing primitive conversion may be used if all of the following conditions are satisfied:

- The expression is a constant expression of type `int`.
- The type of the variable is `byte`, `short`, or `char`.
- The value of the expression (which is known at compile time, because it is a constant expression) is representable in the type of the variable.

If the type of the expression cannot be converted to the type of the variable by a conversion permitted in an assignment context, then a compile-time error occurs.

If the type of the variable is `float` or `double`, then value set conversion is applied after the type conversion:

- If the value is of type `float` and is an element of the float-extended-exponent value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If the value is of type `double` and is an element of the double-extended-exponent value set, then the implementation must map the value to the nearest ele-

ment of the double value set. This conversion may result in overflow or underflow.

If the type of an expression can be converted to the type a variable by assignment conversion, we say the expression (or its value) is *assignable to* the variable or, equivalently, that the type of the expression is *assignment compatible with* the type of the variable.

. . .

5.3 Method Invocation Conversion

Method invocation conversion is applied to each argument value in a method or constructor invocation (§15.8, §15.11): the type of the argument expression must be converted to the type of the corresponding parameter. Method invocation contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4).

If the type of an argument expression is either `float` or `double`, then value set conversion (§5.1.8) is applied after the type conversion:

- If an argument value of type `float` is an element of the float-extended-exponent value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If an argument value of type `double` is an element of the double-extended-exponent value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

. . .

5.5 Casting Conversion

Sing away sorrow, cast away care.
—Miguel de Cervantes (1547–1616),
Don Quixote (Lockhart's translation), Chapter viii

Casting conversion is applied to the operand of a cast operator (§15.15): the type of the operand expression must be converted to the type explicitly named by the cast operator. Casting contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), a narrowing primitive conversion (§5.1.3), a widening reference conversion (§5.1.4), or a narrowing reference conversion (§5.1.5). Thus casting conversions are more inclusive than assignment or method invocation conversions: a cast can do any permitted conversion other than a string conversion.

Value set conversion (§5.1.8) is applied after the type conversion.

Some casts can be proven incorrect at compile time; such casts result in a compile-time error.

. . .

5.6.1 Unary Numeric Promotion

Some operators apply *unary numeric promotion* to a single operand, which must produce a value of a numeric type:

- If the operand is of compile-time type `byte`, `short`, or `char`, unary numeric promotion promotes it to a value of type `int` by a widening conversion (§5.1.2).
- Otherwise, a unary numeric operand remains as is and is not converted.

In either case, value set conversion (§5.1.8) is then applied.

. . .

5.6.2 Binary Numeric Promotion

When an operator applies *binary numeric promotion* to a pair of operands, each of which must denote a value of a numeric type, the following rules apply, in order, using widening conversion (§5.1.2) to convert operands as necessary:

- If either operand is of type `double`, the other is converted to `double`.
- Otherwise, if either operand is of type `float`, the other is converted to `float`.
- Otherwise, if either operand is of type `long`, the other is converted to `long`.
- Otherwise, both operands are converted to type `int`.

After the type conversion, if any, value set conversion (§5.1.8) is then applied to each operand.

. . .

8.1.2 Class Modifiers

A class declaration may include *class modifiers*.

> *ClassModifiers:*
>   *ClassModifier*
>   *ClassModifiers ClassModifier*
>
> *ClassModifier: one of*
>   `public abstract final strictfp`

The access modifier `public` is discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in a class declaration. If two or more class modifiers appear in a class declaration, then it is customary, though not required, that they appear in the order consistent with that shown above in the production for *ClassModifier.*

8.3 Field Declarations

Poetic fields encompass me around,
And still I seem to tread on classic ground.
—Joseph Addison (1672–1719), *A Letter from Italy*

The variables of a class type are introduced by *field declarations*:

> *FieldDeclaration:*
>   *FieldModifiers$_{opt}$ Type VariableDeclarators* `;`
>
> *VariableDeclarators:*
>   *VariableDeclarator*
>   *VariableDeclarators* `,` *VariableDeclarator*
>
> *VariableDeclarator:*
>   *VariableDeclaratorId*
>   *VariableDeclaratorId* `=` *VariableInitializer*
>
> *VariableDeclaratorId:*
>   *Identifier*
>   *VariableDeclaratorId* `[ ]`
>
> *VariableInitializer:*
>   *Expression*
>   *ArrayInitializer*

The *FieldModifiers* are described in §8.3.1. The *Identifier* in a *FieldDeclarator* may be used in a name to refer to the field. The name of a field has as its scope (§6.3) the entire body of the class declaration in which it is declared. More than one field may be declared in a single field declaration by using more than one declarator; the *FieldModifiers* and *Type* apply to all the declarators in the declaration. Variable declarations involving array types are discussed in §10.2.

It is a compile-time error for the body of a class declaration to contain declarations of two fields with the same name. Methods and fields may have the same name, since they are used in different contexts and are disambiguated by the different lookup procedures (§6.5).

