CHAPTER 15

Expressions

When you can measure what you are speaking about,
and express it in numbers, you know something about it;
but when you cannot measure it, when you cannot express it in numbers,
your knowledge of it is of a meager and unsatisfactory kind:
it may be the beginning of knowledge, but you have scarcely,
in your thoughts, advanced to the stage of science.
—William Thompson, Lord Kelvin

MUCH of the work in a program is done by evaluating *expressions*, either for their side effects, such as assignments to variables, or for their values, which can be used as arguments or operands in larger expressions, or to affect the execution sequence in statements, or both.

This chapter specifies the meanings of expressions and the rules for their evaluation.

15.1 Evaluation, Denotation, and Result

When an expression in a 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 (§15.12) 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 (§14.8), 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 either:

- The declaration of some (class or interface) type that is being declared: in a field initializer, in a static initializer, in a constructor declaration, in an annotation, or in the code for a method.
- An annotation of a package or of a top-level type declaration .

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.

15.3 Type of an Expression

If an expression denotes a variable or a value, then the expression has a type known at compile time. The rules for determining the type of an expression are explained separately below for each kind of expression.

The value of an expression is assignment compatible (§5.2) with the type of the expression, unless heap pollution (§4.12.2.1) occurs. Likewise the value stored in a variable is always compatible with the type of the variable, unless heap pollution occurs.

In other words, the value of an expression whose type is *`T`* is always suitable for assignment to a variable of type *`T`*.

Note that an expression whose type is a class type *`F`* that is declared `final` is guaranteed to have a value that is either a null reference or an object whose class is *`F`* itself, because `final` types have no subclasses.

15.4 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 is 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 (§15.28) 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.

If a class, interface, or method, *X*, is declared `strictfp`, then *X* and any class, interface, method, constructor, instance initializer, static initializer or variable initializer within *X* is said to be *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` 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.

15.5 Expressions and Run-Time Checks

If the type of an expression is a primitive type, then the value of the expression is of that same primitive type. But if the type of an expression is a reference type, then the class of the referenced object, or even whether the value is a reference to an object rather than `null`, is not necessarily known at compile time. There are a few places in the Java programming language where the actual class of a referenced object affects program execution in a manner that cannot be deduced from the type of the expression. They are as follows:

- Method invocation (§15.12). The particular method used for an invocation `o.m(...)` is chosen based on the methods that are part of the class or interface that is the type of `o`. For instance methods, the class of the object referenced by the run-time value of `o` participates because a subclass may override a specific method already declared in a parent class so that this overriding method is invoked. (The overriding method may or may not choose to further invoke the original overridden `m` method.)

- The `instanceof` operator (§15.20.2). An expression whose type is a reference type may be tested using `instanceof` to find out whether the class of the object referenced by the run-time value of the expression is assignment compatible (§5.2) with some other reference type.
- Casting (§5.5, §15.16). The class of the object referenced by the run-time value of the operand expression might not be compatible with the type specified by the cast. For reference types, this may require a run-time check that throws an exception if the class of the referenced object, as determined at run time, is not assignment compatible (§5.2) with the target type.
- Assignment to an array component of reference type (§10.10, §15.13, §15.26.1). The type-checking rules allow the array type `S[]` to be treated as a subtype of `T[]` if `S` is a subtype of `T`, but this requires a run-time check for assignment to an array component, similar to the check performed for a cast.
- Exception handling (§14.19). An exception is caught by a `catch` clause only if the class of the thrown exception object is an `instanceof` the type of the formal parameter of the `catch` clause.

Situations where the class of an object is not statically known may lead to run-time type errors.

In addition, there are situations where the statically known type may not be accurate at run-time. Such situations can arise in a program that gives rise to unchecked warnings. Such warnings are given in response to operations that cannot be statically guaranteed to be safe, and cannot immediately be subjected to dynamic checking because they involve non-reifiable (§4.7) types. As a result, dynamic checks later in the course of program execution may detect inconsistencies and result in run-time type errors.

A run-time type error can occur only in these situations:

- In a cast, when the actual class of the object referenced by the value of the operand expression is not compatible with the target type specified by the cast operator (§5.5, §15.16); in this case a `ClassCastException` is thrown.
- In an implicit, compiler-generated cast introduced to ensure the validity of an operation on a non-reifiable type.
- In an assignment to an array component of reference type, when the actual class of the object referenced by the value to be assigned is not compatible

with the actual run-time component type of the array (§10.10, §15.13, §15.26.1); in this case an `ArrayStoreException` is thrown.

- When an exception is not caught by any `catch` handler (§11.3); in this case the thread of control that encountered the exception first invokes the method `uncaughtException` for its thread group and then terminates.

15.6 Normal and Abrupt Completion of Evaluation

No more: the end is sudden and abrupt.
—William Wordsworth, *Apology for the Foregoing Poems* (1831)

Every expression has a normal mode of evaluation in which certain computational steps are carried out. The following sections describe the normal mode of evaluation for each kind of expression. If all the steps are carried out without an exception being thrown, the expression is said to *complete normally*.

If, however, evaluation of an expression throws an exception, then the expression is said to *complete abruptly*. An abrupt completion always has an associated *reason*, which is always a `throw` with a given value.

Run-time exceptions are thrown by the predefined operators as follows:

- A class instance creation expression (§15.9), array creation expression (§15.10), or string concatenation operator expression (§15.18.1) throws an `OutOfMemoryError` if there is insufficient memory available.
- An array creation expression throws a `NegativeArraySizeException` if the value of any dimension expression is less than zero (§15.10).
- A field access (§15.11) throws a `NullPointerException` if the value of the object reference expression is `null`.
- A method invocation expression (§15.12) that invokes an instance method throws a `NullPointerException` if the target reference is `null`.
- An array access (§15.13) throws a `NullPointerException` if the value of the array reference expression is `null`.
- An array access (§15.13) throws an `ArrayIndexOutOfBoundsException` if the value of the array index expression is negative or greater than or equal to the `length` of the array.
- A cast (§15.16) throws a `ClassCastException` if a cast is found to be impermissible at run time.

- An integer division (§15.17.2) or integer remainder (§15.17.3) operator throws an `ArithmeticException` if the value of the right-hand operand expression is zero.
- An assignment to an array component of reference type (§15.26.1) throws an `ArrayStoreException` when the value to be assigned is not compatible with the component type of the array.

A method invocation expression can also result in an exception being thrown if an exception occurs that causes execution of the method body to complete abruptly. A class instance creation expression can also result in an exception being thrown if an exception occurs that causes execution of the constructor to complete abruptly. Various linkage and virtual machine errors may also occur during the evaluation of an expression. By their nature, such errors are difficult to predict and difficult to handle.

If an exception occurs, then evaluation of one or more expressions may be terminated before all steps of their normal mode of evaluation are complete; such expressions are said to complete abruptly. The terms "complete normally" and "complete abruptly" are also applied to the execution of statements (§14.1). A statement may complete abruptly for a variety of reasons, not just because an exception is thrown.

If evaluation of an expression requires evaluation of a subexpression, abrupt completion of the subexpression always causes the immediate abrupt completion of the expression itself, with the same reason, and all succeeding steps in the normal mode of evaluation are not performed.

15.7 Evaluation Order

Let all things be done decently and in order.
—I Corinthians 14:40

The Java programming language guarantees that the operands of operators appear to be evaluated in a specific *evaluation order*, namely, from left to right.

DISCUSSION

It is recommended that code not rely crucially on this specification. Code is usually clearer when each expression contains at most one side effect, as its outermost operation, and when code does not depend on exactly which exception arises as a consequence of the left-to-right evaluation of expressions.

15.7.1 Evaluate Left-Hand Operand First

The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated. For example, if the left-hand operand contains an assignment to a variable and the right-hand operand contains a reference to that same variable, then the value produced by the reference will reflect the fact that the assignment occurred first.

Discussion

Thus:

```
class Test {
    public static void main(String[] args) {
        int i = 2;
        int j = (i=3) * i;
        System.out.println(j);
    }
}
```

prints:

```
9
```

It is not permitted for it to print 6 instead of 9.

If the operator is a compound-assignment operator (§15.26.2), then evaluation of the left-hand operand includes both remembering the variable that the left-hand operand denotes and fetching and saving that variable's value for use in the implied combining operation. So, for example, the test program:

```
class Test {
    public static void main(String[] args) {
        int a = 9;
        a += (a = 3);                // first example
        System.out.println(a);
        int b = 9;
        b = b + (b = 3);             // second example
        System.out.println(b);
    }
}
```

prints:

```
12
12
```

because the two assignment statements both fetch and remember the value of the left-hand operand, which is `9`, before the right-hand operand of the addition is evaluated, thereby setting the variable to `3`. It is not permitted for either example to produce the result `6`. Note that both of these examples have unspecified behavior in C, according to the ANSI/ISO standard.

If evaluation of the left-hand operand of a binary operator completes abruptly, no part of the right-hand operand appears to have been evaluated.

Discussion

Thus, the test program:

```
class Test {
    public static void main(String[] args) {
        int j = 1;
        try {
            int i = forgetIt() / (j = 2);
        } catch (Exception e) {
            System.out.println(e);
            System.out.println("Now j = " + j);
        }
    }
    static int forgetIt() throws Exception {
        throw new Exception("I’m outta here!");
    }
}
```

prints:

```
java.lang.Exception: I'm outta here!
Now j = 1
```

That is, the left-hand operand `forgetIt()` of the operator `/` throws an exception before the right-hand operand is evaluated and its embedded assignment of `2` to `j` occurs.

15.7.2 Evaluate Operands before Operation

The Java programming language also guarantees that every operand of an operator (except the conditional operators `&&`, `||`, and `? :`) appears to be fully evaluated before any part of the operation itself is performed.

If the binary operator is an integer division `/` (§15.17.2) or integer remainder `%` (§15.17.3), then its execution may raise an `ArithmeticException`, but this exception is thrown only after both operands of the binary operator have been evaluated and only if these evaluations completed normally.

DISCUSSION

So, for example, the program:

```
class Test {
    public static void main(String[] args) {
        int divisor = 0;
        try {
            int i = 1 / (divisor * loseBig());
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    static int loseBig() throws Exception {
        throw new Exception("Shuffle off to Buffalo!");
    }

}
```

always prints:

```
java.lang.Exception: Shuffle off to Buffalo!
```

and not:

```
java.lang.ArithmeticException: / by zero
```

since no part of the division operation, including signaling of a divide-by-zero exception, may appear to occur before the invocation of `loseBig` completes, even though the implementation may be able to detect or infer that the division operation would certainly result in a divide-by-zero exception.

15.7.3 Evaluation Respects Parentheses and Precedence

Java programming language implementations must respect the order of evaluation as indicated explicitly by parentheses and implicitly by operator precedence. An implementation may not take advantage of algebraic identities such as the associative law to rewrite expressions into a more convenient computational order unless it can be proven that the replacement expression is equivalent in value and in its observable side effects, even in the presence of multiple threads of execution (using the thread execution model in §17), for all possible computational values that might be involved.

In the case of floating-point calculations, this rule applies also for infinity and not-a-number (NaN) values. For example, `!(x<y)` may not be rewritten as `x>=y`, because these expressions have different values if either `x` or `y` is NaN or both are NaN.

Specifically, floating-point calculations that appear to be mathematically associative are unlikely to be computationally associative. Such computations must not be naively reordered.

DISCUSSION

For example, it is not correct for a Java compiler to rewrite `4.0*x*0.5` as `2.0*x`; while roundoff happens not to be an issue here, there are large values of `x` for which the first expression produces infinity (because of overflow) but the second expression produces a finite result.

So, for example, the test program:

```
strictfp class Test {
    public static void main(String[] args) {
        double d = 8e+307;
        System.out.println(4.0 * d * 0.5);
        System.out.println(2.0 * d);
    }
}
```

prints:

```
Infinity
1.6e+308
```

because the first expression overflows and the second does not.

In contrast, integer addition and multiplication *are* provably associative in the Java programming language.

For example `a+b+c`, where `a`, `b`, and `c` are local variables (this simplifying assumption avoids issues involving multiple threads and `volatile` variables), will always produce the same answer whether evaluated as `(a+b)+c` or `a+(b+c)`; if the expression `b+c` occurs nearby in the code, a smart compiler may be able to use this common subexpression.

15.7.4 Argument Lists are Evaluated Left-to-Right

In a method or constructor invocation or class instance creation expression, argument expressions may appear within the parentheses, separated by commas. Each argument expression appears to be fully evaluated before any part of any argument expression to its right.

DISCUSSION

Thus:

```
class Test {
    public static void main(String[] args) {
        String s = "going, ";
        print3(s, s, s = "gone");
    }

    static void print3(String a, String b, String c) {
        System.out.println(a + b + c);
    }

}
```

always prints:

```
going, going, gone
```

because the assignment of the string `"gone"` to `s` occurs after the first two arguments to `print3` have been evaluated.

If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated.

DISCUSSION

Thus, the example:

```
class Test {
            static int id;
    public static void main(String[] args) {
        try {
            test(id = 1, oops(), id = 3);
        } catch (Exception e) {
            System.out.println(e + ", id=" + id);
        }
    }

    static int oops() throws Exception {
        throw new Exception("oops");
    }

    static int test(int a, int b, int c) {
        return a + b + c;
    }

}
```

prints:

```
java.lang.Exception: oops, id=1
```

because the assignment of `3` to `id` is not executed.

15.7.5 Evaluation Order for Other Expressions

The order of evaluation for some expressions is not completely covered by these general rules, because these expressions may raise exceptional conditions at times that must be specified. See, specifically, the detailed explanations of evaluation order for the following kinds of expressions:

- class instance creation expressions (§15.9.4)
- array creation expressions (§15.10.1)
- method invocation expressions (§15.12.4)
- array access expressions (§15.13.1)
- assignments involving array components (§15.26)

15.8 Primary Expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, class literals, field accesses, method invocations, and array accesses. A parenthesized expression is also treated syntactically as a primary expression.

Primary:
  PrimaryNoNewArray
  ArrayCreationExpression

PrimaryNoNewArray:
  Literal
  Type `.` *class*
  `void` `.` *class*
  `this`
  ClassName`.`*`this`*
  `(` *Expression* `)`
  ClassInstanceCreationExpression
  FieldAccess
  MethodInvocation
  ArrayAccess

15.8.1 Lexical 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 float-

ing-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 lexical literal always completes normally.

15.8.2 Class Literals

A *class literal* is an expression consisting of the name of a class, interface, array, or primitive type, or the pseudo-type `void`, followed by a '.' and the token `class`. The type of a class literal, `C.Class`, where `C` is the name of a class, interface or array type, is `Class<C>.` If `p` is the name of a primitive type, let `B` be the type of an expression of type `p` after boxing conversion (§5.1.7). Then the type of `p.class` is `Class<B>`. The type of `void.class` is `Class<Void>`.

A class literal evaluates to the `Class` object for the named type (or for void) as defined by the defining class loader of the class of the current instance.

It is a compile time error if any of the following occur:

- The named type is a type variable (§4.4) or a parameterized type (§4.5) or an array whose element type is a type variable or parameterized type.
- The named type does not denote a type that is accessible and in scope at the point where the class literal appears.

The method `Object.getClass()` must be treated specially by a Java compiler. The type of a method invocation `e.getClass()`, where the expression `e` has the static type `T`, is defined to be `Class<? extends |T|>`, where `|T|` is the erasure (§4.6) of `T`.

15.8.3 `this`

The keyword `this` may be used only in the body of an instance method, instance initializer or constructor, or in the initializer of an instance variable of a class. If it appears anywhere else, a compile-time error occurs.

When used as a primary expression, the keyword `this` denotes a value that is a reference to the object for which the instance method was invoked (§15.12), or to the object being constructed. The type of `this` is the class *C* within which the keyword `this` occurs. At run time, the class of the actual object referred to may be the class *C* or any subclass of *C*.

DISCUSSION

In the example:

```
class IntVector {
              int[] v;
    boolean equals(IntVector other) {
        if (this == other)
            return true;
        if (v.length != other.v.length)
            return false;
        for (int i = 0; i < v.length; i++)
            if (v[i] != other.v[i])
              return false;
        return true;
    }
}
```

the class `IntVector` implements a method `equals`, which compares two vectors. If the `other` vector is the same vector object as the one for which the `equals` method was invoked, then the check can skip the length and value comparisons. The `equals` method implements this check by comparing the reference to the `other` object to `this`.

The keyword `this` is also used in a special explicit constructor invocation statement, which can appear at the beginning of a constructor body (§8.8.5).

15.8.4 Qualified `this`

Any lexically enclosing instance can be referred to by explicitly qualifying the keyword `this`.

Let *C* be the class denoted by *ClassName*. Let *n* be an integer such that *C* is the *n*th lexically enclosing class of the class in which the qualified `this` expression appears. The value of an expression of the form *ClassName*.`this` is the *n*th lexically enclosing instance of `this` (§8.1.2). The type of the expression is *C*. It is a compile-time error if the current class is not an inner class of class *C* or *C* itself.

15.8.5 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. If the contained expression denotes a variable then the parenthesized expression also denotes that variable.

The use of parentheses only effects the order of evaluation. In particular, the presence or absence of parentheses around an expression does not affect in any way:

- the choice of value set (§4.2.3) for the value of an expression of type `float` or `double`.
- whether a variable is definitely assigned, defintely assigned when true, definitely assigned when false, definitely unassigned, definitely unassigned when true, or definitely unassigned when false (§16).

15.9 Class Instance Creation Expressions

And now a new object took possession of my soul.
—Edgar Allen Poe, *A Tale of the Ragged Mountains* (1844)

A class instance creation expression is used to create new objects that are instances of classes.

ClassInstanceCreationExpression:
  `new` *NonWildTypeArguments*$_{opt}$ *ClassOrInterfaceType* *NonWildTypeArguments*$_{opt}$ `(` *ArgumentList*$_{opt}$ `)` *ClassBody*$_{opt}$
  Primary. `new` *NonWildTypeArguments*$_{opt}$ *Identifier* *NonWildTypeArguments*$_{opt}$ `(` *ArgumentList*$_{opt}$ `)` *ClassBody*$_{opt}$

ArgumentList:
  Expression
  ArgumentList `,` *Expression*

It is a compile-time error if any of the optional type arguments immediately preceding the argument list include any wildcard type arguments (§4.5.1). Class instance creation expressions have two forms:

- *Unqualified class instance creation expressions* begin with the keyword `new`. An unqualified class instance creation expression may be used to create an instance of a class, regardless of whether the class is a top-level (§7.6), member (§8.5, §9.5), local (§14.3) or anonymous class (§15.9.5).