If the class declares a field with a certain name, then the declaration of that field is said to *hide* (§6.3.1) any and all accessible declarations of fields with the same name in the superclasses and superinterfaces of the class.

If a field declaration hides the declaration of another field, the two fields need not have the same type.

A class inherits from its direct superclass and direct superinterfaces all the fields of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.

It is possible for a class to inherit more than one field with the same name (§8.3.3.3). Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the class to refer to any such field by its simple name will result in a compile-time error, because such a reference is ambiguous.

There might be several paths by which the same field declaration might be inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

A hidden field can be accessed by using a qualified name (if it is `static`) or by using a field access expression (§15.10) that contains the keyword `super` or a cast to a superclass type. See §15.10.2 for discussion and an example.

A value stored in a field of type `float` is always an element of the float value set (§4.2.3); similarly, a value stored in a field of type `double` is always an element of the double value set. It is not permitted for a field of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a field of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

8.4.1 Formal Parameters

The *formal parameters* of a method, if any, are specified by a list of comma-separated parameter specifiers. Each parameter specifier consists of a type and an identifier (optionally followed by brackets) that specifies the name of the parameter:

FormalParameterList:
FormalParameter
FormalParameterList `,` *FormalParameter*

FormalParameter:
Type VariableDeclaratorId

The following is repeated from §8.3 to make the presentation here clearer:

VariableDeclaratorId:
Identifier
VariableDeclaratorId `[ ]`

If a method has no parameters, only an empty pair of parentheses appears in the method's declaration.

If two formal parameters are declared to have the same name (that is, their declarations mention the same *Identifier*), then a compile-time error occurs.

When the method is invoked (§15.11), the values of the actual argument expressions initialize newly created parameter variables, each of the declared *Type,* before execution of the body of the method. The *Identifier* that appears in the *DeclaratorId* may be used as a simple name in the body of the method to refer to the formal parameter.

The scope of formal parameter names is the entire body of the method. These parameter names may not be redeclared as local variables or exception parameters within the method; that is, hiding the name of a parameter is not permitted.

Formal parameters are referred to only using simple names, never by using qualified names (§6.6).

A method parameter of type `float` always contains an element of the float value set (§4.2.3); similarly, a method parameter of type `double` always contains an element of the double value set. It is not permitted for a method parameter of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a method parameter of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

Where an actual argument expression corresponding to a parameter variable is not FP-strict (§15.28), evaluation of that actual argument expression is permitted to use intermediate values drawn from the appropriate extended-exponent value sets. Prior to being stored in the parameter variable the result of such an expression is mapped to the nearest value in the corresponding standard value set by method invocation conversion (§5.3).

8.4.3 Method Modifiers

MethodModifiers:
    MethodModifier
    MethodModifiers MethodModifier

MethodModifier: one of
```
public protected private abstract static
final synchronized native strictfp
```

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in a method declaration, or if a method declaration has more than one of the access modifiers `public`, `protected`, and `private`. A compile-time error occurs if a method declaration that contains the keyword `abstract` also contains any one of the keywords `private`, `static`, `final`, `native`, `strictfp`, or `synchronized`. A compile-time error occurs if a method declaration that contains the keyword `native` also contains `strictfp`.

If two or more method modifiers appear in a method declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for *MethodModifier.*

8.4.6.1 Overriding (By Instance Methods)

If a class declares an instance method, then the declaration of that method is said to *override* any and all methods with the same signature in the superclasses and superinterfaces of the class that would otherwise be accessible to code in the class. Moreover, if the method declared in the class is not `abstract`, then the declaration of that method is said to *implement* any and all declarations of `abstract` methods with the same signature in the superclasses and superinterfaces of the class that would otherwise be accessible to code in the class.

A compile-time error occurs if an instance method overrides a `static` method. In this respect, overriding of methods differs from hiding of fields (§8.3), for it is permissible for an instance variable to hide a `static` variable.

An overridden method can be accessed by using a method invocation expression (§15.11) that contains the keyword `super`. Note that a qualified name or a cast to a superclass type is not effective in attempting to access an overridden method; in this respect, overriding of methods differs from hiding of fields. See §15.11.4.10 for discussion and examples of this point.

The presence or absence of the `strictfp` modifier has absolutely no effect on the rules for overriding methods and implementing abstract methods. For example, it is permitted for a method that is not FP-strict to override an FP-strict method and it is permitted for an FP-strict method to override a method that is not FP-strict.

8.6.3 Constructor Modifiers

ConstructorModifiers:
  ConstructorModifier
  ConstructorModifiers ConstructorModifier

ConstructorModifier: one of
```
public  protected  private
```

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in a constructor declaration, or if a constructor declaration has more than one of the access modifiers `public`, `protected`, and `private`.

Unlike methods, a constructor cannot be `abstract`, `static`, `final`, `native`, or `synchronized`. A constructor is not inherited, so there is no need to declare it `final` and an `abstract` constructor could never be implemented. A constructor is always invoked with respect to an object, so it makes no sense for a constructor to be `static`. There is no practical need for a constructor to be `synchronized`, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their work. The lack of `native` constructors is an arbitrary language design choice that makes it easy for an implementation of the Java Virtual Machine to verify that superclass constructors are always properly invoked during object creation.

A *ConstructorModifier* may *not* be `strictfp`. A compile-time error occurs if `strictfp` appears as a constructor modifier. This difference in the definitions for *ConstructorModifier* and *MethodModifier* (§8.4.3) is an intentional language design choice; it effectively ensures that a constructor is FP-strict if and only if its class is FP-strict, so to speak.

9.1.2 Interface Modifiers

An interface declaration may be preceded by *interface modifiers*:

InterfaceModifiers:
  InterfaceModifier
  InterfaceModifiers InterfaceModifier

InterfaceModifier: one of
```
public  abstract  strictfp
```

The access modifier `public` is discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in an interface declaration.

9.4 Abstract Method Declarations

AbstractMethodDeclaration:
  AbstractMethodModifiers$_{opt}$ ResultType MethodDeclarator Throws$_{opt}$;

AbstractMethodModifiers:
  AbstractMethodModifier
  AbstractMethodModifiers AbstractMethodModifier

AbstractMethodModifier: one of
  `public abstract`

The access modifier `public` is discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in an abstract method declaration.

Every method declaration in the body of an interface is implicitly `abstract`, so its body is always represented by a semicolon, not a block. For compatibility with older versions of Java, it is permitted but discouraged, as a matter of style, to redundantly specify the `abstract` modifier for methods declared in interfaces.

Every method declaration in the body of an interface is implicitly `public`. It is permitted, but strongly discouraged as a matter of style, to redundantly specify the `public` modifier for interface methods.

Note that a method declared in an interface must not be declared `static`, or a compile-time error occurs, because in Java `static` methods cannot be `abstract`.

Note that a method declared in an interface must not be declared `strictfp` or `native` or `synchronized`, or a compile-time error occurs, because those keywords describe implementation properties rather than interface properties. However, a method declared in an interface may be implemented by a method that is declared `strictfp` or `native` or `synchronized` in a class that implements the interface.

Note that a method declared in an interface must not be declared `final` or a compile-time error occurs. However, a method declared in an interface may be implemented by a method that is declared `final` in a class that implements the interface.

Chapter 10

. . .

All the components of an array have the same type, called the *component type* of the array. If the component type of an array is `T`, then the type of the array itself is written `T[]`.

An array component of type `float` is always an element of the float value set (§4.2.3); similarly, an array component of type `double` is always an element of the double value set. It is not permitted for an array component of type `float` to be an element of the float-extended-exponent value set that is not also an element of the float value set, nor for an array component of type `double` to be an element of the double-extended-exponent value set that is not also an element of the double value set.

. . .

14.3.1 Local Variable Declarators and Types

Each *declarator* in a local variable declaration declares one local variable, whose name is the *Identifier* that appears in the declarator.

The type of the variable is denoted by the *Type* that appears at the start of the local variable declaration, followed by any bracket pairs that follow the *Identifier* in the declarator. Thus, the local variable declaration:

```
int a, b[], c[][];
```

is equivalent to the series of declarations:

```
int a;
int[] b;
int[][] c;
```

Brackets are allowed in declarators as a nod to the tradition of C and C++. The general rule, however, also means that the local variable declaration:

```
float[][] f[][], g[][][], h[];             // Yechh!
```

is equivalent to the series of declarations:

```
float[][][][] f;
float[][][][][] g;
float[][][] h;
```

We do not recommend such “mixed notation” for array declarations.

A local variable of type `float` always contains a value that is an element of the float value set (§4.2.3); similarly, a local variable of type `double` always contains a value that is an element of the double value set. It is not permitted for a local variable of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a local variable of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

14.15 The `return` Statement

A `return` statement returns control to the invoker of a method (§8.4, §15.11) or constructor (§8.6, §15.8).

ReturnStatement:
    `return` *Expression*$_{opt}$ `;`

A `return` statement with no *Expression* must be contained in the body of a method that is declared, using the keyword `void`, not to return any value (§8.4), or in the body of a constructor (§8.6). A compile-time error occurs if a `return` statement appears within a static initializer (§8.5). A `return` statement with no *Expression* attempts to transfer control to the invoker of the method or constructor that contains it. To be precise, a `return` statement with no *Expression* always completes abruptly, the reason being a `return` with no value.

A `return` statement with an *Expression* must be contained in a method declaration that is declared to return a value (§8.4) or a compile-time error occurs. The *Expression* must denote a variable or value of some type *T*, or a compile-time error occurs. The type *T* must be assignable (§5.2) to the declared result type of the method, or a compile-time error occurs.