- *Qualified class instance creation expressions* begin with a *Primary*. A qualified class instance creation expression enables the creation of instances of inner member classes and their anonymous subclasses.

A class instance creation expression can throw an exception type *E* iff either:

- The expression is a qualified class instance creation expression and the qualifying expression can throw *E*; or
- Some expression of the argument list can throw *E*; or
- *E* is listed in the throws clause of the type of the constructor that is invoked; or
- The class instance creation expression includes a *ClassBody*, and some instnance initializer block or instance variable initializer expression in the *ClassBody* can throw *E*.

Both unqualified and qualified class instance creation expressions may *optionally end with a class body. Such a class instance creation expression* declares an *anonymous class* (§15.9.5) and creates an instance of it.

We say that a class is *instantiated* when an instance of the class is created by a class instance creation expression. Class instantiation involves determining what class is to be instantiated, what the enclosing instances (if any) of the newly created instance are, what constructor should be invoked to create the new instance and what arguments should be passed to that constructor.

15.9.1 Determining the Class being Instantiated

If the class instance creation expression ends in a class body, then the class being instantiated is an anonymous class. Then:

- If the class instance creation expression is an unqualified class instance creation expression, then *let* `T` be the *ClassOrInterfaceType* after the `new` token. It is a compile-time error if the class or interface named by `T` is not accessible (§6.6) or if `T` is an enum type (§8.9). If `T` is the name of a class, then an anonymous direct subclass of the class named by `T` is declared. It is a compile-time error if the class named by `T` is a `final` class. If `T` is the name of an interface then an anonymous direct subclass of `Object` that implements the interface named by `T` is *declared. In either case, the body of the subclass is the*

ClassBody given in the class instance creation expression. The class being instantiated is the anonymous subclass.

- Otherwise, the class instance creation expression is a qualified class instance creation expression. Let `T` be the name of the *Identifier* after the `new` token. It is a compile-time error if `T` is not the simple name (§6.2) of an accessible (§6.6) non-`final` inner class (§8.1.2) that is a member of the compile-time type of the *Primary*. It is also a compile-time error if `T` is ambiguous (§8.5) or if `T` denotes an enum type. An anonymous direct subclass of the class named by `T` is declared. The body of the subclass is the *ClassBody* given in the class instance creation expression. The class being instantiated is the anonymous subclass.

If a class instance creation expression does not declare an anonymous class, then:

- If the class instance creation expression is an unqualified class instance creation expression, then the *ClassOrInterfaceType* must name a class that is accessible (§6.6) and is not an enum type and not `abstract`, or a compile-time error occurs. In this case, the class being instantiated is the class denoted by *ClassOrInterfaceType*.
- Otherwise, the class instance creation expression is a qualified class instance creation expression. It is a compile-time error if *Identifier* is not the simple name (§6.2) of an accessible (§6.6) non-`abstract` inner class (§8.1.2) `T` that is a member of the compile-time type of the *Primary*. It is also a compile-time error if *Identifier* is ambiguous (§8.5), or if *Identifier* denotes an enum type (§8.9). The class being instantiated is the class denoted by *Identifier*.

The type of the class instance creation expression is the class type being instantiated.

15.9.2 Determining Enclosing Instances

Let `C` be the class being instantiated, and let `i` the instance being created. If `C` is an inner class then `i` may have an immediately enclosing instance. The immediately enclosing instance of `i` (§8.1.2) is determined as follows:

- If `C` is an anonymous class, then:
  - If the class instance creation expression occurs in a static context (§8.1.2), then `i` has no immediately enclosing instance.
  - Otherwise, the immediately enclosing instance of `i` is `this`.

- If `C` is a local class (§14.3), then let `O` be the innermost lexically enclosing class of `C`. Let *n* be an integer such that `O` is the *n*th lexically enclosing class of the class in which the class instance creation expression appears. Then:
  - If `C` occurs in a static context, then `i` has no immediately enclosing instance.
  - Otherwise, if the class instance creation expression occurs in a static context, then a compile-time error occurs.
  - Otherwise, the immediately enclosing instance of `i` is the *n*th lexically enclosing instance of `this` (§8.1.2).
- Otherwise, `C` is an inner member class (§8.5).
  - If the class instance creation expression is an unqualified class instance creation expression, then:
    - If the class instance creation expression occurs in a static context, then a compile-time error occurs.
    - Otherwise, if `C` is a member of an enclosing class then let `O` be the innermost lexically enclosing class of which `C` is a member, and let *n* be an integer such that `O` is the *n*th lexically enclosing class of the class in which the class instance creation expression appears. The immediately enclosing instance of `i` is the *n*th lexically enclosing instance of `this`.
    - Otherwise, a compile-time error occurs.
  - Otherwise, the class instance creation expression is a qualified class instance creation expression. The immediately enclosing instance of `i` is the object that is the value of the *Primary* expression.

In addition, if `C` is an anonymous class, and the direct superclass of `C`, `S`, is an inner class then `i` may have an immediately enclosing instance with respect to `S` which is determined as follows:

- If `S` is a local class (§14.3), then let `O` be the innermost lexically enclosing class of `S`. Let *n* be an integer such that `O` is the *n*th lexically enclosing class of the class in which the class instance creation expression appears. Then:
  - If `S` occurs within a static context, then `i` has no immediately enclosing instance with respect to `S`.
  - Otherwise, if the class instance creation expression occurs in a static context, then a compile-time error occurs.
  - Otherwise, the immediately enclosing instance of `i` with respect to `S` is the *n*th lexically enclosing instance of `this`.

- Otherwise, *S* is an inner member class (§8.5).
  - If the class instance creation expression is an unqualified class instance creation expression, then:
    - If the class instance creation expression occurs in a static context, then a compile-time error occurs.
    - Otherwise, if *S* is a member of an enclosing class then let *O* be the innermost lexically enclosing class of which *S* is a member, and let *n* be an integer such that *O* is the *n*th lexically enclosing class of the class in which the class instance creation expression appears. The immediately enclosing instance of *i* with respect to *S* is the *n*th lexically enclosing instance of `this`.
    - Otherwise, a compile-time error occurs.
  - Otherwise, the class instance creation expression is a qualified class instance creation expression. The immediately enclosing instance of *i* with respect to *S* is the object that is the value of the *Primary* expression.

15.9.3 Choosing the Constructor and its Arguments

Let *C* be the class type being instantiated. To create an instance of *C*, *i*, a constructor of *C* is chosen at compile-time by the following rules:

- First, the actual arguments to the constructor invocation are determined.
  - If *C* is an anonymous class, and the direct superclass of *C*, *S*, is an inner class, then:
    - If the *S* is a local class and *S* occurs in a static context, then the arguments in the argument list, if any, are the arguments to the constructor, in the order they appear in the expression.
    - Otherwise, the immediately enclosing instance of *i* with respect to *S* is the first argument to the constructor, followed by the arguments in the argument list of the class instance creation expression, if any, in the order they appear in the expression.
  - Otherwise the arguments in the argument list, if any, are the arguments to the constructor, in the order they appear in the expression.
- Once the actual arguments have been determined, they are used to select a constructor of *C*, using the same rules as for method invocations (§15.12). As

in method invocations, a compile-time method matching error results if there is no unique most-specific constructor that is both applicable and accessible.

Note that the type of the class instance creation expression may be an anonymous class type, in which case the constructor being invoked is an anonymous constructor.

15.9.4 Run-time Evaluation of Class Instance Creation Expressions

At run time, evaluation of a class instance creation expression is as follows.

First, if the class instance creation expression is a qualified class instance creation expression, the qualifying primary expression is evaluated. If the qualifying expression evaluates to `null`, a `NullPointerException` is raised, and the class instance creation expression completes abruptly. If the qualifying expression completes abruptly, the class instance creation expression completes abruptly for the same reason.

Next, space is allocated for the new class instance. If there is insufficient space to allocate the object, evaluation of the class instance creation expression completes abruptly by throwing an `OutOfMemoryError` (§15.9.6).

The new object contains new instances of all the fields declared in the specified class type and all its superclasses. As each new field instance is created, it is initialized to its default value (§4.5.5).

Next, the actual arguments to the constructor are evaluated, left-to-right. If any of the argument evaluations completes abruptly, any argument expressions to its right are not evaluated, and the class instance creation expression completes abruptly for the same reason.

Next, the selected constructor of the specified class type is invoked. This results in invoking at least one constructor for each superclass of the class type. This process can be directed by explicit constructor invocation statements (§8.8) and is described in detail in §12.5.

The value of a class instance creation expression is a reference to the newly created object of the specified class. Every time the expression is evaluated, a fresh object is created.

15.9.5 Anonymous Class Declarations

An anonymous class declaration is automatically derived from a class instance creation expression by the compiler.

An anonymous class is never `abstract` (§8.1.1.1). An anonymous class is always an inner class (§8.1.2); it is never `static` (§8.1.1, §8.5.2). An anonymous class is always implicitly `final` (§8.1.1.2).

15.9.5.1 *Anonymous Constructors*

An anonymous class cannot have an explicitly declared constructor. Instead, the compiler must automatically provide an *anonymous constructor* for the anonymous class. The form of the anonymous constructor of an anonymous class `C` with direct superclass `S` is as follows:

- If `S` is not an inner class, or if `S` is a local class that occurs in a static context, then the anonymous constructor has one formal parameter for each actual argument to the class instance creation expression in which `C` is declared. The actual arguments to the class instance creation expression are used to determine a constructor `cs` of `S`, using the same rules as for method invocations (§15.12). The type of each formal parameter of the anonymous constructor must be identical to the corresponding formal parameter of `cs`.

  The body of the constructor consists of an explicit constructor invocation (§8.8.5.1) of the form `super`(...), where the actual arguments are the formal parameters of the constructor, in the order they were declared.
- Otherwise, the first formal parameter of the constructor of `C` represents the value of the immediately enclosing instance of `i` with respect to `S`. The type of this parameter is the class type that immediately encloses the declaration of `S`. The constructor has an additional formal parameter for each actual argument to the class instance creation expression that declared the anonymous class. The *n*th formal parameter e corresponds to the $n-1$ st actual argument. The actual arguments to the class instance creation expression are used to determine a constructor `cs` of `S`, using the same rules as for method invocations (§15.12). The type of each formal parameter of the anonymous constructor must be identical to the corresponding formal parameter of `cs`. The body of the constructor consists of an explicit constructor invocation (§8.8.5.1) of the form `o.super`(...), where `o` is the first formal parameter of the constructor, and the actual arguments are the subsequent formal parameters of the constructor, in the order they were declared.

In all cases, the `throws` clause of an anonymous constructor must list all the checked exceptions thrown by the explicit superclass constructor invocation statement contained within the anonymous constructor, and all checked exceptions thrown by any instance initializers or instance variable initializers of the anonymous class.

Discussion

Note that it is possible for the signature of the anonymous constructor to refer to an inaccessible type (for example, if such a type occurred in the signature of the superclass constructor *cs*). This does not, in itself, cause any errors at either compile time or run time.

15.9.6 Example: Evaluation Order and Out-of-Memory Detection

If evaluation of a class instance creation expression finds there is insufficient memory to perform the creation operation, then an `OutOfMemoryError` is thrown. This check occurs before any argument expressions are evaluated.

So, for example, the test program:

```
class List {
    int value;
    List next;
    static List head = new List(0);
    List(int n) { value = n; next = head; head = this; }
}

class Test {
    public static void main(String[] args) {
        int id = 0, oldid = 0;
        try {
            for (;;) {
              ++id;
              new List(oldid = id);
            }
        } catch (Error e) {
            System.out.println(e + ", " + (oldid==id));
        }
    }
}
```

prints:

```
java.lang.OutOfMemoryError: List, false
```

because the out-of-memory condition is detected before the argument expression `oldid = id` is evaluated.

Compare this to the treatment of array creation expressions (§15.10), for which the out-of-memory condition is detected after evaluation of the dimension expressions (§15.10.3).

15.10 Array Creation Expressions

This was all as it should be, and I went out in my new array . . .
—Charles Dickens, *Great Expectations* (1861)

An array instance creation expression is used to create new arrays (§10).

ArrayCreationExpression:
    `new` *PrimitiveType DimExprs* $Dims_{opt}$
    `new` *TypeName DimExprs* $Dims_{opt}$
    `new` *PrimitiveType Dims ArrayInitializer*
    `new` *TypeName Dims ArrayInitializer*

DimExprs:
    DimExpr
    DimExprs DimExpr

DimExpr:
    `[` *Expression* `]`

Dims:
    `[ ]`
    Dims `[ ]`

An array creation expression creates an object that is a new array whose elements are of the type specified by the *PrimitiveType* or *TypeName*. It is a compile-time error if the *TypeName* does not denote a reifiable type (§4.7). Otherwise, the *TypeName* may name any named reference type, even an `abstract` class type (§8.1.1.1) or an interface type (§9).

DISCUSSION

Note that the syntax ensures that the element type in an array creation expression cannot be a parameterized type, other than an unbounded wildcard.

The type of the creation expression is an array type that can denoted by a copy of the creation expression from which the `new` keyword and every *DimExpr* expression and array initializer have been deleted.

DISCUSSION

For example, the type of the creation expression:

```
new double[3][3][]
```

is:

```
double[][][]
```

The type of each dimension expression within a *DimExpr* must be a type that is convertible (§5.1.8) to an integral type, or a compile-time error occurs. Each expression undergoes unary numeric promotion (§5.6.1). The promoted type must be `int`, or a compile-time error occurs; this means, specifically, that the type of a dimension expression must not be `long`.

If an array initializer is provided, the newly allocated array will be initialized with the values provided by the array initializer as described in §10.6.

15.10.1 Run-time Evaluation of Array Creation Expressions

At run time, evaluation of an array creation expression behaves as follows. If there are no dimension expressions, then there must be an array initializer. The value of the array initializer is the value of the array creation expression. Otherwise:

First, the dimension expressions are evaluated, left-to-right. If any of the expression evaluations completes abruptly, the expressions to the right of it are not evaluated.

Next, the values of the dimension expressions are checked. If the value of any *DimExpr* expression is less than zero, then an `NegativeArraySizeException` is thrown.

Next, space is allocated for the new array. If there is insufficient space to allocate the array, evaluation of the array creation expression completes abruptly by throwing an `OutOfMemoryError`.

Then, if a single *DimExpr* appears, a single-dimensional array is created of the specified length, and each component of the array is initialized to its default value (§4.5.5).

If an array creation expression contains *N* *DimExpr* expressions, then it effectively executes a set of nested loops of depth $N - 1$ to create the implied arrays of arrays.

DISCUSSION

For example, the declaration:

```
float[][] matrix = new float[3][3];
```

is equivalent in behavior to:

```
float[][] matrix = new float[3][];
for (int d = 0; d < matrix.length; d++)
    matrix[d] = new float[3];
```

and:

```
Age[][][][][] Aquarius = new Age[6][10][8][12][];
```

is equivalent to:

```
Age[][][][][] Aquarius = new Age[6][][][][];
for (int d1 = 0; d1 < Aquarius.length; d1++) {
    Aquarius[d1] = new Age[10][][][];
    for (int d2 = 0; d2 < Aquarius[d1].length; d2++) {
        Aquarius[d1][d2] = new Age[8][][];
        for (int d3 = 0; d3 < Aquarius[d1][d2].length; d3++) {
            Aquarius[d1][d2][d3] = new Age[12][];
        }
    }
}
```

with *d*, *d1*, *d2* and *d3* replaced by names that are not already locally declared. Thus, a single `new` expression actually creates one array of length 6, 6 arrays of length 10, $6 \times 10 = 60$ arrays of length 8, and $6 \times 10 \times 8 = 480$ arrays of length 12. This example leaves the fifth dimension, which would be arrays containing the actual array elements (references to `Age` objects), initialized only to null references. These arrays can be filled in later by other code, such as:

```
Age[] Hair = { new Age("quartz"), new Age("topaz") };
Aquarius[1][9][6][9] = Hair;
```

A multidimensional array need not have arrays of the same length at each level.

Thus, a triangular matrix may be created by:

```
float triang[][] = new float[100][];
for (int i = 0; i < triang.length; i++)
    triang[i] = new float[i+1];
```

15.10.2 Example: Array Creation Evaluation Order

In an array creation expression (§15.10), there may be one or more dimension expressions, each within brackets. Each dimension expression is fully evaluated before any part of any dimension expression to its right.

Thus:

```
class Test {
    public static void main(String[] args) {
        int i = 4;
        int ia[][] = new int[i][i=3];
        System.out.println(
            "[" + ia.length + "," + ia[0].length + "]");
    }
}
```

prints:

```
[4,3]
```

because the first dimension is calculated as `4` before the second dimension expression sets `i` to `3`.

If evaluation of a dimension expression completes abruptly, no part of any dimension expression to its right will appear to have been evaluated. Thus, the example:

```
class Test {
    public static void main(String[] args) {
        int[][] a = { { 00, 01 }, { 10, 11 } };
        int i = 99;
        try {
            a[val()][i = 1]++;
        } catch (Exception e) {
            System.out.println(e + ", i=" + i);
        }
    }

    static int val() throws Exception {
        throw new Exception("unimplemented");
    }

}
```

prints:

```
java.lang.Exception: unimplemented, i=99
```

because the embedded assignment that sets `i` to `1` is never executed.

15.10.3 Example: Array Creation and Out-of-Memory Detection

If evaluation of an array creation expression finds there is insufficient memory to perform the creation operation, then an `OutOfMemoryError` is thrown. This check occurs only after evaluation of all dimension expressions has completed normally.

So, for example, the test program:

```
class Test {
    public static void main(String[] args) {
        int len = 0, oldlen = 0;
        Object[] a = new Object[0];
        try {
            for (;;) {
              ++len;
              Object[] temp = new Object[oldlen = len];
              temp[0] = a;
              a = temp;
            }
        } catch (Error e) {
            System.out.println(e + ", " + (oldlen==len));
        }
    }
}
```

prints:

```
java.lang.OutOfMemoryError, true
```

because the out-of-memory condition is detected after the dimension expression `oldlen` = `len` is evaluated.

Compare this to class instance creation expressions (§15.9), which detect the out-of-memory condition before evaluating argument expressions (§15.9.6).

15.11 Field Access Expressions

A field access expression may access a field of an object or array, a reference to which is the value of either an expression or the special keyword `super`. (It is also possible to refer to a field of the current instance or current class by using a simple name; see §6.5.6.)