A `return` statement with an *Expression* attempts to transfer control to the invoker of the method that contains it; the value of the *Expression* becomes the value of the method invocation. More precisely, execution of such a `return` statement first evaluates the *Expression*. If the evaluation of the *Expression* completes abruptly for some reason, then the `return` statement completes abruptly for that reason. If evaluation of the *Expression* completes normally, producing a value *V*, then the `return` statement completes abruptly, the reason being a `return` with value *V*. (If the expression is of type `float` and is not FP-strict (§15.28), then the value may be an element of either the float value set or the float-extended-exponent value set (§4.2.3). If the expression is of type `double` and is not FP-strict, then the value may be an element of either the double value set or the double-extended-exponent value set.)

It can be seen, then, that a `return` statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.18) within the method or constructor whose `try` blocks contain the `return` statement, then any `finally` clauses of those `try` statements will be executed, in order, innermost to outermost, before control is transferred to the invoker of the method or constructor. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `return` statement.

15.1 Evaluation, Denotation, and Result

When an expression in a Java program is *evaluated* (*executed*), the *result* denotes one of three things:

- A variable (§4.5) (in C, this would be called an *lvalue*)
- A value (§4.2, §4.3)
- Nothing (the expression is said to be `void`)

Evaluation of an expression can also produce side effects, because expressions may contain embedded assignments, increment operators, decrement operators, and method invocations.

An expression denotes nothing if and only if it is a method invocation (§15.11) that invokes a method that does not return a value, that is, a method declared `void` (§8.4). Such an expression can be used only as an expression statement (§14.7), because every other context in which an expression can appear requires the expression to denote something. An expression statement that is a method invocation may also invoke a method that produces a result; in this case the value returned by the method is quietly discarded.

Value set conversion (§5.1.8) is applied to the result of every expression that produces a value.

Each expression occurs in the declaration of some (class or interface) type that is being declared: in a field initializer, in a static initializer, in a constructor declaration, or in the code for a method.

15.2 Variables as Values

If an expression denotes a variable, and a value is required for use in further evaluation, then the value of that variable is used. In this context, if the expression denotes a variable or a value, we may speak simply of the *value* of the expression.

If the value of a variable of type `float` or `double` is used in this manner, then value set conversion (§5.1.8) is applied to the value of the variable.

15.7.1 Literals

A literal (§3.10) denotes a fixed, unchanging value.

The following production from §3.10 is repeated here for convenience:

Literal:
 IntegerLiteral
 FloatingPointLiteral
 BooleanLiteral
 CharacterLiteral
 StringLiteral
 NullLiteral

The type of a literal is determined as follows:

- The type of an integer literal that ends with `L` or `l` is `long`; the type of any other integer literal is `int`.
- The type of a floating-point literal that ends with `F` or `f` is `float` and its value must be an element of the float value set (§4.2.3). The type of any other floating-point literal is `double` and its value must be an element of the double value set.
- The type of a boolean literal is `boolean`.
- The type of a character literal is `char`.
- The type of a string literal is `String`.
- The type of the null literal `null` is the null type; its value is the null reference.

Evaluation of a literal always completes normally.

15.7.3 Parenthesized Expressions

A parenthesized expression is a primary expression whose type is the type of the contained expression and whose value at run time is the value of the contained expression.

Parentheses do not affect in any way the choice of value set (§4.2.3) for the value of an expression of type `float` or `double`.

15.11.4.5 Create Frame, Synchronize, Transfer Control

A method *m* in some class *S* has been identified as the one to be invoked.

Now a new *activation frame* is created, containing the target reference (if any) and the argument values (if any), as well as enough space for the local variables and stack for the method to be invoked and any other bookkeeping information that may be required by the implementation (stack pointer, program counter, reference to previous activation frame, and the like). If there is not sufficient memory available to create such an activation frame, an `OutOfMemoryError` is thrown.

The newly created activation frame becomes the current activation frame. The effect of this is to assign the argument values to corresponding freshly created parameter variables of the method, and to make the target reference available as `this`, if there is a target reference. Before each argument value is assigned to its corresponding parameter variable, it is subjected to method invocation conversion (§5.3), which includes any required value set conversion (§5.1.8).

. . .

15.13.2 Postfix Increment Operator `++`

PostIncrementExpression:
    PostfixExpression `++`

A postfix expression followed by a `++` operator is a postfix increment expression. The result of the postfix expression must be a variable of a numeric type, or a compile-time error occurs. The type of the postfix increment expression is the type of the variable. The result of the postfix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value `1` is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (§5.6.2) is performed on the value `1` and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (§5.1.3) to the type of the variable before it is stored. The value of the postfix increment expression is the value of the variable *before* the new value is stored.

Note that the binary numeric promotion mentioned above may include value set conversion (§5.1.8). If necessary, value set conversion is applied to the sum prior to its being stored in the variable.

A variable that is declared `final` cannot be incremented, because when an access of a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a postfix increment operator.

15.13.3 Postfix Decrement Operator `--`

PostDecrementExpression:
    PostfixExpression `--`

A postfix expression followed by a `--` operator is a postfix decrement expression. The result of the postfix expression must be a variable of a numeric type, or a compile-time error occurs. The type of the postfix decrement expression is the type of the variable. The result of the postfix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value `1` is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (§5.6.2) is performed on the value `1` and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (§5.1.3) to the type of the variable before it is stored. The value of the postfix decrement expression is the value of the variable *before* the new value is stored.

Note that the binary numeric promotion mentioned above may include value set conversion (§5.1.8). If necessary, value set conversion is applied to the difference prior to its being stored in the variable.

A variable that is declared `final` cannot be decremented, because when an access of a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a postfix decrement operator.

15.14.1 Prefix Increment Operator ++

A unary expression preceded by a `++` operator is a prefix increment expression. The result of the unary expression must be a variable of a numeric type, or a compile-time error occurs. The type of the prefix increment expression is the type of the variable. The result of the prefix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value `1` is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (§5.6.2) is performed on the value `1` and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (§5.1.3) to the type of the variable before it is stored. The value of the prefix increment expression is the value of the variable *after* the new value is stored.

Note that the binary numeric promotion mentioned above may include value set conversion (§5.1.8). If necessary, value set conversion is applied to the sum prior to its being stored in the variable.

A variable that is declared `final` cannot be incremented, because when an access of a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a prefix increment operator.

15.14.2 Prefix Decrement Operator `--`

He must increase, but I must decrease.
—John 3:30

A unary expression preceded by a `--` operator is a prefix decrement expression. The result of the unary expression must be a variable of a numeric type, or a compile-time error occurs. The type of the prefix decrement expression is the type of the variable. The result of the prefix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value `1` is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (§5.6.2) is performed on the value `1` and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (§5.1.3) to the type of the variable before it is stored. The value of the prefix decrement expression is the value of the variable *after* the new value is stored.

Note that the binary numeric promotion mentioned above may include value set conversion (§5.1.8). If necessary, value set conversion is applied to the difference prior to its being stored in the variable.

A variable that is declared `final` cannot be decremented, because when an access of a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a prefix decrement operator.

15.14.4 Unary Minus Operator `-`

It is so very agreeable to hear a voice and to see all the signs of that expression.
—Gertrude Stein, *Rooms* (1914), in *Tender Buttons*

The type of the operand expression of the unary `-` operator must be a primitive numeric type, or a compile-time error occurs. Unary numeric promotion (§5.6.1) is performed on the operand. The type of the unary minus expression is the promoted type of the operand.

Note that unary numeric promotion performs value set conversion (§5.1.8). Whatever value set the promoted operand value is drawn from, the unary negation operation is carried out and the result is drawn from that same value set. That result is then subject to further value set conversion.

At run time, the value of the unary plus expression is the arithmetic negation of the promoted value of the operand.

For integer values, negation is the same as subtraction from zero. Java uses two's-complement representation for integers, and the range of two's-complement values is not symmetric, so negation of the maximum negative `int` or `long` results in that same maximum negative number. Overflow occurs in this case, but no exception is thrown. For all integer values `x`, `-x` equals `(~x)+1`.

For floating-point values, negation is not the same as subtraction from zero, because if `x` is `+0.0`, then `0.0-x` equals `+0.0`, but `-x` equals `-0.0`. Unary minus merely inverts the sign of a floating-point number. Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.

15.15 Cast Expressions

My days among the dead are passed;
Around me I behold,
Where'er these casual eyes are cast,
The mighty minds of old . . .
—Robert Southey (1774–1843),
Occasional Pieces, xviii

A cast expression converts, at run time, a value of one numeric type to a similar value of another numeric type; or confirms, at compile time, that the type of an expression is `boolean`; or checks, at run time, that a reference value refers to an object whose class is compatible with a specified reference type.

CastExpression:
`(` *PrimitiveType Dims*$_{opt}$ `)` *UnaryExpression*
`(` *ReferenceType* `)` *UnaryExpressionNotPlusMinus*

See §15.14 for a discussion of the distinction between *UnaryExpression* and *UnaryExpressionNotPlusMinus*.

The type of a cast expression is the type whose name appears within the parentheses. (The parentheses and the type they contain are sometimes called the *cast operator.*) The result of a cast expression is not a variable, but a value, even if the result of the operand expression is a variable.

A cast operator has no effect on the choice of value set (§4.2.3) for a value of type `float` or type `double`. Consequently, a cast to type `float` within an expression that is not FP-strict does not necessarily cause its value to be converted to an element of the float value set, and a cast to type `double` within an expression that is not FP-strict does not necessarily cause its value to be converted to an element of the double value set.

At run time, the operand value is converted by casting conversion (§5.4) to the type specified by the cast operator.

. . .

15.16 Multiplicative Operators

The operators `*`, `/`, and `%` are called the *multiplicative operators*. They have the same precedence and are syntactically left-associative (they group left-to-right).