FieldAccess:
  Primary `.` *Identifier*
  `super .` *Identifier*
  ClassName `.super .` *Identifier*

The meaning of a field access expression is determined using the same rules as for qualified names (§6.6), but limited by the fact that an expression cannot denote a package, class type, or interface type.

15.11.1 Field Access Using a Primary

The type of the *Primary* must be a reference type *T*, or a compile-time error occurs. The meaning of the field access expression is determined as follows:

- If the identifier names several accessible member fields of type *T*, then the field access is ambiguous and a compile-time error occurs.
- If the identifier does not name an accessible member field of type *T*, then the field access is undefined and a compile-time error occurs.
- Otherwise, the identifier names a single accessible member field of type *T* and the type of the field access expression is the type of the member field after capture conversion (§5.1.10). At run time, the result of the field access expression is computed as follows:
  - If the field is `static`:
    - The *Primary* expression is evaluated, and the result is discarded. If evaluation of the *Primary* expression completes abruptly, the field access expression completes abruptly for the same reason.
    - If the field is `final`, then the result is the value of the specified class variable in the class or interface that is the type of the *Primary* expression.
    - If the field is not `final`, then the result is a variable, namely, the specified class variable in the class that is the type of the *Primary* expression.
  - If the field is not `static`:
    - The *Primary* expression is evaluated. If evaluation of the *Primary* expression completes abruptly, the field access expression completes abruptly for the same reason.
    - If the value of the *Primary* is `null`, then a `NullPointerException` is thrown.
    - If the field is `final`, then the result is the value of the specified instance variable in the object referenced by the value of the *Primary*.
    - If the field is not `final`, then the result is a variable, namely, the specified instance variable in the object referenced by the value of the *Primary*.

Discussion

Note, specifically, that only the type of the *Primary* expression, not the class of the actual object referred to at run time, is used in determining which field to use.

Thus, the example:

```
class S { int x = 0; }
class T extends S { int x = 1; }
class Test {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.x=" + t.x + when("t", t));
        S s = new S();
        System.out.println("s.x=" + s.x + when("s", s));
        s = t;
        System.out.println("s.x=" + s.x + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a "
            + t.getClass() + " at run time.";
    }
}
```

produces the output:

```
t.x=1 when t holds a class T at run time.
s.x=0 when s holds a class S at run time.
s.x=0 when s holds a class T at run time.
```

The last line shows that, indeed, the field that is accessed does not depend on the run-time class of the referenced object; even if `s` holds a reference to an object of class `T`, the expression `s.x` refers to the `x` field of class `S`, because the type of the expression `s` is `S`. Objects of class `T` contain two fields named `x`, one for class `T` and one for its superclass `S`.

This lack of dynamic lookup for field accesses allows programs to be run efficiently with straightforward implementations. The power of late binding and overriding is available, but only when instance methods are used. Consider the same example using instance methods to access the fields:

```
class S { int x = 0; int z() { return x; } }
class T extends S { int x = 1; int z() { return x; } }
class Test {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.z()=" + t.z() + when("t", t));
        S s = new S();
        System.out.println("s.z()=" + s.z() + when("s", s));
        s = t;
        System.out.println("s.z()=" + s.z() + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a "
            + t.getClass() + " at run time.";
    }
}
```

Now the output is:

```
t.z()=1 when t holds a class T at run time.
s.z()=0 when s holds a class S at run time.
s.z()=1 when s holds a class T at run time.
```

The last line shows that, indeed, the method that is accessed *does* depend on the run-time class of referenced object; when `s` holds a reference to an object of class `T`, the expression `s.z()` refers to the `z` method of class `T`, despite the fact that the type of the expression `s` is `S`. Method `z` of class `T` overrides method `z` of class `S`.

The following example demonstrates that a null reference may be used to access a class (`static`) variable without causing an exception:

```
class Test {
            static String mountain = "Chocorua";
    static Test favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        System.out.println(favorite().mountain);
    }
}
```

It compiles, executes, and prints:

```
Mount Chocorua
```

Even though the result of `favorite()` is `null`, a `NullPointerException` is *not* thrown. That “`Mount `” is printed demonstrates that the *Primary* expression is indeed fully evaluated at run time, despite the fact that only its type, not its value, is used to determine which field to access (because the field `mountain` is `static`).

15.11.2 Accessing Superclass Members using `super`

The special forms using the keyword `super` are valid only in an instance method, instance initializer or constructor, or in the initializer of an instance variable of a class; these are exactly the same situations in which the keyword `this` may be used (§15.8.3). The forms involving `super` may not be used anywhere in the class `Object`, since `Object` has no superclass; if `super` appears in class `Object`, then a compile-time error results.

Suppose that a field access expression `super.`*`name`* appears within class *`C`*, and the immediate superclass of *`C`* is class *`S`*. Then `super.`*`name`* is treated exactly as if it had been the expression `((`*`S`*`)this).`*`name`*; thus, it refers to the field named *`name`* of the current object, but with the current object viewed as an instance of the superclass. Thus it can access the field named *`name`* that is visible in class *`S`*, even if that field is hidden by a declaration of a field named *`name`* in class *`C`*.

DISCUSSION

The use of `super` is demonstrated by the following example:

```
interface I { int x = 0; }
class T1 implements I { int x = 1; }
class T2 extends T1 { int x = 2; }
class T3 extends T2 {
   int x = 3;
   void test() {
      System.out.println("x=\t\t"+x);
      System.out.println("super.x=\t\t"+super.x);
      System.out.println("((T2)this).x=\t"+((T2)this).x);
      System.out.println("((T1)this).x=\t"+((T1)this).x);
      System.out.println("((I)this).x=\t"+((I)this).x);
   }
}
class Test {
   public static void main(String[] args) {
      new T3().test();
   }
}
```

which produces the output:

```
x=          3
super.x=    2
((T2)this).x=2
```

```
((T1)this).x=1
((I)this).x=    0
```

Within class `T3`, the expression `super.x` is treated exactly as if it were:

```
((T2)this).x
```

Suppose that a field access expression `T.super.`*`name`* appears within class *`C`*, and the immediate superclass of the class denoted by *`T`* is a class whose fully qualified name is *`S`*. Then `T.super.`*`name`* is treated exactly as if it had been the expression `((`*`S`*`)T.this).`*`name`*.

Thus the expression `T.super.`*`name`* can access the field named *`name`* that is visible in the class named by *`S`*, even if that field is hidden by a declaration of a field named *`name`* in the class named by *`T`*.

It is a compile-time error if the current class is not an inner class of class *`T`* or *`T`* itself.

15.12 Method Invocation Expressions

A method invocation expression is used to invoke a class or instance method.

MethodInvocation:
MethodName `(` *ArgumentList*$_{opt}$ `)`
Primary `.` *NonWildTypeArguments*$_{opt}$ *Identifier* `(` *ArgumentList*$_{opt}$ `)`
`super` `.` *NonWildTypeArguments*$_{opt}$ *Identifier* `(` *ArgumentList*$_{opt}$ `)`
ClassName `.` `super` `.` *NonWildTypeArguments*$_{opt}$ *Identifier* `(`
ArgumentList$_{opt}$ `)`
TypeName `.` *NonWildTypeArguments* *Identifier* `(` *ArgumentList*$_{opt}$ `)`

The definition of *ArgumentList* from §15.9 is repeated here for convenience:

ArgumentList:
Expression
ArgumentList `,` *Expression*

DISCUSSION

Resolving a method name at compile time is more complicated than resolving a field name because of the possibility of method overloading. Invoking a method at run time is also more complicated than accessing a field because of the possibility of instance method overriding.

Determining the method that will be invoked by a method invocation expression involves several steps. The following three sections describe the compile-time processing of a method invocation; the determination of the type of the method invocation expression is described in §15.12.3.

15.12.1 Compile-Time Step 1: Determine Class or Interface to Search

The first step in processing a method invocation at compile time is to figure out the name of the method to be invoked and which class or interface to check for definitions of methods of that name. There are several cases to consider, depending on the form that precedes the left parenthesis, as follows:

- If the form is *MethodName*, then there are three subcases:
  - If it is a simple name, that is, just an *Identifier*, then the name of the method is the *Identifier*. If the *Identifier* appears within the scope (§6.3) of a visible method declaration with that name, then there must be an enclosing type declaration of which that method is a member. Let `T` be the innermost such type declaration. The class or interface to search is `T`.
  - If it is a qualified name of the form *TypeName* `.` *Identifier*, then the name of the method is the *Identifier* and the class to search is the one named by the *TypeName*. If *TypeName* is the name of an interface rather than a class, then a compile-time error occurs, because this form can invoke only `static` methods and interfaces have no `static` methods.
  - In all other cases, the qualified name has the form *FieldName* `.` *Identifier*; then the name of the method is the *Identifier* and the class or interface to search is the declared type `T` of the field named by the *FieldName*, if `T` is a class or interface type, or the upper bound of `T` if `T` is a type variable.
- If the form is *Primary* `.` *Identifier*, then the name of the method is the *Identifier*. Let `T` be the type of the *Primary* expression; then the class or interface to be searched is `T` if `T` is a class or interface type, or the upper bound of `T` if `T` is a type variable.
- If the form is `super .` *Identifier*, then the name of the method is the *Identifier* and the class to be searched is the superclass of the class whose declaration contains the method invocation. Let `T` be the type declaration immediately

enclosing the method invocation. It is a compile-time error if any of the following situations occur:

- *T* is the class `Object`.
- *T* is an interface.
-
-

- If the form is *ClassName*`.super .` *Identifier*, then the name of the method is the *Identifier* and the class to be searched is the superclass of the class *C* denoted by *ClassName*. It is a compile-time error if *C* is not a lexically enclosing class of the current class. It is a compile-time error if *C* is the class `Object`. Let *T* be the type declaration immediately enclosing the method invocation. It is a compile-time error if any of the following situations occur:
  - *T* is the class `Object`.
  - *T* is an interface.
- If the form is *TypeName* `.` *TypeArguments Identifier*, then the name of the method is the *Identifier* and the class to be searched is the class *C* denoted by *TypeName*. If *TypeName* is the name of an interface rather than a class, then a compile-time error occurs, because this form can invoke only `static` methods and interfaces have no `static` methods.

15.12.2 Compile-Time Step 2: Determine Method Signature

The hand-writing experts were called upon for their opinion of the signature . . .
—Agatha Christie, *The Mysterious Affair at Styles* (1920), Chapter 11

The second step searches the type determined in the previous step for member methods. This step uses the name of the method and the types of the argument expressions to locate methods that are both *accessible* and *applicable*, that is, declarations that can be correctly invoked on the given arguments. There may be more than one such method, in which case the *most specific* one is chosen. The descriptor (signature plus return type) of the most specific method is one used at run time to perform the method dispatch.

A method is *applicable* if it is either applicable by subtyping (§15.12.2.2), applicable by method invocation conversion (§15.12.2.3), or it is an applicable variable arity method (§15.12.2.4).

The process of determining applicability begins by determining the potentially applicable methods (§15.12.2.1). The remainder of the process is split into three phases.

DISCUSSION

The purpose of the division into phases is to ensure compatibility with older versions of the Java programming language.

The first phase (§15.12.2.2) performs overload resolution without permitting boxing or unboxing conversion, or the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the second phase.

DISCUSSION

This guarantees that any calls that were valid in older versions of the language are not considered ambiguous as a result of the introduction of variable arity methods, implicit boxing and/or unboxing.

The second phase (§15.12.2.3) performs overload resolution while allowing boxing and unboxing, but still precludes the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the third phase.

DISCUSSION

This ensures that a variable arity method is never invoked if an applicable fixed arity method exists.

The third phase (§15.12.2.4) allows overloading to be combined with variable arity methods, boxing and unboxing.

Deciding whether a method is applicable will, in the case of generic methods (§8.4.4), require that actual type arguments be determined. Actual type arguments may be passed explicitly or implicitly. If they are passed implicitly, they must be inferred (§15.12.2.7) from the types of the argument expressions.

If several applicable methods have been identified during one of the three phases of applicability testing, then the *most specific* one is chosen, as specified in section §15.12.2.5. See the following subsections for details.

15.12.2.1 *Identify Potentially Applicable Methods*

A member method is *potentially applicable* to a method invocation if and only if all of the following are true:

- The name of the member is identical to the name of the method in the method invocation.
- The member is accessible (§6.6) to the class or interface in which the method invocation appears.
- If arity of the member is lesser or equal to the arity of the method invocation.
- If the member is a variable arity method with arity *n*, the arity of the method invocation is greater or equal to *n-1*.
- If the member is a fixed arity method with arity *n*, the arity of the method invocation is equal to *n*.
- If the method invocation includes explicit type parameters, and the member is a generic method, then the number of actual type parameters is equal to the number of formal type parameters.

Whether a member method is accessible at a method invocation depends on the access modifier (`public`, none, `protected`, or `private`) in the member's declaration and on where the method invocation appears.

The class or interface determined by compile-time step 1 (§15.12.1) is searched for all member methods that are potentially applicable to this method invocation; members inherited from superclasses and superinterfaces are included in this search.

In addition, if the method invocation has, before the left parenthesis, a *MethodName* of the form *Identifier,* then the search process also examines all methods that are (a) imported by single-static-import declarations (§7.5.3) and static-import-on-demand declarations (§7.5.4) within the compilation unit (§7.3) within which the method invocation occurs, and (b) not shadowed (§6.3.1) at the place where the method invocation appears.

If the search does not yield at least one method that is potentially applicable, then a compile-time error occurs.

15.12.2.2 *Phase 1: Identify Matching Arity Methods Applicable by Subtyping*

Let m be a potentially applicable method (§15.12.2.1), let $e_1, ..., e_n$ be the actual argument expressions of the method invocation and let A_i be the type of e_i, $1 \le i \le n$, and let $R_1 ... R_p$ $p \ge 1$, be the formal type parameters of m, and let B_l be the declared bound of R_l, $1 \le l < p$. Then:

- If m is a generic method, then let $F_1 ... F_n$ be the formal parameter types of m. Then:
  - If the method invocation does not provide explicit type arguments then let $U_1 ... U_p$ be the actual type arguments inferred (§15.12.2.7) for this invocation of m, using a set of initial constraints consisting of the constraints $A_i << F_i$ for each actual argument expression e_i whose type is a reference type, $1 \le i \le n$.
  - Otherwise let $U_1 ... U_p$ be the explicit type arguments given in the method invocation.

  Then let $S_i = F_i[R_1 = U_1, ..., R_p = U_p]$ $1 \le i < n$, be the formal parameter types inferred for m.
- Otherwise, let $S_1 ... S_n$ be the formal parameter types of m.

The method m is *applicable by subtyping* if and only if both of the following conditions hold:

- For $1 \le i \le n$, either:
  - A_i is a subtype (§4.10) of S_i ($A_i <: S_i$) or
  - A_i is convertible to some type C_i by unchecked conversion (§5.1.9), and $C_i <: S_i$.
- If m is a generic method as described above then $U_l <: B_l[R_1 = U_1, ..., R_p = U_p]$, $1 \le i < n$.

If no method applicable by subtyping is found, the search for applicable methods continues with phase 2 (§15.12.2.3). Otherwise, the most specific method (§15.12.2.5) is chosen among the methods that are applicable by subtyping.

15.12.2.3 *Phase 2: Identify Matching Arity Methods Applicable by Method Invocation Conversion*

Let *m* be a potentially applicable method (§15.12.2.1), let $e_1, ..., e_n$ be the actual argument expressions of the method invocation and let A_i be the type of e_i, $1 \le i \le n$. Then:

- If *m* is a generic method, then let $F_1 ... F_n$ be the formal parameter types of *m*, and let $R_1 ... R_p$ $p \ge 1$, be the formal type parameters of *m*, and let B_l be the declared bound of R_l, $1 \le l < p$. Then:
  - If the method invocation does not provide explicit type arguments then let $U_1 ... U_p$ be the actual type arguments inferred (§15.12.2.7) for this invocation of *m*, using a set of initial constraints consisting of the constraints $A_i << F_i$, $1 \le i \le n$.
  - Otherwise let $U_1 ... U_p$ be the explicit type arguments given in the method invocation.
- Then let $S_i = F_i[R_1 = U_1, ..., R_p = U_p]$ $1 \le i < n$, be the formal parameter types inferred for *m*..
- Otherwise, let $S_1 ... S_n$ be the formal parameter types of *m*.

The method *m* is *applicable by method invocation conversion* if and only if both of the following conditions hold:

- For $1 \le i \le n$, the type of e_i, A_i, can be converted by method invocation conversion (§5.3) to S_i.
- If *m* is a generic method as described above then $U_l <: B_l[R_1 = U_1, ..., R_p = U_p]$, $1 \le i < n$.

If no method applicable by method invocation conversion is found, the search for applicable methods continues with phase 3 (§15.12.2.4). Otherwise, the most specific method (§15.12.2.5) is chosen among the methods that are applicable by method invocation conversion.

15.12.2.4 *Phase 3: Identify Applicable Variable Arity Methods*

Let *m* be a potentially applicable method (§15.12.2.1) with variable arity, let $e_1, ..., e_k$ be the actual argument expressions of the method invocation and let A_i be the type of e_i, $1 \le i \le k$. Then:

- If *m* is a generic method, then let F_1 ... F_n, where $k \geq n - 1$, be the formal parameter types of *m*, where $F_n = T[]$ for some type *T*, and let R_1 ... R_p $p \geq 1$, be the formal type parameters of *m*, and let B_l be the declared bound of R_l, $1 \leq l < p$. Then:
  - If the method invocation does not provide explicit type arguments then let U_1 ... U_p be the actual type arguments inferred (§15.12.2.7) for this invocation of *m*, using a set of initial constraints consisting of the constraints $A_i << F_i$, $1 \leq i < n$ and the constraints $A_j << T$, $n \leq j \leq k$.
  - Otherwise let U_1 ... U_p be the explicit type arguments given in the method invocation.

  Then let $S_i = F_i[R_1 = U_1, ..., R_p = U_p]$ $1 \leq i < n$, be the formal parameter types inferred for *m*.
- Otherwise, let S_1 ... S_n, where $k \geq n - 1$, be the formal parameter types of *m*.

The method *m* is an *applicable variable-arity method* if and only if all three of the following conditions hold:

- For $1 \leq i < n$, the type of e_i, A_i, can be converted by method invocation conversion to S_i.
- If $k \geq n$, then for $n \leq i \leq k$, the type of e_i, A_i, can be converted by method invocation conversion to the component type of S_n.
- If *m* is a generic method as described above then $U_l <: B_l[R_1 = U_1, ..., R_p = U_p]$, $1 \leq i < n$.

If no applicable variable arity method is found, a compile-time error occurs. Otherwise, the most specific method (§15.12.2.5) is chosen among the applicable variable-arity methods.

15.12.2.5 *Choosing the Most Specific Method*

If more than one member method is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the run-time method dispatch. The Java programming language uses the rule that the *most specific* method is chosen.

DISCUSSION

The informal intuition is that one method is more specific than another if any invocation handled by the first method could be passed on to the other one without a compile-time type error.

One fixed-arity member method named `m` is *more specific* than another member method of the same name and arity if all of the following conditions hold:

- The declared types of the parameters of the first member method are $T_1, \ldots, T_n$.
- The declared types of the parameters of the other method are $U_1, \ldots, U_n$.
- If the second method is generic then let $S_1, \ldots, S_n$ be the types inferred (§15.12.2.7) for its parameters under the initial constraints $T_i \ll U_i$ $1 \le i \le n$; otherwise let $1 \le i \le n$. Then, for all j from 1 to n, $T_j <: S_j$.

In addition, one variable arity member method named `m` is *more specific* than another variable arity member method of the same name if either:

- One member method has n parameters and the other has k parameters, where $n \ge k$. The types of the parameters of the first member method are $T_1, \ldots, T_{n-1}, T_n[]$, the types of the parameters of the other method are $U_1, \ldots, U_{k-1}, U_k[]$. If the second method is generic then let $S_1, \ldots, S_k$ be the types inferred (§15.12.2.7) for its parameters under the initial constraints $T_i \ll U_i$ $1 \le i \le k-1$, $T_i \ll U_k$ $k \le i \le n$; otherwise let $S_i = U_i$ $1 \le i \le k$. Then:
  - for all j from 1 to $k-1$, $T_j <: S_j$, and,
  - for all j from k to n, $T_j <: S_k$.
- One member method has k parameters and the other has n parameters, where $n \ge k$. The types of the parameters of the first method are $U_1, \ldots, U_{k-1}, U_k[]$, the types of the parameters of the other method are $T_1, \ldots, T_{n-1}, T_n[]$. If the second method is generic then let $S_1, \ldots, S_n$ be the types inferred (§15.12.2.7) for its parameters under the initial constraints $U_i \ll T_i$ $1 \le i \le k-1$, $U_k \ll T_i$ $k \le i \le n$; otherwise let $S_i = T_i$ $1 \le i \le n$. Then:
  - for all j from 1 to $k-1$, $U_j <: S_j$, and,
  - for all j from k to n, $U_k <: S_j$.

The above conditions are the only circumstances under which one method may be more specific than another.

A method m_1 is *strictly more specific* than another method m_2 if and only if m_1 is more specific than m_2 and m_2 is not more specific than m_1

A method is said to be *maximally specific* for a method invocation if it is accessible and applicable and there is no other method that is applicable and accessible that is strictly more specific.

If there is exactly one maximally specific method, then that method is in fact *the most specific method*; it is necessarily more specific than any other accessible method that is applicable. It is then subjected to some further compile-time checks as described in §15.12.3.

It is possible that no method is the most specific, because there are two or more methods that are maximally specific. In this case:

- If all the maximally specific methods have override-equivalent (§8.4.2) signatures, then:
    - If exactly one of the maximally specific methods is not declared `abstract`, it is the most specific method.
    - Otherwise, if all the maximally specific methods are declared `abstract`, and the signatures of all of the maximally specific methods have the same erasure (§4.6), then the most specific method is chosen arbitrarily among the subset of the maximally specific methods that have the most specific return type. However, the most specific method is considered to throw a checked exception if and only if that exception or its erasure is declared in the `throws` clauses of each of the maximally specific methods.
- Otherwise, we say that the method invocation is *ambiguous*, and a compile-time error occurs.

15.12.2.6 *Method Result and Throws Types*

- The result type of the chosen method is determined as follows:
    - If the method being invoked is declared with a return type of `void`, then the result is `void`.
    - Otherwise, if unchecked conversion was necessary for the method to be applicable then the result type is the erasure (§4.6) of the method's declared return type.
    - Otherwise, if the method being invoked is generic, then for $1 \le i \le n$, let A_i be the formal type parameters of the method, let T_i be the actual type arguments inferred for the method invocation, and let R be the declared return

type of the method being invoked. The result type is obtained by applying capture conversion (§5.1.10) to $R[A_1 := T_1, ..., A_n := T_n]$.

- Otherwise, the result type is obtained by applying capture conversion (§5.1.10) to the type given in the method declaration.

The exception types of the throws clause of the chosen method are determined as follows:

- If unchecked conversion was necessary for the method to be applicable then the throws clause is composed of the erasure (§4.6) of the types in the method's declared throws clause.
- Otherwise, if the method being invoked is generic, then for $1 \le i \le n$, let A_i be the formal type parameters of the method, let T_i be the actual type arguments inferred for the method invocation, and let E_j, $1 \le j \le m$ be the exception types declared throws clause of the method being invoked. The throws clause consists of the types to $E_j[A_1 := T_1, ..., A_n := T_n]$.
- Otherwise, the type of the throws clause is the type given in the method declaration.

A method invocation expression can throw an exception type E iff either:

- The method to be invoked is of the form *Primary.Identifier* and the *Primary* expression can throw *E*; or
- Some expression of the argument list can throw *E*; or
- *E* is listed in the throws clause of the type of method that is invoked.

15.12.2.7 *Inferring Type Arguments Based on Actual Arguments*

In this section, we describe the process of inferring type arguments for method and constructor invocations. This process is invoked as a subroutine when testing for method (or constructor) applicability (§15.12.2.2 - §15.12.2.4).

DISCUSSION

The process of type inference is inherently complex. Therefore, it is useful to give an informal overview of the process before delving into the detailed specification.

Inference begins with an initial set of constraints. Generally, the constraints require that the statically known types of the actual arguments are acceptable given the declared formal argument types. We discuss the meaning of “acceptable” below.

Given these initial constraints, one may derive a set of supertype and/or equality constraints on the formal type parameters of the method or constructor.

Next, one must try and find a solution that satisfies the constraints on the type parameters. As a first approximation, if a type parameter is constrained by an equality constraint, then that constraints give its solution. Bear in mind that the constraint may equate one type parameter with another, and only when the entire set of constraints on all type variables is resolved will we have a solution.

A supertype constraint T :> X implies that the solution is one of supertypes of X. Given several such constraints on T, we can intersect the sets of supertypes implied by each of the constraints, since the type parameter must be a member of all of them. We can then choose the most specific type that is in the intersection.

Computing the intersection is more complicated than one might first realize. Given that a type parameter is constrained to be a supertype of two distinct invocations of a generic type, say List<Object> and List<String>, the naive intersection operation might yield Object. However, a more sophisticated analysis yields a set containing List<?>. Similarly, if a type parameter is constrained to be a supertype of two unrelated interfaces I and J., we might infer T must be Object, or we might obtain a tighter bound of I & J. These issues are discussed further later in this section.

We will use the following notational conventions in this section:

- Type expressions are represented using the letters A, F, U, V and W. The letter A is only used to denote the type of an actual parameter, and F is only used to denote the type of a formal parameter.
- Type parameters are represented using the letters S and T
- Arguments to parameterized types are represented using the letters X, Y.
- Generic type declarations are represented using the letters G and H.

Inference begins with a set of *initial constraints* of the form *A << F*, *A = F*, or *A >> F*, where *U << V* indicates that type *U* is convertible to type *V* by method invocation conversion (§5.3), and *U >> V* indicates that type *V* is convertible to type *U* by method invocation conversion.

Discussion

In a simpler world, the constraints could be of the forrm A <: F. - simply requiring that the actual argument types be subtypes of the formal ones. However, reality is more involved. As discussed earlier, method applicability testing consists of up to three phases; this is required for compatibility reasons. Each phase imposes slightly different constraints. If a method is applicable by subtyping (§15.12.2.2), the constraints are indeed subtyping constraints. If a method is applicable by method invocation conversion (§15.12.2.3), the constraints imply that the actual type is convertible to the formal type by method invocation conversion. The situation is similar for the third phase (§15.12.2.4), but the exact form of the constraints differ due to the variable arity.

These constraints are then reduced to a set of simpler constraints of the forms *T :> X, T = X* or *T <: X,* where *T* is a type parameter of the method. This reduction is achieved by the procedure given below:

It may be that the initial constraints are unsatisfiable; we say that inference is *overconstrained*. In that case, we do not necessarily derive unsatisfiable constraints on the type parameters. Instead, we may infer actual type arguments for the invocation, but once we substitute the actual type arguments for the formal type parameters, the applicability test may fail because the actual argument types are not acceptable given the substituted formals.

An alternative strategy would be to have type inference itself fail in such cases. Compilers may choose to do so, provided the effect is equivalent to that specified here.

Given a constraint of the form *A << F, A = F,* or *A >> F:*

- If *F* involves a type parameter T_j.
  - If *A* is the type of `null`, no constraint is implied on T_j.
  - Otherwise, if the constraint has the form *A <<F*
    - If *A* is a primitive type, then *A* is converted to a reference type *U* via boxing conversion and this algorithm is applied recursively to the constraint *U << F*.
    - Otherwise, if *F = Tj*, then the constraint T_j *:> A* is implied.
    - If *F = U[],* where the type *U* involves T_j, then if *A* is an array type *V[],* or a type variable with an upper bound that is an array type *V[],* where *V* is a reference type, this algorithm is applied recursively to the constraint *V <<U.*

    - If F has the form $G<..., Y_{k-1}, U, Y_{k+1}, ...>$, $1 \leq k \leq n$ where U is a type expression that involves T_j, then if A has a supertype of the form $G<..., X_{k-1}, V, X_{k+1}, ...>$ where V is a type expression, this algorithm is applied recursively to the constraint $V = U$.
    - If F has the form $G<..., Y_{k-1}$, *? extends* $U, Y_{k+1}, ...>$, where U involves T_j, then if A has a supertype that is one of:
        - $G<..., X_{k-1}, V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V << U$.
        - $G<..., X_{k-1}$, *? extends* $V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V << U$.
        - Otherwise, no constraint is implied on T_j.
    - If F has the form $G<..., Y_{k-1}$, *? super* $U, Y_{k+1}, ...>$, where U involves T_j, then if A has a supertype that is one of:
        - $G<..., X_{k-1}, V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V >> U$.
        - $G<..., X_{k-1}$, *? super* $V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V >> U$.
        - Otherwise, no constraint is implied on T_j.
    - Otherwise, no constraint is implied on T_j.
- Otherwise, if the constraint has the form $A = F$
    - If $F = T_j$, then the constraint $T_j = A$ is implied.
    - If $F = U[]$ where the type U involves T_j, then if A is an array type $V[]$, or a type variable with an upper bound that is an array type $V[]$, where V is a reference type, this algorithm is applied recursively to the constraint $V = U$.
    - If F has the form $G<..., Y_{k-1}, U, Y_{k+1}, ...>$, $1 \leq k \leq n$ where U is type expression that involves T_j, then if A is of the form $G<..., X_{k-1}, V, X_{k+1},...>$ where V is a type expression, this algorithm is applied recursively to the constraint $V = U$.
    - If F has the form $G<..., Y_{k-1}$, *? extends* $U, Y_{k+1}, ...>$, where U involves T_j, then if A is one of:
        - $G<..., X_{k-1}$, *? extends* $V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V = U$.

    - Otherwise, no constraint is implied on T_j.
  - If F has the form $G<..., Y_{k-1}, ? \textit{ super } U, Y_{k+1} ,...>$, where U involves T_j, then if A is one of:
    - $G<..., X_{k-1}, ? \textit{ super } V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V = U$.
    - Otherwise, no constraint is implied on T_j.
  - Otherwise, no constraint is implied on T_j.
- Otherwise, if the constraint has the form $A >> F$
  - If $F = T_j$, then the constraint $T_j <: A$ is implied.
  - If $F = U[]$, where the type U involves T_j, then if A is an array type $V[]$, or a type variable with an upper bound that is an array type $V[]$, where V is a reference type, this algorithm is applied recursively to the constraint $V >> U$. Otherwise, no constraint is implied on T_j.
  - If F has the form $G<..., Y_{k-1}, U, Y_{k+1}, ...>$, where U is a type expression that involves T_j, then:
    - If A is an instance of a non-generic type, then no constraint is implied on T_j.
    - If A is an invocation of a generic type declaration H, where H is either G or superclass or superinterface of G, then:
      - If $H \neq G$, then let $S_1, ..., S_n$ be the formal type parameters of G, and let $H<U_1, ..., U_l>$ be the unique invocation of H that is a supertype of $G<S_1 , ..., S_n>$, and let $V = H<U_1, ..., U_l>[S_k = U]$. Then, if $V :> F$ this algorithm is applied recursively to the constraint $A >> V$.
      - Otherwise, if A is of the form $G<..., X_{k-1}, W, X_{k+1}, ...>$, where W is a type expression this algorithm is applied recursively to the constraint $W = U$.
      - Otherwise, if A is of the form $G<..., X_{k-1}, ? \textit{ extends } W, X_{k+1}, ...>$, this algorithm is applied recursively to the constraint $W >> U$.
      - Otherwise, if A is of the form $G<..., X_{k-1}, ? \textit{ super } W, X_{k+1}, ...>$, this algorithm is applied recursively to the constraint $W << U$.
      - Otherwise, no constraint is implied on T_j.
  - Otherwise, no constraint is implied on T_j.

- If F has the form $G<..., Y_{k-1}, ?\ extends\ U, Y_{k+1}, ...>$, where U is a type expression that involves T_j, then:
  - If A is an instance of a non-generic type, then no constraint is implied on T_j.
  - If A is an invocation of a generic type declaration H, where H is either G or superclass or superinterface of G, then:
    - If $H \neq G$, then let $S_1, ..., S_n$ be the formal type parameters of G, and let $H<U_1, ..., U_l>$ be the unique invocation of H that is a supertype of $G<S_1 , ..., S_n>$, and let $V = H<U_1, ..., U_l>[S_k = ?\ extends\ U]$. Then this algorithm is applied recursively to the constraint $A >> V$.
    - Otherwise, if A is of the form $G<..., X_{k-1}, ?\ extends\ W, X_{k+1}, ...>$, this algorithm is applied recursively to the constraint $W >>U$.
    - Otherwise, no constraint is implied on T_j.
- If F has the form $G<... , Y_{k-1}, ?\ super\ U, Y_{k+1}, ...>$, where U is a type expression that involves T_j, then A is either:
  - If A is an instance of a non-generic type, then no constraint is implied on T_j.
  - If A is an invocation of a generic type declaration H, where H is either G or superclass or superinterface of G, then:
    - If $H \neq G$, then let $S_1, ..., S_n$ be the formal type parameters of G, and let $H<U_1, ..., U_l>$ be the unique invocation of H that is a supertype of $G<S_1 , ..., S_n>$, and let $V = H<U_1, ..., U_l>[S_k = ?\ super\ U]$. Then this algorithm is applied recursively to the constraint $A >> V$.
    - Otherwise, if A is of the form $G<..., X_{k-1}, ?\ super\ W, ..., X_{k+1}, ...>$, this algorithm is applied recursively to the constraint $W << U$.
    - Otherwise, no constraint is implied on T_j.

- Otherwise, no constraint is implied on T_j.

DISCUSSION

Note that this does not impose any constraints on the type parameters based on their declared bounds. Once the actual type arguments are inferred, they will be tested against the declared bounds of the formal type parameters as part of applicability testing.

Next, for each type variable T_j, $1 \leq j \leq n$, the implied equality constraints are resolved as follows:

For each implied equality constraint $T_j = U$ or $U = T_j$:

- If U is not one of the type parameters of the method, then U is the type inferred for T_j. Then all remaining equality constraints involving T_j are rewritten such that T_j is replaced with U. There are necessarily no further equality constraints involving T_j, and processing continues with the next type parameter, if any.
- Otherwise, if U is T_j, then this constraint carries no information and may be discarded.
- Otherwise, the constraint is of the form $T_j = T_k$ for $k \neq j$. Then all constraints involving T_j are rewritten such that T_j is replaced with T_k, and processing continues with the next type variable.

Then, for each remaining type variable T_j, the constraints $T_j :> U$ are considered. Given that these constraints are $T_j :> U_1 ... T_j :> U_k$, the type of T_j is inferred as $lub(U_1 ... U_k)$, computed as follows:

For a type U, we write $ST(U)$ for the set of supertypes of U, and define the erased supertype set of U,

$EST(U) = \{ V \mid W \text{ in } ST(U) \text{ and } V = |W| \}$

where |W| is the erasure (§4.6) of *W*.

DISCUSSION

The reason for computing the set of erased supertypes is to deal with situations where a type variable is constrained to be a supertype of several distinct invocations of a generic type declaration, For example, if T :> List<String> and T :> List<Object>, simply intersecting the sets ST(List<String>) = {List<String>, Collection<String>, Object} and ST(List<Object>) = {List<Object>), Collection<Object>, Object} would yield a set {Object}, and we would have lost track of the fact that T can safely be assumed to be a List.

In contrast, intersecting EST(List<String>) = {List, Collection, Object} and EST(List<Object>) = {List, Collection, Object} yields {List, Collection, Object}, which we will eventually enable us to infer T = List<?> as described below.

The erased candidate set for type parameter T_j , *EC*, is the intersection of all the sets *EST(U)* for each *U* in U_1 .. U_k. The minimal erased candidate set for T_j is

MEC = { V | V in EC, and for all $W \neq V$ *in EC, it is not the case that W <: V}*

DISCUSSION

Because we are seeking to infer more precise types, we wish to filter out any candidates that are supertypes of other candidates. This is what computing MEC accomplishes.

In our running example, we had EC = List, Collection, Object}, and now MEC = {List}.

The next step will be to recover actual type arguments for the inferred types.

For any element *G* of *MEC* that is a generic type declaration, define the relevant invocations of *G*, *Inv(G)* to be:

Inv(G) = { V | $1 \le i \le k$ *, V in ST(*U_i*), V = G<...>}*

DISCUSSION

In our running example, the only generic element of MEC is List, and Inv(List) = {List<String>, List<Object>}. We now will seek to find a type argument for List that contains (§4.5.1.1) both String and Object.