MultiplicativeExpression:
UnaryExpression
MultiplicativeExpression `*` *UnaryExpression*
MultiplicativeExpression `/` *UnaryExpression*
MultiplicativeExpression `%` *UnaryExpression*

The type of each of the operands of a multiplicative operator must be a primitive numeric type, or a compile-time error occurs. Binary numeric promotion is performed on the operands (§5.6.2). The type of a multiplicative expression is the promoted type of its operands. If this promoted type is `int` or `long`, then integer arithmetic is performed; if this promoted type is `float` or `double`, then floating-point arithmetic is performed.

Note that binary numeric promotion performs value set conversion (§5.1.8).

15.16.1 Multiplication Operator *

. . .

The result of a floating-point multiplication is governed by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result is positive if both operands have the same sign, and negative if the operands have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity. The sign is determined by the rule stated above.

- In the remaining cases, where neither an infinity or NaN is involved, the exact mathematical product is computed. A floating-point value set is then chosen:
  - If the multiplication expression is FP-strict (§15.28):
    - If the type of the multiplication expression is `float`, then the float value set must be chosen.
    - If the type of the multiplication expression is `double`, then the double value set must be chosen.
  - If the multiplication expression is not FP-strict:
    - If the type of the multiplication expression is `float`, then the either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
    - If the type of the multiplication expression is `double`, then the either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

  Next, a value must be chosen from the chosen value set to represent the product. If the magnitude of the product is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. Otherwise, the product is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

Despite the fact that overflow, underflow, or loss of information may occur, evaluation of a multiplication operator * never throws a run-time exception.

15.16.2 Division Operator /

. . .

The result of a floating-point division is determined by the specification of IEEE arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity. The sign is determined by the rule stated above.

- Division of a finite value by an infinity results in a signed zero. The sign is determined by the rule stated above.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero. The sign is determined by the rule stated above.
- Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule stated above.
- In the remaining cases, where neither an infinity or NaN is involved, the exact mathematical quotient is computed. A floating-point value set is then chosen:
  - If the division expression is FP-strict (§15.28):
    - If the type of the division expression is `float`, then the float value set must be chosen.
    - If the type of the division expression is `double`, then the double value set must be chosen.
  - If the division expression is not FP-strict:
    - If the type of the division expression is `float`, then the either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
    - If the type of the division expression is `double`, then the either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

  Next, a value must be chosen from the chosen value set to represent the quotient. If the magnitude of the quotient is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. Otherwise, the quotient is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

Despite the fact that overflow, underflow, division by zero, or loss of information may occur, evaluation of a floating-point division operator `/` never throws a run-time exception.

15.17.2 Additive Operators (+ and -) for Numeric Types

The binary `+` operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The binary `-` operator performs subtraction, producing the difference of two numeric operands.

Binary numeric promotion is performed on the operands (§5.6.2). The type of an additive expression on numeric operands is the promoted type of its operands. If this promoted type is `int` or `long`, then integer arithmetic is performed; if this promoted type is `float` or `double`, then floating-point arithmetic is performed.

Note that binary numeric promotion performs value set conversion (§5.1.8).

Addition is a commutative operation if the operand expressions have no side effects. Integer addition is associative when the operands are all of the same type, but floating-point addition is not associative.

If an integer addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two's-complement format. If overflow occurs, then the sign of the result is not the same as the sign of the mathematical sum of the two operand values.

The result of a floating-point addition is determined using the following rules of IEEE arithmetic:

- If either operand is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two zeros of opposite sign is positive zero.
- The sum of two zeros of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the exact mathematical sum is computed. A floating-point value set is then chosen:
  - If the addition expression is FP-strict (§15.28):
    - If the type of the addition expression is `float`, then the float value set must be chosen.
    - If the type of the addition expression is `double`, then the double value set must be chosen.
  - If the addition expression is not FP-strict:

- If the type of the addition expression is `float`, then the either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
- If the type of the addition expression is `double`, then the either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

Next, a value must be chosen from the chosen value set to represent the sum. If the magnitude of the sum is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. Otherwise, the sum is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

. . .

15.19.1 Numerical Comparison Operators <, <=, >, and >=

The type of each of the operands of a numerical comparison operator must be a primitive numeric type, or a compile-time error occurs. Binary numeric promotion is performed on the operands (§5.6.2). If the promoted type of the operands is `int` or `long`, then signed integer comparison is performed; if this promoted type is `float` or `double`, then floating-point comparison is performed.

Note that binary numeric promotion performs value set conversion (§5.1.8). Comparison is carried out accurately on floating-point values, no matter what value sets their representing values were drawn from.

. . .

15.20.1 Numerical Equality Operators == and `!=`

If the operands of an equality operator are both of primitive numeric type, binary numeric promotion is performed on the operands (§5.6.2). If the promoted type of the operands is `int` or `long`, then an integer equality test is performed; if the promoted type is `float` or `double`, then a floating-point equality test is performed.