This is done by means of the least containing invocation (lci) operation defined below. The first line defines lci() on a set, such as Inv(List), as an operation on a list of the elements of the set. The next line defines the operation on such lists, as a pairwise reduction on the elements of the list. The third line is the definition of lci() on pairs of parameterized types, which in turn relies on the notion of least containing type argument (lcta).

lcta() is defined for all six possible cases. Then *CandidateInvocation(G)* defines the most specific invocation of the generic G that is contains all the invocations of G that are known to be supertypes of T_j. This will be our candidate invocation of G in the bound we infer for T_j

---

and let *CandidateInvocation(G) = lci(Inv(G))* where *lci*, the least containing invocation is defined

lci(S) = lci(e_1*, ...,* e_n*)* where e_i in *S*, $1 \le i \le n$

lci(e_1*, ...,* e_n*) = lci(lci(*e_1*,* e_2*),* e_3*, ...,* e_n*)*

lci(G<X_1*, ...,* X_n*>, G<*Y_1*, ...,* Y_n*>) = G<lcta(*X_1*,* Y_1*),..., lcta(*X_n*,* Y_n*)>*

where *lcta()* is the the least containing type argument function defined (assuming U and V are type expressions) as:

lcta(U, V) = U if *U = V, ? extends lub(U, V)* otherwise
lcta(U, ? extends V) = ? extends lub(U, V)
lcta(U, ? super V) = ? super glb(U, V)
lcta(? extends U, ? extends V) = ? extends lub(U, V)
lcta(? extends U, ? super V) = U if *U = V, ?* otherwise
lcta(? super U, ? super V) = ? super glb(U, V)

where *glb()* is as defined in (§5.1.10).

DISCUSSION

Finally, we define a bound for T_j based on on all the elements of the minimal erased candidate set of its supertypes. If any of these elements are generic, we use the CandidateInvocation() function to recover the type argument information.

Then, define *Candidate(W) = CandidateInvocation(W)* if *W* is generic, *W* otherwise.

Then the inferred type for T_j is

$lub(U_1 \ldots U_k) = Candidate(W_1) \,\&\, \ldots \,\&\, Candidate(W_r)$ where W_i, $1 \le i \le r$, are the elements of *MEC*.

It is possible that the process above yields an infinite type. This is permissible, and Java compilers must recognize such situations and represent them appropriately using cyclic data structures.

DISCUSSION

The possibility of an infinite type stems from the recursive calls to lub().

Readers familiar with recursive types should note that an infinite type is not the same as a recursive type.

15.12.2.8 *Inferring Unresolved Type Arguments*

If any of the method's type arguments were not inferred from the types of the actual arguments, they are now inferred as follows.

- If the method result occurs in a context where it will be subject to assignment conversion (§5.2) to a type *S*, then let *R* be the declared result type of the method, and let $R' = R[T_1 = B(T_1) ... T_n = B(T_n)]$ where $B(T_i)$ is the type inferred for T_i in the previous section, or T_i if no type was inferred.

  Then, a set of initial constraints consisting of:

- the constraint $S >> R'$, and
- additional constraints $B_i[T_1 = B(T_1) ... T_n = B(T_n)] << T_i$, where B_i is the declared bound of T_i,

is created and used to infer constraints on the type arguments using the algorithm of section (§15.12.2.7). Any equality constraints are resolved, and then, for each remaining constraint of the form $T_i <: U_k$, the argument T_i is inferred to be $glb(U_1, ..., U_k)$ (§5.1.10).

Any remaining type variables that have not yet been inferred are then inferred to have type `Object.`

- Otherwise, the unresolved type arguments are inferred by invoking the procedure described in this section under the assumption that the method result was assigned to a variable of type `Object`.

15.12.2.9 *Examples*

DISCUSSION

In the example program:

```
public class Doubler {
           static int two() { return two(1); }
           private static int two(int i) { return 2*i; }
}
class Test extends Doubler {
           public static long two(long j) {return j+j; }
   public static void main(String[] args) {
      System.out.println(two(3));
```

```
        System.out.println(Doubler.two(3)); // compile-time error
    }
}
```

for the method invocation `two(1)` within class `Doubler`, there are two accessible methods named `two`, but only the second one is applicable, and so that is the one invoked at run time. For the method invocation `two(3)` within class `Test`, there are two applicable methods, but only the one in class `Test` is accessible, and so that is the one to be invoked at run time (the argument `3` is converted to type `long`). For the method invocation `Doubler.two(3)`, the class `Doubler`, not class `Test`, is searched for methods named `two`; the only applicable method is not accessible, and so this method invocation causes a compile-time error.

Another example is:

```
class ColoredPoint {
    int x, y;
    byte color;
    void setColor(byte color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        byte color = 37;
        cp.setColor(color);
        cp.setColor(37);                // compile-time error
    }
}
```

Here, a compile-time error occurs for the second invocation of `setColor`, because no applicable method can be found at compile time. The type of the literal `37` is `int`, and `int` cannot be converted to `byte` by method invocation conversion. Assignment conversion, which is used in the initialization of the variable `color`, performs an implicit conversion of the constant from type `int` to `byte`, which is permitted because the value `37` is small enough to be represented in type `byte`; but such a conversion is not allowed for method invocation conversion.

If the method `setColor` had, however, been declared to take an `int` instead of a `byte`, then both method invocations would be correct; the first invocation would be allowed because method invocation conversion does permit a widening conversion from `byte` to `int`. However, a narrowing cast would then be required in the body of `setColor`:

```
        void setColor(int color) { this.color =
(byte)color; }
```

DISCUSSION

15.12.2.10 *Example: Overloading Ambiguity*

Consider the example:

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }

class Test {
   static void test(ColoredPoint p, Point q) {
      System.out.println("(ColoredPoint, Point)");
   }
   static void test(Point p, ColoredPoint q) {
      System.out.println("(Point, ColoredPoint)");
   }
   public static void main(String[] args) {
      ColoredPoint cp = new ColoredPoint();
      test(cp, cp);                  // compile-time error
   }
}
```

This example produces an error at compile time. The problem is that there are two declarations of *test* that are applicable and accessible, and neither is more specific than the other. Therefore, the method invocation is ambiguous.

If a third definition of `test` were added:

```
   static void test(ColoredPoint p, ColoredPoint q) {
      System.out.println("(ColoredPoint, ColoredPoint)");
   }
```

then it would be more specific than the other two, and the method invocation would no longer be ambiguous.

15.12.2.11 *Example: Return Type Not Considered*

As another example, consider:

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
   static int test(ColoredPoint p) {
      return p.color;
   }
```

```
      static String test(Point p) {
         return "Point";
      }
      public static void main(String[] args) {
         ColoredPoint cp = new ColoredPoint();
         String s = test(cp);              // compile-time error
      }
   }
```

Here the most specific declaration of method `test` is the one taking a parameter of type `ColoredPoint`. Because the result type of the method is `int`, a compile-time error occurs because an `int` cannot be converted to a `String` by assignment conversion. This example shows that the result types of methods do not participate in resolving overloaded methods, so that the second `test` method, which returns a `String`, is not chosen, even though it has a result type that would allow the example program to compile without error.

15.12.2.12 *Example: Compile-Time Resolution*

The most applicable method is chosen at compile time; its descriptor determines what method is actually executed at run time. If a new method is added to a class, then source code that was compiled with the old definition of the class might not use the new method, even if a recompilation would cause this method to be chosen.

So, for example, consider two compilation units, one for class `Point`:

```
   package points;
   public class Point {
              public int x, y;
              public Point(int x, int y) { this.x = x; this.y =
y; }
              public String toString() { return toString(""); }
      public String toString(String s) {
         return "(" + x + "," + y + s + ")";
      }
   }
```

and one for class `ColoredPoint`:

```
   package points;
   public class ColoredPoint extends Point {
      public static final int
         RED = 0, GREEN = 1, BLUE = 2;
```

```
        public static String[] COLORS =
            { "red", "green", "blue" };
                public byte color;
        public ColoredPoint(int x, int y, int color) {
            super(x, y); this.color = (byte)color;
        }
        /** Copy all relevant fields of the argument into
            this ColoredPoint object. */
        public void adopt(Point p) { x = p.x; y = p.y; }
        public String toString() {
            String s = "," + COLORS[color];
            return super.toString(s);
        }
    }
```

Now consider a third compilation unit that uses `ColoredPoint`:

```
    import points.*;
    class Test {
        public static void main(String[] args) {
            ColoredPoint cp =
                new ColoredPoint(6, 6, ColoredPoint.RED);
            ColoredPoint cp2 =
                new ColoredPoint(3, 3, ColoredPoint.GREEN);
            cp.adopt(cp2);
            System.out.println("cp: " + cp);
        }
    }
```

The output is:

```
    cp: (3,3,red)
```

The application programmer who coded class `Test` has expected to see the word `green`, because the actual argument, a `ColoredPoint`, has a `color` field, and `color` would seem to be a "relevant field" (of course, the documentation for the package `Points` ought to have been much more precise!).

Notice, by the way, that the most specific method (indeed, the only applicable method) for the method invocation of `adopt` has a signature that indicates a method of one parameter, and the parameter is of type `Point`. This signature becomes part of the binary representation of class `Test` produced by the compiler and is used by the method invocation at run time.

Suppose the programmer reported this software error and the maintainer of the `points` package decided, after due deliberation, to correct it by adding a method to class `ColoredPoint`:

```
    public void adopt(ColoredPoint p) {
        adopt((Point)p); color = p.color;
    }
```

If the application programmer then runs the old binary file for `Test` with the new binary file for `ColoredPoint`, the output is still:

```
    cp: (3,3,red)
```

because the old binary file for `Test` still has the descriptor “one parameter, whose type is `Point`; `void`” associated with the method call `cp.adopt(cp2)`. If the source code for `Test` is recompiled, the compiler will then discover that there are now two applicable `adopt` methods, and that the signature for the more specific one is “one parameter, whose type is `ColoredPoint`; `void`”; running the program will then produce the desired output:

```
    cp: (3,3,green)
```

With forethought about such problems, the maintainer of the `points` package could fix the `ColoredPoint` class to work with both newly compiled and old code, by adding defensive code to the old `adopt` method for the sake of old code that still invokes it on `ColoredPoint` arguments:

```
    public void adopt(Point p) {
        if (p instanceof ColoredPoint)
            color = ((ColoredPoint)p).color;
        x = p.x; y = p.y;
    }
```

Ideally, source code should be recompiled whenever code that it depends on is changed. However, in an environment where different classes are maintained by different organizations, this is not always feasible. Defensive programming with careful attention to the problems of class evolution can make upgraded code much more robust. See §13 for a detailed discussion of binary compatibility and type evolution.

15.12.3 Compile-Time Step 3: Is the Chosen Method Appropriate?

If there is a most specific method declaration for a method invocation, it is called the *compile-time declaration* for the method invocation. Three further checks must be made on the compile-time declaration:

- If the method invocation has, before the left parenthesis, a *MethodName* of the form *Identifier*, and the method is an instance method, then:
    - If the invocation appears within a static context (§8.1.2), then a compile-time error occurs. (The reason is that a method invocation of this form can-

not be used to invoke an instance method in places where `this` (§15.8.3) is not defined.)

  - Otherwise, let *`C`* be the innermost enclosing class of which the method is a member. If the invocation is not directly enclosed by *`C`* or an inner class of *`C`*, then a compile-time error occurs

- If the method invocation has, before the left parenthesis, a *MethodName* of the form *TypeName* `.` *Identifier*, then the compile-time declaration should be `static`. If the compile-time declaration for the method invocation is for an instance method, then a compile-time error occurs. (The reason is that a method invocation of this form does not specify a reference to an object that can serve as `this` within the instance method.)

- If the method invocation has, before the left parenthesis, a *MethodName* of the form `super .` *Identifier*, then:

  - If the method is `abstract`, a compile-time error occurs

  - If the method invocation occurs in a static context, a compile-time error occurs

- If the method invocation has, before the left parenthesis, a *MethodName* of the form *ClassName*`.super .` *Identifier*, then:

  - If the method is `abstract`, a compile-time error occurs

  - If the method invocation occurs in a static context, a compile-time error occurs

  - Otherwise, let *`C`* be the class denoted by *ClassName*. If the invocation is not directly enclosed by *`C`* or an inner class of *`C`*, then a compile-time error occurs

- If the compile-time declaration for the method invocation is `void`, then the method invocation must be a top-level expression, that is, the *Expression* in an expression statement (§14.8) or in the *ForInit* or *ForUpdate* part of a `for` statement (§14.13), or a compile-time error occurs. (The reason is that such a method invocation produces no value and so must be used only in a situation where a value is not needed.)

The following compile-time information is then associated with the method invocation for use at run time:

- The name of the method.

- The qualifying type of the method invocation (§13.1).

- The number of parameters and the types of the parameters, in order.
- The result type, or `void`.
- The invocation mode, computed as follows:
  - If the compile-time declaration has the `static` modifier, then the invocation mode is `static`.
  - Otherwise, if the compile-time declaration has the `private` modifier, then the invocation mode is `nonvirtual`.
  - Otherwise, if the part of the method invocation before the left parenthesis is of the form `super .` *Identifier* or of the form *ClassName*`.super.`*Identifier* then the invocation mode is `super`.
  - Otherwise, if the compile-time declaration is in an interface, then the invocation mode is `interface`.
  - Otherwise, the invocation mode is `virtual`.

If the compile-time declaration for the method invocation is not `void`, then the type of the method invocation expression is the result type specified in the compile-time declaration.

15.12.4 Runtime Evaluation of Method Invocation

At run time, method invocation requires five steps. First, a *target reference* may be computed. Second, the argument expressions are evaluated. Third, the accessibility of the method to be invoked is checked. Fourth, the actual code for the method to be executed is located. Fifth, a new activation frame is created, synchronization is performed if necessary, and control is transferred to the method code.

15.12.4.1 *Compute Target Reference (If Necessary)*

There are several cases to consider, depending on which of the four productions for *MethodInvocation* (§15.12) is involved:

- If the first production for *MethodInvocation*, which includes a *MethodName*, is involved, then there are three subcases:
  - If the *MethodName* is a simple name, that is, just an *Identifier*, then there are two subcases:
    - If the invocation mode is `static`, then there is no target reference.

        - Otherwise, let `T` be the enclosing type declaration of which the method is a member, and let *n* be an integer such that `T` is the *n*th lexically enclosing type declaration (§8.1.2) of the class whose declaration immediately contains the method invocation. Then the target reference is the *n*th lexically enclosing instance (§8.1.2) of `this`. It is a compile-time error if the *n*th lexically enclosing instance (§8.1.2) of `this` does not exist.
    - If the *MethodName* is a qualified name of the form *TypeName* `.` *Identifier*, then there is no target reference.
    - If the *MethodName* is a qualified name of the form *FieldName* `.` *Identifier*, then there are two subcases:
        - If the invocation mode is `static`, then there is no target reference. . The expression *FieldName* is evaluated, but the result is then discarded.
        - Otherwise, the target reference is the value of the expression *FieldName*.
- If the second production for *MethodInvocation*, which includes a *Primary*, is involved, then there are two subcases:
    - If the invocation mode is `static`, then there is no target reference. The expression *Primary* is evaluated, but the result is then discarded.
    - Otherwise, the expression *Primary* is evaluated and the result is used as the target reference.

  In either case, if the evaluation of the *Primary* expression completes abruptly, then no part of any argument expression appears to have been evaluated, and the method invocation completes abruptly for the same reason.
- If the third production for *MethodInvocation*, which includes the keyword `super`, is involved, then the target reference is the value of `this`.
- If the fourth production for *MethodInvocation*, *ClassName*.`super`, is involved, then the target reference is the value of *ClassName*.`this`.

15.12.4.2 *Evaluate Arguments*

The process of evaluating of the argument list differs, depending on whether the method being invoked is a fixed arity method or a variable arity method (§8.4.1).

If the method being invoked is a variable arity method (§8.4.1) *m*, it necessarily has $n > 0$ formal parameters. The final formal parameter of *m* necessarily has type *T[]* for some *T*, and *m* is necessarily being invoked with $k \geq 0$ actual argument expressions.

If *m* is being invoked with $k \neq n$ actual argument expressions, or, if *m* is being invoked with $k = n$ actual argument expressions and the type of the *k*th argument expression is not assignment compatible with *T[]*, then the argument list $(e_1, \ldots, e_{n-1}, e_n, \ldots e_k)$ is evaluated as if it were written as $(e_1, \ldots, e_{n-1}, new\ T[]\{e_n, \ldots, e_k\})$.

The argument expressions (possibly rewritten as described above) are now evaluated to yield *argument value*s. Each argument value corresponds to exactly one of the method's *n* formal parameters.

The argument expressions, if any, are evaluated in order, from left to right. If the evaluation of any argument expression completes abruptly, then no part of any argument expression to its right appears to have been evaluated, and the method invocation completes abruptly for the same reason.The result of evaluating the *j*th argument expression is the *j*th argument value, for $1 \leq j \leq n$. Evaluation then continues, using the argument values, as described below.

15.12.4.3 *Check Accessibility of Type and Method*

Let `C` be the class containing the method invocation, and let `T` be the qualifying type of the method invocation (§13.1), and `m` be the name of the method, as determined at compile time (§15.12.3). An implementation of the Java programming language must insure, as part of linkage, that the method `m` still exists in the type `T`. If this is not true, then a `NoSuchMethodError` (which is a subclass of `IncompatibleClassChangeError`) occurs. If the invocation mode is `interface`, then the implementation must also check that the target reference type still implements the specified interface. If the target reference type does not still implement the interface, then an `IncompatibleClassChangeError` occurs.

The implementation must also insure, during linkage, that the type `T` and the method `m` are accessible. For the type `T`:

- If `T` is in the same package as `C`, then `T` is accessible.
- If `T` is in a different package than `C`, and `T` is `public`, then `T` is accessible.
- If `T` is in a different package than `C`, and `T` is `protected`, then `T` is accessible if and only if `C` is a subclass of `T`.

For the method `m`:

- If `m` is `public`, then `m` is accessible. (All members of interfaces are `public` (§9.2)).
- If `m` is `protected`, then `m` is accessible if and only if either `T` is in the same package as `C`, or `C` is `T` or a subclass of `T`.

- If *m* has default (package) access, then *m* is accessible if and only if *T* is in the same package as *C*.
- If *m* is `private`, then *m* is accessible if and only if *C* is *T*, or *C* encloses *T*, or *T* encloses *C*, or *T* and *C* are both enclosed by a third class.

If either *T* or *m* is not accessible, then an `IllegalAccessError` occurs (§12.3).

15.12.4.4 *Locate Method to Invoke*

Here inside my paper cup,
Everything is looking up.
—Jim Webb, *Paper Cup* (1967)

The strategy for method lookup depends on the invocation mode.

If the invocation mode is `static`, no target reference is needed and overriding is not allowed. Method *m* of class *T* is the one to be invoked.

Otherwise, an instance method is to be invoked and there is a target reference. If the target reference is `null`, a `NullPointerException` is thrown at this point. Otherwise, the target reference is said to refer to a *target object* and will be used as the value of the keyword `this` in the invoked method. The other four possibilities for the invocation mode are then considered.

If the invocation mode is `nonvirtual`, overriding is not allowed. Method *m* of class *T* is the one to be invoked.

Otherwise, the invocation mode is `interface`, `virtual`, or `super`, and overriding may occur. A *dynamic method lookup* is used. The dynamic lookup process starts from a class *S*, determined as follows:

- If the invocation mode is `interface` or `virtual`, then *S* is initially the actual run-time class *R* of the target object. This is true even if the target object is an array instance. (Note that for invocation mode `interface`, *R* necessarily implements *T*; for invocation mode `virtual`, *R* is necessarily either *T* or a subclass of *T*.)
- If the invocation mode is `super`, then *S* is initially the qualifying type (§13.1) of the method invocation.

The dynamic method lookup uses the following procedure to search class *S*, and then the superclasses of class *S*, as necessary, for method *m*.

Let *X* be the compile-time type of the target reference of the method invocation.

1. If class *S* contains a declaration for a non-abstract method named *m* with the same descriptor (same number of parameters, the same parameter types, and

the same return type) required by the method invocation as determined at compile time (§15.12.3), then:

- If the invocation mode is `super` or `interface`, then this is the method to be invoked, and the procedure terminates.
- If the invocation mode is `virtual`, and the declaration in *S* overrides (§8.4.6.1) *X.m*, then the method declared in *S* is the method to be invoked, and the procedure terminates.

2. Otherwise, if *S* has a superclass, this same lookup procedure is performed recursively using the direct superclass of *S* in place of *S*; the method to be invoked is the result of the recursive invocation of this lookup procedure.

The above procedure will always find a non-abstract, accessible method to invoke, provided that all classes and interfaces in the program have been consistently compiled. However, if this is not the case, then various errors may occur. The specification of the behavior of a Java virtual machine under these circumstances is given by *The Java Virtual Machine Specification, Second Edition*.

We note that the dynamic lookup process, while described here explicitly, will often be implemented implicitly, for example as a side-effect of the construction and use of per-class method dispatch tables, or the construction of other per-class structures used for efficient dispatch.

15.12.4.5 *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).

If the erasure of the type of the method being invoked differs in its signature from the erasure of the type of the compile-time declaration for the method invocation (§15.12.3), then if any of the argument values is an object which is not an instance of a subclass or subinterface of the erasure of the corresponding formal parameter type in the compile-time declaration for the method invocation, then a `ClassCastException` is thrown.

DISCUSSION

As an example of such a situation, consider the declarations:

```
class C<T> { abstract T id(T x); }
class D extends C<String> { String id(String x) { return x; } }
```

Now, given an invocation

```
C c = new D();
c.id(new Object()); // fails with a ClassCastException
```

The erasure of the actual method being invoked, `D.id()`, differs in its signature from that of the compile-time method declaration, `C.id()`. The former takes an argument of type `String` while the latter takes an argument of type `Object`. The invocation fails with a `ClassCastException` before the body of the method is executed.

Such situations can only arise if the program gives rise to an unchecked warning (§5.1.9).

Implementations can enforce these semantics by creating *bridge methods*. In the above example, the following bridge method would be created in class D:

```
Object id(Object x) { return id((String) x); }
```

This is the method that would actually be invoked by the Java virtual machine in response to the call `c.id(new Object())` shown above, and it will execute the cast and fail, as required.

If the method *m* is a `native` method but the necessary native, implementation-dependent binary code has not been loaded or otherwise cannot be dynamically linked, then an `UnsatisfiedLinkError` is thrown.

If the method *m* is not `synchronized`, control is transferred to the body of the method *m* to be invoked.

If the method *m* is `synchronized`, then an object must be locked before the transfer of control. No further progress can be made until the current thread can obtain the lock. If there is a target reference, then the target must be locked; otherwise the `Class` object for class *S*, the class of the method *m*, must be locked. Control is then transferred to the body of the method *m* to be invoked. The object is automatically unlocked when execution of the body of the method has completed, whether normally or abruptly. The locking and unlocking behavior is exactly as if the body of the method were embedded in a `synchronized` statement (§14.18).

15.12.4.6 *Example: Target Reference and Static Methods*

When a target reference is computed and then discarded because the invocation mode is `static`, the reference is not examined to see whether it is `null`:

```
class Test {
            static void mountain() {
            System.out.println("Monadnock");
            }
    static Test favorite(){
        System.out.print("Mount ");
        return null;
    }

    public static void main(String[] args) {
        favorite().mountain();
    }

}
```

which prints:

```
Mount Monadnock
```

Here `favorite` returns `null`, yet no `NullPointerException` is thrown.

15.12.4.7 *Example: Evaluation Order*

As part of an instance method invocation (§15.12), there is an expression that denotes the object to be invoked. This expression appears to be fully evaluated before any part of any argument expression to the method invocation is evaluated.

So, for example, in:

```
class Test {
    public static void main(String[] args) {
        String s = "one";
        if (s.startsWith(s = "two"))
            System.out.println("oops");
    }
}
```

the occurrence of `s` before “`.startsWith`” is evaluated first, before the argument expression `s="two"`. Therefore, a reference to the string `"one"` is remembered as the target reference before the local variable `s` is changed to refer to the string `"two"`. As a result, the `startsWith` method is invoked for target object `"one"` with argument `"two"`, so the result of the invocation is `false`, as the string `"one"` does not start with `"two"`. It follows that the test program does not print “`oops`”.

15.12.4.8 *Example: Overriding*

In the example:

```
class Point {
    final int EDGE = 20;
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy;
        if (Math.abs(x) >= EDGE || Math.abs(y) >= EDGE)
            clear();
    }

    void clear() {
        System.out.println("\tPoint clear");
        x = 0; y = 0;
    }

}

class ColoredPoint extends Point {
            int color;
    void clear() {
        System.out.println("\tColoredPoint clear");
        super.clear();
        color = 0;
    }

}
```

the subclass `ColoredPoint` extends the `clear` abstraction defined by its superclass `Point`. It does so by overriding the `clear` method with its own method, which invokes the `clear` method of its superclass, using the form `super.clear`.

This method is then invoked whenever the target object for an invocation of `clear` is a `ColoredPoint`. Even the method `move` in `Point` invokes the `clear` method of class `ColoredPoint` when the class of `this` is `ColoredPoint`, as shown by the output of this test program:

```
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println("p.move(20,20):");
        p.move(20, 20);
        ColoredPoint cp = new ColoredPoint();
        System.out.println("cp.move(20,20):");
        cp.move(20, 20);
        p = new ColoredPoint();
        System.out.println("p.move(20,20), p colored:");
        p.move(20, 20);
        }
    }
```

which is:

```
    p.move(20,20):
        Point clear
    cp.move(20,20):
        ColoredPoint clear
        Point clear
    p.move(20,20), p colored:
        ColoredPoint clear
        Point clear
```

Overriding is sometimes called “late-bound self-reference”; in this example it means that the reference to `clear` in the body of `Point.move` (which is really syntactic shorthand for `this.clear`) invokes a method chosen “late” (at run time, based on the run-time class of the object referenced by `this`) rather than a method chosen “early” (at compile time, based only on the type of `this`). This provides the programmer a powerful way of extending abstractions and is a key idea in object-oriented programming.

15.12.4.9 *Example: Method Invocation using* `super`

An overridden instance method of a superclass may be accessed by using the keyword `super` to access the members of the immediate superclass, bypassing any overriding declaration in the class that contains the method invocation.

When accessing an instance variable, `super` means the same as a cast of `this` (§15.11.2), but this equivalence does not hold true for method invocation. This is demonstrated by the example:

```
    class T1 {
        String s() { return "1"; }
    }
    class T2 extends T1 {
        String s() { return "2"; }
    }
    class T3 extends T2 {
                String s() { return "3"; }
        void test() {
            System.out.println("s()=\t\t"+s());
            System.out.println("super.s()=\t"+super.s());
            System.out.print("((T2)this).s()=\t");
                System.out.println(((T2)this).s());
            System.out.print("((T1)this).s()=\t");
            System.out.println(((T1)this).s());
        }
    }
    class Test {
        public static void main(String[] args) {
            T3 t3 = new T3();
            t3.test();
        }
    }
```

which produces the output:

```
s()=        3
super.s()=  2
((T2)this).s()=3
((T1)this).s()= 3
```

The casts to types `T1` and `T2` do not change the method that is invoked, because the instance method to be invoked is chosen according to the run-time class of the object referred to be `this`. A cast does not change the class of an object; it only checks that the class is compatible with the specified type.

15.13 Array Access Expressions

An array access expression refers to a variable that is a component of an array.

ArrayAccess:
    ExpressionName `[` *Expression* `]`
    PrimaryNoNewArray `[` *Expression* `]`

An array access expression contains two subexpressions, the *array reference expression* (before the left bracket) and the *index expression* (within the brackets). Note that the array reference expression may be a name or any primary expression that is not an array creation expression (§15.10).

The type of the array reference expression must be an array type (call it *T*`[]`, an array whose components are of type *T*) or a compile-time error results. Then the type of the array access expression is the result of applying capture conversion (§5.1.10) to *T*.

The index expression undergoes unary numeric promotion (§5.6.1); the promoted type must be `int`.

The result of an array reference is a variable of type *T*, namely the variable within the array selected by the value of the index expression. This resulting variable, which is a component of the array, is never considered `final`, even if the array reference was obtained from a `final` variable.

15.13.1 Runtime Evaluation of Array Access

An array access expression is evaluated using the following procedure:

- First, the array reference expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason and the index expression is not evaluated.
- Otherwise, the index expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason.
- Otherwise, if the value of the array reference expression is `null`, then a `NullPointerException` is thrown.
- Otherwise, the value of the array reference expression indeed refers to an array. If the value of the index expression is less than zero, or greater than or equal to the array's length, then an `ArrayIndexOutOfBoundsException` is thrown.
- Otherwise, the result of the array access is the variable of type *T*, within the array, selected by the value of the index expression. (Note that this resulting variable, which is a component of the array, is never considered `final`, even if the array reference expression is a `final` variable.)

DISCUSSION

15.13.2 Examples: Array Access Evaluation Order

In an array access, the expression to the left of the brackets appears to be fully evaluated before any part of the expression within the brackets is evaluated. For example, in the (admittedly monstrous) expression `a[(a=b)[3]]`, the expression `a` is fully evaluated before the expression `(a=b)[3]`; this means that the original value of `a` is fetched and remembered while the expression `(a=b)[3]` is evaluated. This array referenced by the original value of `a` is then subscripted by a value that is element `3` of another array (possibly the same array) that was referenced by `b` and is now also referenced by `a`.

Thus, the example:

```
class Test {
    public static void main(String[] args) {
        int[] a = { 11, 12, 13, 14 };
        int[] b = { 0, 1, 2, 3 };
        System.out.println(a[(a=b)[3]]);
    }
}
```

prints:

```
14
```

because the monstrous expression's value is equivalent to `a[b[3]]` or `a[3]` or `14`.

If evaluation of the expression to the left of the brackets completes abruptly, no part of the expression within the brackets will appear to have been evaluated. Thus, the example:

```
class Test {
    public static void main(String[] args) {
        int index = 1;
        try {
            skedaddle()[index=2]++;
        } catch (Exception e) {
            System.out.println(e + ", index=" + index);
        }
    }
    static int[] skedaddle() throws Exception {
        throw new Exception("Ciao");
    }
}
```

prints:

```
java.lang.Exception: Ciao, index=1
```

because the embedded assignment of `2` to `index` never occurs.

If the array reference expression produces `null` instead of a reference to an array, then a `NullPointerException` is thrown at run time, but only after all parts of the array access expression have been evaluated and only if these evaluations completed normally. Thus, the example:

```
class Test {
    public static void main(String[] args) {
        int index = 1;
        try {
            nada()[index=2]++;
        } catch (Exception e) {
            System.out.println(e + ", index=" + index);
        }
    }
```

```
                static int[] nada() { return null; }
    }
```

prints:

```
    java.lang.NullPointerException, index=2
```

because the embedded assignment of `2` to `index` occurs before the check for a null pointer. As a related example, the program:

```
    class Test {
        public static void main(String[] args) {
            int[] a = null;
            try {
                int i = a[vamoose()];
                System.out.println(i);
            } catch (Exception e) {
                System.out.println(e);
            }
        }

        static int vamoose() throws Exception {
            throw new Exception("Twenty-three skidoo!");
        }

    }
```

always prints:

```
    java.lang.Exception: Twenty-three skidoo!
```

A `NullPointerException` never occurs, because the index expression must be completely evaluated before any part of the indexing operation occurs, and that includes the check as to whether the value of the left-hand operand is `null`.

15.14 Postfix Expressions

Postfix expressions include uses of the postfix ++ and `--` operators. Also, as discussed in §15.8, names are not considered to be primary expressions, but are handled separately in the grammar to avoid certain ambiguities. They become interchangeable only here, at the level of precedence of postfix expressions.

PostfixExpression:
  Primary
  ExpressionName
  PostIncrementExpression
  PostDecrementExpression

15.14.1 Expression Names

The rules for evaluating expression names are given in §6.5.6.

15.14.2 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 type that is convertible (§5.1.8) to 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) and/or subjected to boxing conversion (§5.1.7) 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 unboxing conversion (§5.1.8) and 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 (unless it is a definitely unassigned (§16) blank final variable (§4.5.4)), because when an access of such 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.

15.14.3 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 ype that is convertible (§5.1.8) to 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) and/or subjected to boxing conversion (§5.1.7) 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 unboxing conversion (§5.1.8) and 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 (unless it is a definitely unassigned (§16) blank final variable (§4.5.4)), because when an access of such 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.

15.15 Unary Operators

The *unary operators* include `+`, `-`, `++`, `--`, `~`, `!`, and cast operators. Expressions with unary operators group right-to-left, so that `-~x` means the same as `-(~x)`.

UnaryExpression:
  PreIncrementExpression
  PreDecrementExpression
  `+` *UnaryExpression*
  `-` *UnaryExpression*
  UnaryExpressionNotPlusMinus

PreIncrementExpression:
  `++` *UnaryExpression*

PreDecrementExpression:
  `--` *UnaryExpression*

UnaryExpressionNotPlusMinus:
  PostfixExpression
  `~` *UnaryExpression*
  `!` *UnaryExpression*
  CastExpression

The following productions from §15.16 are repeated here for convenience:

CastExpression:
  `(` *PrimitiveType* `)` *UnaryExpression*
  `(` *ReferenceType* `)` *UnaryExpressionNotPlusMinus*

15.15.1 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 type that is convertible (§5.1.8) to 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) and/or subjected to boxing conversion (§5.1.7) 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 unboxing conversion (§5.1.8) and 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 (unless it is a definitely unassigned (§16) blank final variable (§4.5.4)),, because when an access of such 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.

15.15.2 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 type that is convertible (§5.1.8) to 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) and/or subjected to boxing conversion (§5.1.7) 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 unboxing conversion (§5.1.8) and value set conversion (§5.1.8). If necessary, format conversion is applied to the difference prior to its being stored in the variable.

A variable that is declared `final` cannot be decremented (unless it is a definitely unassigned (§16) blank final variable (§4.5.4)), because when an access of such 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.

15.15.3 Unary Plus Operator `+`

The type of the operand expression of the unary `+` operator must be a type that is convertible (§5.1.8) to 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 plus expression is the promoted type of the operand. The result of the unary plus expression is not a variable, but a value, even if the result of the operand expression is a variable.

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

15.15.4 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 type that is convertible (§5.1.8) to 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 minus expression is the arithmetic negation of the promoted value of the operand.

For integer values, negation is the same as subtraction from zero. The Java programming language 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` is `+0.0`, but `-x` is `-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.

15.15.5 Bitwise Complement Operator ~

The type of the operand expression of the unary `~` operator must be a type that is convertible (§5.1.8) to a primitive integral type, or a compile-time error occurs. Unary numeric promotion (§5.6.1) is performed on the operand. The type of the unary bitwise complement expression is the promoted type of the operand.

At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand; note that, in all cases, `~x` equals `(-x)-1`.

15.15.6 Logical Complement Operator !

The type of the operand expression of the unary `!` operator must be `boolean` or `Boolean`, or a compile-time error occurs. The type of the unary logical complement expression is `boolean`.

At run time, the operand is subject to unboxing conversion (§5.1.8) if necessary; the value of the unary logical complement expression is `true` if the (possibly converted) operand value is `false` and `false` if the (possibly converted) operand value is `true`.

15.16 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 §15.15 for a discussion of the distinction between *UnaryExpression* and *UnaryExpressionNotPlusMinus*.

The type of a cast expression is the result of applying capture conversion (§5.1.10) to 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 (§15.4) 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.

It is a compile-time error if the compile-time type of the operand may never be cast to the type specified by the cast operator according to the rules of casting conversion (§5.5). Otherwise, at run-time, the operand value is converted (if necessary) by casting conversion to the type specified by the cast operator.

Discussion

Some casts result in an error at compile time. For example, a primitive value may not be cast to a reference type. Some casts can be proven, at compile time, always to be correct at run time. For example, it is always correct to convert a value of a class type to the type of its superclass; such a cast should require no special action at run time. Finally, some casts cannot be proven to be either always correct or always incorrect at compile time. Such casts require a test at run time. See for §5.5 details.

A `ClassCastException` is thrown if a cast is found at run time to be impermissible.

15.17 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 type that is convertible (§5.1.8) to 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 unboxing conversion (§5.1.8) and value set conversion (§5.1.13).

15.17.1 Multiplication Operator *

Entia non sunt multiplicanda praeter necessitatem.
—William of Occam (c. 1320)

The binary `*` operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects. While integer multiplication is associative when the operands are all of the same type, floating-point multiplication is not associative.

If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values.

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 nor NaN is involved, the exact mathematical product is computed. A floating-point value set is then chosen:
  - If the multiplication expression is FP-strict (§15.4):
    - 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 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 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 programming 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.