Note that binary numeric promotion performs value set conversion (§5.1.8). Comparison is carried out accurately on floating-point values, no matter what value sets their representing values were drawn from.

15.24 Conditional Operator `? :`

The conditional operator `? :` uses the boolean value of one expression to decide which of two other expressions should be evaluated.

The conditional operator is syntactically right-associative (it groups right-to-left), so that `a?b:c?d:e?f:g` means the same as `a?b:(c?d:(e?f:g))`.

ConditionalExpression:
  ConditionalOrExpression
  ConditionalOrExpression `?` *Expression* `:` *ConditionalExpression*

The conditional operator has three operand expressions; `?` appears between the first and second expressions, and `:` appears between the second and third expressions.

The first expression must be of type `boolean`, or a compile-time error occurs.

The conditional operator may be used to choose between second and third operands of numeric type, or second and third operands of type `boolean`, or second and third operands that are each of either reference type or the null type. All other cases result in a compile-time error.

Note that it is not permitted for either the second or the third operand expression to be an invocation of a `void` method. In fact, it is not permitted for a conditional expression to appear in any context where an invocation of a `void` method could appear (§14.7).

The type of a conditional expression is determined as follows:

- If the second and third operands have the same type (which may be the null type), then that is the type of the conditional expression.
- Otherwise, if the second and third operands have numeric type, then there are several cases:
  - If one of the operands is of type `byte` and the other is of type `short`, then the type of the conditional expression is `short`.
  - If one of the operands is of type *T* where *T* is `byte`, `short`, or `char`, and the other operand is a constant expression of type `int` whose value is representable in type *T*, then the type of the conditional expression is *T*.
  - Otherwise, binary numeric promotion (§5.6.2) is applied to the operand types, and the type of the conditional expression is the promoted type of the second and third operands. Note that binary numeric promotion performs value set conversion (§5.1.8).

- If one of the second and third operands is of the null type and the type of the other is a reference type, then the type of the conditional expression is that reference type.
- If the second and third operands are of different reference types, then it must be possible to convert one of the types to the other type (call this latter type *T*) by assignment conversion (§5.2); the type of the conditional expression is *T*. It is a compile-time error if neither type is assignment compatible with the other type.

. . .

15.25.1 Simple Assignment Operator =

A compile-time error occurs if the type of the right-hand operand cannot be converted to the type of the variable by assignment conversion (§5.2).

At run time, the expression is evaluated in one of two ways. If the left-hand operand expression is not an array access expression, then three steps are required:

- First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.
- Otherwise, the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the value of the right-hand operand is converted to the type of the left-hand variable, is subjected to value set conversion (§5.1.8) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the variable.

If the left-hand operand expression is an array access expression (§15.12), then many steps are required:

- First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the index subexpression (of the left-hand operand array access expression) and the right-hand operand are not evaluated and no assignment occurs.
- Otherwise, the index subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assign-

ment expression completes abruptly for the same reason and the right-hand operand is not evaluated and no assignment occurs.

- Otherwise, the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, if the value of the array reference subexpression is `null`, then no assignment occurs and a `NullPointerException` is thrown.
- Otherwise, the value of the array reference subexpression indeed refers to an array. If the value of the index subexpression is less than zero, or greater than or equal to the length of the array, then no assignment occurs and an `IndexOutOfBoundsException` is thrown.
- Otherwise, the value of the index subexpression is used to select a component of the array referred to by the value of the array reference subexpression. This component is a variable; call its type *SC*. Also, let *TC* be the type of the left-hand operand of the assignment operator as determined at compile time.
  - If *TC* is a primitive type, then *SC* is necessarily the same as *TC*. The value of the right-hand operand is converted to the type of the selected array component, is subjected to value set conversion (§5.1.8) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the array component.
  - If *T* is a reference type, then *SC* may not be the same as *T*, but rather a type that extends or implements *TC*. Let *RC* be the class of the object referred to by the value of the right-hand operand at run time.

    The compiler may be able to prove at compile time that the array component will be of type *TC* exactly (for example, *TC* might be `final`). But if the compiler cannot prove at compile time that the array component will be of type *TC* exactly, then a check must be performed at run time to ensure that the class *RC* is assignment compatible (§5.2) with the actual type *SC* of the array component. This check is similar to a narrowing cast (§5.4, §15.15), except that if the check fails, an `ArrayStoreException` is thrown rather than a `ClassCastException`. Therefore:
    - If class *RC* is not assignable to type *SC*, then no assignment occurs and an `ArrayStoreException` is thrown.
    - Otherwise, the reference value of the right-hand operand is stored into the selected array component.

. . .

15.25.2 Compound Assignment Operators

All compound assignment operators require both operands to be of primitive type, except for +=, which allows the right-hand operand to be of any type if the left-hand operand is of type `String`.