15.17.2 Division Operator `/`

Gallia est omnis divisa in partes tres.
—Julius Caesar, *Commentaries on the Gallic Wars* (58 B.C.)

The binary `/` operator performs division, producing the quotient of its operands. The left-hand operand is the dividend and the right-hand operand is the divisor.

Integer division rounds toward `0`. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d \cdot q| \le |n|$; moreover, q is positive when $|n| \ge |d|$ and n and d have the same sign, but q is negative when $|n| \ge |d|$ and n and d have opposite signs. There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is `-1`, then integer overflow occurs and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is `0`, then an `ArithmeticException` is thrown.

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 nor NaN is involved, the exact mathematical quotient is computed. A floating-point value set is then chosen:
  - If the division expression is FP-strict (§15.4):
    - 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 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 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 programming 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

15.17.3 Remainder Operator `%`

And on the pedestal these words appear:
“My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!”
Nothing beside remains.

—Percy Bysshe Shelley, *Ozymandias* (1817)

The binary `%` operator is said to yield the remainder of its operands from an implied division; the left-hand operand is the dividend and the right-hand operand is the divisor.

DISCUSSION

In C and C++, the remainder operator accepts only integral operands, but in the Java programming language, it also accepts floating-point operands.

DISCUSSION

The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that `(a/b)*b+(a%b)` is equal to `a`. This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is `-1` (the remainder is `0`). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive; moreover, the magnitude of the result is always less than the magnitude of the divisor. If the value of the divisor for an integer remainder operator is `0`, then an `ArithmeticException` is thrown.

Examples:

`5%3` produces `2` (note that `5/3` produces `1`)
`5%(-3)` produces `2` (note that `5/(-3)` produces `-1`)
`(-5)%3` produces `-2` (note that `(-5)/3` produces `-1`)
`(-5)%(-3)` produces `-2` (note that `(-5)/(-3)` produces `1`)

The result of a floating-point remainder operation as computed by the `%` operator is *not* the same as that produced by the remainder operation defined by IEEE 754. The IEEE 754 remainder operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java programming language defines `%` on floating-point operations to behave in a manner analogous to that of the integer remainder operator; this may be compared with the C library function `fmod`. The IEEE 754 remainder operation may be computed by the library routine `Math.IEEEremainder`.

The result of a floating-point remainder operation is determined by the rules of IEEE arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the floating-point remainder *r* from the division of a dividend *n* by a

divisor *d* is defined by the mathematical relation $r = n - (d \cdot q)$ where *q* is an integer that is negative only if n/d is negative and positive only if n/d is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of *n* and *d*.

Evaluation of a floating-point remainder operator `%` never throws a run-time exception, even if the right-hand operand is zero. Overflow, underflow, or loss of precision cannot occur.

DISCUSSION

Examples:

`5.0%3.0` produces `2.0`

`5.0%(-3.0)` produces `2.0`

`(-5.0)%3.0` produces `-2.0`

(-5.0)%(-3.0) produces -2.0

15.18 Additive Operators

The operators `+` and `-` are called the *additive operators*. They have the same precedence and are syntactically left-associative (they group left-to-right).

AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression `+` *MultiplicativeExpression*
  AdditiveExpression `-` *MultiplicativeExpression*

If the type of either operand of a `+` operator is `String`, then the operation is string concatenation.

Otherwise, the type of each of the operands of the `+` operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

In every case, the type of each of the operands of the binary `-` operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

15.18.1 String Concatenation Operator +

"The fifth string was added after an unfortunate episode in the Garden of Eden . . ."
—John Philip Sousa, *The Fifth String* (1902), Chapter 6

If only one operand expression is of type `String`, then string conversion is performed on the other operand to produce a string at run time. The result is a reference to a `String` object (newly created, unless the expression is a compile-time constant expression (§15.28))that is the concatenation of the two operand strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string. If an operand of type `String` is `null`, then the string "`null`" is used instead of that operand.

15.18.1.1 *String Conversion*

Any type may be converted to type `String` by *string conversion*.

A value *x* of primitive type *T* is first converted to a reference value as if by giving it as an argument to an appropriate class instance creation expression:

- If *T* is `boolean`, then use `new Boolean(`*x*`)`.
- If *T* is `char`, then use `new Character(`*x*`)`.
- If *T* is `byte`, `short`, or `int`, then use `new Integer(`*x*`)`.
- If *T* is `long`, then use `new Long(`*x*`)`.
- If *T* is `float`, then use `new Float(`*x*`)`.
- If *T* is `double`, then use `new Double(`*x*`)`.

This reference value is then converted to type `String` by string conversion.

Now only reference values need to be considered. If the reference is `null`, it is converted to the string "`null`" (four ASCII characters `n`, `u`, `l`, `l`). Otherwise, the conversion is performed as if by an invocation of the `toString` method of the referenced object with no arguments; but if the result of invoking the `toString` method is `null`, then the string "`null`" is used instead.

The `toString` method is defined by the primordial class `Object`; many classes override it, notably `Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double,` and `String`.

15.18.1.2 *Optimization of String Concatenation*

An implementation may choose to perform conversion and concatenation in one step to avoid creating and then discarding an intermediate `String` object. To increase the performance of repeated string concatenation, a Java compiler may use the `StringBuffer` class or a similar technique to reduce the number of intermediate `String` objects that are created by evaluation of an expression.

For primitive types, an implementation may also optimize away the creation of a wrapper object by converting directly from a primitive type to a string.

DISCUSSION

15.18.1.3 *Examples of String Concatenation*

The example expression:

```
"The square root of 2 is " + Math.sqrt(2)
```

produces the result:

```
"The square root of 2 is 1.4142135623730952"
```

The + operator is syntactically left-associative, no matter whether it is later determined by type analysis to represent string concatenation or addition. In some cases care is required to get the desired result. For example, the expression:

```
a + b + c
```

is always regarded as meaning:

```
(a + b) + c
```

Therefore the result of the expression:

```
1 + 2 + " fiddlers"
```

is:

```
"3 fiddlers"
```

but the result of:

```
"fiddlers " + 1 + 2
```

is:

```
"fiddlers 12"
```

In this jocular little example:

```
class Bottles {
    static void printSong(Object stuff, int n) {
        String plural = (n == 1) ? "" : "s";
        loop: while (true) {
            System.out.println(n + " bottle" + plural
              + " of " + stuff + " on the wall,");
            System.out.println(n + " bottle" + plural
                + " of " + stuff + ";");
              System.out.println("You take one down "
                + "and pass it around:");
              --n;
              plural = (n == 1) ? "" : "s";
              if (n == 0)
                break loop;
              System.out.println(n + " bottle" + plural
                + " of " + stuff + " on the wall!");
              System.out.println();
            }
            System.out.println("No bottles of " +
                          stuff + " on the wall!");
        }
    }
```

the method `printSong` will print a version of a children's song. Popular values for stuff include `"pop"` and `"beer"`; the most popular value for `n` is `100`. Here is the output that results from `Bottles.printSong("slime", 3)`:

```
3 bottles of slime on the wall,
3 bottles of slime;
You take one down and pass it around:
2 bottles of slime on the wall!

2 bottles of slime on the wall,
2 bottles of slime;
You take one down and pass it around:
1 bottle of slime on the wall!

1 bottle of slime on the wall,
1 bottle of slime;
You take one down and pass it around:
No bottles of slime on the wall!
```

In the code, note the careful conditional generation of the singular "`bottle`" when appropriate rather than the plural "`bottles`"; note also how the string concatenation operator was used to break the long constant string:

```
"You take one down and pass it around:"
```

into two pieces to avoid an inconveniently long line in the source code.

15.18.2 Additive Operators (`+` and `-`) for Numeric Types

We discern a grand force in the lover which he lacks whilst a free man; but there is a breadth of vision in the free man which in the lover we vainly seek. Where there is much bias there must be some narrowness, and love, though added emotion, is subtracted capacity.
—Thomas Hardy, *Far from the Madding Crowd* (1874), Act IV, scene i

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 (§15.4):
  - 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 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 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 programming language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

The binary `-` operator performs subtraction when applied to two operands of numeric type producing the difference of its operands; the left-hand operand is the minuend and the right-hand operand is the subtrahend. For both integer and floating-point subtraction, it is always the case that `a-b` produces the same result as `a+(-b)`.

Note that, for integer values, subtraction from zero is the same as negation. However, for floating-point operands, subtraction from zero is *not* the same as negation, because if `x` is `+0.0`, then `0.0-x` is `+0.0`, but `-x` is `-0.0`.

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

15.19 Shift Operators

What, I say, is to become of those wretches?
. . . What more can you say to them than “shift for yourselves?”

—Thomas Paine, *The American Crisis* (1780)

The *shift operators* include left shift `<<`, signed right shift `>>`, and unsigned right shift `>>>`; they are syntactically left-associative (they group left-to-right). The left-hand operand of a shift operator is the value to be shifted; the right-hand operand specifies the shift distance.

ShiftExpression:
  AdditiveExpression
  ShiftExpression `<<` *AdditiveExpression*
  ShiftExpression `>>` *AdditiveExpression*
  ShiftExpression `>>>` *AdditiveExpression*

The type of each of the operands of a shift operator must be a type that is convertible (§5.1.8) to a primitive integral type, or a compile-time error occurs. Binary numeric promotion (§5.6.2) is *not* performed on the operands; rather, unary numeric promotion (§5.6.1) is performed on each operand separately. The type of the shift expression is the promoted type of the left-hand operand.

If the promoted type of the left-hand operand is `int`, only the five lowest-order bits of the right-hand operand are used as the shift distance. It is as if the right-hand operand were subjected to a bitwise logical AND operator `&` (§15.22.1) with the mask value `0x1f`. The shift distance actually used is therefore always in the range 0 to 31, inclusive.

If the promoted type of the left-hand operand is `long`, then only the six lowest-order bits of the right-hand operand are used as the shift distance. It is as if the right-hand operand were subjected to a bitwise logical AND operator `&` (§15.22.1) with the mask value `0x3f`. The shift distance actually used is therefore always in the range 0 to 63, inclusive.

At run time, shift operations are performed on the two's complement integer representation of the value of the left operand.

The value of `n<<s` is `n` left-shifted `s` bit positions; this is equivalent (even if overflow occurs) to multiplication by two to the power `s`.

The value of `n>>s` is `n` right-shifted `s` bit positions with sign-extension. The resulting value is $\lfloor n/2^s \rfloor$. For nonnegative values of `n`, this is equivalent to truncating integer division, as computed by the integer division operator `/`, by two to the power `s`.

The value of `n>>>s` is `n` right-shifted `s` bit positions with zero-extension. If `n` is positive, then the result is the same as that of `n>>s`; if `n` is negative, the result is equal to that of the expression `(n>>s)+(2<<~s)` if the type of the left-hand operand is `int`, and to the result of the expression `(n>>s)+(2L<<~s)` if the type of the left-hand operand is `long`. The added term `(2<<~s)` or `(2L<<~s)` cancels out the propagated sign bit. (Note that, because of the implicit masking of the right-hand operand of a shift operator, `~s` as a shift distance is equivalent to `31-s` when shifting an `int` value and to `63-s` when shifting a `long` value.)

15.20 Relational Operators

The *relational operators* are syntactically left-associative (they group left-to-right), but this fact is not useful; for example, `a<b<c` parses as `(a<b)<c`, which is always a compile-time error, because the type of `a<b` is always `boolean` and `<` is not an operator on `boolean` values.

RelationalExpression:
  ShiftExpression
  RelationalExpression `<` *ShiftExpression*
  RelationalExpression `>` *ShiftExpression*
  RelationalExpression `<=` *ShiftExpression*
  RelationalExpression `>=` *ShiftExpression*
  RelationalExpression `instanceof` *ReferenceType*

The type of a relational expression is always `boolean`.

15.20.1 Numerical Comparison Operators `<`, `<=`, `>`, and `>=`

The type of each of the operands of a numerical comparison operator must be a type that is convertible (§5.1.8) to 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.

The result of a floating-point comparison, as determined by the specification of the IEEE 754 standard, is:

- If either operand is NaN, then the result is `false`.
- All values other than NaN are ordered, with negative infinity less than all finite values, and positive infinity greater than all finite values.
- Positive zero and negative zero are considered equal. Therefore, `-0.0<0.0` is `false`, for example, but `-0.0<=0.0` is `true`. (Note, however, that the meth-

ods `Math.min` and `Math.max` treat negative zero as being strictly smaller than positive zero.)

Subject to these considerations for floating-point numbers, the following rules then hold for integer operands or for floating-point operands other than NaN:

- The value produced by the `<` operator is `true` if the value of the left-hand operand is less than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `<=` operator is `true` if the value of the left-hand operand is less than or equal to the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `>` operator is `true` if the value of the left-hand operand is greater than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `>=` operator is `true` if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, and otherwise is `false`.

15.20.2 Type Comparison Operator `instanceof`

The type of a *RelationalExpression* operand of the `instanceof` operator must be a reference type or the null type; otherwise, a compile-time error occurs. The *ReferenceType* mentioned after the `instanceof` operator must denote a reference type; otherwise, a compile-time error occurs. It is a compile-time error if the *ReferenceType* mentioned after the `instanceof` operator does not denote a reifiable type (§4.7).

At run time, the result of the `instanceof` operator is `true` if the value of the *RelationalExpression* is not `null` and the reference could be cast (§15.16) to the *ReferenceType* without raising a `ClassCastException`. Otherwise the result is `false`.

If a cast of the *RelationalExpression* to the *ReferenceType* would be rejected as a compile-time error, then the `instanceof` relational expression likewise produces a compile-time error. In such a situation, the result of the `instanceof` expression could never be `true`.

DISCUSSION

Consider the example program:

```
class Point { int x, y; }
class Element { int atomicNumber; }
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        Element e = new Element();
        if (e instanceof Point) {       // compile-time error
            System.out.println("I get your point!");
            p = (Point)e;               // compile-time error
        }
    }
}
```

This example results in two compile-time errors. The cast `(Point)e` is incorrect because no instance of `Element` or any of its possible subclasses (none are shown here) could possibly be an instance of any subclass of `Point`. The `instanceof` expression is incorrect for exactly the same reason. If, on the other hand, the class `Point` were a subclass of `Element` (an admittedly strange notion in this example):

```
class Point extends Element { int x, y; }
```

then the cast would be possible, though it would require a run-time check, and the `instanceof` expression would then be sensible and valid. The cast `(Point)e` would never raise an exception because it would not be executed if the value of `e` could not correctly be cast to type `Point`.

15.21 Equality Operators

The equality operators are syntactically left-associative (they group left-to-right), but this fact is essentially never useful; for example, `a==b==c` parses as `(a==b)==c`. The result type of `a==b` is always `boolean`, and `c` must therefore be of type `boolean` or a compile-time error occurs. Thus, `a==b==c` does *not* test to see whether `a`, `b`, and `c` are all equal.

EqualityExpression:
    RelationalExpression
    EqualityExpression `==` *RelationalExpression*
    EqualityExpression `!=` *RelationalExpression*

The == (equal to) and the!= (not equal to) operators are analogous to the relational operators except for their lower precedence. Thus, `a<b==c<d` is `true` whenever `a<b` and `c<d` have the same truth value.

The equality operators may be used to compare two operands that are convertible (§5.1.8) to numeric type, or two operands of type `boolean` or `Boolean`, or two operands that are each of either reference type or the null type. All other cases result in a compile-time error. The type of an equality expression is always `boolean`.

In all cases, `a!=b` produces the same result as `!(a==b)`. The equality operators are commutative if the operand expressions have no side effects.

15.21.1 Numerical Equality Operators `==` and `!=`

If the operands of an equality operator are both of numeric type, or one is of numeric type and the other is convertible (§5.1.8) to 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.

Floating-point equality testing is performed in accordance with the rules of the IEEE 754 standard:

- If either operand is NaN, then the result of `==` is `false` but the result of `!=` is `true`. Indeed, the test `x!=x` is true if and only if the value of `x` is NaN. (The methods `Float.isNaN` and `Double.isNaN` may also be used to test whether a value is NaN.)
- Positive zero and negative zero are considered equal. Therefore, `-0.0==0.0` is `true`, for example.
- Otherwise, two distinct floating-point values are considered unequal by the equality operators. In particular, there is one value representing positive infinity and one value representing negative infinity; each compares equal only to itself, and each compares unequal to all other values.

Subject to these considerations for floating-point numbers, the following rules then hold for integer operands or for floating-point operands other than NaN:

- The value produced by the == operator is `true` if the value of the left-hand operand is equal to the value of the right-hand operand; otherwise, the result is `false`.
- The value produced by the `!=` operator is `true` if the value of the left-hand operand is not equal to the value of the right-hand operand; otherwise, the result is `false`.

15.21.2 Boolean Equality Operators == and `!=`

If the operands of an equality operator are both of type `boolean`, or if one operand is of type `boolean` and the other is of type `Boolean`, then the operation is boolean equality. The boolean equality operators are associative.

If one of the operands is of type `Boolean` it is subjected to unboxing conversion (§5.1.8).

The result of == is `true` if the operands (after any required unboxing conversion) are both `true` or both `false`; otherwise, the result is `false`.

The result of `!=` is `false` if the operands are both `true` or both `false`; otherwise, the result is `true`. Thus `!=` behaves the same as `^` (§15.22.2) when applied to boolean operands.

15.21.3 Reference Equality Operators == and `!=`

Things are more like they are now than they ever were before.
—Dwight D. Eisenhower

If the operands of an equality operator are both of either reference type or the null type, then the operation is object equality.

A compile-time error occurs if it is impossible to convert the type of either operand to the type of the other by a casting conversion (§5.5). The run-time values of the two operands would necessarily be unequal.

At run time, the result of == is `true` if the operand values are both `null` or both refer to the same object or array; otherwise, the result is `false`.

The result of `!=` is `false` if the operand values are both `null` or both refer to the same object or array; otherwise, the result is `true`.

DISCUSSION

While == may be used to compare references of type `String`, such an equality test determines whether or not the two operands refer to the same `String` object. The result is `false` if the operands are distinct `String` objects, even if they contain the same sequence of characters. The contents of two strings `s` and `t` can be tested for equality by the method invocation `s.equals(t)`. See also §3.10.5.

15.22 Bitwise and Logical Operators

The *bitwise operators* and *logical operators* include the AND operator `&`, exclusive OR operator `^`, and inclusive OR operator `|`. These operators have different precedence, with `&` having the highest precedence and `|` the lowest precedence. Each of these operators is syntactically left-associative (each groups left-to-right). Each operator is commutative if the operand expressions have no side effects. Each operator is associative.

AndExpression:
  EqualityExpression
  AndExpression `&` *EqualityExpression*

ExclusiveOrExpression:
  AndExpression
  ExclusiveOrExpression `^` *AndExpression*

InclusiveOrExpression:
  ExclusiveOrExpression
  InclusiveOrExpression `|` *ExclusiveOrExpression*

The bitwise and logical operators may be used to compare two operands of numeric type or two operands of type `boolean`. All other cases result in a compile-time error.

15.22.1 Integer Bitwise Operators &, ^, and |

When both operands of an operator `&`, `^`, or `|` are of a type that is convertible (§5.1.8) to a primitive integral type, binary numeric promotion is first performed on the operands (§5.6.2). The type of the bitwise operator expression is the promoted type of the operands.

For `&`, the result value is the bitwise AND of the operand values.

For `^`, the result value is the bitwise exclusive OR of the operand values.

For `|`, the result value is the bitwise inclusive OR of the operand values.

DISCUSSION

For example, the result of the expression `0xff00 & 0xf0f0` is `0xf000`. The result of `0xff00 ^ 0xf0f0` is `0x0ff0`.The result of `0xff00 | 0xf0f0` is `0xfff0`.

15.22.2 Boolean Logical Operators &, ^, and |

When both operands of a `&`, `^`, or `|` operator are of type `boolean` or `Boolean`, then the type of the bitwise operator expression is `boolean`. In all cases, the operands are subject to unboxing conversion (§5.1.8) as necessary.

For `&`, the result value is `true` if both operand values are `true`; otherwise, the result is `false`.

For `^`, the result value is `true` if the operand values are different; otherwise, the result is `false`.

For `|`, the result value is `false` if both operand values are `false`; otherwise, the result is `true`.

15.23 Conditional-And Operator &&

The `&&` operator is like `&` (§15.22.2), but evaluates its right-hand operand only if the value of its left-hand operand is `true`. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions *a*, *b*, and *c*, evaluation of the expression `((`*a*`)&&(`*b*`))&&(`*c*`)` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `(`*a*`)&&((`*b*`)&&(`*c*`))`.

ConditionalAndExpression:
  InclusiveOrExpression
  ConditionalAndExpression `&&` *InclusiveOrExpression*

Each operand of `&&` must be of type `boolean` or `Boolean`, or a compile-time error occurs. The type of a conditional-and expression is always `boolean`.

At run time, the left-hand operand expression is evaluated first; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8); if the resulting value is `false`, the value of the conditional-and expression is `false` and the right-hand operand expression is not evaluated. If the value of the left-hand operand is `true`, then the right-hand expression is evaluated; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8); the resulting value becomes the value of the conditional-and expression. Thus, `&&` computes the same result as `&` on `boolean` operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

15.24 Conditional-Or Operator `||`

The `||` operator is like `|` (§15.22.2), but evaluates its right-hand operand only if the value of its left-hand operand is `false`. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions *a*, *b*, and *c*, evaluation of the expression `((`*a*`)||(`*b*`))||(`*c*`)` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `(`*a*`)||((`*b*`)||(`*c*`))`.

ConditionalOrExpression:
  ConditionalAndExpression
  ConditionalOrExpression `||` *ConditionalAndExpression*

Each operand of `||` must be of type `boolean` or `Boolean`, or a compile-time error occurs. The type of a conditional-or expression is always `boolean`.

At run time, the left-hand operand expression is evaluated first; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8); if the resulting value is `true`, the value of the conditional-or expression is `true` and the right-hand operand expression is not evaluated. If the value of the left-hand operand is `false`, then the right-hand expression is evaluated; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8); the resulting value becomes the value of the conditional-or expression.

Thus, `||` computes the same result as `|` on `boolean` or `Boolean` operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

15.25 Conditional Operator `? :`

But be it as it may. I here entail
The crown to thee and to thine heirs for ever;
Conditionally . . .
—William Shakespeare, *Henry VI, Part III* (1623), Act I, scene i

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 `Boolean`, or a compile-time error occurs.

Note that it is a compile-time error 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 (§14.8).

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.
- If one of the second and third operands is of type `boolean` and the type of the other is of type `Boolean`, then the type of the conditional expression is `boolean`.
- 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.
- Otherwise, if the second and third operands have types that are convertible (§5.1.8) to numeric types, then there are several cases:
  - If one of the operands is of type `byte` or `Byte` and the other is of type `short` or `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*.
  - If one of the operands is of type `Byte` and the other operand is a constant expression of type `int` whose value is representable in type `byte`, then the type of the conditional expression is `byte`.
  - If one of the operands is of type `Short` and the other operand is a constant expression of type `int` whose value is representable in type `short`, then the type of the conditional expression is `short`.

    - If one of the operands is of type `Character` and the other operand is a constant expression of type `int` whose value is representable in type `char`, then the type of the conditional expression is `char`.
    - 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 unboxing conversion (§5.1.8) and value set conversion (§5.1.8).
- Otherwise, the second and third operands are of types *S1* and *S2* respectively. Let *T1* be the type that results from applying boxing conversion to *S1*, and let *T2* be the type that results from applying boxing conversion to *S2*. The type of the conditional expression is the result of applying capture conversion (§5.1.10) to *lub(T1, T2)* (§15.12.2.7).

At run time, the first operand expression of the conditional expression is evaluated first; if necessary, unboxing conversion is performed on the result; the resulting `boolean` value is then used to choose either the second or the third operand expression:

- If the value of the first operand is `true`, then the second operand expression is chosen.
- If the value of the first operand is `false`, then the third operand expression is chosen.

The chosen operand expression is then evaluated and the resulting value is converted to the type of the conditional expression as determined by the rules stated above. This conversion may include boxing (§5.1.7) or unboxing conversion. The operand expression not chosen is not evaluated for that particular evaluation of the conditional expression.

15.26 Assignment Operators

There are 12 *assignment operators*; all are syntactically right-associative (they group right-to-left). Thus, `a=b=c` means `a=(b=c)`, which assigns the value of `c` to `b` and then assigns the value of `b` to `a`.

AssignmentExpression:
    ConditionalExpression
    Assignment

Assignment:
  LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:
  ExpressionName
  FieldAccess
  ArrayAccess

AssignmentOperator: one of
```
= *= /= %= += -= <<= >>= >>>= &= ^= |=
```

The result of the first operand of an assignment operator must be a variable, or a compile-time error occurs. This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access (§15.11) or an array access (§15.13). The type of the assignment expression is the type of the variable after capture conversion (§5.1.10).

At run time, the result of the assignment expression is the value of the variable after the assignment has occurred. The result of an assignment expression is not itself a variable.

A variable that is declared `final` cannot be assigned to (unless it is a definitely unassigned (§16) blank final variable (§4.5.4)), because when an access of such a `final` variable is used as an expression, the result is a value, not a variable, and so it cannot be used as the first operand of an assignment operator.

15.26.1 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 three ways:

- If the left-hand operand expression is a field access expression (§15.11) *e.f*, possibly enclosed in one or more pairs of parentheses, then:
  - First, the expression *e* is evaluated. If evaluation of *e* completes abruptly, the assignment expression completes abruptly for the same reason.
  - Next, the right hand operand is evaluated. If evaluation of the right hand expression completes abruptly, the assignment expression completes abruptly for the same reason.
  - Then, if the field denoted by *e.f* is not `static` and the result of the evaluationof *e* above is `null`, then a `NullPointerException` is thrown.

    - Otherwise, the variable denoted by *e.f* is assigned the value of the right hand operand as computed above.

- If the left-hand operand is an array access expression (§15.13), possibly enclosed in one or more pairs of parentheses, then:

    - 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, 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 `ArrayIndexOutOfBoundsException` 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 *`TC`* is a reference type, then *`SC`* may not be the same as *`TC`*, 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 (§5.5, §15.16), 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.

- Otherwise, 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.

The rules for assignment to an array component are illustrated by the following example program:

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow extends RuntimeException { }
class RightHandSideThrow extends RuntimeException { }
class IllustrateSimpleArrayAssignment {
    static Object[] objects = { new Object(), new Object() };
    static Thread[] threads = { new Thread(), new Thread() };
    static Object[] arrayThrow() {
        throw new ArrayReferenceThrow();
    }
    static int indexThrow() { throw new IndexThrow(); }
    static Thread rightThrow() {
        throw new RightHandSideThrow();
    }
    static String name(Object q) {
        String sq = q.getClass().getName();
        int k = sq.lastIndexOf('.');
        return (k < 0) ? sq : sq.substring(k+1);
    }
    static void testFour(Object[] x, int j, Object y) {
        String sx = x == null ? "null" : name(x[0]) + "s";
        String sy = name(y);
        System.out.println();
        try {
            System.out.print(sx + "[throw]=throw => ");
            x[indexThrow()] = rightThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[throw]=" + sy + " => ");
            x[indexThrow()] = y;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]=throw => ");
            x[j] = rightThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]=" + sy + " => ");
            x[j] = y;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
    }
    public static void main(String[] args) {
        try {
            System.out.print("throw[throw]=throw => ");
            arrayThrow()[indexThrow()] = rightThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]=Thread => ");
            arrayThrow()[indexThrow()] = new Thread();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]=throw => ");
            arrayThrow()[1] = rightThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]=Thread => ");
            arrayThrow()[1] = new Thread();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        testFour(null, 1, new StringBuffer());
        testFour(null, 1, new StringBuffer());
        testFour(null, 9, new Thread());
        testFour(null, 9, new Thread());
        testFour(objects, 1, new StringBuffer());
        testFour(objects, 1, new Thread());
        testFour(objects, 9, new StringBuffer());
        testFour(objects, 9, new Thread());
        testFour(threads, 1, new StringBuffer());
        testFour(threads, 1, new Thread());
        testFour(threads, 9, new StringBuffer());
        testFour(threads, 9, new Thread());
    }
}
```

This program prints:

```
throw[throw]=throw => ArrayReferenceThrow
throw[throw]=Thread => ArrayReferenceThrow
throw[1]=throw => ArrayReferenceThrow
throw[1]=Thread => ArrayReferenceThrow

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
```

```
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException
Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=StringBuffer => Okay!
Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=Thread => Okay!
Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=StringBuffer => ArrayIndexOutOfBoundsException
Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=Thread => ArrayIndexOutOfBoundsException
Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=StringBuffer => ArrayStoreException
Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=Thread => Okay!
Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=StringBuffer => ArrayIndexOutOfBoundsException
Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=Thread => ArrayIndexOutOfBoundsException
```

The most interesting case of the lot is the one thirteenth from the end:

```
Threads[1]=StringBuffer => ArrayStoreException
```

which indicates that the attempt to store a reference to a `StringBuffer` into an array whose components are of type `Thread` throws an `ArrayStoreException`. The code is type-correct at compile time: the assignment has a left-hand side of type `Object[]` and a right-hand side of type `Object`. At run time, the first actual argument to method `testFour` is a reference to an instance of "array of `Thread`" and the third actual argument is a reference to an instance of class `StringBuffer`.

15.26.2 Compound Assignment Operators

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.

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, 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 (§15.13), 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 `ArrayIndexOutOfBoundsException` 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 §15.17.2), 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.

DISCUSSION

The rules for compound assignment to an array component are illustrated by the following example program:

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow extends RuntimeException { }
class RightHandSideThrow extends RuntimeException { }
class IllustrateCompoundArrayAssignment {
   static String[] strings = { "Simon", "Garfunkel" };
   static double[] doubles = { Math.E, Math.PI };
   static String[] stringsThrow() {
      throw new ArrayReferenceThrow();
   }
   static double[] doublesThrow() {
      throw new ArrayReferenceThrow();
   }
   static int indexThrow() { throw new IndexThrow(); }
   static String stringThrow() {
      throw new RightHandSideThrow();
   }
   static double doubleThrow() {
        throw new RightHandSideThrow();
    }
    static String name(Object q) {
        String sq = q.getClass().getName();
        int k = sq.lastIndexOf('.');
        return (k < 0) ? sq : sq.substring(k+1);
    }
    static void testEight(String[] x, double[] z, int j) {
        String sx = (x == null) ? "null" : "Strings";
        String sz = (z == null) ? "null" : "doubles";
        System.out.println();
        try {
            System.out.print(sx + "[throw]+=throw => ");
            x[indexThrow()] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[throw]+=throw => ");
            z[indexThrow()] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[throw]+=\"heh\" => ");
            x[indexThrow()] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[throw]+=12345 => ");
            z[indexThrow()] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]+=throw => ");
            x[j] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[" + j + "]+=throw => ");
            z[j] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]+=\"heh\" => ");
            x[j] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[" + j + "]+=12345 => ");
            z[j] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
    }
    public static void main(String[] args) {
        try {
            System.out.print("throw[throw]+=throw => ");
            stringsThrow()[indexThrow()] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=throw => ");
            doublesThrow()[indexThrow()] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=\"heh\" => ");
            stringsThrow()[indexThrow()] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=12345 => ");
            doublesThrow()[indexThrow()] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=throw => ");
            stringsThrow()[1] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=throw => ");
            doublesThrow()[1] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=\"heh\" => ");
            stringsThrow()[1] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=12345 => ");
            doublesThrow()[1] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        testEight(null, null, 1);
```

```
            testEight(null, null, 9);
            testEight(strings, doubles, 1);
            testEight(strings, doubles, 9);
        }
    }
```

This program prints:

```
    throw[throw]+=throw => ArrayReferenceThrow
    throw[throw]+=throw => ArrayReferenceThrow
    throw[throw]+="heh" => ArrayReferenceThrow
    throw[throw]+=12345 => ArrayReferenceThrow
    throw[1]+=throw => ArrayReferenceThrow
    throw[1]+=throw => ArrayReferenceThrow
    throw[1]+="heh" => ArrayReferenceThrow
    throw[1]+=12345 => ArrayReferenceThrow

    null[throw]+=throw => IndexThrow
    null[throw]+=throw => IndexThrow
    null[throw]+="heh" => IndexThrow
    null[throw]+=12345 => IndexThrow
    null[1]+=throw => NullPointerException
    null[1]+=throw => NullPointerException
    null[1]+="heh" => NullPointerException
    null[1]+=12345 => NullPointerException

    null[throw]+=throw => IndexThrow
    null[throw]+=throw => IndexThrow
    null[throw]+="heh" => IndexThrow
    null[throw]+=12345 => IndexThrow
    null[9]+=throw => NullPointerException
    null[9]+=throw => NullPointerException
    null[9]+="heh" => NullPointerException
    null[9]+=12345 => NullPointerException

    Strings[throw]+=throw => IndexThrow
    doubles[throw]+=throw => IndexThrow
    Strings[throw]+="heh" => IndexThrow
    doubles[throw]+=12345 => IndexThrow
    Strings[1]+=throw => RightHandSideThrow
    doubles[1]+=throw => RightHandSideThrow
    Strings[1]+="heh" => Okay!
    doubles[1]+=12345 => Okay!

    Strings[throw]+=throw => IndexThrow
    doubles[throw]+=throw => IndexThrow
    Strings[throw]+="heh" => IndexThrow
    doubles[throw]+=12345 => IndexThrow
    Strings[9]+=throw => ArrayIndexOutOfBoundsException
    doubles[9]+=throw => ArrayIndexOutOfBoundsException
```

```
Strings[9]+="heh" => ArrayIndexOutOfBoundsException
doubles[9]+=12345 => ArrayIndexOutOfBoundsException
```

The most interesting cases of the lot are tenth and eleventh from the end:

```
Strings[1]+=throw => RightHandSideThrow
doubles[1]+=throw => RightHandSideThrow
```

They are the cases where a right-hand side that throws an exception actually gets to throw the exception; moreover, they are the only such cases in the lot. This demonstrates that the evaluation of the right-hand operand indeed occurs after the checks for a null array reference value and an out-of-bounds index value.

The following program illustrates the fact that the value of the left-hand side of a compound assignment is saved before the right-hand side is evaluated:

```
class Test {
    public static void main(String[] args) {
        int k = 1;
        int[] a = { 1 };
        k += (k = 4) * (k + 2);
        a[0] += (a[0] = 4) * (a[0] + 2);
        System.out.println("k==" + k + " and a[0]==" + a[0]);
    }
}
```

This program prints:

```
k==25 and a[0]==25
```

The value `1` of `k` is saved by the compound assignment operator `+=` before its right-hand operand `(k = 4) * (k + 2)` is evaluated. Evaluation of this right-hand operand then assigns `4` to `k`, calculates the value `6` for `k + 2`, and then multiplies `4` by `6` to get `24`. This is added to the saved value `1` to get `25`, which is then stored into `k` by the `+=` operator. An identical analysis applies to the case that uses `a[0]`. In short, the statements

```
k += (k = 4) * (k + 2);
a[0] += (a[0] = 4) * (a[0] + 2);
```

behave in exactly the same manner as the statements:

```
k = k + (k = 4) * (k + 2);
a[0] = a[0] + (a[0] = 4) * (a[0] + 2);
```

15.27 Expression

An *Expression* is any assignment expression:

Expression:
AssignmentExpression

Unlike C and C++, the Java programming language has no comma operator.

15.28 Constant Expression

. . . the old and intent expression was a constant, not an occasional, thing . . .
—Charles Dickens, *A Tale of Two Cities* (1859)

ConstantExpression:
Expression

A compile-time *constant expression* is an expression denoting a value of primitive type or a `String` that does not complete abruptly and is composed using only the following:

- Literals of primitive type and literals of type `String` (§3.10.5)
- 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 `? :`
- Parenthesized expressions whose contained expression is a constant expression.
- Simple names that refer to constant variables (§4.12.4).
- Qualified names of the form *TypeName* `.` *Identifier* that refer to constant variables (§4.12.4).

Compile-time constant expressions are used in `case` labels in `switch` statements (§14.10) and have a special significance for assignment conversion (§5.2). Compile-time constants of type `String` are always “interned” so as to share unique instances, using the method `String.intern`.

A compile-time constant expression is always treated as FP-strict (§15.4), even if it occurs in a context where a non-constant expression would not be considered to be FP-strict.

Discussion

Examples of constant expressions:

```
true
(short)(1*2*3*4*5*6)
Integer.MAX_VALUE / 2
2.0 * Math.PI
"The integer " + Long.MAX_VALUE + " is mighty big."
```

. . . when faces of the throng turned toward him and ambiguous eyes stared into his, he assumed the most romantic of expressions . . .
—F. Scott Fitzgerald, *This Side of Paradise* (1920)