A compound assignment expression of the form *`E1 op= E2`* is equivalent to *`E1 = (T)((E1) op (E2))`*, where *`T`* is the type of *`E1`*, except that *`E1`* is evaluated only once. Note that the implied cast to type *`T`* may be either an identity conversion (§5.1.1) or a narrowing primitive conversion (§5.1.3). For example, the following code is correct:

```
short x = 3;
x += 4.6;
```

and results in `x` having the value `7` because it is equivalent to:

```
short x = 3;
x = (short)(x + 4.6);
```

At run time, the expression is evaluated in one of two ways. If the left-hand operand expression is not an array access expression, then four steps are required:

- First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.
- Otherwise, the value of the left-hand operand is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the saved value of the left-hand variable and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator. If this operation completes abruptly (the only possibility is an integer division by zero—see §15.16.2), then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the result of the binary operation is converted to the type of the left-hand variable, subjected to value set conversion (§5.1.8) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the variable.

If the left-hand operand expression is an array access expression (§15.12), then many steps are required:

- First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the index subexpression (of the left-hand operand array access expression) and the right-hand operand are not evaluated and no assignment occurs.
- Otherwise, the index subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and the right-hand operand is not evaluated and no assignment occurs.
- Otherwise, if the value of the array reference subexpression is `null`, then no assignment occurs and a `NullPointerException` is thrown.
- Otherwise, the value of the array reference subexpression indeed refers to an array. If the value of the index subexpression is less than zero, or greater than or equal to the length of the array, then no assignment occurs and an `IndexOutOfBoundsException` is thrown.
- Otherwise, the value of the index subexpression is used to select a component of the array referred to by the value of the array reference subexpression. The value of this component is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs. (For a simple assignment operator, the evaluation of the right-hand operand occurs before the checks of the array reference subexpression and the index subexpression, but for a compound assignment operator, the evaluation of the right-hand operand occurs after these checks.)
- Otherwise, consider the array component selected in the previous step, whose value was saved. This component is a variable; call its type *S*. Also, let *T* be the type of the left-hand operand of the assignment operator as determined at compile time.
    - If *T* is a primitive type, then *S* is necessarily the same as *T*.
        - The saved value of the array component and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator. If this operation completes abruptly (the only possibility is an integer division by zero—see §15.16.2), then the assignment expression completes abruptly for the same reason and no assignment occurs.
        - Otherwise, the result of the binary operation is converted to the type of the selected array component, subjected to value set conversion (§5.1.8) to

the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the array component.

- If *T* is a reference type, then it must be `String`. Because class `String` is a `final` class, *S* must also be `String`. Therefore the run-time check that is sometimes required for the simple assignment operator is never required for a compound assignment operator.
  - The saved value of the array component and the value of the right-hand operand are used to perform the binary operation (string concatenation) indicated by the compound assignment operator (which is necessarily +=). If this operation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
  - Otherwise, the `String` result of the binary operation is stored into the array component.

. . .

15.27 Constant Expression

ConstantExpression:
  Expression

A compile-time *constant expression* is an expression denoting a value of primitive type or a `String` that is composed using only the following:

- Literals of primitive type and literals of type `String`
- Casts to primitive types and casts to type `String`
- The unary operators +, `-`, ~, and `!` (but not ++ or `--`)
- The multiplicative operators `*`, `/`, and `%`
- The additive operators + and `-`
- The shift operators <<, >>, and >>>
- The relational operators <, <=, >, and `>=` (but not `instanceof`)
- The equality operators == and `!=`
- The bitwise and logical operators &, `^`, and `|`
- The conditional-and operator && and the conditional-or operator `||`
- The ternary conditional operator `? :`

- Simple names that refer to `final` variables whose initializers are constant expressions
- Qualified names of the form *TypeName* `.` *Identifier* that refer to `final` variables whose initializers are constant expressions

Compile-time constant expressions are used in `case` labels in `switch` statements (§14.9) and have a special significance for assignment conversion (§5.2).

A compile-time constant expression is always treated as FP-strict (§15.28), even if it occurs in a context where a non-constant expression would not be considered to be FP-strict.

. . .

15.28 FP-strict Expressions

If the type of an expression is `float` or `double`, then there is a question as to what value set (§4.2.3) the value of the expression may be drawn from. This is governed by the rules of value set conversion (§5.1.8); these rules in turn depend on whether or not the expression is *FP-strict*.

Every compile-time constant expression (§15.27) is FP-strict. If an expression is not a compile-time constant expression, then consider all the class declarations, interface declarations, and method declarations that contain the expression. If *any* such declaration bears the `strictfp` modifier, then the expression is FP-strict.

It follows that an expression is not FP-strict if and only if it is not a compile-time constant expression *and* it does not appear within any declaration that has the `strictfp` strict modifier.

Within an FP-strict expression, all intermediate values must be elements of the float value set or the double value set, implying that the results of all FP-strict expressions must be those predicted by IEEE 754 arithmetic on operands represented using single and double formats. Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results; the net effect, 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.